Radboud University Nijmegen

Master Thesis

Software Architectural Styles in the Internet of Things

Author:
Evertson Croes
4241754

Supervisors:
dr. Jaap-Henk Hoepman
MSc. Anna Krasnova
MSc. Angelo van der Sijpt (Luminis)

A thesis submitted in fulfilment of the requirements for the degree of Master of Computer Science

in the

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

August 2015

Radboud University

RADBOUD UNIVERSITY NIJMEGEN

Abstract

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Master of Computer Science

Software Architectural Styles in the Internet of Things

by Evertson CROES

The Internet of Things (IoT) is an ambiguous term. There are different definitions for the concept, ranging from any system that has sensors and actuators to a single interconnected network of physical objects. Despite this ambiguity, there are reference architectures proposed for the "Internet of Things" as a single type of system. This thesis shows that this term does not provide enough information to base a software architecture on. This is accomplished by taking an in-depth look at the IoT described in literature as well as the types of applications that exist on the market today and using the concept of software architectural styles to show how different areas in the IoT will need varying styles.

Software architectural styles are a labeled set of design decisions that have proven to elicit quality attribute benefits given the right context and are considered to be the first step in designing an architecture for a software system. However, during the course of this research it has become clear that the term Internet of Things is not enough to provide a verdict for the effects of software architectural styles. For this reason, this thesis proceeded to classify solutions in the Internet of Things into multiple classes. The results are that for a subset of the classes there is a clear "best" style, however for remaining classes there are still different choices where more context information is needed.

The analysis itself provides a list of important IoT related factors when choosing a software architectural style, which can be used as a basis for future IoT projects and reference architectures. However the conclusion of this thesis is that the term "Internet of Things" should not be used as a basis for software architecture. This was proven by showing that even for the different classes, which are subsets of the IoT, there are needs for different styles.

Keywords: The Internet of Things, software architectural styles, quality attributes, software architecture evaluation

Acknowledgements

This document presents my Master's Thesis "Software Architectural Styles in the Internet of Things". This project took place from February to August of 2015. This was one of the most difficult projects I have worked on during my six years of studying Computer Science, due to the ambiguity and high level of abstraction of the topics that play a role in this thesis. Thankfully, I received help from various people to whom I wish to express my gratitude towards in this section.

First and foremost, I would like to thank Luminis for providing this research topic and a comfortable working environment. During my thesis period I have always felt at home there.

I would like to thank Anna Krasnova for supervising me during this thesis and providing me with feedback regarding the structure and motivation for this thesis. I would also like to thank Jaap-Henk Hoepman for providing me with feedback on how to finish the thesis and for agreeing to be the primary assessor for this project.

I am also grateful to Hans Bossenbroek for the discussions about the importance of software architecture as a discipline in the world of software development. I would also like to take this opportunity to express gratitude to Hans Gringhuis en Geert Schuring for allowing me to give a presentation about my results halfway through the thesis and for providing me with constructive feedback.

I would not have been able to complete this thesis without the constant moral support from my friends and family, who always remind me to never give up and why the effort is worth it.

Last but certainly not least, I would like to thank Angelo van der Sijpt for the constant feedback, brainstorm sessions and weekly discussions. His remarks were often enough to push me in the right direction while still granting me the freedom to conduct my own research.

Evertson Croes,
August 2015

Contents

Abstract i

Acknowledgements ii

Contents iii

1 Introduction 1
- 1.1 Related Work . . . 3
- 1.2 Research Questions . . . 3

2 The Internet of Things 5
- 2.1 Introduction . . . 5
- 2.2 The IoT Vision . . . 5
- 2.3 The IoT Reality . . . 7
- 2.4 Conclusion . . . 9

3 Quality Attributes 11
- 3.1 Introduction . . . 11
- 3.2 Quality attributes . . . 11
- 3.3 Relevance of Quality Attributes to the Internet of Things . . . 13
- 3.4 Quality Attributes in the Internet of Things Reality . . . 19
- 3.5 Conclusion . . . 19

4 Internet of Things Solution Classes 20
- 4.1 Introduction . . . 20
- 4.2 Classification . . . 20
- 4.3 Classes . . . 23
- 4.4 Conclusion . . . 25

5 Software Architectural Styles in the Internet of Things 26
- 5.1 Introduction . . . 26
- 5.2 Software Architectural Styles and Evaluation . . . 26
- 5.3 Mapping . . . 28
  - 5.3.1 Integration . . . 29
    - 5.3.1.1 Class A. Location Constrained Heterogeneous Devices . . 31
    - 5.3.1.2 Class B. Location Free Heterogeneous Devices . . . 47

5.4 Observation/ Class C. Body Sensors 51
5.4.1 Aggregation 57
5.4.1.1 Class D. Active User Collective Data Solutions 58
5.4.1.2 Class E. Passive User Collective Data Solutions 65
5.4.1.3 Class F. Stationary Homogeneous Sensors 65
5.4.1.4 Class G. Mobile Homogeneous Sensors 66
5.4.2 Automation 67
5.4.2.1 Class H. Smart Systems 68
5.4.2.2 Class I. User-Controlled Actuators 75
5.5 Conclusion 75

6 Conclusion 81
6.1 The Internet of Things 81
6.2 Software Architectural Styles 82
6.3 Quality Attributes Evaluation 82
6.4 Software Architectural Styles in the Internet of Things 82
6.5 Future research 83

A IoT Definition List 85

B IoT Solutions Analysis 89

C Software Architectural Styles 136

D Software Architecture Evaluation Methods 157

E Discussion Panel 163

Bibliography 165

Chapter 1

Introduction

The Internet of Things (IoT) is an ambiguous concept. The term is considered a buzzword[1] and is generally used to describe systems that connect the physical world to the digital world. This is achieved by giving real-world physical objects, which were previously disconnected from any type of network, connectivity to the digital world in various ways. Advancements in the fields of small processing units leading to sensors that can measure attributes of real world objects, actuators that can change the state of the real world, RFID for identifying and locating objects and Cloud technology for computation and data storage all contributed to the current state of the Internet of Things. Applications in the IoT can range from connecting the human body by use of a wearable sensor to a smart city solution with multiple sensors distributed throughout the city.

The advent of the Internet of Things brings with it many possibilities and challenges. One of the areas of research in the Internet of Things is software architecture. There have been several proposals of reference architectures[2] for the Internet of Things as a single type of system. However, given the ambiguity of the term and the diverse applications regarded as being part of the Internet of Things, it seems unlikely that a "one size fits all" reference architecture can exist.

It is the first goal of this thesis to explore the Internet of Things and infer a definition in order to bring some clarity to the term. This research will take into account the IoT described in literature as well as the current state of the IoT by looking at various solutions that exist today. The differences and similarities between the vision for and the reality of the Internet of Things will be highlighted. By finding the common denominators between the various literature and IoT solutions, it becomes possible to derive a single definition for the term Internet of Things in order to reduce the ambiguity. This makes

[1] A word or phrase, often an item of jargon, that is fashionable at a particular time or in a particular context

[2] Reference Architecture: A template solution for an architecture for a particular domain

it possible to reason about the architecture design choices in the Internet of Things to be made regarding software architecture moving forward.

This thesis will argue that there can be no single reference architecture for the Internet of Things, because the term does not provide the context information that is needed to make many design decisions. This is an important statement to make as it shows the incompatibility of the term Internet of Things and software architecture, i.e. the term should not be used to describe a type of system when talking about software architecture. Future papers will need to be more specific about which type of solutions their reference architectures are proposed for in order for them to be useful.

In order to show this, a set of IoT solutions(applications) will be grouped into classes in order to illustrate how diverse the IoT solution space can be. This is done by analyzing several IoT solutions in detail and illustrating the differences and similarities between them. This thesis will not provide a complete set of classes as this is out of scope and cannot be proven to be complete.

While the partial set of classes provides more information on the type of applications considered as being part of the Internet of Things, it is not enough to prove that a single reference architecture cannot exist. It must be shown that these classes require different types of architectures. This will be done by using the concept of software architectural styles. A software architectural style is a labeled set of components[3] and connectors[4], and a set of constraints on how they can interact [GS94]. These constraints can be topological, for example not allowing cycles, or it can regard execution semantics. The latter refers to the meaning of such an interaction between two components, which could be a procedure call or a notification for example. All styles come with trade-offs, explicitly mentioning which quality attributes[5] are gained and which are given away, however this also depends on the context of the system to be built.

The choice of which architectural style to use, based on a given set of quality requirements, should be the first step in software design[PH]. If it can be shown that for at least two different IoT solution classes there is a need for different styles at the same level, then the statement that a single reference architecture for the Internet of Things cannot exist is supported. This is because the choice of style is considered to be at a higher level (first step) than a reference architecture(collection of steps), which can be seen as an instantiation of one or multiple styles. This thesis also contributes to the architecture in the Internet of Things research area by illustrating the (in)compatibility of certain software architectural styles in the IoT and what the factors are for this verdict.

[3]Component: A constituent part; element. `http://dictionary.reference.com/browse/component`
[4]Connectors: Description of the interactions between components. [GS94]
[5]Quality attribute: Non-Functional Requirement

1.1 Related Work

There were no papers found that look at software architectural styles in the Internet of Things. However, as mentioned, there are reference architectures provided for the IoT as a single type of system. The IoT-A is a group consisting of researchers from large companies and institutions with the goal to find provide such a reference architecture. The book "Enabling things to talk", written by members of this institution, describes the reference architecture as well as the research leading up to it [BM]. The problem with this approach is that they provide a single reference architecture for the entire IoT as a single domain.

The same approach is taken in an article describing the "Distributed Internet-like Architecture for Things" or DIAT for short, which presents a layered architecture where all components in the Internet of Things must implement the architecture in order for it to work[SR14]. This paper, written recently in 2014, also reviews other reference architectures proposed for the IoT in the past, all of which try to achieve a single reference architecture for the entire Internet of Things.

This thesis will provide evidence that this goal of achieving such a "one size fits all" architecture is not possible for the Internet of Things, thanks to the ambiguity of the term, diversity in applications and continuous evolution of the concept. This will be done by using software architectural styles and evaluating the quality effects on a system using them.

1.2 Research Questions

The main research question that will be answered in this thesis is the following:

Which software architectural styles are suitable to apply when designing Internet of Things applications, and can they be used as a basis for a single reference architecture?

To answer this question, this thesis will also address the following sub-questions:

- What is the Internet of Things?
- What classes of Internet of Things applications can be discerned?
- Which software architectural styles best fit each of these classes and why?
- Can the set of best styles be merged into a single reference architecture?

The hypothesis in this case is that there are different scenarios in the Internet of Things that require various types of architectures (and thus styles), which will show that a single reference architecture for the entire domain cannot exist.

The contributions of this thesis will include an analysis on the IoT, a list of classes for IoT solutions and a mapping of software architectural styles to this list of classes. The goal is to reduce the ambiguity of the term "Internet of Things", provide an analysis on the usage of software architectural styles in the IoT and show that the term does not give enough information to make design choices.

The remainder of this document contains the analysis and results of this thesis. Chapter 2 provides a description of the Internet of Things based on literature and examples of applications. Chapter 3 discusses the concepts of quality attributes and why there are important in the IoT and for this thesis. The classification of the IoT is presented in chapter 4. Chapter 5 contains the mapping and analysis of software architectural styles to the Internet of Things classes. Finally, chapter 6 presents the conclusion of this thesis and suggestions for future research.

Chapter 2

The Internet of Things

2.1 Introduction

In order to reason about the software architecture of the Internet of Things, it is necessary to understand what the term means. This chapter describes the IoT defined in literature as well as the reality that exists today. The goal of this chapter is to derive a definition for the IoT based on this information.

2.2 The IoT Vision

The Internet of Things (IoT) is a term that was first mentioned in 1999 by Kevin Ashton, a founder of the original MIT Auto-ID center[SW10]. Since then, the IoT has become a buzzword much like "Big Data" in the sense that it is used frequently but seems to be ambiguous to the users of the term. This section provides a brief view on the vision for the Internet of Things as described in literature and other web sources. The definitions for the IoT provided by these sources are listed in appendix A.

For many of the papers written about the Internet of Things, two characteristics have been mentioned the most. The first is that the IoT is a network. It is always referred to in the singular form, which may suggest that there is only one IoT. The second most mentioned characteristic in the definitions refer to the "things" in the IoT. These are (everyday) physical objects.

These physical objects are to be embedded with technology so that they can be connected to the IoT, have their state measured and changed, are uniquely identifiable and can communicate with each other to achieve greater value and service.

The goal of achieving greater value and service is extremely broad. This is due to the fact that the variety in applications for the IoT is big. This greater value and service

provided by the IoT is either the creation of new functionality previously not possible without the network of physical objects or the increase of the quality of existing processes with the help of the IoT.

The novelty of the IoT is present in the name. The term “Things” refers to the everyday physical objects that will now become connected. This will make new type of data available that was previously not possible. The term Internet is used to illustrate the interconnection between these networks of heterogeneous objects.

