6/6/2012

Smart home

Integration of controllable, power consuming components in an intelligent home

Andreas Rask Jensen, s083165

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
IMM-B.Eng-2012-XX

Table of contents

Table of contents 2

Abstract 5

Resume 5

Introduction 6

Contributions 7

Organization of the report 8

Analysis 10

Problem definition 10

Domain 10

Components 11

Sensors and actuators 12

Requirements 16

Actors 16

Formalized requirements 16

Use cases 18

Architecture analysis 20

Distribution of logic 21

Communication and hierarchy 21

The message bus 23

App web service 23

Design 25

Component design 25

Subsystem integration 28

Sensors and actuators by subsystem 31

Data model 32
High level API 33
Lights 34
Wall plugs 34
HVAC 35
Location 36
Alarm 36
Scenarios 37
Data 37
Messages 38
Message types 38
State 40
Model 40
Bus 41
Message topics 42
Security 43
Implementation 45
Bus 45
MSMQ Utilization 45
Subscribe 46
Publish 48
Receiving subscribed items 49
Generics 49
Authentication and Encryption 50
Data model 51
Messages 54
Topic space 54
Evaluation 61

Operation and further development 63

Conclusion 65

Litterature.. 66

Appendix.. 67

Smart home

Integration of controllable, power consuming components in an intelligent home

Abstract

As a part of DTU's participation in the competition Solar Decathlon Europe 2012 a group of students, supervisors and sponsors are working to design and construct a sustainable, energy efficient and innovative home for the future. At the department of Informatics and Mathematical Modeling 8 students including myself, are making the control system of the house supervised by Christian D. Jensen. This dissertation is to document the analysis, design and implementation of my contributions to this system.

The project was made in the spring semester of 2012 from February 6th to June 6th. The project is my final project on Diploma in information technology as DTU and it adheres to the standard terms of DTU for Diploma dissertations.

Resume

Som en del af DTUs deltagelse i konkurrencen Solar Decathlon Europe 2012 skal der designes og bygges et lavenergihus. Dette hus skal indeholde et kontrolsystem som muliggør central styring af lys og indeklima via et enkelt system der skal kunne styres af en enkelt enhed. Denne rapport vil gennemgå hvordan man ved brug af en fælles datamodel, et aftalt beskedformat og en ”Message Bus” kan integrere forskellige kommercielle produkter i et samlet system der kan præsenteres som et samlet API i f.eks. en webservice. Projektet er udført i samarbejde med 7 andre studerende under vejledning af Christian D. Jensen

Denne rapport vil dokumentere analyse, design og implementering af mit bidrag til dette system samt vurdere hvor godt løsningen der er udviklet i mit projekt kan anvendes i den færdige løsning.

Projektet er udført i forårssemestret 2012 fra den 6. februar til den 6. juni og er mit eksamensprojekt på Diplom IT uddannelsen. Rapporten er underlagt DTUs standardaftale for diplom -afgangsprojekter.

Introduction

This project is a part of DTU's entry for the competition Solar Decathlon Europe 2012, which is a quest for students to build a sustainable, energy efficient and innovative home for the future. To be able to explain the scope of my project I will start by explaining about the full project and how this relates to the task that I am trying to solve.

The main reason that I decided to do my dissertation in collaboration with this project is that the project is going to be carried out in full scale. We are building a real house with a functioning control system and this is an excellent opportunity to make a project that is going to be operating and used and where success is not only a matter of how good the idea is, but also how well it is carried out. For my part of the project this is also a challenge because the rest of the team is relying on the solution my project will propose. The team at DTU working on the project consists of approximately 30 students, a handful of professors and corporate sponsors supplying us with domain knowledge and products that we can use. The products used in and around my project are the IHC and a Modicon M340 PLC and these will be described in detail further on.

The team at DTU is divided into sections each responsible for carrying out a specified task. The group that I am a part of is called *installations* and our task is to get everything related to powering and controlling installations in the house. This group is divided further into collaborating subgroups: HVAC, waste and supply water, electricity, entertainment and finally the control system group, of which I am part. We are 8 persons in the group, all working on different parts of the control system and although we are assigned to different tasks and our projects vary in content and score, there will be parts where our projects overlap and I will specify the common parts as well as the contributor where this applies. Where the other peoples' projects focus on implementing a specific subsystem such as the database or the IHC integration, my project focuses on facilitating integration between the different components of the system. Making sure they can communicate and present all sensor information and actuator possibilities as one integrated system: a distributed system.

The house should be energy efficient and innovative which call for an intelligent control system and this is what we are trying to build. I will explain about the specific requirements for the control system later and so for now the requirements are just controlling the electrical appliances in the house in a combined system, saving as much energy as possible. The usage of the above mentioned products and the desire for integrating online services such as weather and calendar data resulted in a distributed, service oriented architecture. Based on availability of hardware drivers and programming APIs for the physical components we made a decision to use C# and .net in a Windows Server 2k8 environment. The nature of

a distributed architecture incites a possibility for use of other platform possibilities at least for the clients, and although we believe it should be a possibility, we have only paid very little attention to this when designing the architecture.

The system has been designed as a multi-tier, decoupled, semi-hierarchical, decentralized and distributed structure allowing all information to be gathered in one system allowing both centralized state-aware decision making as well as local autonomous sense and act logic like closing the windows when it starts raining. The system is presented to the user as a catalogue of services independent of the hardware implementing them in an API intended for mobile app development.

Looking at the nature of the components we have to integrate into our solution both physically connected as the PLC and IHC but also service subsystems as the weather forecast and calendar integration we found that it was a natural choice to use an event based distributed message system and after looking at existing commercial message bus solutions we decided to make our own, as this gives the advantage of full control of the components at least in scope of a single framework namely WCF on .net.

Contributions

This report will document the analysis, design and implementation of the architectural basis of the control system in the house. The criterions for success in my project are the system's ability to integrate the different components in the system: PLC, IHC, a central control unit, and information services such as weather and calendar. They should be able to share information both upon occurrence of an event (asynchronous) or request based (both synchronous and asynchronous). The system should be able to save messages even if subsystems or the transport does not respond at the time the message is transmitted. The messages should be encrypted, authenticated and it should be possible for the message broker as well as the receiving system to identify the sender of a message. Finally it should also be possible to add new hardware as well as information subsystems without having to reprogram the entire system. This should be done by providing documentation and an API that facilitates integration with the bus.

Being the first in the group to finish the project, the system is not ready to deploy at the time this report is delivered. The system does however have a foundation and a way of communicating and based on my project the rest of the group will be able to make their parts of the project work together. My project revolved around the integration of all the systems by combining the information and functionality and presenting this as a single API. This API will be used by Philip designing the app and Carsten and Morten

designing and building the central logic unit. The backbone of the system – the message bus – was designed and developed by both Søren and me. My part has focused on the design and his on the implementation, with mutual contributions to each other's work. I can therefore only take partial credit for the implementation of the bus.

The literature referenced has been used to freely adapt the ideas and concepts that they describe. The message bus concept was developed from group discussions and backed up with theory from the book, Enterprise Integration Patterns[1], and advice from Christian. Design principles taken directly from the literature will be referenced otherwise they are considered common knowledge in the field of IT. Other books have been used for inspiration and guideline to documentation[2] and nature of distributed systems [3]and their influence should be acknowledged, but cannot be referenced precisely to a principle or a section in this report.

At the end of my project we have a running system that integrates test instances of the physical components. The programming and installation of these has not been completed yet as the persons in charge of them still have 2 months left of their respective projects. The bus is using the data model that I have created with Emil and the topic space and message types I have developed. The weather service integration has also been implemented and at the time of writing the only working "passenger" on the bus. The test instances of the physical devices can utilize the message system and the data model to communicate over the bus, and it is possible to share data and execute commands over the bus.

Organization of the report

The report is organized in the following way:

The analysis chapter describes the domain of home control with the possibilities in sensors and actuators and how subsystems can be integrated to be able to share information. It is described how the architecture was decided and which possibilities it gives.

The design chapter focuses on how the structure of the system was designed and how the subsystems were integrated using messages and a common data model to transfer data. It also describes the design

---

[1] G. Hopfe & B. Woolf et al. Enterprise Integration Patterns [Designing, building and deploying messaging solutions] Pp 137
[2] B. Bruegge & A. Dutoit, Object-Oriented Software Engineering [Using UML, Patterns and Java™]
[3] A. Tanenbaum & M. Steen, Distributed Systems [Principles and Paradigms]

characteristics of the bus and the part of the subsystems connected to the bus: middleware. It also covers the design of security on the bus.

The implementation chapter shows how the bus was implemented and how it can be communicated through. It shows how the security mechanisms on the bus work and how the data model can be used by the message hierarchy to send information and commands over the bus.

The evaluation chapter evaluates and discusses the system with regards to its possibilities and limits. It proposes how it can be further developed and how the surrounding project can reach its goals using my findings.

The conclusion chapter summarizes the results of my project by measuring what has been achieved compared to the goals of the project.

Analysis

Home control systems have become more and more popular over the years and regular old-fashioned interfaces as wall switches and thermostats are being substituted with touch screen interfaces, smart phone/tablet apps centralizing control and allowing users to easily control the surrounding environment. Embracing this trending phenomenon, we, in the control group will try to make a versatile system that could integrate an arbitrary number of standards, products or protocols in one system that could be controlled and configured by the consumer.

Problem definition

The apparent problem definition is to develop a home automation system for the contribution to the Solar Decathlon competition in Madrid. The requirements for this specific purpose are, of course, defined by the context of the competition, but the system developed should be universally deployable and hence it should be able to be used on residential premises as well as office buildings and factories.

The task we are trying to solve in the control group is making a system that can integrate the physical installations in a control system. These physical installations have changed a lot during the project and it became obvious that the system should be made so it could include all kinds of devices connected in all kinds of ways. The unknown number and format of the devices handling control of different parts of the system called for a hierarchical solution where these devices would have to be split from the common logic that could integrate them and this again had to be split from the user interface (UI).

Domain

Home automation has rapidly been evolving over the past 20 years and a lot has happened since C. R. Stevens and D. E. Reamer invented “Method and apparatus for activating switches in response to different acoustic signals”[4] also known as “the clapper”. Here in Denmark home automation has not had its big break through but commercial solutions for home control does exist. Some of the more well known systems include Z-Wave, Zigbee and IHC. They are all wireless and can connect sensors and actuators in a single system. The difference is that the IHC is a commercial packet solution where the other 2 are open standards with multiple producers.

[4] http://www.patentstorm.us/patents/5493618.html

Through the past couple of years more and more producers of single-service components like radiators, air condition units and window systems have joined in and developed their systems. These systems can be operated by a touch screen or even a smart phone or tablet app. This development is very interesting, but the problem is however that soon the hallway of a modern home will have a wall mosaic with different home control interfaces each controlling their own part of the home. This can make it hard for the normal consumer to operate with the result of a bad user experience. Another problem with these separate systems is that they do not know about each other and can therefore not share data.

So what do you do if your home in equipped with multiple protocols and devices that are not integrated? This is exactly the problem that we face in the competition. Our solution is to build an integration system on top that allows exchange of information and a central way of controlling the system with a single interface that can be customized to fit the installation.

Components

The components being used in the solution have different advantages and can integrate different sensors and actuators.

IHC

The IHC is a product from Schneider Electric that can be programmed to enable and disable electric outputs (230V and 12V) based on input and conditions. The IHC will be used to control lighting and receive input from sensors available for this system. These sensors will be described in the design section.

The IHC has USB and Ethernet interfaces which can be used to program, configure and interact with the system.

PLC

The PLC is in this project a Modicon M340 from Schneider Electric. The PLC will be used to control HVAC (heating, ventilation and air conditioning). The sensors and actuators available for this device will be discussed in the design section.

The PLC has USB and Ethernet interfaces which can be used to program, configure and interact the system. The PLC is delivered with an OPC server that handles the communication over Ethernet.

Nilan Unit

The Nilan unit uses water and air to control the indoor climate conditions. We will use it for mechanical ventilation and heating of the domestic hot water tank.

The Nilan unit has a serial interface which can be used to interact with the system.

WindowMaster

The window master system will control natural ventilation using one or more windows. The system is partly autonomous and uses its own sensors to determine when it should ventilate. In addition to that it is possible to manually override the system. Natural and mechanical ventilation should be coordinated to avoid conflicts which cause increased power usage. It should not be possible to open enterable windows when the house is uninhabited.

The Window Master system runs on KNX which can be connected to through Ethernet.

BCPM

The BCPM will measure power generation- and consumption.

The BCPM has an Ethernet interface and will also use the OPC server to communicate with the PC.

Sensors and actuators

Sensors and actuators are the backbone of a control system and to be able to make decisions about how to monitor and control the house I will present different sensors and actuators that could have relevance to our system and suggest how it can be used in an intelligent home control system.

Sensors

- **Air temperature**
  If we want to be able to regulate the temperature by a set point instead of based on user perception it has to be known if the temperature is below or above the set point. It would be optimal to have multiple sensors placed where occupants spend most time.
  Outside temperature is also important for deciding when to use natural ventilation instead of mechanical. Although we are using heat exchange in the mechanical ventilation, this will use more power and may not be advantageous if the temperature outside temperature is not too far from the set point.
- **Water temperature**
  Water temperature can be measured multiple places as we are going to have a vast number of water circuits. The temperature should be measured in heat exchanging circuits as the solar panel

cooling circuit and the ground heat exchange circuit as well as the domestic hot water tank. It is though, up to the HVAC group will decide where and when to measure water temperature. Although these values will, most likely, only be used in autonomous processes we would still want this information to be shared.

One place where it could be relevant for the higher levels of the control system to know about temperature is in the domestic hot water tank. In this tank excess energy can be stored and based on the weather forecast it can be decided when to cool the solar panels and at the same time heat the domestic hot water tank's water.

- **CO2**

  Air quality is important and must not exceed the limits defined in the competition rules. Multiple sensors should be placed in the house both where occupants spend most time but also where the competition reference sensor is placed.

- **Humidity**

  As well as for CO2 there are rules for how humid the air can be. In order to know when to ventilate we need to measure humidity where occupants spend most time.

- **Air pressure**

  The air pressure can indicate how the weather is going to be. As we have access to a weather forecast this is not believed to be important.

- **LUX**

  The competition specifies rules for how much light there should be at the work station in the house and therefore we need to measure the LUX level. When not in competition this sensor could be used to determine if there is light enough. This could be relevant for controlling lights and blinds.

- **Smoke**

  It is advised to have one or more smoke sensors in your house, and to be able to make an alert that is not only audible within the premises we need to have a smoke sensor that can integrate with the control system.

- **Power consumption**

  This can be used to make the inhabitant aware of what uses power and when.

- **Power generation**

  Monitoring power generation could be used to determine what to do with excess power. Do we store it in a battery or capacities like the domestic hot water tank or do we sell to the grid.

Power generation data could also be compared to cloud cover rates from the weather forecast to see if we can forecast generation of power and save energy.