The IoT also has to be dynamic, since nodes will be added and removed constantly from this network. New types of devices will emerge and the IoT has to be able to handle this change. The IoT should be self-configuring, always adapting to this change.

The terms global, world-wide, ubiquitous are mentioned for the IoT. The IoT has to be available everywhere. With the Internet being as pervasive and ubiquitous as it is today, this is not an unreasonable requirement.

A pervasive system has the following characteristics [Hoe12]:

- **Invisible by design**: Pervasive systems are not explicitly there. They are often integrated common objects
- **Networked** : Devices are interconnected by a seamless communication infrastructure
- **Many-to-many**: As opposed to one-to-one or one-to-many relationships. The devices are not restricted to one user and a user is not restricted to one device.
- **Always on**: Devices do not need to be actively switched before interaction can be had.
- **Distributed**: The computing intelligence is a combined computing effort of multiple devices
- **Context-aware**: Can measure their environment and is aware of other pervasive devices in their vicinity.
- **Adaptive**: The actions of the system are triggered by implicit actions rather than explicit user interaction.
- **Natural human interface**: People should not need to think about how to interact with the system, this should be natural through speech, touch or movement.

One source goes as far as to say that the IoT should be able to react to any and every event in the real-world and the devices should be able to communicate with each other.

Finally, some sources mention which technologies and protocols can be used for the IoT, such as the Internet, Cloud computing, RFID, IPv6 and much more, but these will be omitted since it is out of the scope for this thesis.

From my analysis of the sources I conclude that the Internet of Things is a singular, dynamic, global and pervasive network which contains physical objects embedded with technology that are uniquely identifiable. These objects work together, are autonomous and can react to and change the real world in order to provide some value to its users.

However, this is considered a vision as the IoT is currently not at this state. It can be argued whether or not the Internet of Things can ever be a "singular" network. At a technical level, anything connected to the Internet is part of a single network, however what the literature suggests is that the physical objects can also all exchange information with each other. As we will see in the analysis of the reality of the IoT, this will not be possible for several reasons.

There is much more written about the Internet of Things, however this thesis will focus on the software architecture aspect. For this reason, it is enough to know what is generally meant by the Internet of Things in literature. However, we will also need information on what the quality/non-functional requirements can be for the IoT. This will be covered in chapter 3.

2.3 The IoT Reality

This section describes the current state of the IoT by looking at the solutions that are available at the moment of writing this thesis. Because there is a big variety in IoT solutions, the data set has to be representative of all "types" of solutions.

Postscapes[1] categorizes the IoT into six domains. These categories are the connected home, connected body, connected retail, connected transportation, smart city and industrial application.

These categories almost correspond completely with the domains mentioned in the IoT-A research, except the IoT-A also includes E-Health and Smart Energy as an IoT domain. The Smart grid which is the only proposed application for smart energy, will be referred to later in the thesis. The E-Health domain overlaps with some of the other domains, so this will also be omitted. From each domain, three solutions will be taken and analyzed in depth. The detail in which the analysis is done can differ between solutions, as some are less likely to share information than others. This analysis can be found in appendix B. An overview of the solutions analyzed can be seen in table 2.1.

[1] http://postscapes.com/internet-of-things-award/2014/

Domain	**Solutions**
Connected home	Nest Thermostat, Homeseer, SmartThings
Connected body	Angel Wristband, Nymi Wristband, Zebra Motionworks
Connected retail	Scanalytics floor sensors, S5 Electronic Shelf labels, Nomi Brickstream live
Connected transportation	Weather Cloud, Truvolo Car Solution, Veniam Vehicular Networking
Smart city	Bitlock bycicle lock, Array of Things, Enevo waste collection
Industrial application	Farmobile Fleet Management, Condeco Workspace Occupancy Sensor, DAQRI Smart Helmet

Table 2.1: Overview of domains and IoT solutions

The methodology for analyzing the solutions was done systematically by looking at the following variables:

- Identify the Physical Entity[2] being measured. This is done to verify that the system can be classified as an IoT solution.
- Identify the attribute(s) of the physical entity that is being measured. This is also done to verify that the system can be classified as an IoT solution.
- Identify if the type of IoT Connector(s) present in the solution [3]
- Identify the components and the topology of the network.
- Illustrate the topology in a diagram showing the relationship between components (one-to-one, one-to-many, many-to-many)
- Illustrate the direction of messages passed between the components in a diagram
- Identify the location of application logic and data storage for the solution. The application logic and data storage locations do not refer to logic and data needed to network between nodes, rather it refers to the logic and data that are specific to the solution. This helps to illustrate how centralized the solution is.
- Identify the user interaction possibilities of the solution.
- Make an estimation of the scalability requirement of each component. The scale can be fixed or potentially increasing.
- Identify the Internet-Dependency of the solution.

[2]A real world physical object or environment

[3]IoT Connector: sensor or actuator

- Identify the type of invocation. (Explicit vs. Implicit)
- Identify whether the system anticipates interoperability with other systems and how this is done.
- Identify the battery life of the devices in the solution.
- Identify if there is any autonomous behavior that the solution exhibits.
- List any other interesting characteristics specific to the solution if any are found.

The goal of this analysis is to a) show that for all IoT solutions the physical entity and measured and changed attributes can be identified b) gather important characteristics of IoT solutions to be used for recommending software architectural styles later in the thesis. Software architectural styles are explained in detail in appendix C.

For all IoT Solutions, at least one physical entity could be identified which was being given a digital presence. For all IoT Solutions, at least one attribute measured or changed in the physical entity being given a digital presence could be identified. These are all listed in appendix B.

In order to talk about Internet of Things Solutions, we must be able to identify if an application is an IoT or non-IoT solution. To do this, a common denominator must be found for IoT solutions. What has been noticed by looking at the 18 solutions that are deemed IoT-related by Postscapes, is that they all measure an attribute of an entity in the real world, change the state of an entity in the real world or replace an entity in the real world by adding a digital and connected version of the entity.

The variety of types of IoT Solutions is so big that no other commonalities could be identified. From this we can infer that the trait that discriminates IoT solutions from non-IoT application is the connecting of a physical entity to the digital world via a network. Note that by definition a network must only contain a minimum of two nodes, so a solution containing a wristband sensor that communicates to a smartphone via Bluetooth is also considered an IoT solution for example.

2.4 Conclusion

The IoT described in literature has a set of interconnected objects on one network that can communicate to each other. However, in reality it is the case that each type of object is manufactured by a certain company that provide their own environment (UI applications, servers, protocols, architecture choices) for communicating with the objects. There is not enough incentive for the solutions to expose their physical objects/devices to an open network. Instead, what we will see moving forward is multiple networks of interconnected solutions where each solution decides which group of other solutions they want/need to communicate with.

The first sub-question of this thesis is "what is the Internet of things?". Based on all of the information collected and analyzed, I propose to look at *the* IoT as an event in motion instead of as a singular network as it is described in literature so far.

Because of the variety in IoT solutions and the divide between literature and reality, it is hard to find a common denominator. However I have found that the defining characteristic of the IoT is the binding of the physical world to the digital world through sensors and actuators. While interoperability between solutions will play a big part, it will not be necessary for all scenarios where we are connecting the physical world to the digital world.

I propose the following definition for the Internet of Things:

"The Internet of Things is the continuous increase of connectivity between the physical world and the digital world by connecting physical entities to a network".

This definition describes the IoT as a trend instead of a network. However, we can still speak of *an* Internet of Things as being an interconnected network of physical entities/objects/things, all belonging to the same or different vendors.

The point is that using the Internet of Things in the singular form to describe one network provides confusion as it is not likely to exist in the way that it is described in literature. By using it in the singular form in this way, it gives the illusion that a single reference architecture might be possible.

This chapter has shown that the common denominator in all the literature and the solutions examined is that the IoT connects the physical world to the digital world. Besides this fact, every other attribute of application in the IoT is variable. The next chapter will show how diverse IoT solutions can be in regards to their quality attribute requirements, which are important for software architecture evaluation as will be explained in chapter 5.

Chapter 3

Quality Attributes

3.1 Introduction

The goal of software architecture is to design a system that meets the requirements set for certain quality attributes. Choosing one software architectural style over another will exhibit different levels of quality attributes for the same system. This chapter provides a brief description of the quality attributes that will be used as indicators for style choices in the remainder of the thesis. How important these quality attributes can be in the context of the Internet of Things will also be described, as well as how different applications have varying quality attribute requirements.

3.2 Quality attributes

When designing a system, there are a number of functional and non-functional requirements present that need to be fulfilled. The functional requirements relate to the actual functionality that the system to be developed needs to have while the non-functional relate to the quality of the system. The trend has been to use other terms such as "extra-functional", "quality attributes" or even "illities", because most of them end with the suffix "ility" [PW01]. For this thesis they are referred to as quality attributes. While functionality is provided by the implementation of the system, the quality attributes are introduced in large part by the software architecture.

The standard reference for software quality characteristics is the ISO 25010 standard[1]. This lists Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintanability, Portability as main quality characteristics of software systems. Each of these have a set of sub-characteristics that are described in one sentence.

[1]http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

A research done to illustrate the important requirements for the IoT based on the opinions of stakeholders for the IoT was conducted by the Internet of Things Architecture (IoT-A) institute [BM]. The most important requirements found were Interoperability, Evolvability, Performance, Scalability, Availability, Resiliency, Security and Privacy. Various other studies support the notion that these are important requirements for the IoT, however they will be mentioned in the next section.

For the remainder of this thesis, scalability will be considered as part of performance while resiliency will be seen as part of availability. The reason for this choice is because in software evaluation methods, which will be explained later in the thesis, these attributes are also merged in this way.

Interoperability is mentioned as a sub-characteristic of compatibility in the ISO standard. Compatibility is the ability of a system to exchange information with other systems while sharing the same hardware or software environment. Interoperability is defined as the ability of two or more systems to exchange information and also use the information that has been changed.

Evolvability is not mentioned in the ISO-standard, but can be described as a combination of modularity[2] and modifiability[3].

Performance is defined by three sub-characteristics. These are time-behaviour, resource utilization and capacity. The architecture of a system can be have an effect on all three of these sub categories.

Scalability can be mapped to capacity, which is described as the degree to which the maximum limits of a system meet requirements. For a system to meet the required work load, it might have to be scalable depending on the context.

Availability is mentioned in the ISO standard as a sub characteristic of Reliability. Resiliency can be seen as a combination of fault tolerance[4] and recoverability[5].

Security and all of its sub-characteristics ar mentioned in the ISO standard. These are confidentiality, integrity, non-repudiation, authenticity and accountability. Privacy is not a part of the standard. While confidentiality covers a part of privacy, there are also other aspects of privacy that need attention in the IoT.

---

[2] The degree to which a system is composed of discrete components such that a change to one component has a minimal impact on other components

[3] The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality

[4] Degree to which a system, product or component operates as intended despite the presence of hardware or software faults

[5] Degree to which, in the event of an interruption or a failure, a product or a system can recover the data directly affected and re-establish the desired state of the system

3.3 Relevance of Quality Attributes to the Internet of Things

This section will show why the aforementioned quality attributes can be important in the Internet of Things.

Interoperability is important in the IoT as it will contain different heterogeneous objects, networks and thus systems that will need to work together to create an Internet of Things.

Interoperability can be reached by integrating one IoT solution as a subsystem in another solution (by communicating in the same format), or by building a bridge (mediator) through which the key functionalities of each solution can be used [BM] [BS11]. The type of architecture cannot guarantee interoperability, however it is an important tool in helping to achieve it. Design choices can have an impact on how easily a solution can interoperate with another solution.

One of the key problems in the IoT is that data exchange will happen among large-scale heterogeneous network elements [Ma11]. The more interoperable these elements are, the more efficient the data exchange can be. An example of such an optimization can be that if devices that are close to each other can communicate directly to each other without having to go through a network and still get the data that is needed (device-to-device interoperability).

Interoperability ties in directly with Security and Privacy in the IoT [BS11]. The easier the devices at the edge of the network can communicate with each other, the more data they will exchange. This will make it harder to have an overview of the where the user's data is going. It also makes it easier for an attacker to pretend to be a trusted device.

Interoperability can be defined as having four levels. These are the technical, syntactic, semantic and organizational level [KS]. Technical interoperability is associated with hardware software component. The discussion here is usually centered on communication protocols, but ever since TCP/IP became available, this layer of interoperability has not been an issue.

Syntactic interoperability is associated with data formats. The data being transferred must have some well-defined syntax and encoding. Examples of high-level transfer syntaxes include HTML, XML or ASN.

Semantic interoperability concerns the understanding of meaning of the exchanged information. "Semantic interoperability enables systems to combine received information with other information resources and to process it in a meaningful manner" [KS]. This is the level of interoperability that is needed for the IoT, where the providers and requesters of information can communicate meaningfully despite the heterogeneity [BS11].

The final layer is the organizational interoperability layer. This layer concerns automatic linkage of *processes* among different systems. The state of knowledge on this layer is still lacking and is omitted for this reason.

Evolvability. The paper "Evolvability as a Quality Attribute of Software Architectures" explains what the term evolvability means for software, why it should be considered a quality attribute and how it relates to other quality attributes that refer to change in a system [CB06].

There is a need for designs that can withstand and adapt to new requirements and changes, which is inherent in a system that has high evolvability. Systems can become evolvable by having component-exchangeability and increased component distance.

Architectures that are evolvable should be designed in a way that allow changes without damaging the integrity of systems and which can evolve in a controllable way. The level in which a system architecture allows this can be seen as the evolvability of the initial system.

The reason Evolvability is important in the IoT is directly tied to Interoperability. Once a system is deployed, it is important that it can easily be adapted to communicate with emerging technologies and new IoT applications in its domain. If a system is easily changed and extended, than it can easier be adapted to communicate with these new applications.

Requirements for systems are constantly changing and this is the same for IoT Solutions. As technology that supports IoT progresses, the solutions developed for the IoT must be able to evolve with it.

Performance measures the responsiveness and stability of a system. Performance is measured in response time, throughput (number of transactions per time unit), resource efficiency and scalability.

The reason why this is important for the IoT is because the edge of the network is extended with devices with varying levels of processing power and storage. Despite this variety of devices, the IoT should perform at an acceptable level. Designer of the systems will have to decide how much of the application logic and data storage is pushed to the edge and how much is done on centralized servers in order to achieve acceptable performance.

One way to improve performance with respect to response time is to through the Fog computing platform. While Cloud computing frees the end user and enterprise from many details, it becomes a problem for latency-sensitive applications.

"Fog computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers" [BA12].

It can be seen as a Cloud platform closer to the edge of the network, hence the term Fog. The idea is that the devices at the very edge of the network that need external computing power or storage can access these on Fog nodes instead of the Cloud. This leads to increased performance with regards to response time. From this we can infer that moving application logic and data storage to the edge, decentralizing the system, increases performance in the IoT. For this performance gain you have to pay the cost of having smarter nodes near the edge of the network and the security risks of attackers being able to connect easier to the nodes on the ground.

Software architecture can only play a part in the performance of a system. Something that also plays a part in performance is systems it the way software is written, which is out of scope for this thesis.

Scalability is the ability of a system to handle increased workload by applying cost-effective strategies for extending the system's capacity [Wei06]. Gartner[6] estimates that around 4.9 billion "things" are connected at the moment. The IoT has to be able to perform at an acceptable level with this scale of devices.

When talking about scalability of a system, some of the terms that show up are scaling up, scaling out, caching and load balancers. Scaling up means increasing the resources of a particular node. Scaling out means getting more nodes to share the workload. A load balancer has the task of balancing the workload between these nodes. Caching can help with scalability by reducing the number of actual requests to the system.

It is often hard to make a statement about the scalability of an entire system. It is usually the case that some parts of the system has to be more scalable than others. In a centralized system for example, the central component would need to be more scalable than the other non-central components. The centralized component needs to have a scaling strategy, which usually comes down to adding more resources while also designing how the system handles these new resources.

Cloud computing has made is possible to pay for scalability on demand, meaning that extra resources can be added at any time when there is more work to be done by the system running in the Cloud. The Cloud is considered an important part in the IoT, since the IoT will be driven by a lot of data. The Cloud provides the resources needed to analyze and handle this amount of data.

For this thesis, scalability is considered to be a part of performance.

Availability is the ability of a system to be fully or partly operational as and when required [BM]. In the IoT, the user can be human but also another device or system. A system has high availability if it can be maintained or updated while the core is still running. The aim for systems is to have as less downtime as possible. In safety critical IoT solutions such as E-Health, availability could be very important.

[6] http://www.gartner.com/newsroom/id/2905717

Systems that will depend on data gathered from many different solutions may become unable to provide their services if some these solutions are not available. In the IoT, this is a scenario that needs to be anticipated, as the strength of the connectivity of devices at the edge of the network may be unstable at times.

Redundancy in sensors can help with availability. If a solution measures a certain attribute of a city by placing approximately 50 sensors and then 3 of them lose connection or stop working, the overall solution can still be deemed operational as 47 sensors are enough to make an estimation of the current levels of the attribute being measured for the city.

Due to the power constraints on some sensors (battery life), some devices may not always be on. They might push their state once every time interval in order to save battery life as opposed to continuously streaming.

Resiliency. A system that is resilient is one that can effectively handle failures that could affect system availability [BM]. The system should avoid single points of failure and adapt (keep running) to individual component failures.

If the system is resilient than it has high availability. In the same way, redundancy of nodes at the edge can help increase resiliency, since it removes the single point of failure in the system. If a device in the IoT fails, a replacement should be found that offers a similar services. In this way the IoT recovers from failure in a graceful manner. This is especially important in the IoT, since it may contain sensors that are vulnerable and out in the open and can be physically damaged, which can cause a denial of service.

Complexity of systems can lead to less resiliency, since there are more ways in which the system can fail. Combining two or more systems, which is the case in the IoT, lead to greater level of complexity [Axe09]. Resiliency is also the ability to recover quickly from an attack. If security measures fail, the system should still be able to recover to an acceptable state.

Backup systems play a role in resiliency. If a node should fail, a backup node can take over its transactions and make it unapparent to the user that anything has gone wrong.

For this thesis moving forward, resiliency will be considered as a part of availability.

Security. The Internet of Things will introduce many small connected devices at the edge of the network. Therefore, new security concerns arise that are specific to the IoT. The main goals of security are confidentiality, integrity, authenticity, availability, non-repudiation and accountability.

These devices might store sensitive information which would make it sometimes easier for the attacker to access said information. Since the devices in some cases might be physically reachable by attackers (or even stolen), sensitive information might be easier to extract and thus compromising confidentiality of the data.

Because of the scale of the IoT, there are more communication channels than ever between the devices, middleware and servers. This means that there are more entry points for attackers to be able to "flip some bits" and compromise the integrity of certain messages.

The fact that these devices might have limited computation power means that they sometimes cannot run complicated cryptography. One of the challenges for the IoT is to make encryption algorithms faster and less energy-consuming [SW10]. The lack of computation power could make it easier for an attacker to authenticate himself to a device. This causes problems for confidentiality, authenticity, but also availability. By gaining control of the device, the attacker can cause a denial-of-service by either making the device itself unavailable or by sending constant requests to a service or device to which it is connected. Note that there is a difference between availability as a security goal and availability as a quality factor for a system. The former refers to the system being secure in such a way that a denial of service cannot be initiated by an attacker. The latter refers to availability of the system in the absence of an attacker.

In the case of non-repudiation, an unattended device could be used without leaving a trace [BM]. Though if devices first identify who is using their service, this problem might be avoided. However, for some services users might want to operate anonymously. The right balance should be found between non-repudiation (security) and anonymity (privacy).

Privacy. The Internet of Things will bring technology closer to users. New type of data will become available in the IoT, some of which contain personal information. Since the IoT runs on data gathered by sensors, privacy has become a big concern in this field.

First of all, in the IoT a person might not always be aware what personal data is being captured. A person has some control over their own home, they can even choose to make their household completely disconnected from the IoT. However, eventually people have to leave their homes, maybe in a car or public transportation that is now connected to the IoT to reach their work, where the workspace is now also entirely connected to the IoT and monitor their arrival for example. Eventually people will be forced to take part of the IoT in some way or another.

In a non-IoT application, the user is usually granted the control over their data or is at least informed how their data will be used if they choose to use the application. However, in the IoT the data could be gathered without consent, and even if consent is given to use certain data by an application it becomes hard to track the flow of the data since devices might communicate with other systems. What is the rule of user consent if objects may be able to talk to each other spontaneously[BM]?

The reason privacy is diminished in the IoT is because of the pervasiveness and the scale. People will not realize that systems are acquiring information about them and because

of the huge scale in some scenarios, it becomes hard to keep up the data flow even if the user is interested in their own privacy.

In the future it might become a standard to use IoT solutions to accomplish everyday objectives, meaning there is no privacy-friendly non-IoT alternative. It could very well be possible that all cars of the future will be connected to the IoT. In this case the user has no choice but to take part in the IoT ecosystem unless they can live without the functionality that the car provides.

Another reason why the scale of the IoT is a threat to privacy, is because data that might seem harmless on its own could be combined with other data to infer personal information. The IoT should be developed in a way that such personal information cannot be easily extracted from data from several points.

Many approaches to provide privacy in IoT have focused on data minimization, however this is not applicable as the IoT runs data [Hoe12]. One approach to provide privacy is to keep user profiles and preferences on a personal device instead of in the IoT infrastructure. The personal device acts as a firewall between the user and the IoT. The IoT has to first query the user profile before anything can happen.

The IoT-A suggest that psuedonymisation[7], avoid transmitting identifiers in the clear, minimizing unauthorized access to implicit information, enabling the user to control privacy settings and provide privacy-aware identification[8] are some design choices that can be made in favor of privacy in the IoT[BM].

In “Internet of things - New security and privacy challenges” [Web10], they argue that the scale and heterogeneity make privacy enhancing technologies such as Virtual Private Networks, Transport Layer Security, DNS Security Extensions, Onion Routing and Private Information Retrieval unfitting for the IoT.

In “Reference Architectures for Privacy Preservation in Cloud-Based IoT Applications” [Add14], they propose a Secure, Private and Trustworthy protocol (SPTP) which allows end-users to tag private data with a access control list (ACL). The protocol makes sure that any web page or system that wants to access that data is validated with the ACL. This protocol should be administered by a third party, to provide an unbiased regulation of privacy. However this means that this would be centralized and would have to be a very scalable component in the IoT. They also propose a reference architecture, which will be discussed later in this thesis. However, the problem with these reference architectures is that they only achieve their goals, which in this case is privacy, if all systems incorporate this architecture and protocol.

The discussion about privacy in the IoT is very broad. On the legal front, the question of whether existing/traditional legislation is sufficient or if there is a need for new laws.

[7] Creation of a fictional identity
[8] Only the responding host needs to be authenticated

However this discussion is out of scope for this thesis, for now it is enough to note that privacy is an important concern in the IoT.

3.4 Quality Attributes in the Internet of Things Reality

The previous section stated why the quality attributes can be important in the Internet of Things. However, analyzing the set of IoT solutions reveal scenarios with varying requirements of these attributes. These are covered in detail in appendix B.

For solutions such as a wristband that measures a users heart rate and sends it to a mobile phone via Bluetooth, scalability is not as important as security and privacy for example. However, for another solution where many users access an open database that contains data collected from many sensors distributed throughout a city, scalability plays a bigger role than privacy. Some scenarios have to connect a great deal of heterogeneous devices from many vendors, such as the smart home domain, making interoperability an important requirement.

The result is that there is a great variety of solutions that have different quality attribute requirements. This has to be taken into consideration when reasoning about software architecture in the Internet of Things, instead of just saying that all attributes are important.

3.5 Conclusion

This section has introduced the quality attributes that will be used in the analysis in this thesis moving forward. The reasons why these quality attributes could be important in the IoT have been described, however it has also been shown that not all of these attributes are important to all IoT solutions.

Even if a single IoT network could exist, it would not be possible to provide an architecture that exhibits all of the quality attributes mentioned. As we will see later in the thesis, software architecture is all about making trade-offs between these attributes.

The analysis of IoT solutions have shown that there are scenarios where certain quality attributes are not important, for example scalability in a body sensor network or privacy in a system that provides open data. The varying levels of quality attribute priorities in the IoT supports the claim that a single reference architecture cannot exist. The next chapter will divide the IoT solution space into different classes in order to be more specific about the types of systems that are considered to be IoT solutions and also to provide more context information in order to perform an analysis of the compatibility of styles.

Chapter 4

Internet of Things Solution Classes

4.1 Introduction

The Internet of Things contains many types of solutions. This chapter will aim to group the solutions analyzed in appendix B into classes. These classes of IoT solution will give more context information in order to reason about software architectural styles, which is not possible by only using the term "Internet of Things".

The IoT-A and Postscapes group IoT solutions into domains, as shown in chapter 2. However, the solutions and their requirements within these domains can still vary greatly. There are also many solutions from different domains that are very similar to each other. For this reason, this grouping of solutions into domains provided by these sources is not desirable for this thesis. Instead we look at the differences between solutions and use these to create classes. These classes will be used to illustrate the variety of IoT solutions. They will also be used to illustrate the effects of software architectural styles on the quality attributes.

4.2 Classification

The first step into showing the effect of software architectural styles in the IoT while simultaneously illustrating that the IoT should not be seen as one system is to divide it into classes. In order to do this, a number of classifying attributes must be identified. The classification was made with the following constraints in mind:

- The classification is made primarily using the dataset of 18 solutions analyzed.

- The architecture of these solutions shall not be used as classifying attributes. This includes published architectures as well as the component topology that can be derived from the description of the solutions. If this were to be included, then the choices for possible styles would be restricted.
- No quality attribute requirements will explicitly be used as classifying attributes.