- **Wind**

  Wind data can be obtained from the weather forecast or live from outside. The latter can be used to determine if opening the windows is a bad idea.

- **Motion**

  Placement of motion sensors allows us to control things based on presence and timeouts. These kinds of sensors can be perceived as privacy invasive and should be placed with that in mind.

- **Door opening**

  This sensor can be used to detect a break-in but it could also be used to inform the system that someone left or entered the premises for use in light or heat control.

- **Window opening**

  This information can be used to detect a break-in but it could also be used to inform the system that natural ventilation is ongoing and mechanical ventilation should therefore be turned off.

- **Floor sensors**

  To detect presence this is more accurate than motion as motion sensors by nature only detects motion. These kinds of sensors can be perceived as privacy invasive and should be placed with that in mind.

- **Sound**

  Also known as microphones. This sensor can be used for voice control of the system. These kinds of sensors can be perceived as privacy invasive and should be placed with that in mind.

Human Interface Sensors

- Light switches
- Other switches
- Wall panels
- Touch screens
- Remote controls
- Gesture sensors (Kinect)

Actuators

- **Light**

  Lights can be turned on or off and some lights can even have variable intensity using a dimmer.

- **Door lock**

The door can be (un)locked remotely by using electromagnetism.

- **Window motor**

  Windows can be opened and closed using a motor. The motor controller can be with or without wire.

- **Blind motor / smart glass**

  Motor controlled blinds enable the house to automatically keep sunlight out of the house. The same goes for smart glass.

- **Water Valve**

  Valves can direct the flow in a pipe circuit. This could be used for floor heating, photo voltaic cooling, and ground heat exchange.

- **Pump**

  The pump works with the valves to control the flow in a pipe circuit. Increased flow will have greater effect on the exchange of the heat, but the difference in temperature from inlet to outlet will be lower.

Services

Services can be used to gather information from network services to enhance the systems knowledge.

- **Calendar**

  One or more calendars can be used to determine when the premises are inhabited and use this information to save energy by turning off ventilation and lights. It could be an idea to simulate presence when the house is uninhabited for a longer period of time to scare off burglars.

- **Weather forecast**

  The weather forecast can be used to plan energy usage as it could indicate when the photovoltaics will produce power.

- **Environmental warnings**

  In case the local authorities send out a warning for instance about toxic smoke, it could be used to close the windows automatically. These warnings do unfortunately not exist yet, but when they do the system should be able to handle them.

Requirements

The requirements of the system are decided by the competition rules[5] and the local project management[6]. The requirements from the competition rules contain a set of measurable intervals that should be obtained to score the maximum amount of points in the competition. These desired conditions can be described as a scenario – a combination of settings. These can be activated when the house enters the competition. Other scenarios like family dinner could also be made from a collection of different settings. The competition scenario and custom scenarios will be explained in this chapter.

Actors

The actors are mostly decided by the requirements of the system and they will therefore not be questioned, but only described. For the actors we have introduced I will describe why they were introduced and what part of the task that they will be doing.

The user

The user is specified as an inhabitant or a guest of the premises in which the system is installed. A person controlling the building.

The intelligent home control system (CCU)

The intelligent home control system, also referred to as the Central Control Unit (CCU) is the system that performs actions based on configuration and prediction. The requirements mention a central intelligent controlling unit responsible for saving energy and receiving the user's requests: "*…to make it possible for occupants to create certain indoor climate conditions, but letting the CCU decide the best method for obtaining it.*"[7]. This unit is defined as a computer that connects all information from the physical devices being used.

Formalized requirements

"The control system should be made so the occupants have to do a minimum of work to obtain the desired indoor climate conditions, with a minimum of energy consumed."[8] So the vision for the house is written. The document presents some characteristics and requirements which will serve as foundation for the

---

[5] Rule 19 in Appendix "SDE 2012 Rules"
[6] Described in Appendix "Control System Description"
[7] "Control System Description" pp2 l17-19
[8] Vision from Appendix "Control System Description"

requirements for the system. In this section I will try to elaborate and formalize these requirements into something that could serve as a requirement specification for an IT system.

Functional requirements

1. The occupants should be able to control the indoor climate conditions[9]
2. The system should be able to receive indoor climate condition *requests* and decide on the best, most energy efficient way of obtaining this.
3. The system should be automatic and there should be little or no need to do manual overrides if they are considered to increase energy consumption
4. The system should be configurable, allowing the user to specify a set of indoor climate condition values and choose between these scenarios.
5. There should be a scenario with the settings set up by the competition rules.
6. It should be possible to control all functions via its natural, manual interface.
7. The system should be able to schedule energy heavy tasks to be performed when it is favorable to do so with regards to energy usage, price, etc.
8. If a manual action that uses energy is started the user should be notified about the cost of doing this with regards to energy usage.
9. The system should learn the habits of the occupants.

Non-functional requirements

1. Besides natural control interfaces, all functions of the house should be able to be controlled by a single device.

There are no requirements with regards to language, response time or QOS.

Optional requirements

1. The system could have a hibernation function that would save as much power as possible without any damage being inflicted to any components while still being fully aware of what is happening.

   Damage being:

   a. Broken pipes caused by ice.
   b. Rotten or molded food caused by too high temperature in freezer or refrigerator.

[9] Indoor climate conditions include CO_2, humidity, temperature and lux levels

    c. Break in caused by the house looking uninhabited.
2. The system could be remote controlled allowing the user to notify it that someone is coming home and it should wake from hibernation and return to normal indoor climate conditions.
3. The system could be controlled by voice or gestures.

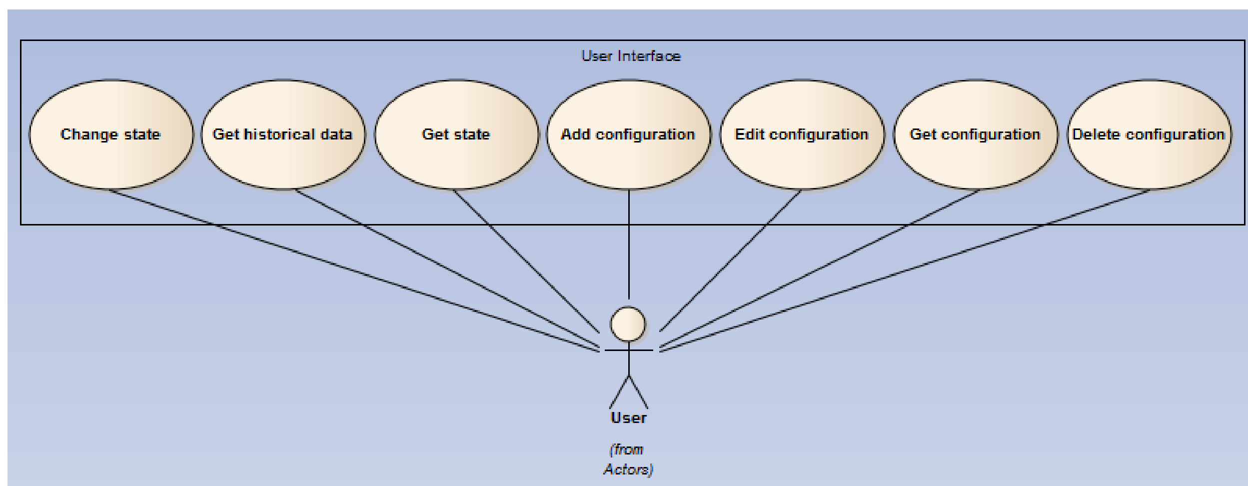
Use cases

The requirements described above can be implemented in a series of use cases defining the system's required capabilities. The need for monitoring devices proposes use cases related to passively or actively receiving sensor data.

Changing the indoor climate conditions based on a single request like turning up the heat means that we need use cases related to make an action to an actuator by changing its state.

The requirements have dictated a way of setting multiple criteria at a time to fit a scenario. In theory this is just multiplying the use case “Change State” but these scenarios could be subject to change and therefore proposes use cases for creating and changing configurations as lists of actuator actions.

Finally the system should learn from the behavior and this necessitates the capability of gathering historical data.



Change state

Change the state of an actuator. For instance turn on the light in the kitchen or open windows. This use case is very broad.

This use case also includes configurations that can be described as multiple actuator state changes.

Prerequisites:

- Knowledge of available actuators
- Knowledge of valid values.

Get state

Get the state of a sensor. That could be getting the living room temperature or check if it is raining.

Prerequisites:

- Knowledge of available sensors.
- Uniform resource identifier of the actuator.

Get historical data

Get historical sensor data for use in analysis or visualization. E.g. get last month's power generation data or who turned off the alarm yesterday?

Prerequisites:

- Knowledge of available data.

Add configuration

Make a list of actuator settings that can be activated at once.

Prerequisites:

- Knowledge of available actuators.
- Knowledge of valid values for actuators.

Edit configuration

Edit actuator settings for an existing configuration.

Prerequisites:

- Knowledge of valid values
- Uniform resource identifier of the actuator

Get configuration

Get existing configurations. This is used when displaying and editing configurations.

Delete configuration

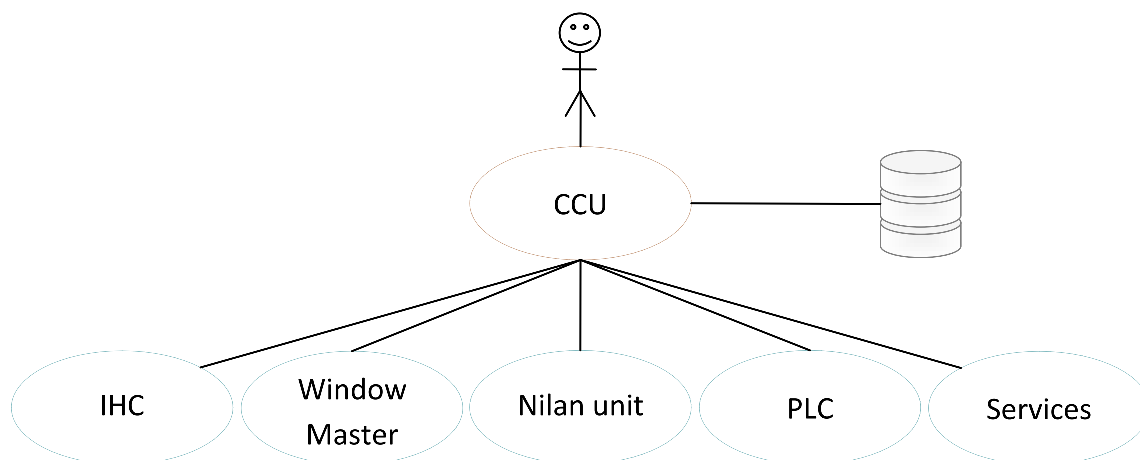
Delete a configuration

Prerequisites:

- Knowledge of existing configurations

Architecture analysis

The system can consists of a selection of the actors already described positioned on the different subsystems. In addition to that there should be a database to store historical data and configurations. The components have different interfaces and can be connected in several different ways where some may be more meaningful than other. The basic structure can be seen on the figure below where the CCU connects the subsystems and the database.



The organization above is strictly hierarchical with the CCU controlling the system with regards to data flow. This simple architecture proposal has some immediate advantages: all communication goes through the CCU which makes monitoring and controlling the system simpler because the CCU will have all information. Another advantage is that the subsystems are all on the same level and thus they do not know about the presence of each other. This makes the subsystem easier to program as they only have to

consider interaction with the CCU. This also means that the CCU should make sure that no two subsystems are counteracting on each other.

Distribution of logic

The requirements state that the system should be able to control itself and this pattern could be cascaded all the way through to the sensors and actuators. Like the human anatomy control functions are divided in multiple levels the system can be designed to react on multiple levels as well. Although maintenance of a single reasoning knowledgebase is much easier it could be an advantage to be able to react quickly to an occurring event. An example could be when putting a hand on a hot boiling plate. It would be a normal reaction by that the reflexes will make sure that the hand is removed immediately minimizing the risk of permanent damage. If the information was to travel all the way to the brain where one could reflect on the psychosocial consequences of removing the hand, then the hand would probably be melted or at least much burned. It is believed that the distribution of logic should have maintenance in mind, but be placed as low as possible, but where enough information is available to make the decision. Distribution of logic will be further discussed and elaborated in the design section.

Communication and hierarchy

Normally when doing communication between applications you would use some kind of client/server pattern in a two way communication. That could be implemented with RPC, web services, DCOM or something else. This way of communicating has a lot of advantages, as you know where your counterpart is located and what it can do for you as this has been configures beforehand. It has however also some limitations: First of all the communication is point to point which necessitates multiple connections if there are more than two participants. In the diagram above that would necessitate the CCU to have connections to all 5 subsystems and the database. Another thing to notice is the tight coupling between the endpoints where you have to have a reference to all the data sources you have contact with and with many of those it can be hard to keep a good overview of your application especially when adding and removing subsystems. This architecture does not support the subsystems to communicate directly and thus there is a single point of failure: the CCU. If the CCU becomes unavailable because of maintenance or a breakdown, the entire system will not be able to communicate and will only rely on autonomous functionality.

In the following I will specify some requirements or desired characteristics for the transport to be able to find an architecture that supports the projects need the most.

Decoupled: The CCU should be able to connect to an arbitrary number of subsystems and if a subsystem fails it should be possible to replace or restart it without any effect on the rest of the system. The decoupling should also regard location as subsystems may not be connected to the same physical computer.

Event driven: The nature of the subsystems is based on an event and a following action. If you press a light switch you will expect a light to turn on. This pattern should be propagated all the way up through the system and therefore the communication should mainly be asynchronous. This does not mean that the CCU should not be able to perform calculations or make decisions synchronously, but as events occur on both the user and the subsystem side of the CCU the logic should mainly be based on events and state within the system being invoked.

Many events can happen at the same time or very close in time to each other and we want to avoid bottle necks both when sending and receiving events. This means that both the sender and receiver must make sure that the process of sending and receiving events does not block or queue other events going in and out. If this is not possible a queue would be preferred over lost events.

Because of the decoupling in the system, two-way communications need special attention. Procedure calls are not reliable and therefore it should be possible to choose both synchronous and asynchronous invocations when requesting information. If synchronous communication is used it would be of great importance to specify a timeout in order to be able to close the thread waiting for the response.

Robust: The system consists of many different subsystems each having own standards and way of communicating not only by protocol but also by data types and the way data becomes available. Therefore it becomes necessary to translate the subsystem-specific implementations into a common standard. This should be done in middleware whose primary task is to ensure the robustness of the system so that the CCU can focus on business logic.

Reliable communication: When the events are transmitted they must not get lost even if the network fails. The transport should guarantee delivery.

Multicast enabled messages: To be able to support a complex and changing data model the data should be distributed as some kind of versatile messages identified by who should receive the message. As some events might be of interest to other subsystems it should be possible to send the events to more than one recipient. The messages will hold only a small amount of data and should not be divided into more messages so that the system would have to deal with sequencing.