Within these constraints I began the classification by comparing the solutions to each other. There are of course many ways in which each solution differs from the next, however I will refer to this subset:

- **Heterogeneous devices**[1] **from multiple vendors**. The solution can either be a closed environment provided by only one vendor or it can be a solution that aims to provide interoperability between multiple vendors in one application for the user.
- **User interface on devices**. The devices at the edge of the network can have a user interface or be controlled by an application on another device. This is important as it shows the way data flows in the system.
- **Sensors and Actuators**. The solutions can either have only sensors or also have actuators. This is also important to show the data flow in the solutions as well as the importance of data aggregation.
- **Devices connected directly to a network**. The devices can be directly connected to a network or might make use of other ways to interact with the system.
- **Device is stationary or mobile**. This can have an effect on availability. If a device is stationary, it is more likely to have a stable connection if this is required. This might not be relevant if the device is not directly connected to a network.
- **User interface application**. Some solutions do not come with a user interface application. This is important as it can classify solutions as being a IoT product or a complete solution.
- **Data per user or collective**. Some solutions provide data gathered for a specific user while other systems provide data collected by all sensors for all users.
- **Number of devices per user**. The values can be *one*, *one to few (constrained)*, *many*. This attribute is important for scalability. This might not be relevant if the data that the user is interested in is collective of all sensors in the system.
- **Devices battery or plugged into power outlet**. If the devices use batteries it might be in the best interest to limit computation and communication to the devices as much as possible. However if they are connected to a power outlet then this becomes less important.

[1]Devices at the edge of the node; sensors, actuators or hubs

- **Devices constrained to a location**: If all of the devices are constrained to a location such as a home or a workspace, then certain design choices can be made in order to improve scalability.
- **Autonomous behavior**: Solutions can exhibit autonomous behavior, by this we are especially interested in actuators being controlled without the use of human interaction.

Not all of these attributes are used in the classification. It is also the case that some of these attributes are dependent on each other. For example, the constrained to a location attribute is false if the devices are not stationary. This will be taken into account in the classification. Figure 4.1 shows the resulting decision tree after several revisions.

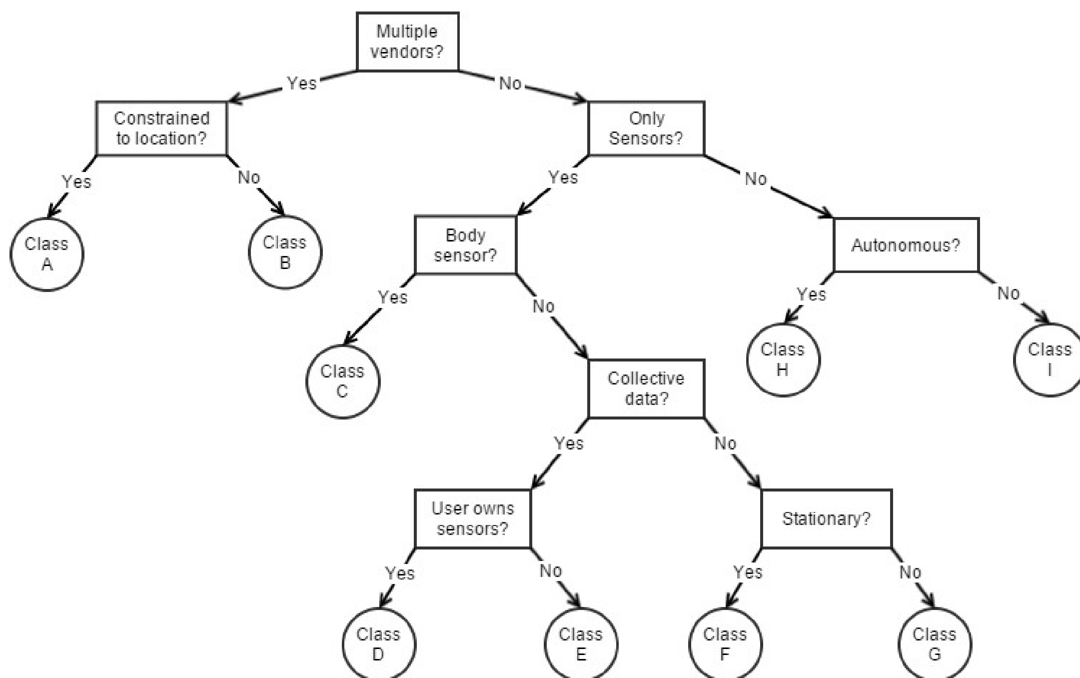

FIGURE 4.1: IoT Classification Decision Tree

Note that a decision tree might not be considered an optimal classification technique for this purpose, as several of classifying attributes can be placed at other areas of the tree since they are not dependent on each other. The stationary attribute could also be placed to split class G or H for example. Furthermore, it is not possible to calculate the accuracy of this classifier for the entire solution space as this would require a dataset containing all, or at least a significant amount, of solutions. However, this decision tree should be enough to provide a set of classes that can be used to show the variety of solutions and the need for different styles. The goal is not to provide a complete set of classes for the IoT.

This particular structure gives us a view into two dimensions and four major categories of IoT solutions.

The dimensions correspond to the two nouns present in the term IoT, namely Internet and Things. As illustrated in figure 4.2, solutions located on the left side of the tree are responsible for creating the interconnected network of things by providing interoperability between multiple solutions when this is necessary. The right side of the tree contains the independent solutions that have sensors and/or actuators that connect the physical world to the digital world. The solutions on the right side of the tree can be seen as the building blocks of the IoT, while the solutions at the left side can be seen as the glue that will paste them together.

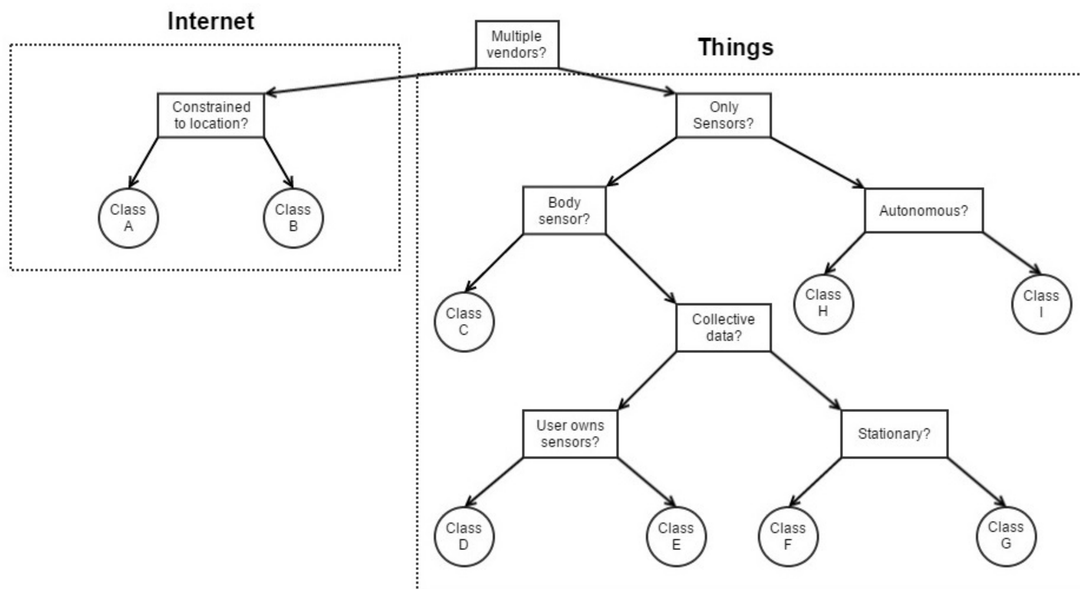

FIGURE 4.2: IoT solution Dimensions

The identified classes can be put into four major categories based on their goal. The first category is integration, which contains solutions aimed at providing interoperability between various solutions. The second is observation, where the goal is to monitor and observe one person using a type of sensor. In this case we are not interested in large-scale data from multiple sensors. This is however the case for the third category, namely aggregation. This category contains solutions that provide value by collecting data from multiple sensors and visualize the data in interesting ways for the user. The final category represents solutions that have actuators in their network. The goal of this category is to provide automation of some kind and does not have to be driven by large-scale data. Figure 4.3 illustrates the four major categories.

4.3 Classes

Up to now we have referred to the classes by an alphabetical ordering, however it is helpful to give them a name. The following is a labeled set of IoT classes that will be used for the remainder of this thesis.

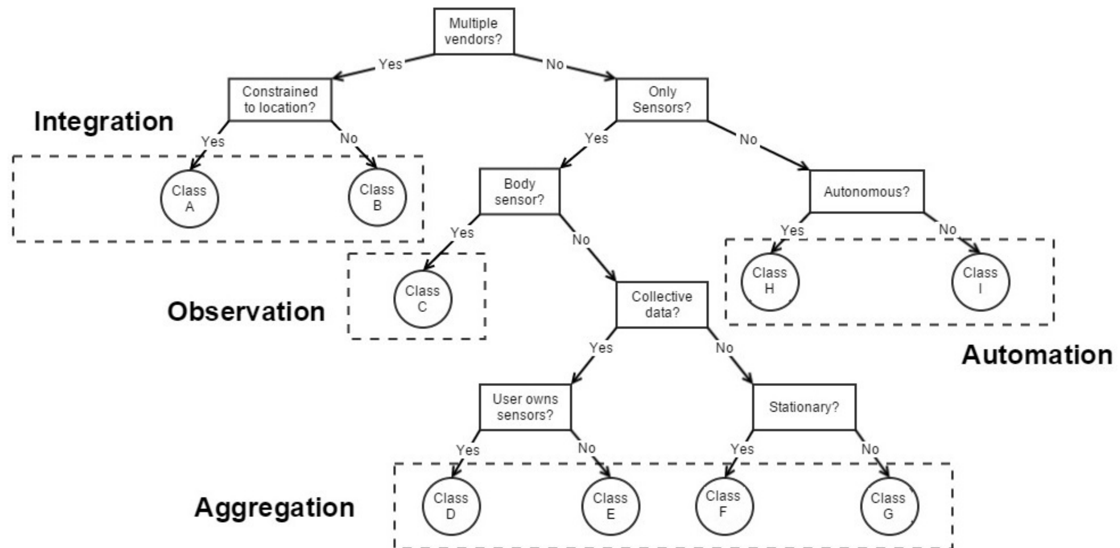

Figure 4.3: IoT solution categories

A. Location Constrained Heterogeneous Devices. This class is categorized by solutions that provide interoperability between many solutions that contain device in the same location, such as a smart home or work office. This class must take into account that performance is not entirely in their control because it is dependent on the architecture and implementation of other solutions. The fact that all devices are in the same area can mean that we can create a more decentralized environment and group connections to a central point on the ground (Hub) instead of the Cloud. Solutions that fall into this class are the Homeseer and SmartThings solutions.

B. Location Free Heterogeneous Devices. This class also provides interoperability between solutions, except the devices can be located anywhere. This means that a central point must be located on a server possibly in the Cloud to provide a single point of interaction. Scalability plays a bigger role in this class. Also, it becomes possible to integrate solutions from the previous class into this class. An example is the SensorFlare[2] which lists SmartThings as one of the solutions it provides interoperability for.

C. Body Sensors. Characterized by the one-to-one relationship between users and devices, this class contains solutions that monitor and observes measurements of the user. This can be for personal uses but also in particular for E-Health. In the latter scenario availability can become important. Scalability is relatively less important for this class. Solutions in this class include the Nymi and the Angel wristband.

D. Active User Collective Data Solutions. This solution aims to collect data from multiple sensors and provide an analysis on the entire set of data as a whole. Users are considered active as they contribute to the data set actively through sensors that they own. An example is the WeatherCloud system where each user equips their car with

[2] https://www.sensorflare.com/

multiple sensors. The aggregation of the data is done by sensors owned by multiple users and is collectively visualized.

E. Passive User Collective Data Solutions. The difference between this class and the previous is that the users of the data are not owners of the sensors. This means that the number of sensors is entirely in control of the party that owns the solution, meaning scalability only has to be handled based on the number of users. Solutions in this class include the Array of Things and the Zebra Motionworks solutions.

F. Stationary Homogeneous Sensors. This class contains solutions that have sensors that are stationary and are of the same type. The difference between the previous two classes and this one is that the data should not be seen as one whole but belongs to a user group. The data belonging to a certain user group should only be visible to them and no one else. In this scenario privacy becomes a more important requirement.

G. Mobile Homogeneous Sensors. These solutions have the same goal as the previous class except that the sensors can be in motion, meaning that providing availability becomes a more important requirement. Solutions that fall into this class are the Farmobile, DAQRI smart helmet, Truvolo and Veniam solutions.

H. Smart Systems. The term smart is used a lot these days to describe any alternative version of a device or system that provides some automation. In this classification we use it to describe autonomous solutions that are able to use data and logic and convert it into decisions that can lead to actuator commands without human intervention. This brings many concerns of security and safety to the table. A big difference between these types of systems and the last four classes is that they do not necessarily need to be driven by large amounts of data, meaning that a more decentralized architecture might be possible for some solutions in this class.

I. User-controlled Actuators. The final group contains solutions that also contain actuators, except that these are explicitly human controlled. An example of this are the S5 electronic shelf labels and the Bitlock Bycicle lock solutions.

4.4 Conclusion

This chapter has provided a classification of the Internet of Things based on an initial set of IoT solutions. The differences and similarities between solutions were used as a foundation for this analysis. The resulting classification tree provides an illustration of four categories of solutions, each with a specific way to provide value to the users of the solutions. The classes provided in this chapter will be used in the next chapter to analyze the effect of software architectural styles on quality attributes in the context of the IoT.

Chapter 5

Software Architectural Styles in the Internet of Things

5.1 Introduction

This chapter contains the mapping of software architectural styles to the IoT classes presented in the previous chapter and analyses the effects on the quality attributes described in chapter 3. The chapter will start off by mentioning which software architectural styles will be considered and how they will be evaluated. Using this information, a mapping and analysis is presented which describes the best software architectural styles to use as starting points for architecture in the different IoT classes.

5.2 Software Architectural Styles and Evaluation

A software architectural style is a labeled set of components[1] and connectors[2], and a set of constraints on how they can interact [GS94]. These constraints can be topological, for example not allowing cycles, or it can regard execution semantics. The latter refers to the meaning of such an interaction between two components, which could be a procedure call or a notification for example. All styles come with trade-offs, explicitly mentioning which quality attributes[3] are gained and which are given away, however this also depends on the context of the system to be built.

The software architectural styles that will be considered in this thesis are Client-Server, Peer-to-Peer, Pipes-and-Filters, Event-Based, Publish-Subscribe, Service-Oriented, REST, Layered and Microkernel. There are other styles that exist, however these are some of

[1] Component: A constituent part; element. http://dictionary.reference.com/browse/component
[2] Connectors: Description of the interactions between components. [GS94]
[3] Quality attribute: Non-Functional Requirement

the most common and well documented ones. In case the reader is not familiar with these styles, a description is given in appendix C.

There are a number of Software Architecture Evaluation Methods that can be used to evaluate software architectures for their fulfillment of quality attribute requirements. Appendix D describes some of these methods and why they cannot be applied directly in this analysis. In short, these evaluation methods are meant to be used at a later stage in the design process where more information is needed about the system to be built. However, in this thesis we analyze the very first step, namely which style to choose, in the design phase. For this analysis it is only necessary to know how quality attributes will be evaluated in this analysis.

For the mapping we will identify what the quality attribute requirements are for each class. The architectural styles provide variations in how these requirements are fulfilled by the architecture, which will allow us to compare them with each other.

Interoperability. For interoperability the requirements could either be primary or secondary. We have seen enough examples of solutions where interoperability is not mentioned at all, however for this analysis we will categorize these solutions as having interoperability as a secondary requirements. Interoperability fulfillment by styles will be measured by the presence or absence of the tactics for this attribute, which are mentioned in the previous section.

Evolvability. Evolvability is about reducing the cost of change to the system. For each class of solution we will indicate some of the likely changes to occur. The choice in style will dictate how and where these changes will occur and thus how evolvable the architecture is. Evolvability can also be evaluated by the presence of an intermediary, which decouples components from each other and centralization which brings logic to one central component where change only needs to occur in one location.

Performance. We will consider latency, throughput, power consumption/energy efficiency, bandwidth efficiency and scalability as characteristics that define performance in the IoT. These will all be affected by the choice of architectural style. Latency can be measured by the number of hops needed to reach the destination. Intermediary components increase latency. Energy efficiency can be measured by looking at how much logic is present in the edge devices and how many messages they have to send. Scalability can be evaluated by estimating the number of devices and users that have to communicate with a central component. Removing unneeded interactions to the central component or decentralization can decrease the need for scalability.

Availability. We can make an estimation of how much impact a single device being unavailable could be. We can also identify single-points of failure inherent in the classes and their goals. Request-Response interaction is considered good for availability because the response can be a confirmation message.

Security. Security is always a priority. For this reason we will not make an estimate on the requirement for this attribute, however we will refer to it later to see if the choice of architectural style has an impact.

Privacy. Some solutions, like the ones that contain collective open data, have less of a privacy requirements than other systems. This will be mentioned per class. Data can be personal or open. There is no known way of measuring privacy in software architecture, however an attempt will be made anyway to show that this is indeed difficult/not possible.

The remainder of this chapter contains the actual analysis of software architectural styles in the Internet of Things solution classes.

5.3 Mapping

This section presents a format with which the mapping will be performed and the actual mapping itself. This will ensure that the analysis can be done in a systematic way, as well as making sure that all possibilities have been considered. The following is the format that was used:

For each category:

- Description
- For each class:
  - Description
  - Functional Requirement(s)
  - Quality Attribute Specifications
  - For each style:
    - Description
    - Quality Attribute Effects
  - Verdict

This means that for each of the four categories we will look at the classes and what the effects of software architectural styles are on them. While this thesis focuses on quality attributes, we will also list a few functional requirements as this will help get a better view of what functionality the system should provide which can help to eliminate styles that are not suitable because their constraints are not compatible. The quality attribute specification indicates what the specific quality attributes mean in the context of that class and how much of a priority they are.

It should be noted that all styles can be used to create any type of system, however this has an effect on the quality attributes that the system will exhibit. An example is using a Peer-to-Peer style for data aggregation. This is possible, however it will not perform as well as using a more centralized style where the data comes together at one point. At the end of each class description there will be a verdict indicating which classes are the best fit.

While this format has been used for the analysis, the description found in this thesis will deviate from this format in order to illustrate important discoveries or to leave out redundant information.

The reasoning used in this document is based on information found in several references regarding software architecture, styles, quality attributes and software architecture evaluation methods. The goal is to illustrate a) which styles should be used in which type of scenarios in the IoT and b) that there is indeed a need for different styles and thus architectures in the IoT, since it contains such a wide variety of applications. The second goal supports the claim that there cannot be a single generic reference architecture for the entire IoT.

The remainder of the document contains the mapping presented in this format.

5.3.1 Integration

The integration category of IoT solution contains systems with the primary goal of providing interconnection between multiple IoT solutions from a variety of vendors. The traditional definition for interoperability is the ability of a component to interact with other components or systems. However, what is unique in the IoT is that there should also be interoperability between solutions from multiple vendors. The reason is that not all companies have the knowledge of all the different types of sensors and actuators. It becomes inevitable to work together with these companies instead of trying to provide a one-company solution. A report on the IoT done by the McKinsey Global Institute states that 40 percent of potential value of the IoT is enabled when integrating multiple IoT solutions [Man]. The potential value in this case is described in economic impact.

There is a big difference between planning for interoperability between different technologies and platforms from the start of a system design and trying to provide interoperability between systems that are already built. The second scenario is what this category deals with. In this case these solutions already have their own architecture that caters to their specific requirements, which can be based on many different styles depending on the application. In the worst case scenario, there is such an architecture where no attention has been paid at all to interoperability.

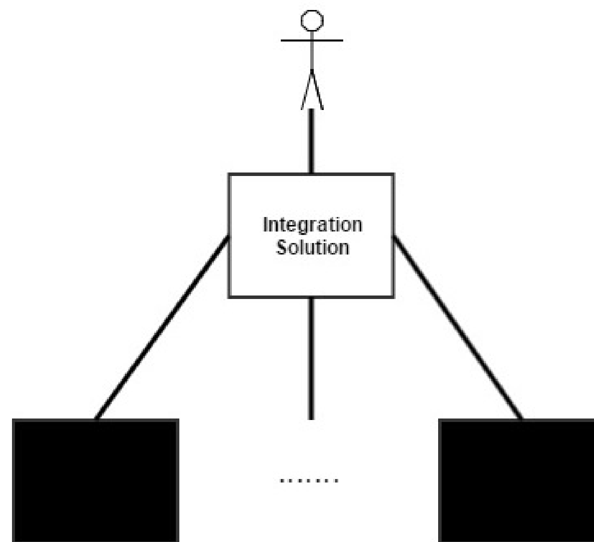

FIGURE 5.1: Integration Category

Figure 5.1 gives an illustration of this category. The different solutions that will be interconnected using this system are depicted as black boxes, since the details might not always be known and should also not be important. The direction in which communication and data flow have been left out on purpose, as this will be decided by the choice of style. The Integration solution component in the diagram represents all of the software to be built for this category, which will be decomposed based on the style chosen. This category of solutions provide a single point in which the user can communicate with and control all of these systems, instead of having to do this separately in different applications. In addition to this, there should be the possibility to add autonomous behavior, which in this case means adding scenarios where certain actuators will respond to measured data by sensors. This removes the human effort while still providing benefits for the user.

We divide this category into two classes, where the classifying attribute is whether or not the devices are constrained to a certain location, meaning they are physically close to each other. A concrete example of such a location is a smart home. This difference can lead to certain architecture choices and even different business models.

In the first case where the devices are close to each other, for example in a smart home, the vendor can choose to sell a hub which will act as a central point for all of the communication within a certain proximity. This cannot be done for the second class, which will probably have to make money with a subscription based business model by providing a service instead of hardware. A big difference is in the need for scalability, as in the first case we can choose a decentralized approach by letting all computation occur in these hubs instead of in some Cloud component for example.

All of the architectures that can be used for Class B can also be applied for Class A, however the inverse is not true. This is because the hub can simply be used as a gateway that relays communication to the Cloud. The hub can in this case be omitted from the architecture, resulting in design that can be used for both Class A and Class B. However, Class B can never take advantage of such a hub because no assumption can be made about the location of the devices. For this reason while analyzing Class A we will only look at the situation where there is a Hub, which acts as a gateway for all of the devices in their location. This means that the hub will be interacting directly with devices instead of their Cloud services. At the end of this category description it should be clear how this affects the system.

5.3.1.1 Class A. Location Constrained Heterogeneous Devices

This class contains solutions that provides interoperability between multiple heterogeneous devices located in close proximity to each other. We make the first design decision by introducing a hub which acts as a central point of communication between these devices. This is to help illustrate the differences between class A and class B. In this way we take advantage of the fact that these devices are contained within a location in order to create an environment of interconnected devices. Figure 5.2 illustrates this class by showing the hardware that will be used. In this case we have replaced the black boxes from the previous figure with devices, which are sensors and actuators. This requires interoperability at device level.

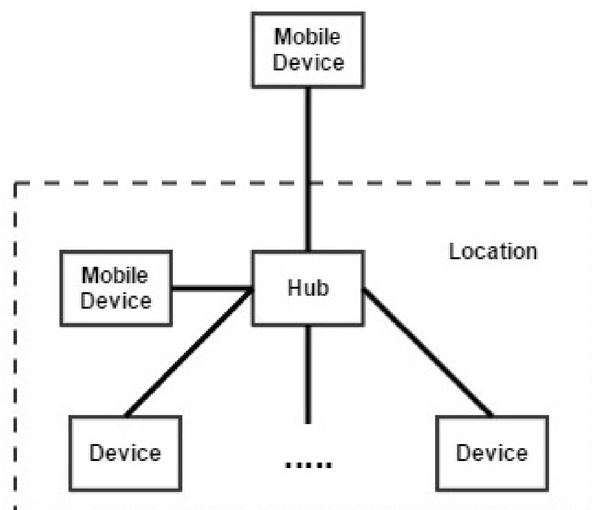

Figure 5.2: Location Constrained Heterogeneous Devices

The devices and the hub are all located in close proximity to each other, which means that they can communicate directly to each other or are at least on the same local network. The user will be able to use the system via a mobile device, which should be able

to communicate with the system from within or anywhere outside the location, meaning that the hub should be connected to the Internet.

Functional Requirements:

- The user can view measured data from sensors and control actuators inside the location from one application.
- The user can choose from a set of smart scenarios in order to create an autonomous smart environment.
- The user can add and remove devices from the system.

Quality Attribute Requirements:

- **Interoperability:** This is a primary requirement for this class, meaning that we should consider styles that provide interactivity and possibly inherent interoperability between components.
- **Evolvability:**
  - S1: Change to UI application
  - S2: Change/Addition autonomous behavior logic
  - S3: New device
  - S4: Change in data format of a device
- **Performance:** Latency depends on the combination of devices and the hub, however the performance of devices are the responsibility of their vendor. The Hub should be able to handle multiple interactions in a second, so throughput should be measured. The Hub will most likely be connected to a power outlet, however the devices may vary. It is important to choose an architecture that will not strain devices unless it is needed. Finally, different types of architectures can make scalability a primary issue or a secondary one as we will see later on.
- **Availability:** The hub is a single point of error, however in the worst case scenario the users should be able to communicate with their devices using their respective applications. The choice of architecture can also have a big impact on availability in this case, specifically when choosing a centralized or decentralized approach.
- **Security:** If the attacker can gain access to the Hub, it can interact with all devices in the location. If the attacker can gain access to a device, they may cause a denial-of-service by sending multiple requests to the Hub.

- **Privacy:** The privacy level of the data depends on the data being measured by the third party devices. However, since all devices are in one location, it is possible to provide privacy by choosing an architecture where the data does not need to be stored on a shared resource such as a Cloud server.