The message bus

A solution to these problems could be using a message bus based on publish/subscribe. This approach still has the advantage of a single point of contact, the bus, but whereas the CCU can be unstable and is subject to change the bus would be static and therefore more robust. This approach also enables the subsystem to communicate and a breakdown of the CCU will not break the system but only the part the CCU is responsible for. The organization of the components will look like this:

Programming and configuring an entire messaging system by ourselves seamed out of scope as we wanted to focus on making the subsystems work so I wanted to analyze the market for existing commercial message frameworks. The only product that suited our needs was a product called NServiceBus. It is a robust framework that supports type subscription the way we intend to do it. The cons are that it is a commercial product and it is not free. As it is a commercial product we are not able to configure it precisely to our needs.

While analyzing the market for commercial solutions Søren made a skeleton to a message bus following the characteristics we had specified. This bus was not able to do much but it showed how a data structure could be sent from one application to another which used a TCP connection to connect to a third application and deliver the data. This served as the foundation of the message bus. Although the commercial solution is believed to be well tested we cannot know for sure if we run into a dead end with a critical functionality that is required. Having tried to use and install the NServiceBus I found the programming model to be cumbersome and with a too steep learning curve. After some discussion about which way to go we decided to use our own bus because we have full power over the functionality it provides and can change it to suit our needs.

App web service

The requirements' only non-functional requirement is that the system should be able to be controlled from a single device and to embrace this requirement there should be some kind of service layer that can be used by various devices. Although the actual app is not a part of the system, the service providing the functionality has to be considered. The AppService could either be built on top of the CCU or it could be a separate system. The advantage of building it on top of the CCU is that there are fewer independent devices and the immediate performance of being connected to the most central part of the system. The disadvantages are that it increases coupling and makes the system harder to maintain as the CCU Service would then get large and potentially heavy and immense.

The service could benefit from using a well known standard to be able to support multiple devices. A solution to this could be a REST web service accessed via http(s) and a common data format as JSON or XML.

The nature of http introduces some challenges that will have to be handled by the person developing the AppService. First of all the asynchronous communication on the bus conflicts with the client driven nature of http, where a user makes a request and the server responds. This can be solved by saving messages on the AppService until the client decides to poll the AppService for pending updates to the UI. This solution exaggerates another problem with http: how multiple users can get dynamic contents based on state. Http is by nature stateless and although it is possible to use sessions it still requires the server to keep track of which client have received what. If there is only one device this is however not the biggest of problems as the AppService would just have to save events until the next time the client polls for updates.

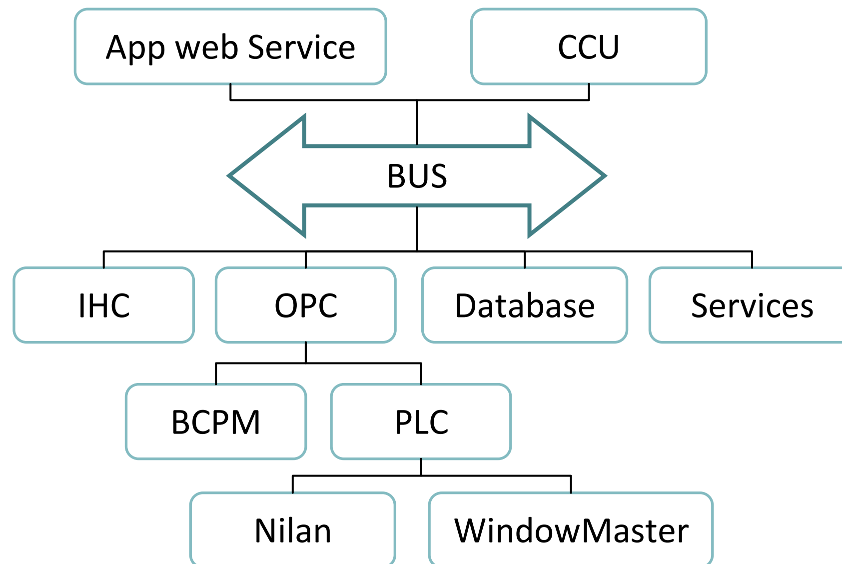
Design

This section will look at the design of the message architecture from design of the bus, the subsystem middleware and the message structure and topic system. In the analysis section the different sensor- and actuator types were presented and in the following I will present the sensors and actuators that is going to be in the system and where. The background for why the sensors belong to the different subsystems was mainly decided by availability and therefore some less obvious placements were made. E.g. the windows can be opened through the WindowMaster system that is attached to the PLC, but the sensor knowing if the windows are open are placed in the IHC. This does however not matter as the bus should not care about where information comes from.

Component design

Until now all subsystems and components have been looked as independent units but when integrating the different components it became apparent that some of the components would have to be arranged hierarchically. This would make the distribution of logic work differently as all information would not have to go over the bus, but could be handled locally at a lower level. In the following I will explain the how devices are connected and why.

Of the physical devices, only the PLC and the IHC are connected directly to the PC running the CCU. The WindowMaster system and the Nilan unit will be connected through the PLC and from the perspective in this report they will be looked at as an autonomous system with a single interface. This interface is accessed through an OPC server which will handle communication from the PLC to the PC. The reason for placing the Nilan unit and the WindowMaster system below the PLC is that logic controlling indoor climate conditions can be moved to a lower level on a platform that is considered highly reliable. This reliability claim is based on the fact that the PLC is used in many existing deployments in industry automation whereas the bus has not yet has its first real deploy. In addition to that the hardware in the PLC is designed to endure variations in temperature and humidity that a normal PC cannot handle. One last thing is that the PLC is running a hard real time operating system that requires low maintenance and has guarantees about how often tasks are attended to, defined as scan time. The PC on the other hand is running on Microsoft Windows Server 2k8, which of course is also considered reliable, but is however not a real time operating system, and requires more frequent maintenance when installing service packs software updates.



The non-physical devices connected to the bus are the database, the CCU and the AppService. Although these subsystems are out of the scope of this report there are subjects specific to these integrations that have changed the way a part of the system works. In addition to that the choices of how these systems are connected have relevance.

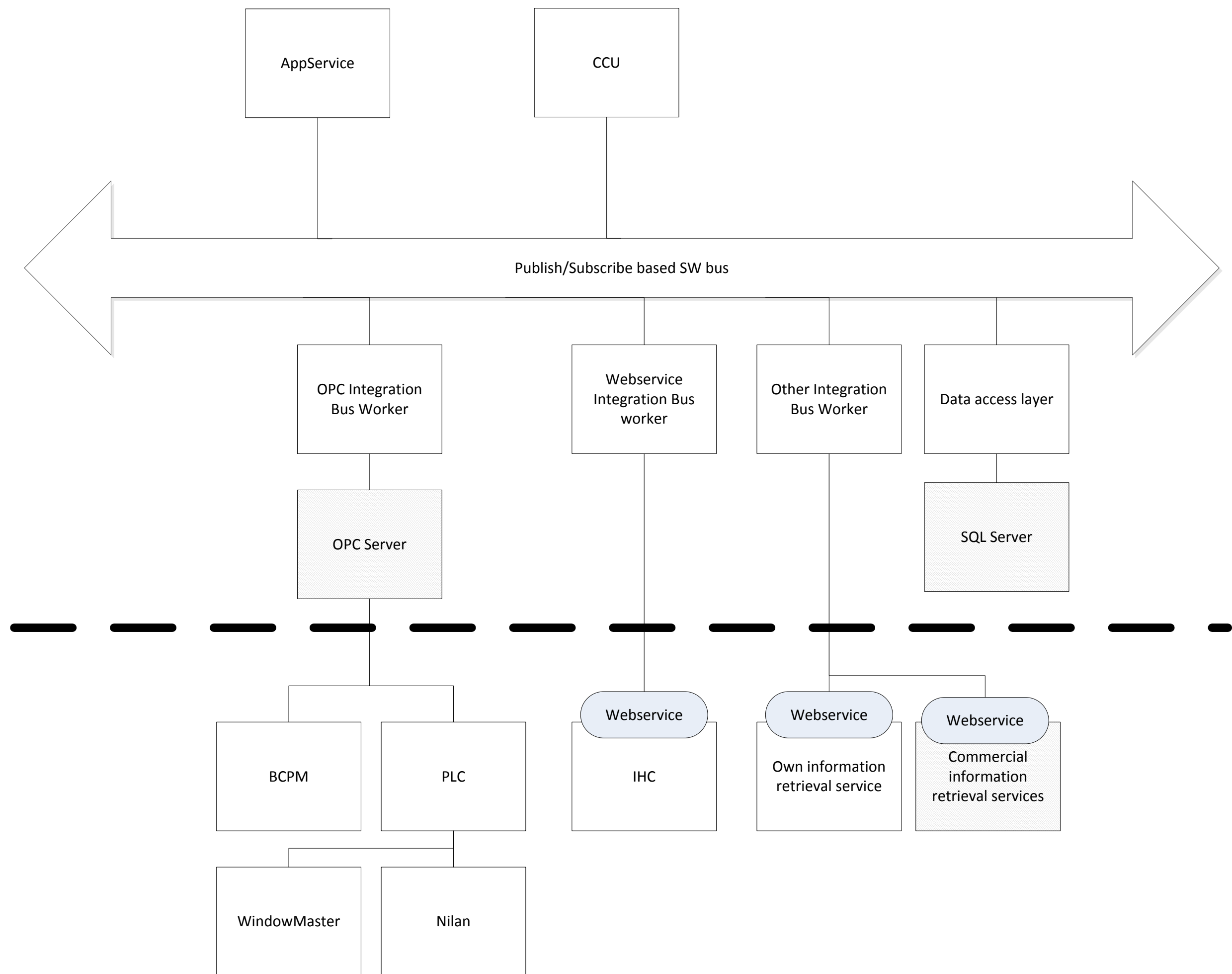
The AppService can be attached either to the CCU or directly on the bus as discussed in the analysis section. Based on the concept of decoupling everything it was a natural choice to integrate this separate from the CCU. The reason for this was in addition to the ones mentioned in analysis, that the AppService has heavy use of the subsystems directly and the goal of a combined API can be achieved using the higher level message types.

The separation of CCU and AppService created a demand for a higher level of requests as the AppService should not know about the complete state of the system such as the CCU. Having mentioned this broadcast problem earlier, this actually comes in handy in this context. If there is a deployment of the system where there are multiple subscribers of requests that could change heat, e.g. a radiator and a floor heating circuit residing in different subsystems. In that case the user may not care where the heat comes from and it is now up to the CCU to decide which provider of heat that provides the best degree/kWh ratio. In fact closing the windows might solve the problems and so the CCU can suggest using this approach. High level messages will be elaborated later.

Direct connection to the bus was also the choice when connecting the database, but this was not as evident as for the CCU. The database is only used by the CCU and the AppService, to store data from sensors and

messages flowing on the bus. This is done to eventually be able to make prediction based logic and to show historical data. As both of those topics are out of scope within this report, the focus will instead be on the consequences of placing the database integration on the bus. A publish/subscribe based messaging system has the advantage of easy access to all communication as any system can subscribe to all items. This is ideal for the database when doing logging as all data is available. The problem is that the database must know something about the system in order to be able to keep the meaning of the data it saves. This problem will be solved by using well known message types and globally identifiable objects. Another problem is that the database works as an autonomous unit and therefore it has to be configured how much data the database should save and how to query data and this is hard to change once the system is deployed. The main argument that settled the decision was that it could be created in such a way that it would be easy to query even using messages as both query and response carrier.

Below is a complete diagram of the system, from now on all subsystems will only be considered as their respective integration service.



Subsystem integration

Live sensor data becomes available in the subsystems in different ways and will have to be handled accordingly. There are several possibilities when dealing with different availability. The two extremes are:

1. To receive data "as is" and handle the differences in format and frequency as late as possible. The main advantage of this approach is that you are not shielding the characteristics of the different polling rates and data types which enable the system to see the raw data at the exact time it becomes available. The cons are that the system will get very hard to maintain as the complexity will only get greater when adding subsystems and/or subsystems components.

2. The other extreme is to normalize the data as close to the source as possible. The main advantage of this approach is that the system can be agnostic with regards to how to monitor different

systems. This makes the system easier to maintain as the systems components gets decoupled and subsystem specific terms are translated to a system standard. This has however also some disadvantages. First of all there is a processing overhead when converting and normalizing data. Secondly you take the risk of losing some of a subsystem's functionality as you are homogenizing data and procedures of how data becomes available. This approach also involves programming the subsystem to use the data types and polling rates best suitable for the other parts of the entire system which emphasizes the disadvantages just described.

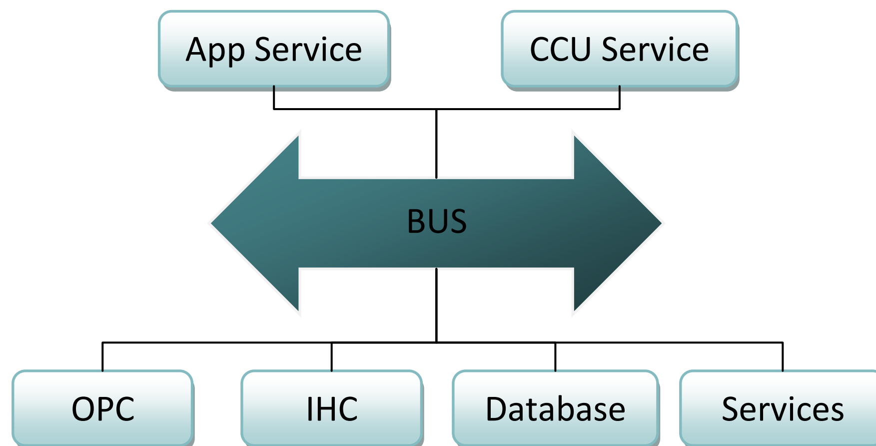
For the components used in the system, both the physical interface and also the protocols and the ability to do conversions on the subsystem side of the wire are different. Therefore it would be necessary to use an approach between the two extremes explained above. Because of the desire for a maintainable and decoupled system some translation layer has to be made. In order increase reliability on the bus it will be of great advantage to make the conversion before it enters the bus and in direct connection to the subsystem's drivers.

When converting the different subsystem's data types to a common data structure using strong typing would be preferred. The primary task is to ensure that the data getting on the bus has been sanitized and validated which would increase the reliability of the bus. Strong typing will be achieved using common namespaces and interfaces. When transporting data we will be using .NET's "service contracts" which also uses common namespaces and it can therefore be validated if data is in the right format.

Each subsystem should have its own integration service that is responsible for translating subsystem specific data to the unified standard using commonly defined sensor and actuator data types. This is done to be able to achieve the decoupling between different subsystems providing the same service. For instance more than one physical device could turn light on and off and although it may be done differently in the two subsystems, it will be presented as the same functionality on the bus. The possibility of multiple providers of the same service can cause some problems when requesting something in a publish/subscribe based system as more subscribers may be invoked by the same request for turning on light. This will be solved using global unique identifiers, GUIDs[10], which will be assigned to every item and message that enters the bus. Although GUIDs, which basically is a 128bit integer, does not guarantee that no two elements will have the same value, it is very unlikely due to its 3,4E38 different values.

[10] http://msdn.microsoft.com/en-us/library/system.guid.aspx

The integration services will use a common interface when connecting to the bus; a piece of message oriented middleware (MOM) where message contents and its type are commonly defined. The different integration systems should be able to use a common code library to make it easier for new subsystems to be integrated and to unify the way the predefined messages are used. It is up to the developers of the integration services to ensure division between subsystem specific logic and the MOM, but the middleware will help facilitating this division by the common library. The integration services should be able to be reused if more than one of the same physical device is used in the system. The common library will consist of message types with predefined fields and a common data model and will be explained in the following 2 sections.



The message bus will be used by the *BusWorkers* seen above, but in theory there could be as many as one would like – including two instances of the same device. They are divided into 2 categories:

- Publishers
- Subscribers

There is of course the possibility of being both a publisher and a subscriber but the properties of such a system is no different than if they were apart. To be able to design the components of the system I will first provide an overview of the subsystems and the sensors and actuators attached to these.

- IHC Service
  This service will be implemented by Anders, but I will propose a message structure that will facilitate communication with the bus.

- PLC Service
  The PLC service will be built by Dainius and Søren, but I will make messages that can be used to transport sensor data and utilize the actuators of the subsystem.
- CCU Service
  The CCU service will be implemented by Morten and Carsten, but I will show how the CCU can receive multiple sensor input and build its own state.
- App Service
  This service is made by Philip who will also be in charge of implementing an app using the app service as service layer.
- Weather forecast service
  I will design and build a Weather forecast service using an online API to get weather and publish it on the bus.
- Calendar service
  The calendar is a part of the CCU, and the implementation will be done by Morten and Carsten.
- Database Service
  The database service will be implemented by Emil and I will collaborate with him when making the data model. He will be in charge of the root elements and I will make the parts used by the message system: actuators, sensors, states, errors and data objects sent over the bus.

Sensors and actuators by subsystem

These are the sensors and actuators that are going to be placed in the house. As mentioned the reason for their position is based on availability from the provider and does not necessarily reflect the best way to do it. This is however not a problem as the purpose of this system is to gather all information and in the end it doesn't matter which system that has specific capabilities as they should be presented as one uniform system.

OPC

Sensors

- Flow
- Temperature
- CO_2
- Temperature in domestic hot water tank.
- Temperature in circulation air

- Temperature (indoor/outdoor)
- Humidity (indoor/outdoor)
- CO2 (indoor/outdoor)
- Wind
- Rain (boolean)
- Power Consumption
- Power Generation

Actuators

- Air temperature set point
- Valves
- Window motors

IHC

Sensors

- Wall switches
- Motion (PIR)
- Door lock
- Magnet sensors (for doors and windows)
- Twilight sensor

Actuators

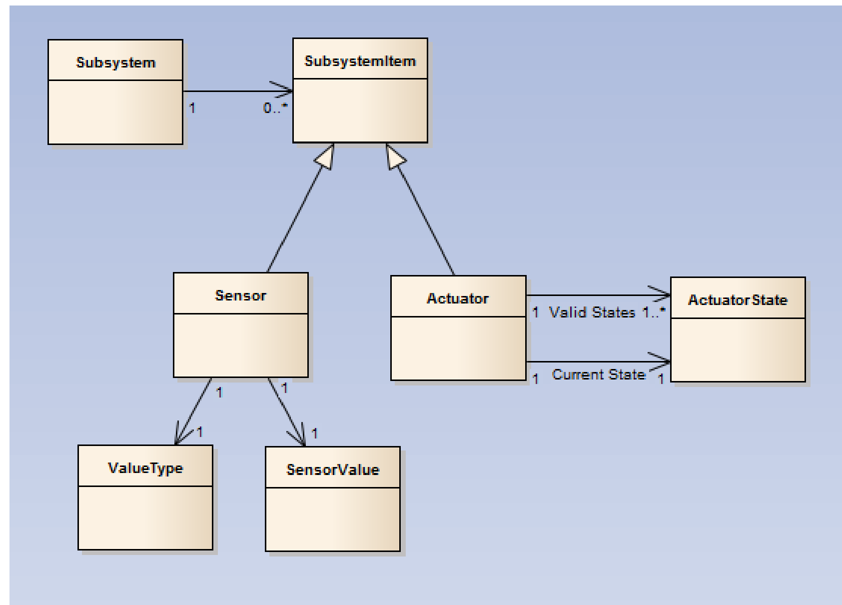
- Lights
- Wall plugs
- Door lock

Data model

The data model has been designed in close collaboration with Emil who is responsible for implementing the database integration service. The model is designed to fit all logical elements of the system which is subsystem, sensors, values and messages.

Sensors and actuators are both subsystem items but a sensor has a value whereas an actuator has a state. It was discussed to merge the types and present them more like the interface the user would know. An example of that is a window which has actuator actions open and close and a sensor can tell if it is open or not. It could therefore be argued that a window was both a sensor and an actuator. Although this would

add a level of abstraction it creates a problem that we do not have time to solve. If the sensor part of an item is located in one subsystem and the actuator part in another then they would have to generate a mutual Id for this device. Although this could be done via some protocol where the CCU would negotiate the Id the value added to the system would not justify the time it would take. The CCU must therefore be configured to know that sensor of window A corresponds to actuator on window A.



High level API

The high level API contains all use cases and they reflect the possibilities the user should have from the app or other user interface. Some of the methods are handled directly by the subsystems and other actions will have to be processed by an informed logic controller as the CCU. In other deployments there could be more CCUs with each their responsibility and they could implement a protocol to distribute information about state and model. In our deployment of the system the high level API has been directed towards our use of the system.

I will describe the different sections of the API and where they should be controlled. The full API documentation can be found in full in the appendix. This is only a suggestion of how the use cases can be realized in the system; it will be up to the developers of the AppService and CCU subsystems to agree on the final interface.

Lights

Lights in this context are not indivisual bulps, but wall outlets leading to a lightsource. If there are multiple lamps on the same outlet they will not be able to be controlled individually. Some lights are dimable which makes it possible to adjust the how much light the lamp should emmit. The control options for lights will therefore be:

- On
  Turns the light on
- Off
  Turns the light off
- Toggle
  State-uninfirmed transition that turns the light on it i was off and the other way around.
- DimSet
  Sets the dim value to a point. E.g. 75%
- DimUp
  State-uninformed transition – regardsless of current level make it brighter.
- DimDown
  State-uninformed transittion – make it darker
- AllOff
  Energy saving function where all light is turned off – regardless of current state.

All requests except AllOff should include a ressource identification so that the system will know what to control. Although lights can be grouped in the IHC configuration it could also be a possibility to make own groups where the system interprets the ressource id as a group and divides the ressource into multiple calls. This would enable controlling light in two seperate subsystems without thinking of the fact that it is two subsystems.

Wall plugs

The interface description of a wall swich is exactly the same as a non-dimmable light, but to follow the principle of adding semantics as low as possible this will be its own type so that the CCU and the UI can distinguish these two. This is very usefull when using the obove mentioned AllOff as some appliances such as a freezer should not be turned off even to save energy. Plugs therefore have the following interface:

- On
  Turns the power in the plug on

- Off
  Turns the power in the plug off
- Toggle
  State-uninformed transition that turns the power in the plug on it i was off and the other way around.

HVAC

Heating, ventilation and airconditioning is controlled autonomously in its own subsystem which makes the CCUs work regarding these values significantly easier. The system is designed to be able to handle communication between multiple providers of heat to prevent these in counteracting on eachother. In our project however, it has been decided that all decistions regarding HVAC should reside in the PLC and the interface description is not complete yet. Therefore this is only a suggestion to the interface.

- HeatSetPoint
  Sets the desired temperature that the system should try to maintain.
- VentilationOn
  Turns on ventilation. The CCU will determine which method is most energy efficient.
- VentilationOff
  Turns on ventilation. The CCU will determine which method is most energy efficient.
- VentilationIncrease
  State-uninformed request to increase the ventilation flow. The CCU will determine if this should be done by opening the windows (more) or if the mechanical ventilation should be increased.
- VentilationDecrease
  State-uninformed request to decrease the flow. The CCU will determinehow this should be done.
- WindowOpen
  Opens the window specified.
- WindowClose
  Closes the window specified
- WindowOpenMore
  Open the window specified more to increase ventilation.
- WindowOpenLess
  Closes the a bit to decrease ventilation.

Location

Information about location can provide the house with information of when the house is inhabited. This information should be used by the CCU to save power. The following methods can help make the house adapt to if it is inhabited or not.

- IamHome
  Tells the system that you are home. This command could be issued when a person is using a device at home.
- WhereIsHome
  Tells the system what should be considered "home" it could be coordinates or a "tag". The tag could be used in the location-field on events in an online calendar service to indicate that you are home
- EnteringHome
  Event that will tell the system that someone is coming home. This method would take an optional time parameter, so that when leaving work the system can begin to climatiz itself and wake from hibernation.
- LeavingHome
  Tells the system that you are leaving the home which could start a prepared, possibly configured, sequence like turn off all lights, turn of ventilation, lock all doors and close all windows.

Alarm

The alarm should be able to be controlled from the system as well. There are of course some security concerns regarding allowing this to be done. The bus' handling of this will be described in the security section. The methods proposed for the alarm:

- EnableAlarm
  Enables the alarm. Any movement detected after the alarm has been enabled will trigger the alarm.
- Deactivate
  By providing a code, it should also be a possibility to deactivate the alarm through the API. The call could take a location parameter that could be limited to a predefined set of locations like home work.
- SetCode
  It should be possible to change the code. Changing the code should only could be done from home providing the previous code.

Scenarios

The scenarios should be coulfigured on by the user and therefore these methods could be needed:

- GetScenarios
  Gets a list of scenario names and Ids to show in a list.
- GetScenation
  Based on the scenario id from "GetScenarios" the user can get all settings for a single scenario.
- CreateScenario
  The user should be able to create new scenarios, specifying specific settings. The settings should not be complete and if a scenario just specifies to lamps to be on then this will be the only change when selecting it.
- EditScenario
  Edit the settings for an existing scenario.
- DeleteScenario
  Deletes the scenario specified by id.
- SetActiveScenario
  Activates a scenario which will send a series of commands for the items involved in the configurations' settings. The settings will be saved in the CCU as it is responsible for packing a scenario out and sending individual commands that the configuration consist of.

Data

Data will be available on the bus through messages, but as discussed earlier there are problems regarding pushing data to mobile devices. Therefore the system should be able to receive information upon request. The request should be of one of the following types:

- Full state
  Publish a broad event asking for everyone to share their state. If there are multiple subsystems this method will return multiple responses that will have to be aggregated in the AppService.
- State of types
  Publish a broad event asking for state on a specific type e.g. Light. Although this method would most likely return a single response this cannot be guaranteed as more subsystems might share state on Lamps. If the system(s) with information about light is down no response will return. As

with the full state, multiple responses will have to be aggregated in the AppService. The case with no response should be handled by a timeout.

- Single item
  Requests the state of a single item. Although it is not guaranteed that only one response will come back aggregation would not be neccesary.

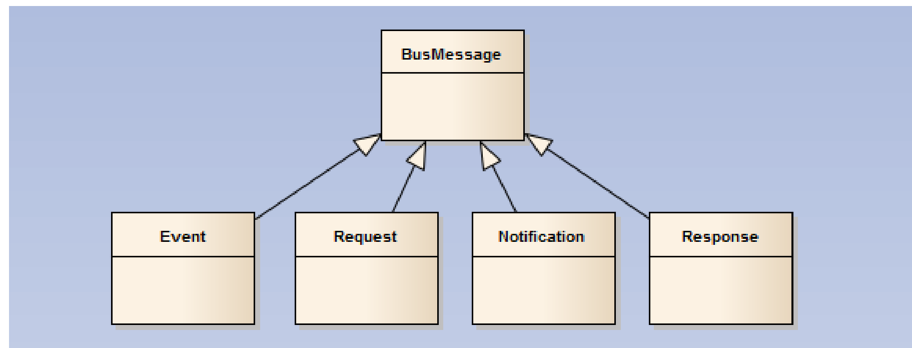
All UI specific methods, higher level events and all logic regarding scenarios will be placed in the CCU and this report will therefore not cover the implementation of these, simply how to achieve the functionality. These methods will be called using messages, more on that in following section.

Messages

The *MessageBus* serves as the backbone of all communication between the attached subsystems. To be able to communicate a common message standard is needed and especially when doing two way communication *BusWorkers* have to agree on a request and response format.

Message types

Messages are designed as hierarchical structure where the messages are grouped in 4 categories:



Event

Messages of type event[11] are defined as time critical messages delivering information about a change or something that has happened in the system. Events can be a sensor exceeding a predefined level or it could be a motion sensor event triggered when motion is detected. This type of messages usually have a

[11] Event message adapted from: [EID] G. Hopfe & B. Woolf et al. Pp 151

limited time of which they are interesting for instance a motion sensor having been activated yesterday does rarely have relevance. The event will therefore have a time specifying when the data is believed not to be interesting anymore.

Sensor data should be published on the bus as events. This enables the information to be distributed when something happens. The CCU and the AppService should subscribe to these events as it enables the CCU to make decisions based on the state information that it has received from the different subsystems and it enables the App to update the user interface if a light was turned on from outside the app on a switch.

The event type can be used in a wide variety of situations, for instance a switch being pressed would create an event from the IHC. This event does not have to be handled locally which enables controlling something residing in another system. This should be achieved by mapping the event type "Switch Event" to a certain action in the CCU.

Notification

Notifications are less time critical than events and although there is nothing in the bus that can make prioritize specified types of messages it could be so at a later stage and therefore the division can be justified. Notifications do not have an expiry and will therefore be equally relevant at any point in time.

Request

A request[12] can be compared to a remote method invocation. When using this message it can be specified whether or not a response is wanted. If a specific response type is wanted then the publisher of the request and the provider of the response must agree on the format of those messages so the target of the invocation subscribes to the request message and the requestor subscribes on the response type.

The complexities of the decoupled publish/subscribe way of doing two-way communication makes it impossible to guarantee if there will be zero or more responses. Therefore the requestor will have to be aware of multiple responses and the lack of a response.

Requests can be used to change state in actuators. This can be achieved in 2 ways: to specify the transition or to specify the goal state. If the action is requested from transition either the current state has to be

[12] Adoption of "Command message" from: [EID] G. Hopfe & B. Woolf et al. Pp 145

known or the transition model has to have same transition possibilities in each state. An example of the latter is to turn up and down the heat where at any point you can still turn it up and down.

Response

The response should fit the request as described above. The response should in addition be versatile enough to display an error if the request was bad. Because of the strong typing in the system, a certain quality can be expected, but invalid data could still result in an exception or a missing ability to process the input. Therefore the requestor must always expect a response with an error message instead of the expected result.