Client-Server

In a Client-Server style the client always initiates the communication with a request to the server, which replies with a response. This means that the client has to know the identity and network address of the server and the server has to be a non-terminating process. Any server can also take the role of a client to another server, however a server may not be a client to its own clients. Because of these constraints, there are two possible topologies using this style for decomposition.

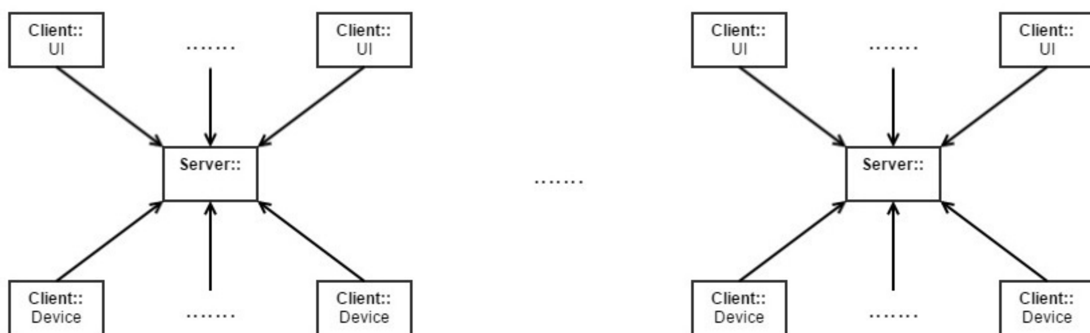

Figure 5.3: Client-Server Topology A: Centralized

Figure 5.3 shows the first topology of components possible with this style. In this topology there is one central server and multiple clients. The clients can be the UI device or the devices at the location. Notice that each location has such a structure, which is why there are multiple instances illustrated. The central component can be located in the Hub or in a Cloud server, this will be discussed later. The second topology is illustrated in figure 5.4, where the *main component* also acts as a client to the multiple devices that now have the server role. This topology can also be seen as a layered architecture with three layers, presentation, control and data. The environment of the location of the devices can be seen as a database, where sensors read the data and actuators write to it. In this sense it is very much like a Model-View-Controller pattern, which is considered a design pattern and not an architectural style in this thesis. The reason for this is because it is an instantiation of the layered style.

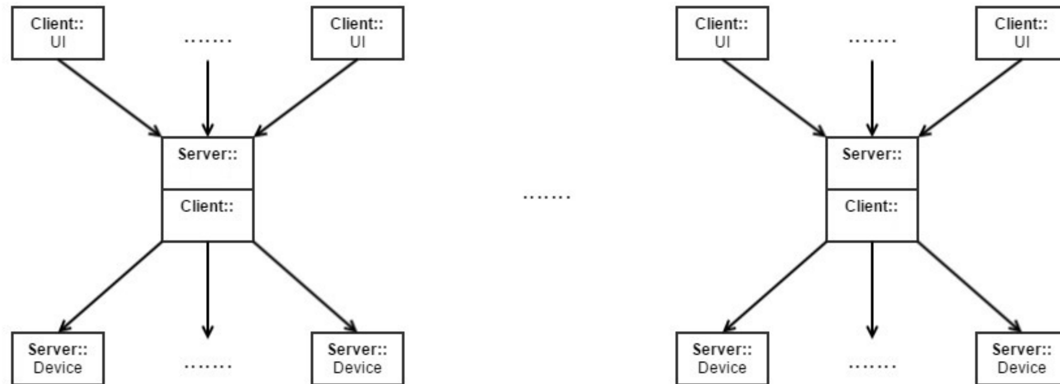

FIGURE 5.4: Client-Server Topology B: Hierarchical

It is important to state the differences between the two topologies:

- In topology A, actuators will constantly have to query to see if there is a request from the user to do something. This results in unnecessary requests to the server which increases the workload on the server and thus reduces performance and scalability. In topology B, the server sends a request to the devices only when they are is a need, either from a user request or logic programmed into the Hub.
- In Topology B, devices are forced to always be in an ON state, since they have the server role. It eliminates the possibility that devices can go into a OFF (almost off) mode to save energy and only start up to update or check on updates every once in a while. This type of behavior is possible in topology A, however this might increase latency as the user would have to wait for an actuator to start up and send a request to see if any commands have been issued by the user before performing the action requested.
- In Topology A, the main component needs to provide an interface and all devices need to be able to communicate with this interface. In Topology B, the devices need to provide an interface which will be used by the main component. The difference is which component needs to make changes when a new device is added to the system.
- Topology A might be better for certain sensors, such as motion detectors. The sensor would only have to send a message when a motion occurs instead of saving an event log and sending it when requested by the main component. This could be an indicator that we might need a style that allows for different types of interaction depending on the type of arrival pattern.

For this class we will choose topology B moving forward, as it presents a more efficient way of querying for data and sending actuator commands. Furthermore we can assume that a majority of the devices are stationary since they are constrained to a location and

are plugged into a power outlet when possible, meaning that having them always on as a non-terminating process is not a huge price to pay. Finally, it is better to make changes in the main component to allow communication, which is in control of the developers of this system, than it is to require change in the sensors and actuators. However, this topology does not perform well for motion detectors or other sensors that detect events instead of constant measurements, as the main component will have to constantly query for changes that might not occur for a long period.

The quality attribute effects of this client-server style on this class of IoT solutions are:

- **Interoperability**: The system uses the devices instead of being used. This means that the devices simply have to provide their functionality and do not need to worry about finding the main component to communicate with. However, this system will need to keep track of all devices it wants to provide interoperability with for the user and translate all communications into a format that the devices understand.
- **Evolvability**:
  - S1. Changes to the UI are isolated to the UI component, which is a typical client-server decomposition.
  - S2. The Autonomous behavior logic will be present in the main component, which is isolated from the UI and devices.
  - S3. A new device potentially means adding a new adapter to the main component which can translate communication to the correct data format. The main component is also responsible for detecting new devices in this scenario.
  - S4. Changes to data format for a certain device are isolated to the main component.
- **Performance**: Latency depends partially on the devices, which are out of control of the developers of this system. The remaining latency is dependent on the architecture of the main component. The more layers and components the communication has to go through, the higher latency becomes.

  The resources such as bandwidth and energy are used efficiently as communication and computation only occurs when it is needed. The managing of scalability of the system depends on which hardware the logic of the main component is located. Putting this logic in the hub means higher performance and less scalability issues, since each location will host their own logic. However, this decreases evolvability as every time there is a new update to a system then this must be pushed to all hubs. Moving this logic to a cloud component and using the hub as a simple gateway will increase evolvability as updates only need to occur in one location, but decrease performance and scalability since communication has to go over a network and all users and devices communicate to a central component type.

- **Availability**: The central component is a single point of failure for the system, however the impact of this failure depends on whether we put the application logic in the Hubs or in the Cloud. Having each Hub host their own logic means that if the main component should fail, only the devices in that location will fail to work. However, if the logic is hosted in the Cloud than the impact of one server going down is much higher.
- **Security**: The main component is an entry point to which the attacker gains access to all devices as it knows their identities. It is not possible to say too much about security without having another style to compare it to.
- **Privacy**: Data flows through the main component, so privacy is depends on whether the application logic is located in the Hub or in a Cloud component. Having it in the Hub means that data never needs to leave the location and does not share resources with other processes.

What we have seen from this first analysis is that some of the quality attribute effects still depend on whether we have a centralized or decentralized approach, i.e. putting the main component logic in the Cloud or in the Hub.

Peer-to-Peer

A Peer-to-Peer contains peers that are both consumers and producers of data and functionality. In this sense, a peer is both a client and a server. The constraints on the peers are that they all provide similar services and use the same communication protocol [Bas]. This is not a good fit for this class as all devices provide different types of services. Sensors provide data while actuators provide functionality. The constraint of having all peers be similar increases availability and performance. Availability is increased because one peer failing does not affect the rest of the system. Performance is increased because peers can always communicate with the closest or least busy peers. However in this system, the peers cannot serve as backup for each other as they provide completely different services. Creating an abstraction layer with which all peer nodes can communicate with each other would be too complex for such a wide array of heterogeneous devices.

Pipes-and-Filters

The Pipe-and-Filter style is used when a system needs to perform a series of transformations on the input data. The main disadvantage of this style in this context is that it does not support interactivity. In this system we have a user interface but also have interactivity between different components. The inherent advantages such as predictability and reusability of components provided by this style is not in high demand in this context. This style is the least appropriate for interoperability in this set of styles and can be omitted.

Event-Based

In an event-based architecture there are event-producers, event-consumers and an event bus which links the two together. The event-producers push their events to the event bus and event-consumers can listen to those events by registering to the event bus. In this style, the event-consumers know who the producers are and register to specific events from specific producers. This is different than the publish-subscribe style, where the consumer is only interested in a type of data and not where it came from.

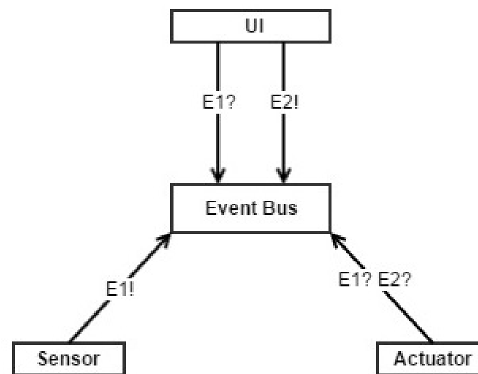

FIGURE 5.5: Event-Based Topology

Figure 5.5 shows the minimum components needed to run this class in an event-based architectural style. The exclamation point means an event has been produced, while a question mark indicates a registration to an event. In the diagram provided, both the UI and actuator listen to event E1 by registering for it on the Event Bus. This means that both must have knowledge of the existence of that particular sensor and what kind of event it produces. Once the sensor produces the E1 event, the Event Bus will notify both the UI and the actuator.

In an Event-Based architecture, the event producers do not know who their consumers are and do not get feedback on whether their event has been processed. This is not a problem for the first event, however the second event might be something like "turn on the lights", which the actuator will respond to. However, the user will get no confirmation that the lights have been turned on, unless the actuator produces an event and the ui registers to this event. Another problem with this style is that the user cannot address individual devices, instead it can only listen to and provide generic events.

For persistence a component can always be created that listens to all events and creates a log or saves data to a database, however this is not a priority in this class.

The quality attribute effects will be compared to the client-server style:

- **Interoperability**: All devices have to be able to communicate with the Event Bus. This means that the third party devices need to change their communication

in order to be part of this system. From the perspective of the developers of this system, the Event-Based style is more interoperability friendly. However, from the perspective of the third party device vendors, the Client-Server style is more interoperability friendly. This is an indicator that stakeholder perspective can also play a role in evaluating quality attribute effects.

- **Evolvability**:
  - S1. There is still a separate component for the UI.
  - S2. The change or addition to new autonomous behavior logic will have to occur in each actuator by having it listen to certain events instead of in a central component. This means that evolving the system is much harder with this topology than it is with the Client-Server style.
  - S3. Adding a new devices does not mean any change for the system itself, the new devices just needs to either provide events or register to events via the event-bus. This means adding new devices is easier with this style in the perspective of the developers of this system but harder for the vendor developers.
  - S4. The devices can change their data format internally however they must continue to interact with the event-bus, so that communication cannot change.
- **Performance**: The latency using this style might be higher than the Client-Server style for certain functionality. For example, if the user is interested in a certain sensor data value, they have to register to the Event Bus and wait for that sensor to post an event. In the Client-Server style, the data is retrieved as soon as the user requests it. However, if the device posts events fast enough than the event-based style will perform better due to less messages needed.

  The Event-Bus should be able to handle more interactions per time unit than the main server component, as it only has to notify listeners that an event has happened and possibly relay some data. The main component might also have to perform some computation and send confirmation responses.

  This style is more efficient for sensors such as motion detectors, which will only send events when there is motion detected instead of constantly sending periodic messages. This is again an indicator that different interaction schemes might be needed depending on the type of sensor.

  This style can also be considered slightly more scalable as the main component only relays messages and depends on the event-consumers to process the data.
- **Availability**: There is no increase or decrease in availability, the Event-Bus still serves as a central point of failure.
- **Security**: The Event-Bus is less of an interesting entry point for an attacker than the main component of the Client-Server style, since it cannot initiate requests on

command and does not keep data stored locally. Communication also only occurs via events, making it harder for attackers to provide malicious data. In this sense, the Event-Based style can be seen as more secure than the Client-Server Style.

- **Privacy**: Data is not stored on the Event-Bus, but is relayed to event-processors. In this sense the event-based style might be considered more privacy friendly than the client-server style granted that only trusted devices are communicating to the event-bus.

This style might not be suitable as a primary choice, however it can be implemented within another style such as the Client-Server style. There can be some event-based interaction inside the main Client-Server component in order to provide autonomous behavior.