State

When the system is started the different *BusWorkers* on the system does not know each other which is not considered a problem, but what is a problem is that the UI needs to know something about the subsystems actuators in order to present it for the user. The CCU is also having a hard time making decisions if not knowing what effects changing the state of an actuator have. This report will not engage in the discussion of how user request messages will map to lower level request messages that changes the state of actuators, but it should be designed how the state is distributed across the system so that the CCU and UI can get a full set of the systems actuators and sensors. Messages as the ones just described, should be able to distribute this information both when the system starts and upon request.

The complete state of the system consists of a list of all sensors and actuators as strong types as these define what data they contain and what they can do. This data includes the latest sensor value as well as the actuators' current state.

Model

The model consists of all installed components in the system:

- The *Subsystems* / *BusWorkers*
  Even though the CCU initially does not know anything about the other workers connected to the bus, the state will be gradually built as the CCU receives sensor input. The same goes for actuators and their respective interfaces. In this part of the project they are working on a static model, but I see no problem utilizing the message system to distribute information about sensors and actuators making it possible to configure the CCU on the fly as you add new components.
- Sensors and actuators
  As I mentioned before, a dynamic way for the subsystems to provide information about their abilities with regards to sensor information and the possibilities obtained with its actuators, could

> be made. This would of course require reconfiguration of the CCU as we would need to add semantics about the new sensor for the CCU to understand the meaning of the new data. Such semantics should include the position (room) and what actions you want to take based on thresholds or values.

When new bus workers are added there should be little or no need reprogram other parts of the system than the new subsystems middleware. The same goes for existing subsystems where a new sensor or actuator is added. If a new type of sensor or actuator is introduced you would of course need to specify the data type and properties for the new device. If it is a sensor you would additionally need to specify what actions to perform from the data you receive. If it is an actuator you would need to specify the valid values or states as an interface so that the system will know how to facilitate the actuator.

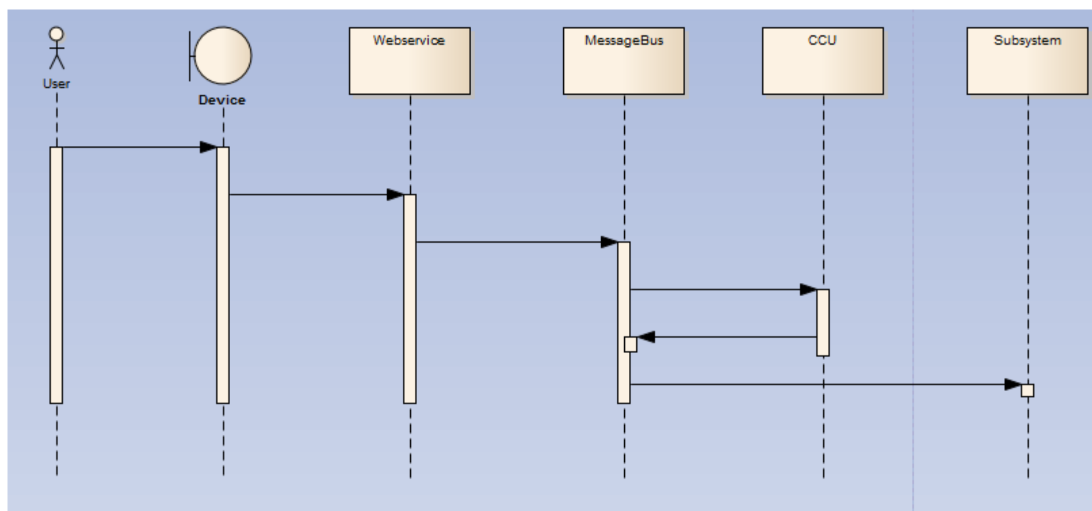
Bus

As mentioned previously the bus uses an implementation of publish/subscribe to distribute the messages. Therefore the bus workers does not know about where messages go on the other side if the bus. The only reference they have is the bus and the bus is then responsible for distributing the message sent (published) to the receivers who are interested (subscribed). This also goes for the recipient which might cause problems if certain messages should only be accepted from approved senders.

When calling a method in your local application you have a good idea of how long it is going to take and can therefore make decisions of when to do it asynchronously and when to wait for the reply before proceeding. With a message bus you cannot be sure if the request is carried out on the local machine where your code is executed or if it is processed on a remote location. This makes it a good idea to make this communication asynchronous. Fortunately this is how the message bus works. You send (publish) a message on the bus and get on with other tasks until the application is invoked by an event subscribed to. When a message is received an event handler is invoked and the message can be processed. This makes it possible for the application to do other things while waiting for a reply opposed to synchronized communication where your thread is blocked while waiting for a response. This does unfortunately also pose some problems. Because of the decoupling in the bus architecture it makes two way communication difficult as you do not know which subsystems subscribe to what, and if they're actually connected to the bus. This will be solved by putting a type in the request that can be used when publishing the response. Unfortunately this has some side effects: this makes it possible for one requestor to receive two replies if two bus workers subscribe to the same request type. Then they will, by definition, both send a reply. The other problem is that the response is not exclusive to you as other bus workers may subscribe to the

response type. In this system this is however not a problem as all information should be available to anyone trusted to be on the bus. A solution could have been to tell the bus not to allow duplicate requests but the design of letting the bus be stateless and without knowledge makes this approach not ideal. The duplicate response would instead be handled by the receiver of the reply and it can discard any duplicate replies.

Delivering messages can be done in two ways: deliver one message at the time and wait for the bus worker to tell when it is ready for the next message. This approach would be thread safe and simpler to program, but by doing this we would not benefit from the possibilities in the event driven architecture. Instead it is now up to the bus worker to decide how to handle messages. Event handlers are called when messages arrives and to make a bus worker where is could only process one message at the time the client would have to make mutual exclusion on event handlers which should not be able to run at the same time. The messages would – if the bus worker does not handle threads – be processed the millisecond they enter the application and thus will be processed simultaneously. This effectively means that when implementing the bus workers one should watch out for race conditions, dirty reads, dead locks and other thread un-safe behavior. If the connection to the actual subsystem is blocked while a request is made then the bus worker should handle other threads trying to acquire this connection and not return an error that the subsystem was unavailable.



Message topics

When designing a public/subscribe architecture one of the first things to look at is how to arrange the messages in order to distribute them correctly. There are two main ways to examine messages in such a system: by content or by topic. Examining the message contents can be very time consuming, but also

comes with extended possibilities to apply sophisticated filters to the messages when processing them. A less time consuming approach is to categorize messages by one or more topics from which the message consumers can subscribe to. An example of this is magazines which are also published and subscribed to. Some people judge the book by its cover and can tell if they are interested by looking at the front page or the publisher whereas other people like to browse through the magazine to see if one or more of the articles are of interest.

A topic space can be modeled to suit the exact needs of the system they are implemented in. The topics will be hierarchically organized so that subscriptions can be made broader and the subscriber will receive a wider variety of messages. In the above example this could mean subscribing to all motorized vehicles.

In the control system the topic system should be hierarchical and divided in logical groups enabling subsystems to get the messages intended for the actuators and sensors they possess. The same goes for the CCU and the App Service as they communicate on a slightly alternate level it should be somewhat obvious what messages belong where.

A Hierarchical strong typed topic space makes it possible to subscribe to an interface or an abstract class and access known fields or test by type to access sub class fields. This is useful for logging and monitoring

The strong types make the system less flexible by adding constraints to the data sent, however it can make it possible to guarantee the structure of the content that is sent.

Security

The bus handles critical tasks in the home control system such as the alarm. Therefore security is an important part of the system and although the system, in our deployment, runs on a single server this may not always be the case. When messages are transported they must therefore be encrypted and the *BusWorkers* allowed to get on the bus must be controlled as anyone on the bus can subscribe to everything and receive all data. The same goes for publishing as there is no restriction on who can publish what on the bus. The only thing we can control is who gets on the bus, and therefore only trusted subsystems should be allowed to connect.

Authentication

To ensure that no messages can get on the bus if the sender is not known and trusted the bus should authenticate the bus workers. This should apply to both publishers and subscribers. Authentication can be obtained in several ways. The two ways I have looked at are using certificates and using Microsoft's Active

Directory. The latter can be implemented on both the Queues directly and on the bus and bus workers message endpoints, the problem is however that it is not a common standard and we could have problems connecting non-windows applications. The other approach would be to use certificates as there are not bound to a specific platform or technology.

Authorization

In order to determine which subsystems has the rights to do what a method to identify the sender is needed. The receiving system can choose to use this information to filter which publishers have the right to do what. This conflicts a little with the idea of decoupling, but in this case the functionality achieved justifies the sacrifice.

The authorization is using the bus workers' canonical names from their certificates and the bus is inserting a field in the message to identify the sender. This approach conflicts with the principle that the bus should not look at or edit the contents of a message, but when the bus has decrypted the message to find its type the cost of updating this field is next to nothing, and when using the canonical name from the certificate we do not add other information, but weakly impersonating the original sender.

Encryption

Encrypting the messages will make the messages unreadable for everyone else than the intended receiver. We have had a lot of discussions about whether to use encryption because most, if not all of the systems will run on the same local network and it would be more feasible to secure the network than to endure the overhead of encrypting every single message. The advantage of encrypting the messages is that if the system at a later point will be distributed across multiple networks or over the internet, we would not want the data to be visible to everyone.

The encryption should be done using x509 certificates, and at this point we have only got self signed certificates. These should of course be switched to real certificates when the solution is deployed.

Implementation

This section covers the development and implementation of the system. The section will mainly cover the bus, but it will also be explained how the message system was implemented and these.

Bus

The transport backbone bus was mainly implemented by Søren and I was in charge of making sure that the bus would meet the requirements for our purpose of distributing events and requests. In this section I will document the implementation of the bus on a rather high level, but explain why it meets the requirements and how.

The service contract for the bus defines the 3 logical methods: publish, subscribe and unsubscribe, but to facilitate publishing data types with generics a separate method was implemented: PublishGeneric. The last method is however hidden for the client, and the client middleware will convert the message received and sent. The reason for having this extra method is that WCF does not handle serialization of generics and we therefore had to do it ourselves. The bus is implemented to be able to use both generic TCP and Microsoft Message Queues (MSMQ). As we will only use MSMQ I will only cover this part of the implementation. As I have previously discussed this has some advantages regarding reliability.

MSMQ Utilization

Microsoft Message Queue Server (MSMQ) works as a server that can receive and hold messages until you get them. I will not go in further detail about how MSMQ works that it will just be treated as a reliable message endpoint where queues can be created and messages can be put in those queues this is out of scope for this assignment so for this section.

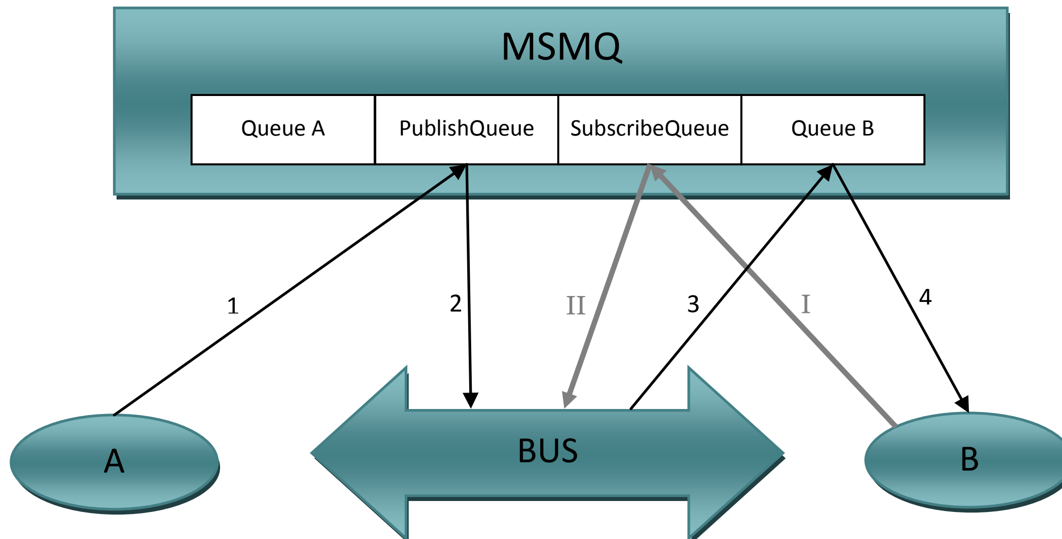
MSMQ is the component that makes durable subscribing[13] a possibility because messages are delivered to the message queue instead of directly to the receiver. If a sub system is down messages will not be lost as long as the message queue server is up and running.

The bus itself has 2 queues. These are treated as input queues for the bus as they handle subscriptions and receiving publications. These are the queues used when the client wants to subscribe and publish

[13] Durable subscriber, adapted from: [EID] G. Hopfe & B. Woolf et al. Pp 522

something on the bus. In addition to the bus' own queues, each bus worker has its own queue. These queues are used by the bus to distribute the events coming into the ingoing publish queue.

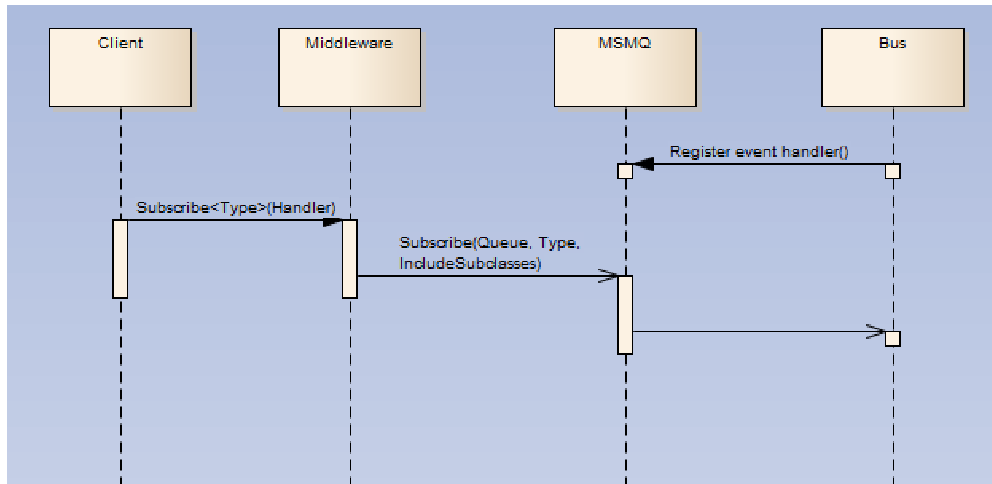
The bold gray arrows show an example of subscribing, where a message is put in the *SubscribeQueue* which only the bus reads from. The thinner black arrows show how a message is published to the bus and distributed to the lone subscriber B.



Subscribe

Subscribe<Message>(EventHandler, ?IncludeSubTypes)

Subscriptions are made by the type of the message and therefore all messages sent have to inherit from *BusMessage*. When subscribing to the type a method taking this type as argument must be supplied. This is obtained by specifying that the delegate's argument types must comply with the generic type argument. The delegate is used as an event handler and fires when messages of the specified type and subtypes arrive. If subtypes should not be included the optional parameter IncludeSubTypes must be set to false. A subscription is initiated from the client like this:



Before the actual subscribe call WCF has registered an event handler on MSMQ when establishing a connection. Therefore messages in the SubscribeQueue will automatically enter the bus. The client is subscribing to events on the bus, but what happens behind the scenes is that the Subscribe-call is altered a bit in the middleware. The middleware handles the transport and if the calling application does not yet have a queue, it will create one. When the middleware make the subscription it sends the queue name defined in the calling application's configuration, the type it wants to subscribe to and an indication of whether subtypes should also be subscribed to. The callback handler has been exchanged with the queue name and therefore the client middleware saves a lookup table with a reference to the type and what method to invoke when the message arrives:

Type	Handler
LightRequest	LightRequestHandler
AlarmRequest	AlarmRequestHandler
StateRequest	StateRequestHandler

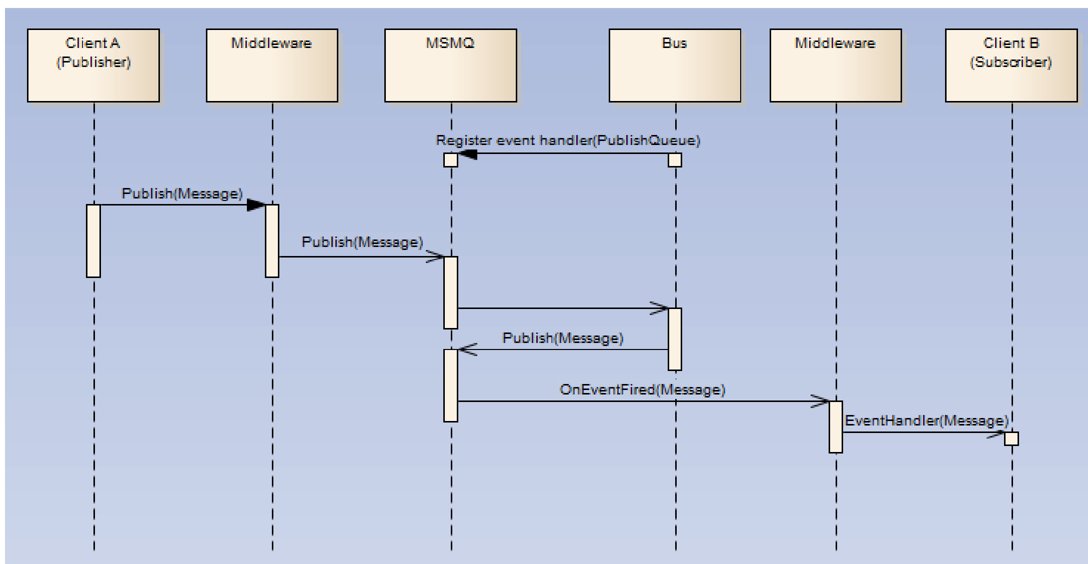
If the "includeSubTypes" flag has not been explicitly set to false, subscriptions will be made for all messages inheriting from that type. Subscribing to BusMessage, which is the root type, therefore means subscribing to all messages. The subscription interpretation is handled on the bus where all message types

inheriting from the subscribed type will be subscribed to. In the clients lookup table all references is also traversed and saved so that sub types are registered to call the same event handler.

Publish

Publish (Message)

Publishing a message is not so different from subscribing as the traffic still goes through MSMQ. Instead of putting the message on the *SubscribeQueue* the, by name, appropriate *PublishQueue* is used. Messages in the input can only inherit from BusMessage and therefore a certain data structure is guaranteed.



Prior to publishing, someone has to have subscribed to the event type that is being sent. If not the message is discarded. As for the subscribe sequence the bus is connected to MSMQ and have had an event handler set up for messages in the *PublishQueue*. When the client publishes the middleware establishes a connection to the MSMQ server through WCF. The message is put in the *PublishQueue* and the bus is invoked with that message. The bus, holding information of who is subscribed to what, distributes the event to all subscribers in their respective queues. The bus maintains the subscriptions in local xml files and can look the subscribers up. The following pseudocode explains the process

```
subscribers = Subscribers[Message.Type]
foreach (subscriber in subscribers)
    proxy = GetProxy(subscriber)
    proxy.OnEventFired(Message)
```

The method GetProxy is a bit more complex than it could seem as it has to set up WCF to make a connection to the client using the service contract "IMsmqSubscriber". In this case the client is the client's queue on MSMQ. When putting the message on the proxy who is actually the receiver's MSMQ, the receiving client will be invoked and will because of the data contract know the format of the message.

Receiving subscribed items

OnEventFired (Message)

When a message subscribed to arrives in the clients queue, WCF will retrieve this message and call the *OnEventFired* method in the client this can be done because the client implements the same service contract: *IMsmqSubscriber*. The type of the message is *BusMessage* or a child type hereof. Based on the actual type of the message, the client middleware can identify which method should be invoked. To ensure that the event handler is not blocking other incoming messages of the same type the call to the event handler has to be done asynchronously.

Generics

When transferring the messages which is just objects in C# we use a the built in serialization. When using data contracts this serialization can be done automatically as long as they share the same contract. For some reason WCF cannot serialize and deserialize objects with generics and therefore we do it ourselves. This leads to a new problem that we can no longer use the same data contract as all data in the contract has to be known before hand so the deserializer knows what to expect. The contract was therefore extended to be able to handle messages with generic types by making a custom serializer and sending strings instead of serialized objects which is by the way also strings; contracted strings. On the other side in the receivers middleware the string is then deserialized and the type is used to generate a new instance of the type with the deserialized XML as its body.

Authentication and Encryption

WCF has a built in authentication and encryption framework and therefore implementing authentication is a matter of configuration rather than programming. WCF provides two basic ways to authenticate the messages: *AD* integrated and using certificates. As discussed earlier *AD* was not an optimal solution so self-signed certificates were generated using *makecert* and put in the machines certificate store under trusted people. The private keys are contained in the certificates, but if the solution was to be deployed on multiple environments then the keys should be available as follows.

	Machine A	Machine B	Machine C
Units on machine:	BusWorker A	Bus Busworker B	BusWorker C
Certificates required:	Public(Bus) Private(A)	Private(Bus) Private (B)	Public(Bus) Private(C)

Encryption in WCF is closely related and in fact also provides two alternatives: encrypting the transport (MSMQ message) or encrypting the message which is the contents of the MSMQ message, the serialized contract data. As we do not use AD authentication on the physical queues on the machine it was enough to encrypt the messages and they are there encrypted with the bus' public key prior to send.

The scenarios works as follows:

Subscribe

Task: A subscribes to messages of type M.

1. A creates the message Subscribe<M>(Handler).
2. A signs the message using own private key (authentication)
3. A encrypts the message with the bus' public key (encryption).
4. A sends the message to the SubscribeQueue
5. The message is retrieved from the queue by the bus
6. The bus decrypts the message using its own private key
7. The bus reads the signature and verifies that it exists in trusted people.
8. The bus accepts the subscription.

In fact anyone could get the message from MSMQ, but only the bus or anyone with the bus' private certificate can decrypt and make sense of it.

Publish

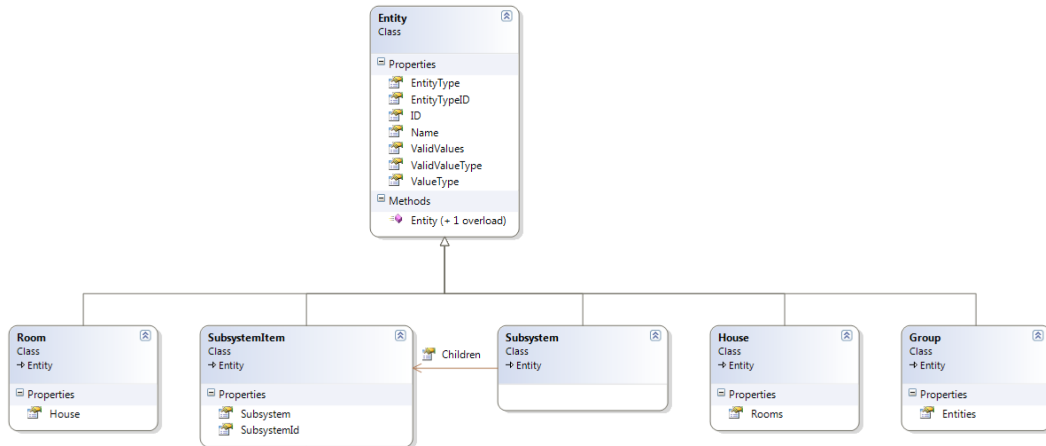
Task: B publishes message M1 of type M (A is subscribed).

1. A creates the message M1.
2. A signs the message M1 using own private key (authentication)
3. A encrypts M1 with the bus' public key (encryption).
4. A sends the message to the PublishQueue
5. The message is retrieved from the queue by the bus
6. The bus decrypts the message using its own private key
7. The bus reads the signature and verifies that it exists in trusted people.
8. The bus adds the canonical name of the authenticated sender's certificate to the message's field: origin
9. The bus finds all subscribers (A)
10. The bus signs the message using own private key
11. The bus encrypts the message with A's public key
12. The bus sends the message to A's queue
13. A is invoked and retrieves the message
14. A decrypts the message using own private key
15. A reads the signature and verifies it using the bus' public key stored on the local machine.
16. A can read the field origin and use it to identify the original sender.

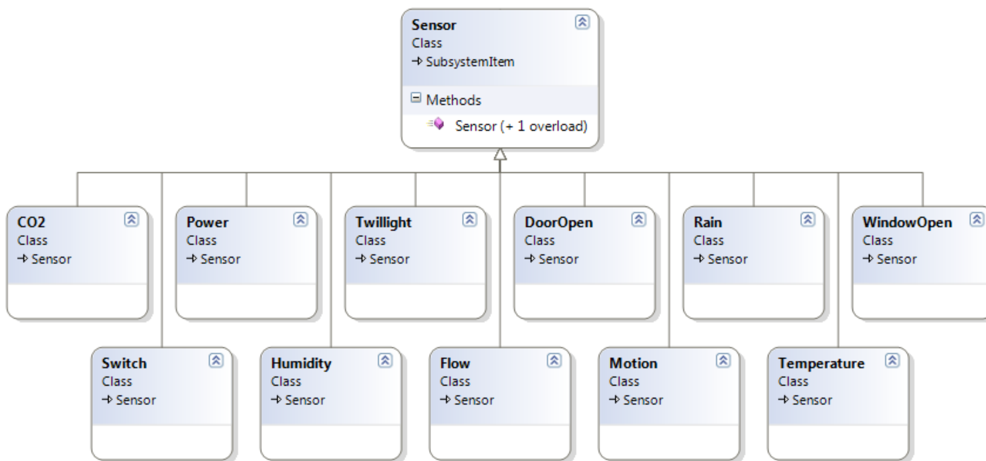
The last point conflicts a little with the decoupling as A should not know about B, but if A wants to limit its resources or use the caller to return context aware material then it should be a possibility. This is a protocol implemented on top of the system, and the origin-field can be ignored for complete decoupling.

Data model

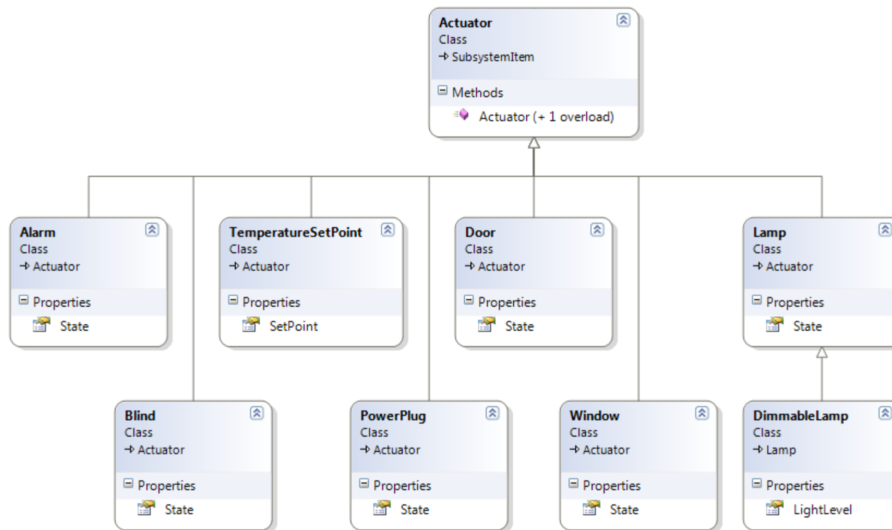
The data model was implemented in collaboration with Emil based on our mutual design. I have augmented the data model with the different sensors and actuators shown below. Although it holds the key elements of the system it is not believed to be complete, but at this stage of the project it reflects the current data.



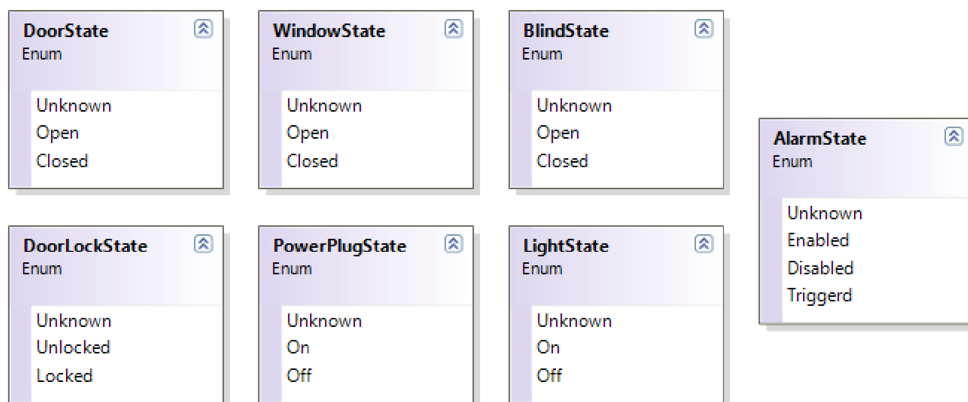
This model show the top of the system and it is used to relate sensors and actuators to a subsystem. In the message space a subsystem is a part of the *StateResponse* message that describes the current state of a subsystem. The distribution of state will be described when the data model and message space has been described in detail.



Sensor inherits from *SubsystemItem* and as described in the design section these strong types identify the subsystem items so that the CCU or the App Service can prepare for the data these sensors publish. All sensors have a property named Value, that holds the value of the latest measurement. This value's type correspond to the type and so a Humidity's value is a floating point number and a WindowOpen's value is both a Boolean to indicate if it is open and a nullable floating point number to represent how open the window is. This number is nullable because not all sensors can provide this value.