Publish-Subscribe

The problem with the Event-Based style is that the producers do not know the consumer, which in this context is necessary if the user wants to address one of its devices individually. In the Publish-Subscribe style, the producers and consumers become even less coupled, where publishers publish a type of data and subscribers are interested in a type of content and do not care about the producers of data. The topology would not look much different than that of the Event-Based style, however it would be harder to address individual devices, which is a functionality requirement. For this reason, this style is not explored any further for this class.

Service-Oriented

In the Service-Oriented style, service providers advertise their functionality to a service registry, which can be used by service consumers to find appropriate services for their needs. Once found, these service consumers can communicate directly to the service providers. Any service provider can be a service consumer and vice versa. A common component found in this style is called an orchestration service, which acts as a service provider to a user interface and acts as a service consumer by using multiple services to reach a goal. The main goal of this style is considered to be interoperability, however that is due to the way it has been used in practice, which is to have all services communicate using one network protocol such as SOAP. This means that there is no inherent complete interoperability that comes with SOA, however it is aided by the registry component which provides discovery, which is one of the tactics for providing interoperability [Bas].

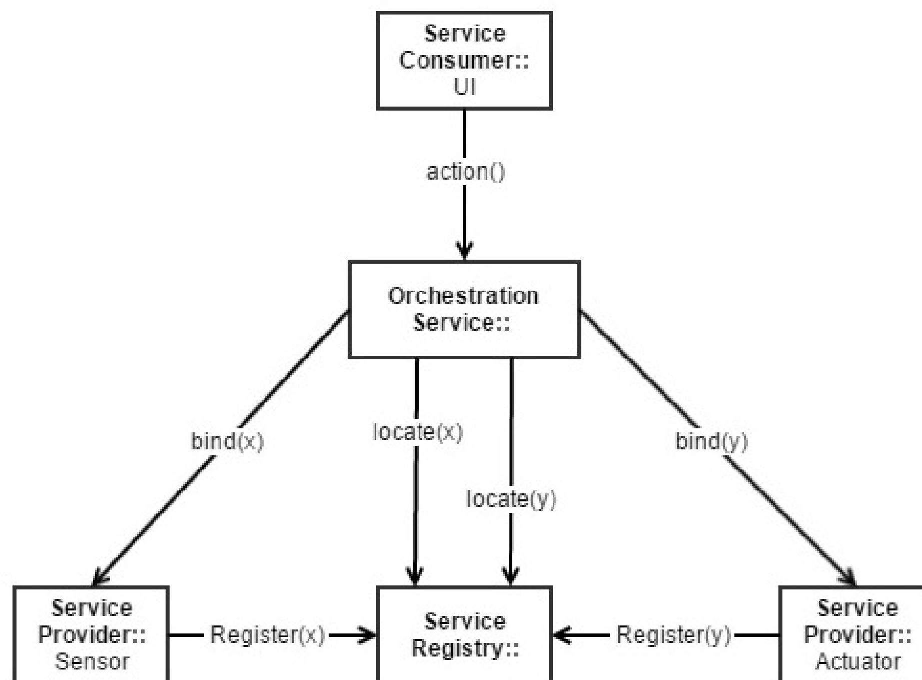

FIGURE 5.6: Service-Oriented Topology

The topology for this system can be viewed in figure 5.6. Note that the orchestration service can either be put in the mobile application, in the Hub or in a Cloud component. This choice makes a big difference for the quality attribute effects.

Orchestration in the mobile device. Having a smart app on the mobile device that communicates directly to a registry, can find your devices and send messages directly to them decreases latency as the UI and orchestration service are co-located, meaning that communication occurs faster and thus one network message is eliminated. The trade-off is that there is a bigger app running on a mobile device, which uses more energy. Another negative point is that the orchestration logic will need to be updated in every mobile app when there is a change, decreasing evolvability.

Orchestration in the hub. This option creates a separation of concerns between user interface logic and application logic, which is typical in a client-server scenario. It increases latency however, since there is an extra layer of communication that has to go over a network. Evolvability is increased compared to the first option since only all hubs need to be updated now.

Orchestration in the Cloud. In this option, the hub is simply used as a gateway to the Cloud component which hosts a central orchestration service and service registry. This is the type of architecture the SmartThings solution has, where all logic is located in the Cloud. The advantage is that all logic is in one place, which makes it easier to evolve the system and also allows all users to have the same version of application logic.

Another advantage is that the hub does not have to compute much, making it possibly cheaper to produce. The disadvantages are lower performance because communication has to occur to a central Cloud, a higher scalability requirement as all devices and users will be using the central Cloud component, higher internet-dependency as the hub cannot perform any actions without an internet connection, lower availability as the single point of failure can cause a complete system crash and less privacy since all data will have to pass through a central Cloud component.

This makes it harder to compare this style with the previous two candidates, since all three options are good in specific situations. However we will try to generalize the quality attribute effects:

- **Interoperability**: This style inherently has a discovery component, which is one of the tactics for providing interoperability in a software architecture. However, the service providers still need to be able to communicate to the service registry and describe their services in the appropriate format for this interoperability to occur. However, once this is done, they will also be discoverable within the context of this system.
- **Evolvability**:
  - S1: The UI has its own component and is decoupled from the logic.
  - S2: Autonomous behavior will be located in the Orchestration service component. This is the same as the Client-Server style, where this logic isolated from the UI and from the sensors and actuators.
  - S3: A new device has to be able to communicate with the registry, however once it is registered the orchestration service or any other service consumer in the system can discover and use it.
  - S4: Changes in the data format can be reported to the service registry by updating the description of the service and how to invoke it. This makes it easier for the vendors of the devices to provide newer versions.
- **Performance**: The best case scenario for latency is if the logic is located in the mobile device, thus allowing direct communication with devices once they are bound. The worst case is having to communicate with a cloud component in order to use this logic. This decision can make latency better or worse than the two other candidate styles.

  The decision of which hardware to map the software on also has an effect on energy consumption, as keeping the edges simple will require them to compute less and make them cheaper to produce.

  Choosing the Cloud first approach will require scalability tactics to be employed, such as replication and layering with a load balancer. Having the logic in the mobile devices or in the hub decreases this need.

- **Availability**: The Cloud first approach creates a single point of failure, while having the logic in the hub only removes functionality for that location. Having the logic in the mobile device means that only one user will be affected if an instance of the orchestration service should fail.
- **Security**: Typically services do not know or care who invokes them. Security in service oriented architectures is typically solved by making agreements on how to sign and authenticate messages within the system. However, this is not a part of the style itself. The registry contains information on all registered services. If an attacker gains access to this component then they will have knowledge on how to properly invoke all devices. They can for example cause denial-of-service by sending multiple messages to a device. The registry should in this case have a way of authenticating its users.
- **Privacy**: Privacy as we have seen with the two previous styles is dependent on where the logic is located. Having it in the Cloud means that valued sensor data and actuator commands need to go through a central cloud component, which the user has to trust is not being used for any other purpose. Keeping the data flow constrained to the location means that the data does not leave the local network except to the mobile device when it leaves the area.

The trend we are beginning to see is that the choice of centralization versus decentralization makes a big difference on the quality attributes regardless of the initial style chosen. However, the service-oriented style is the best choice so far with respect to interoperability and allows us to map the software to hardware in three different ways, which means that it is more flexible and can be adjusted to the needs of the system to be built. I also includes the discovery and orchestration tactics for interoperability.

REST

The REST architectural style will provide the same topology as the client-server style except that the components will have a uniform interface, which increases evolvability by decoupling implementation and the services that components can provide. Furthermore, this increases interoperability as any other component can communicate to each other by using this uniform interface. The trade-off is in performance, since information is transformed to a standardized format rather than one that is specific to the application's needs [Fie00].

The "Web of Things" concept is the idea of having sensors and actuators themselves host web services, thus making them available as resources via standard web mechanisms[Sti08]. The architecture proposed looks similar to the topology we have provided in figure 5.6 for service oriented, where the registry service is referred to as a resource repository containing the URI's of the devices and description of services, which are machine-readable. The repositories have two sets of references, a private set and a public set. This is in

line with the view that each IoT solution is its own private intranet of things which can be combined with other solutions to form a bigger network. The private set includes the location of resources which are only allowed to be shared within the intranet, while the public set contains resources that may be shared.

The advantage of REST and more specifically the Web of Things scenario is that the sensors and actuators can be reused in other applications. It depends on the type and goal of the device if this is wanted or not. The uniform interface increases interoperability and evolvability but reduces performance.

Layered

They layered style is used to decompose a system into cohesive layers, where layers can only communicate with the layer below it and does not have knowledge of the layers "above" it. The layered style can be seen as an extra constraint that can be combined with styles such as Client-Server and Service Oriented. Adding a layer means increased evolvability but a decrease in performance. Using the layered style can yield many different topologies, however the minimum needed is shown in figure 5.7.

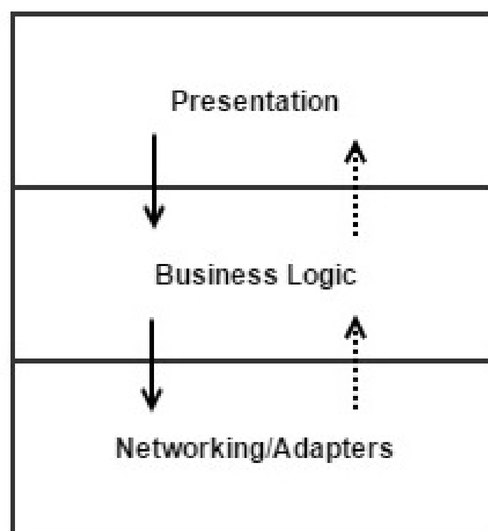

FIGURE 5.7: Layered Topology

The presentation layer contains all UI related components, the application logic is located in the middle layer and the logic to either connect to or accept requests from devices is located in the networking/adapter layer. These layers do not need to be located on the same hardware. The layered style gives us a good overview of the types of components present, however it becomes hard to assess the quality attribute effects without any more information.

We can fit the Client-Server topology in figure 5.4 inside these layers while still adhering to the layered style constraints. This is illustrated in figure 5.8.

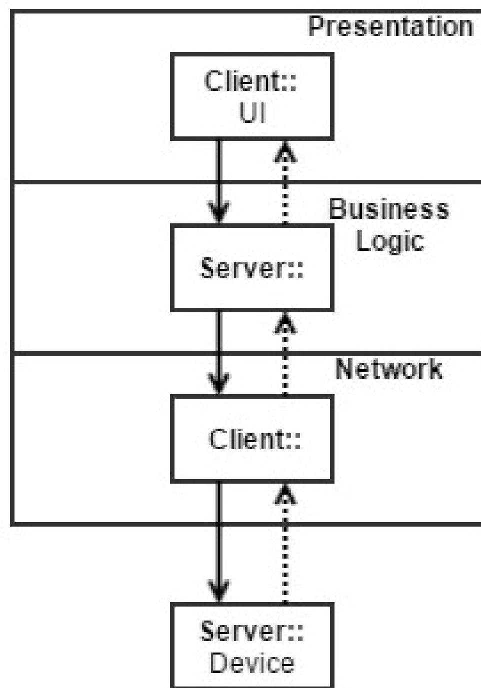

FIGURE 5.8: Layered Client-Server Topology

In this context the layered style is analogous to the Client-Server style for the second topology.

Microkernel

The Microkernel style provides a component with the bare minimum functionalities needed for the application, called the Microkernel, that can be extended using independent plugins. The advantage of this style is that it provides customizability by allowing the user to only install features that are wanted instead of providing a full set of functionalities for all users. This can be a useful style to use to allow users to customize the functionality inside their hub, however is not a good choice when all of the logic is located in one central server since there is no need for customizability.

This is the style that is being used by the Homeseer, where users can download or write their own plugins for communicating with devices or creating autonomous behavior logic. It is possible to design an entire application in the Microkernel style, however this becomes more difficult in a distributed context as the only components in the vocabulary available are the Microkernel and the Plugins. In the context of the IoT, which is inherently distributed, it simply becomes a choice of whether or not to provide a generic or customizable piece of software. The trade-off is that designing an architecture in the Microkernel style is considered to be harder because it requires a lot of up front thinking and an extensible interface for future plugins[Ric15]. This customizability does not necessarily mean an increase in performance, it just shifts the responsibility to the

user. If the user only installs the plugins they need than the system will be wasting less resources.

Using a microkernel increases evolvability as all plugins and the Microkernel are decoupled which can also allow the software to be updated while it is still running and customizability as all users can choose which plugins to install. The disadvantages is that a Microkernel is much harder to design and is less efficient than a monolithic component since they are inherently decoupled, which as we have seen has the side-effect of increasing latency.

However, I do not find it possible to compare this style to others such as the Client-Server and Peer-to-Peer styles. The description of this style mentions that it is a product based style and is difficult to use in a network based setting to describe an entire system[Ric15]. It is always possible to use the Microkernel inside another style however.

As we will see moving forward, this is the only class where having a Microkernel style is really compatible, as each location of the system might have different types of sensors and actuators and thus will need a different subset of plugins located in the Hub. For the next class we will see that this style is not desirable as all users and devices will need to connect to a Cloud component, which should contain all logic and not a subset of the logic.