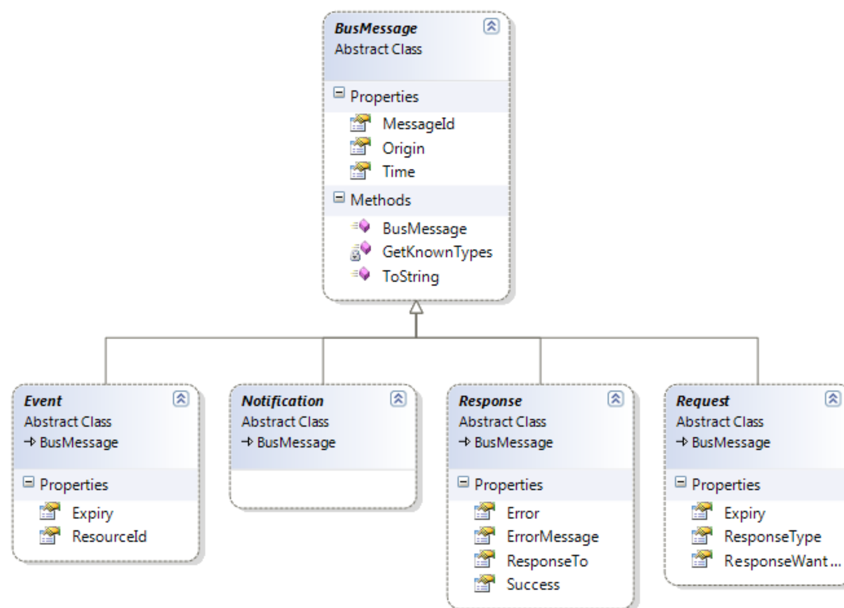
Actuators are like sensors a descendant of *SubsystemItem* . The actuator types hold a current state that corresponds to the available states for the actuator. For instance a door can be Open, Closed and Unknown, an Alarm can be Enabled, Disabled, Triggered and Unknown. The types of the actuators makes it possible not only to see the actuators allowed states but also enables the CCU and the App Service to present and use it based on these types. For instance if a *SubsystemItem* is a Lamp then the icon may be a clickable bulb and if it is a Temperature Setpoint is may be presented as a slider. If the state value is not an enumeration it is strongly suggested to supply a minimum and maximum value.



These enumerations show the suggested valid states of the actuators of the system. The unknown option was implemented as a valid null value and it can be interpreted by the individual subsystem as they wish.

Messages

With the data model in place it is time to get on with the messages. As mentioned all messages should inherit from *BusMessage* and therefore this is the root node. The model below shows the root node and the 4 main message types. There is a full page version in the appendix. As the system is not done this message data model is not final and may be subject to change.



BusMessage is defined as the transport root type and because of that all messages contains at least the fields MessageId, Origin and Time. A message id is used to store the messages in the database and to be able to provide a response to a specific request. Origin is used to identify the subsystem the message comes from. The origin is set by the bus to reflect the canonical name of the sender. This information is gathered from the authenticating certificate. Time is the time that the data was put into the message. This field can be used to sequence messages, but if the message is from two different subsystems it should be handled with care as there is no global clock.

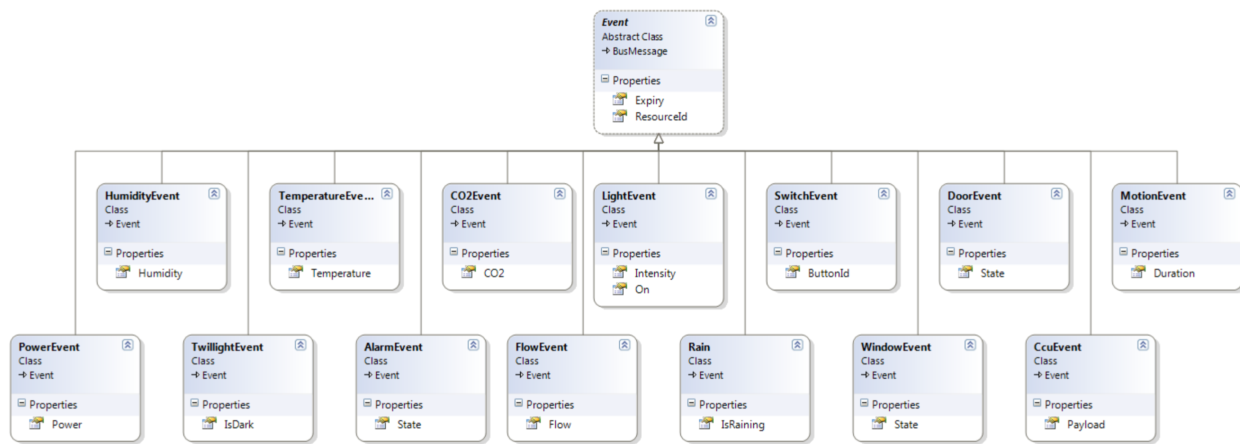
Topic space

The topic space has been implemented using types, which means that subscriptions are handled by the container the messages arrive in. The topic space has been made hierarchical so that it is possible to

request all messages and child messages of a specific type. It even allows subscribing to all messages which is very handy for logging. The topic space is divided into 4 main categories explained below.

Events

Messages of type event has an informing nature and is able to reflect both sensor and actuator events in the system. The class diagram below does not reflect the final Event-library of the system but should be seen as an example of how events can distribute information about the system.



Events exist for all sensors and actuator types and uses data models state or value type to distribute the latest sensor value or actuator state over the bus. The events represent the value of a certain point in time and can be sent by the subsystems to inform the other parts of the system of the current value that a sensor has reported. For actuators these events should be sent after a state change. For sensors this can be done on a regular basis (poll rate) and when certain thresholds are reached. The latter can be implemented by configuration or be implemented as a protocol on top of the message system where other subsystems can request a notification when a certain value is read.

Notification

The notification data type is used for less time critical data as there is no Expiry on these messages. There is not yet a way to prioritize messages on the network, but if at some point the bus was to prioritize messages notifications would be prioritized lower than events.

Request

Requests are method invocations and to be able to implement the API to work over the mesage bus all the high level API methods must have a message that represents this method and the subsystems that can

fulfill these requests must subscribe to these events. Methods can have a return type which can be void, a primitive or an object. In the message hierarchy there are only void and Response. This means that if a method should return a response then the response have to inherit from Response. It is however not unthinkable that a void method such as turn on light, on own initiative publishes an event that informs the entire system that a light was turned on.

The message space will be completed by the designers of the subsystems and the following will serve as an example of how it can be implemented. The parent class of all requests have 3 properties: Expiry[14], ResponseWanted and ResponseType. ResponseWanted and ResponseType are optional and can be used to inform the subscriber about what you expect to get back. ResponseWanted should be thought of as a way to opt out – an so it will only matter if explicitly set to false otherwise a response whould always be sent. The subscriber can choose to use this information to whatever it wants, and some times the ResponseWanted does not apply to the request type and will therefore be ignored. The "ResponseType" gives the possibility of having a request method that support multiple response types. This will only work as long as the publisher has subscribed to the response type supplied. Because WCF cannot serialize types the type will be sent as a string and the subscriber can use reflection to create the appropriate resoponse object. Now for some concrete request ypes.

LightsRequest

LightRequest = {State = LightState.On, RessourceId = GUID}

In stead of implementing a different message for all actions I found it sufficient with one message where the ressource and the requested state would be present. This would only be a problem if two different sub systems is in charge of turning on and off the light as there would not be a way of subscribing to events that could only turn off the light.

HeatSetPointRequest

HeatSetPointRequest = {Temperature = 23.0}

[14] Message Expiration. Adapted from: [EID] G. Hopfe & B. Woolf et al. Pp. 176

As heating, ventilation and airconditioning is controlled autonomously in its own subsystem it is only possible to set a single setpoint that controls the heat in the entire house. This is achieved by sending a HeatSetPointRequest containing the desired set point. Both the CCU and the OPC could subscribe to this event and it would be up to the designers of the CCU to decide if there should be an intermediate layer when changing the set point. An advantage of making this request go through the CCU is that it can reason about how much power the change will cost and advise the user about the consequences before the request is sent to the OPC. This requires a differences in messages as the OPC should not subscribe to the HeatSetPointRequest but a seperate message should be made.

HeatChangeRequest

HeatChangeRequest = {Delta = -5.0}

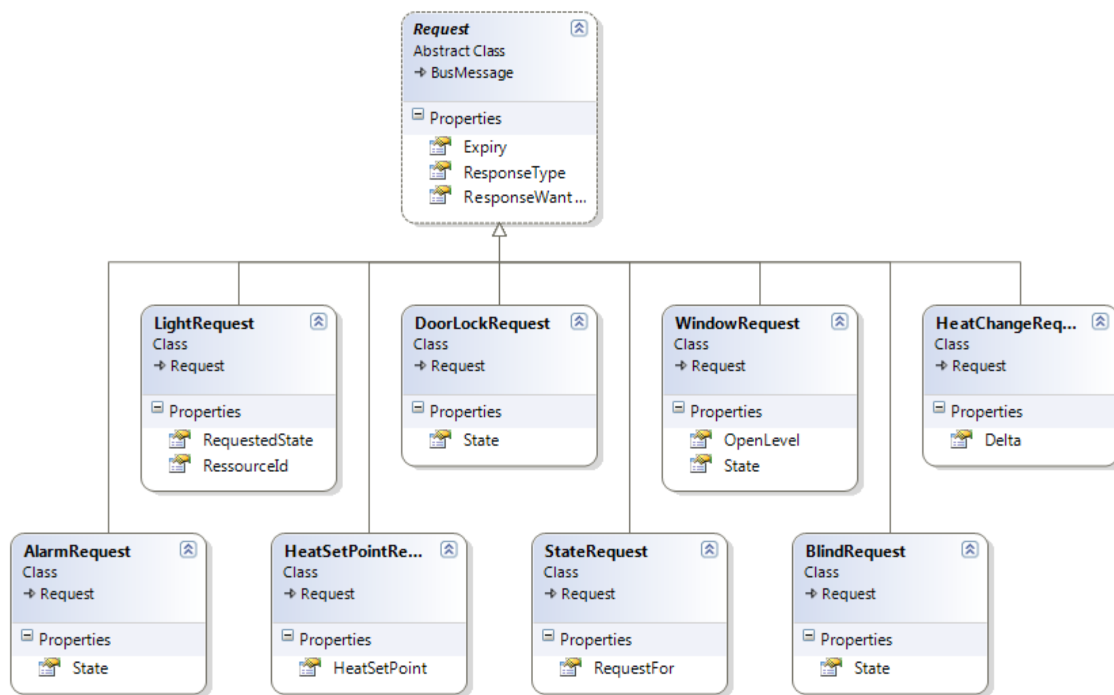
This request reminds a lot about the previous, but there is the difference that this request does not care about the current state but simply asks for the system to lower the temperature. This event could in theory also be subscribed to by both the CCU and the OPC but it would be a good idea to have these requests go through the CCU first. The CCU should know if similar requests have already been made and consider if the new request is a duplicate. It could be that the change has not been effectuated and the CCU would then have to think about other ways to fulfill the request. It might also be the case that more users are issuing the same command and the duplicate requests could just be disregarded. Another thing to notice is how the request should be fulfilled. The CCU would know if the windows has just been opened and can therefore tell the user that the reason for the slight draft is that the house is ventilating to obtain better indoor climate conditions.

StateRequest

StateRequest = {?RequestFor = GUID}

As described data will be available in 2 ways: requested or autonomously from a subsystem. To supplement the automatic events from the subsystems it should also be possible to request data. The only parameter is a GUID that identifies the data requested. The parameter is optional because an empty ID

will be considered a broad cast and all subsystems are requested to answer. It cannot be guaranteed that everyone will answer, but they should.



Response

Responses are as indicated not always required and can in fact freely be neglected despite the publisher's (requestor) wishes. The normal situation would be a fixed request response scheme where the publisher and subscriber have agreed on a format and the requestor knows what to expect.

The response contains a reference to the request message, a so-called "correlation identifier"[15]. Other properties in the Response parent class facilitate error reporting by a simple string explaining the error and an error object that can be used to handle the errors – for instance a *RessourceUnavailableError* should be handled differently than an *InvalidArgumentError*.

[15] G. Hopfe & B. Woolf et al. pp163

Requests changing state of an actuator is by definition void methods as the subsystems are expected to publish an event that a change in state occurred. It makes the most sense to inform broadly about a state change instead of asking everyone to subscribe for a response.

StateResponse

StateResponse = {
?SubSystem = Subsystem,
?ItemGroup = Group
?SubsystemItems = SubsystemItem}

The response for a StateRequest is a StateResponse. The response returns the ressources associated with the request's RequestFor GUID which can be the 3 non-mutually exclusive properties: Subsystem, Itemgroups and *SubsystemItems*. If the GUID in the request is that of a Subsystem, only one response should be returned containing the complete state of the sybsystem specified by the Id. The subsystem is a datastructure containing a name and a list of *SubsystemItems* which can be both a sensor and an actuator. This is the response for the broadcast-request where no id was supplied.

If the GUID supplied is the id of a Group then an entire group will be returned. This datastructure is still on an experimental level and will therefore not be explained in detail in this report. A group can be thought of as a recursive collection that eventually will contain hierarchical *SubsystemItems*.

The final thing a StateResponse can return is a single *SubsystemItem*. For this response type only one response can be extected.

It was mentioned that the properties was not mutually exclusive, and that is only the case if someone abuses its intend. Only one of these fields should be set, but as with the rest of the message hierarchy data types can be used rather freely.

There is no limit on who can subscribe to what and hence multiple *BusWorkers* can receive a response even if the requestor wants the reply to itself. I will discuss how this can be handled in the evaluation section at the end of this report, for now this is just how the bus works as it does not cause any problems, it is just something designers of *BusWorkers* should be aware of. It is considered good practice only to

subscribe to a response just before publishing a request and when all responses have been registered, or after a timeout the *BusWorker* should automatically unsubscribe. There are no strict reasons for this other than respect of privacy and to limit the amount of work the bus has to do and it is therefore considered good practice.

Evaluation

The system has been designed and implemented as a multi-tier, decoupled, semi-hierarchical, decentralized and distributed system. Although the system is not complete, the design principles and implementation demonstrates that when the devices are connected they will be able to utilize the messages and data model to distribute information over the bus.

One of the most important goals was to be able to integrate a number of different components and I believe this has been achieved by providing a versatile message and data model.

The central control unit's role has been made possible using different message types and subscriptions based on those. Actually the message model will make it possible to implement any process flow that might be required as the possibilities with message types and the data model are without significant restrictions. The cost of versatility is of course the lack of constraints that makes the system vulnerable to attacks. All applications being connected to the bus will be able to receive all data and send almost all kinds of data. This is at this point not considered a problem of 2 reasons. The first is that security has been implemented to ensure that only applications that are trusted by certificate will be able to connect to the bus. The other things is that at this point the bus and message system is still new and has not been tested in a real environment and it is therefore not possible to tell if an active bus validating constraints will have any advantages.

A solution to the problem of authorizing subscribing and publishing on a type level could be handled by a separate policy server that the bus and the receiver could communicate with. The receiver could specify when subscribing which publishers should be allowed to invoke the subscriber. The problem with this approach is that it breaks the decoupling. The decoupling could be embraced by using roles in addition to the existing certificates. The policy server would then have to keep track of which publisher roles has access to publish what per subscriber. The latter is necessary as subscriber A may not have the same requirements for messages as subscriber B.

Another unsolved problem is how to facilitate that a publisher can send a message that can only be read by an intended receiver. This requires, because of the decoupling, that they have knowledge of one another based on some kind of discovery message pattern, or that the responder known the requestor because of the request. The problem could be handled with the implementation of a protocol where the intended single receiver would send its public certificate along with the request and then the responder could encapsulate an encrypted field inside a message. This field will because of the fact that the bus

would still have to be able to read the type of the message have to be wrapped in a regular message with a string element consisting of a serialized string of the contents that is signed with the responder's private key and encrypted with the requestor's supplied public key. Of anyone receiving the response the requestor would be the only one to be able to read the contents of the message. It would do that by decrypting the message string with its own private key and after that the string could be deserialized and the message could be restored and processed. The requestor could receive multiple responses all signed and encrypted with the requestors public key, but from the response's origin field set by the bus and based on the signature of the inner-message the requestor would be able to the responder and only use the trusted response.

Currently the system has no global clock which is only neglected because all systems reside on the same physical machine. If some of the bus workers were to be distributed to another machine on the network or to a remote location then it would be of great importance to synchronize the clock across the system. Using the message system proposed it would be possible to make a pseudo-protocol that exchange the time from a managing subsystem on the bus. The bus could also be in charge of informing about the global clock as there is guaranteed to be only one bus. The problem with this approach is that the bus is designed to be naïve and reactive and hence it should not have to do any tasks other than receiving messages and subscription requests and handling those. The bus should not have any control function in the system.

A dead letter[16] manager could be implemented to make sure that unsubscribed messages would not be lost and could be delivered to a later subscriber. The argument would be something like. "You just subscribed to car magazines! Would you like to receive the unattended car magazines of the last month?". This approach would ensure that all messages will be seen by someone and no information is lost. The problem is that what if another subscriber subscribes the second after the first one. Then the subscribers believe to share more or less the same information but there is a huge difference in the amount of data that they have received. At this point there is no reason for us to implement this feature, but I felt it was important to mention that messages are purposely lost as the lack of subscribers is perceived by lack of interest.

The common data model was designed so that every application connected to the bus would share the same knowledge about structure of data. This enables the different applications to communicate using strong types which makes the communication more reliable. There is however a challenge when using this approach. Changes in the data model and the implementation of new message types and data structures

[16] Dead letter queue described in: [EID] G. Hopfe & B. Woolf et al. Pp. 119

have to be distributed in order to publish new types. There is not yet any clever way to do this and this is the backside of our use of strong typing. If we have used weak typing a change in the data structure or new message or data types would be sent as strings and it would only be needed to reprogram the receivers to understand and make use of the new data. This is also one of the main reasons that we decided on strong typing. All our applications send and receive data and therefore the savings on parts of the system that didn't need reprogramming would be inconsiderable.

The quality of the system is hard to measure as the project is still in its preliminary stage. Tests have been performed to identify problems and to validate if the intended purpose was met and not the other way around. This means that when we are building further on this system tests will have to be systemized and it must be ensured that message endpoints as well as the bus has a certain standard. This can be tested by fabricating a wide variety of messages to test the stability of the middleware as well as the transporting bus. It should also be tested if our use of the message queues has any weaknesses that could cause a breakdown. This could be done by sending messages that vary in size and frequency. As this project has focused mainly about facilitating communication tests have only been sporadic and initiated to correct unwanted behavior.

Operation and further development

The project is going to be exhibited on the 13th of June at DTU where students, sponsors and members of the press can come and see the progress of the house and also the control system. At the moment we are working hard on getting the IHC connected to the bus so that we will be able to control actual installations.

The status of the project is that the App Web Service is running and is able to get data from the weather service I have developed. The programming of the IHC is nearly finished and the focus needs to be directed to the PLC and all of its subsystems.

It is hard to tell how the final result will be but I feel that anything is possible with the bus, and therefore it would not be impossible to get a working control system installed in the house before September where the competition takes place.

Until we have a working solution it is hard to prove that this can be used in a larger scale and considering the limited success for home automation it is not within the next couple of years that this system will have any chance to be broadly applied. This has however not been the intention. As mentioned in the introduction the project spawned from the need of integrating different subsystems into one and I believe

it would be better to integrate the subsystems at a lower level so that they can share data wirelessly using a standard protocol. These data should then be available to a developer API so that user interaction and automation can be customized to the surrounding premises and configurable so the user can decide how things should be controlled.

Conclusion

In the preceding report I have accounted for the analysis design and implementation of the message bus and its transport: messages using a shared data model.

The system is able to integrate all kinds of control systems provided that some integration between the system and .NET has been made. The actual devices: the PLC and IHC have not yet been connected to the bus, but software emulation classes have been connected and been able to share state across the system. A single service is so far connected and proves that the integration with online APIs can be made.

Information can be requested or can arrive based on an external event. Messages are guaranteed to be delivered as long as the message queue server is running. This enables subscriptions and published information to be saved even if the bus or a subsystem is not responding.

Messages are encrypted using self-signed certificates and authenticated against the bus. Receivers of messages can identify the sender by canonical name and will therefore have to trust that the bus has validated this information.

New subsystems can be added if the common data model is used and if the new subsystem has received the bus public key and the bus has been configured to accept messages signed by the public key of the new sub system.

A common API has been implemented in two levels where the first level is the message system that facilitates method invocation and sharing data. The second level is the proposed subscribing pattern where messages can be divided into 2 levels where a manager subscribes on another type of message and can in that way translate a higher level message into requests known to perform specific actions such as starting mechanical ventilation. This creates 2 levels of abstraction for the user where the latter can be bypassed so that windows can be opened even though the central logic unit believes it to be a bad idea. The system is thus presented to the user as a catalogue of services independent of the hardware implementing them in an API intended for mobile app development.

Litterature

- MSDN, Securing Messages Using Transport Security
  http://msdn.microsoft.com/en-us/library/ms789030.aspx
- How to: Exchange Messages with WCF Endpoints and Message Queuing Applications
  http://msdn.microsoft.com/en-us/library/ms789008.aspx
- **[EIP]** Enterprise Integration Patterns (Designing, building and deploying messaging solutions)
  G. Hopfe & B. Woolf et al.
- **[OOSE]** Object-Oriented Software Engineering (Using UML, Patterns and Java™)
  B. Bruegge & A. Dutoit
- **[DS]** Distributed Systems (Principles and Paradigms)
  A. Tanenbaum & M. Steen

Appendix

- Contributors by name
- Value Requirements
- Description and vision by Søren Andersen
- Software CD
  The software CD contains the code that I have contributed to. The solution consists of 4 projects:
  1. Bus – the bus code
  2. Common – the common namespace and middleware use to create clients
  3. WeatherService – The weather forecast is using an online API to import weather information into the system.
  4. IHC – The IHC project contains a demo of an IHC with demo sensors and actuators.
  5. lib contains the logging framework we are using.

The purpose of the software is primarily to show the code that is referenced in the report. If readers of this report wishes to run the message bus it will require installation of MSMQ and certificates. I will gladly assist to make the solution run if this is needed.

Contributors by name

Name	Details
Søren	s093030, Søren Olofsson
Emil	s093281, Emil Refn
Philip	s093041, Philip Engberg Nielsen
Carsten	s093751, Carsten Nilsson
Morten	s093270, Morten Schnack
Dainius	s101404, Dainius Griguzauskas
Anders	s093217, Anders Jensen
Christian	IMM, Christian D. Jensen

Value requirements

The indoor climate conditions are set by Solar Decathlon Europe and in order to get the highest amount of points the measurements will have to be in to these intervals. If the measured value is between min and max we will be rewarded 100% and if the measurement is between min and alt. min or max and alt max we will receive a lower score.

Measure	Min	Max	Alt. min	Alt. max
Temperature (c)	23	25	20	28
Humidity (relative %)	40	55	25	60
CO2	0	800	-	1200
Light level (LUX)	500	-	300	-

Control System Description

By Søren Andersen
January 2012

Contact: sa.soerenandersen@gmail.com

Description and requirements of control system

The aim of the document is to create a base for the entire group, for every partipant to be clear on the goal for the contriol system.

Vision

The control system should be made so the occupants have to do a minimum of work to obtain the desired indoor climate conditions, with a minimum of energy consumed.

The control system should be intelligent so human-overwrite isn't necessary.

The drawing shows how al components are connected to each other, in order to reach the indoor climate conditions set by the SDE organization. In the drawing all action-lines starts with a: Occupant controlled questions. During the contest week the answer will be No. However, it is believed such a option is necessary in a "real" house.

Control of the house

The following paragraph discribes how/what the control system perform.

Pre-installed conditions

The aim of pre-installed conditions, is to make it possible for occupants to create certain indoor climate conditions, but letting the CCU decide the best method for obtaining it. The goal is to reduce conflictions between the system and the occupant, where the occupant overrules the CCU, which in most cases will increase the energy consumption.
The following suggestions show how some of these pre-installed conditions could be:

- SDE competition
  - Create the conditions demanded through the competition.

- Indoor climate conditions suggested by DS 15251

- Family dinner
  - Increase the cooling and air change rate.

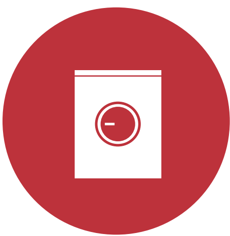
- Mom is home alone.
  - Interior lighting should be dimmed for cosiness.
  - Exterior lighting should be increased, for security.
  - Temperature should be higher than normal.
  - The air change rate should be high, for a fresh feeling.
  - Lock al doors and windows.

- Dad is home alone
  - Increase the air change rate
  - Turn on the TV

Controlling the indoor climate conditions will be done be the CCU and the Nilan VPN unit, but it is believed a interface telling the occupants what is happening and why it is happening is the essence of succeeding, and reaching a very low energy consumption.
The interface could be a central placed screen, a tablet or smartphone app. The important part is that it is easy and not an obstacle for the user. Thus, Iit is believe an app for a smartphone or a tablet would be a very smart solution, as it can be very handy and can be very intuitive.

Can the internet and a website be used as feedback platform? Yes, a website can be used for controlling of overall situations, say when should the pumps run etc. However, this is a subject that no ordinary occupant really have to have the control of, and perhaps don't even care of.

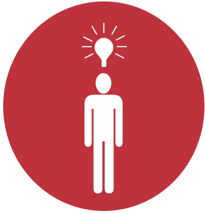
Do we need switches or similar to control the lighting etc.? It is believed the answer should be: Yes, we need both, e.g. a lamp should be controllable through both a switch and an app.



Control of home gods
Using energy when ever it is present is beneficial both in SDE and as a general rule in society. Provide the equipment with a delayed start or a similar controlling must be part of the control system. Controlling the equipment from a decentralised place could be done through a smartphone app or website. To optimize the energy consumption the occupants should be informed about the cost whenever a the dishwasher, the washing mashine or similar products is switched on. The occupants should be given the option to turn on now, or delayed start.

Intelligent house and feedback
FOLD must be an intelligent house.

A comprehensive feedback system should/could be the backbone in the intelligent house. The aim of the feedback system is for the system to "learn" the habits of the occupants.



Ideas

The following is ideas that might can be incorporated in the house, perhaps branded as: Future House Components...?

Check-In

During the SDE competition the indoor air conditions are relatively determined, but during a real life situation the house should be more flexible.
In order to reduce the energy consumption as much as possible, it I desired to "close down" the house when leaving it. When returning to the house, it will be necessary to recreate the desired conditions prior to the arrival of the occupants.
To reach the desired conditions requires time, and this time could be gained by letting people check-in by e.g. using a smartphone and an app when they get on the train home.

Voice and movement controlled

Taking the intelligent house to the next level could be to use movement or perhaps voice controlling.
The voice controlling is known from the Iphone 4S, where SIRI can be asked to do al sorts of things. A command could be: FOLD, increase the temperature.
Movement controlling is seen in the PlayStaion KNX (correct me if I'm wrong), and by moving hands or entire body a command could be carried out.

Description of building components

In the following section house components are described.

In the following text the term, letting the house decide what is the best. This term means that the house will try to obtain the desired the indoor climate conditions with the least consumed energy.
In order to reduce the energy consumption as much as possible, must passive means be used before active means a taken in use, e.g. solar shading in used before the chilled ceiling.

Windows

The function of the windows is to provide an acceptable daylight level and natural ventilation.
Opening of the windows will happen when the outside conditions allow it.
- Outside temperature > inside temp, and outside temp. < Maximum acceptable temp.

Solar shading

The main functions of solar shading is to reduce the heat load from, and thereby reduce the use of active cooling.
The solar shading can be controlled in two ways, the first is letting the occupant do it them self. The second and the preferred is letting the house decide what would be the best.

Lighting
The house will be equipped with different lighting system. It has not be decided the exact lamps yet, but it is acceptable to assume that LED technology is going to be used. The lights should be controlled in two ways.

- **Method 1**: Is letting the occupants decide whether the lights should be on/off, or dimmed in some way.

- **Method 2:** Is controlling the lights according to the illuminance level in the house and minimum desired illuminance, combined by the natural and artificial light.

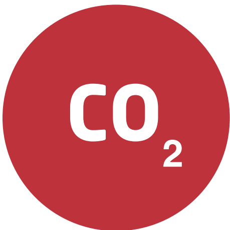
Requirements:
- Method 1 requires a switch, which is occupant controlled.
- Method 2 requires one or more sensors measuring the illuminance level on specified areas of the house. The amount of sensors, are dependent on the ability of each sensor.

Question: How much area can one sensor register, and how can information be send to just one lamp? Do we need a sensor for each lamp?

Needed information: Where will each lamp be placed? How much lux is required from them?

Control: As the house must be equipped with both methods, it is desirable to have a platform where the occupants can decide which method is in use, e.g. an website based app.

Conditioning
The indoor climate should be conditioned so it complies with the requirements of the pre-installed conditions, among these the SDE conditions. Either the sun or the floor heating system will control the heating of the house. The solar shading, natural ventilation and/or the chilled ceiling are used to keep temperature low. In extreme cases can the floor be used for cooling as well.
Controlling the indoor climate conditions should be done measuring the temperature, humidity and CO2 concentration. To control the solar shading a sensor on the outside is necessary. Sensor measurements should be sent from the CCU to the Nilan VPN, Solar shading and window control units.



The humidity of the house must be between 40 - 55 RF% (SDE conditions). The house will be quipped with some kind of passive humidity buffer, but if this isn’t sufficient, the mechanical ventilation must create the acceptable conditions.

Sensor placement
The measurement should be performed in a central place, or in a place that often is in use.