Verdict

The analysis shows that there are four candidate styles for this class, which are the Client-Server, Event-Based, Service-Oriented and REST architectural styles. The topology for SOA can be mapped to a Client-Server style. The topology for the REST style is identical to the Client-Server style. We have already seen that the Event-Based style can be used, however for communicating to specific devices it will have to be used in an unnatural way to stay within the constraints. For this reason we will also eliminate it as a contender. The best styles will be mentioned for each quality attributes.

Interoperability. The Service-Oriented style contains the discovery and orchestration tactic, while the REST style contains the uniform interface tactic. In practice, the two are usually combined where the service-oriented architecture is the main style and the communications occur with the uniform REST interface.

Evolvability. REST has the highest evolvability as components can change however they want internally as long as the interface remains the same. Services in Service-Oriented Architecture can always choose to change their interface as long as they also change the description in the service registry. However this means that all components using this service will have to adjust to these changes.

Performance. The Service-Oriented style has the most potential to achieve high performance by allowing mobile devices to interact directly with sensors and actuators.

This will decrease latency, however comes at the cost of more energy usage on mobile devices which can be a rough trade-off in a commercial setting. However in situations where there is a dedicated mobile device for the system and low-latency is a priority, this trade-off is welcomed.

Availability. All styles have a single point of failure, the important decision here is not which of the three styles to choose but rather the centralization or decentralization of application logic. The more logic is moved the edge of the network, the more overall availability will increase.

Security. The simple Client-Server model is in theory the most secure because it is not easy to acquire information on how to communicate within the system. For Service-Oriented this information is located in the registry service and for REST the attacker already knows how the interface looks like.

Privacy. Privacy remains the same between the three styles. The important decision is again centralization versus decentralization.

Since the main goal of this class is to provide interoperability, it is no surprise to find that Service-Oriented and REST are the main choices.

Different sensors require alternative interaction patterns. As we have seen, styles often constrain the direction of communication and dictates which components can initiate communication. However, sensors might need to use different ways of communicating depending on their type. A motion detector for example can save a lot of bandwidth and energy by only transmitting messages when there is a movement detected instead of constantly sending an update. Another type of sensor that is used in a real-time system might benefit more from sending a constant stream of data such as in a Pipe-and-Filter or periodic messages in a Client-Server style.

Centralization vs. Decentralization. After eliminating styles because their constraints do not match the goal of this class and would have negative effects on the quality attributes, it becomes hard to see many differences between the remaining styles. However, one decision that remains a constant factor for this class regardless of the style chosen is whether to put the application logic in a Cloud component or in the hub. This is best illustrated with the examples of the Homeseer and SmartThings solutions.

SmartThings has a "Cloud first" philosophy, which means that all logic is located in the Cloud. The Cloud is also the first to know when a user or a device makes a request. This has the advantage that the logic can be updated in one location and that all users have access to the same logic. The hub is simply used as a gateway to the Cloud and thus can be considered to be cheaper to produce, which is also an advantage. The disadvantage is that performance is reduced as requests have to go over the network and be answered

by the Cloud, scalability becomes an issues as all users and devices contend for the same resources, privacy also becomes a concern as data has to go through the Cloud instead of staying local, if there is no internet connection than the hub cannot provide any functionality and also the Cloud component becomes a single point of failure for the entire system.

In contrast, the Homeseer puts the logic in the hub, which decreases latency as described in [BA12]. Privacy is ensured by allowing data only to flow locally. The price to pay is evolvability, as updates need to be pushed to all hubs. However, they partially solve this problem by using the Microkernel style. The users themselves are responsible for downloading, purchasing or developing their own plugins. In this way it becomes a more “Do it yourself” product. The developers will only have to push basic updates to the Microkernel. In this decentralized architecture, there is no single point of failure where all users of Homeseer lose functionality. Instead, failure is isolated to its respective location.

Best style for this class. For this class the Service-Oriented style and REST are the best styles. Combining the two to allow direct communications with devices hosting their functionalities as web services such as Web of Things vision is a good way to provide interoperability, which is the main goal of this class.

The difference between using Service-Oriented with REST and using it with SOAP, which is the Web Service standard, is described in [Bas]. SOAP is characterized as being complete while REST offers simplicity. This is because REST interfaces only have four commands and there is no notion of type, so the application themselves are in charge of handling semantics of interaction. Using SOAP and the WS* standards for Service-Oriented style will provide a registry service containing machine readable descriptions of services, which makes sure that they interact with each other in the correct way.

5.3.1.2 Class B. Location Free Heterogeneous Devices

This class of solution has the same goals as the Class A, except no assumptions can be made about the location of the devices. This means that the designers cannot move logic to the edge of the network. The styles proposed for the previous class all apply to this class. Only Service-Oriented and REST styles are further analysed for this class. The Microkernel style will not work because there is no longer room for customizability as all users will have to use the same logic located in a central area. The layering will remain the same as the previous class, so this will also be omitted. The Peer-to-Peer, Pipe-and-Filter and Publish-Subscribe style are all still not compatible as their constraints do not fit the goal of this class.

An important difference to note is that this class has no way of knowing the identity of devices owned by the user. In the previous class this was possible as the hub could

locate the devices in its proximity. This means that this class will have to communicate with an entry point in the third party solutions, which can have any type of architecture. What this also means is that all of the accounts of the third party solutions of the user will have to be linked up to the account for this system, as there is no other way to make the connection between user owned devices.

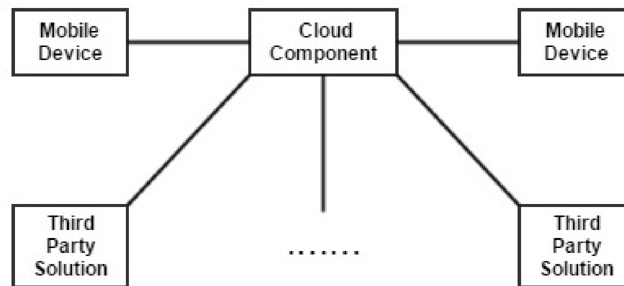

FIGURE 5.9: Location Free Heterogeneous Devices

Figure 5.9 gives an illustration of this class. The Cloud component will contain the software for providing interoperability, syncing accounts and autonomous behavior logic.

Functional Requirements:

- The user can view measured data from sensors and control actuators of any third party solution that they have registered to this system.
- The user can choose from a set of smart scenarios in order to create an autonomous smart environment.
- The user can add and remove third party solution accounts to the system.

Quality Attribute Requirements:

- **Interoperability:** This is a primary requirement for this class, meaning that we should consider styles that provide interactivity and possibly inherent interoperability between components.
- **Evolvability:**
  - S1: Change to UI application
  - S2: Change/Addition autonomous behavior logic
  - S3: New third party solution
  - S4: New device in a third party solution
  - S4: Change in data format of third party solution

- **Performance:**. The latency in this case is dependent on the third party solutions. It is expected to be higher than in a solution of class A, where a Hub can be used. Scalability becomes an issue here as all communication will have to occur in a centralized Cloud platform. Tactics such as layering and replication should be used to handle this.
- **Availability:**. If the Cloud Platform is offline or unreachable, then there is no functionality available. This is in contrast to the previous class, where the hub could still carry on if the application logic is stored there.
- **Security:**. The main Cloud component remains an area of interest for attackers, however since it runs in a Cloud component and not a limited edge device such as a Hub, it can be argued that it has more resources to provide better security.
- **Privacy:**. There is no possibility of keeping data flow constrained to a location, it has to get sent to the Cloud to be processed. The privacy level of data depends on what kind of data is being measured by the third party solutions.

Verdict

The differences amongst styles remain the same in this context as for Class A. The styles do not gain an advantage or disadvantage from having all of the logic moved to a Cloud component except for the Microkernel style, which is not applicable in this context since there is no possibility for customizability as provided by that style. The Service-Oriented style is the best choice for this class because it allows for two different types of mapping to hardware, both of which support interoperability. New solutions simply register their service to the service registry and can be used by this class any time.

The first topology maps the orchestration service and the service registry to the Cloud component. This is shown in figure 5.10.

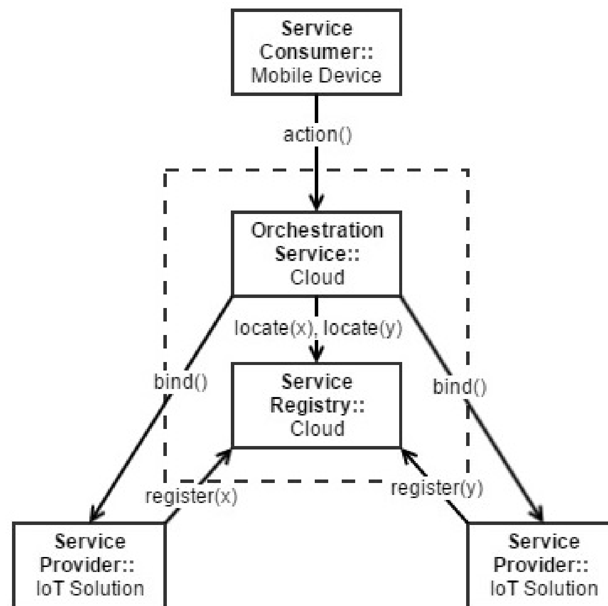

FIGURE 5.10: Service-Oriented mapping A

The advantage of this mapping is that all application logic which is located in the orchestration service is centralized and can be changed in one place. The mobile device only uses energy to show the UI. However, the Cloud component acts as an extra layer increasing latency. Furthermore, scalability is a big issue as all users must constantly communicate with the orchestration service to be able to provide functionality.

The following topology is also possible, which puts all of this logic inside the mobile device. This is illustrated in figure 5.11.

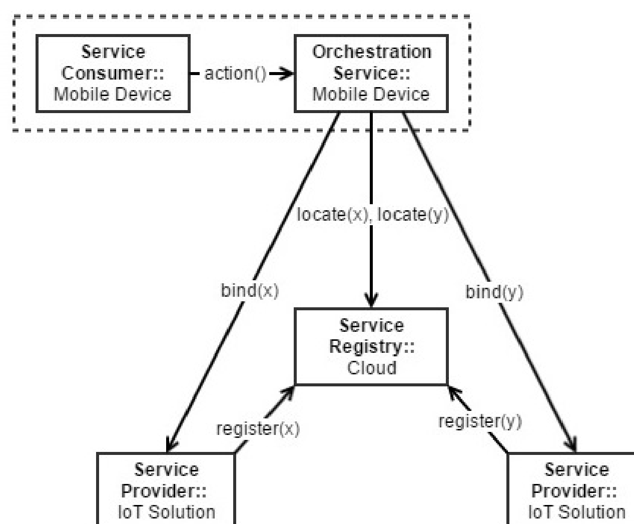

FIGURE 5.11: Service-Oriented mapping B

This has the advantage that the mobile device will be able to communicate directly to services once it has been bound to the services it requires. This decreases latency. This also increases availability of the system, as having the Cloud component go down only removes the functionality of looking up new services. Scalability also becomes less of an issue since the Cloud is only used for registration. The downside is that every update to the orchestration logic must be pushed to all mobile devices. Furthermore, the app size and energy usage increases on the mobile device. This can be a good trade-off for certain specialized environments, where performance is a priority. An example could be a smart hospital environment where all specialists and emergency responders are equipped with special purpose mobile devices that can last an entire shift.

Mediator VS. Direct interoperability. Interoperability can be achieved by either building a bridge between two systems or having the two systems speak the same language. Both classes A and B are an example of building bridges, by acting as a mediator between solutions and also the user. Such a mediator increases latency, has a potential negative effect on privacy, creates an extra entry point for attackers, creates a single point of error for communication, but also increases evolvability as it decouples the systems from each other.

5.4 Observation/ Class C. Body Sensors

The Observation category deals with solutions that measure certain attributes of the human body, where the data from one sensor is used to provide useful data. This is in contrast to the third category of IoT Solutions, where aggregation of data from multiple sensors is used. The user of the solution can be the person that is wearing the sensor, for example for fitness and activity tracking. It is also possible that the user is a health specialist that monitors patients wearing body sensors.

This category only has one class, namely the Body sensors class. One of the differences is the scale of the network, which can be mapped to the Body Area Network (BAN) mentioned by the IoT-A [BM]. It is relatively small in scale compared to the other categories. The communication might be a connection between the sensor and a mobile device using Bluetooth technology, or the sensor could also be equipped with networking capabilities and communicate with a Cloud. One other possibility is to use the mobile device as a gateway to the Cloud. Figure 5.12 illustrates this class of IoT solutions.

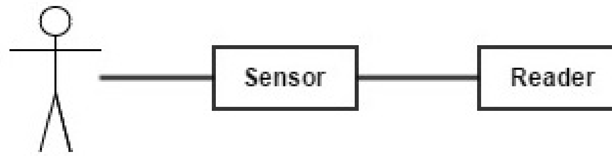

Figure 5.12: Observation Category / Body Sensors Class

The reader component in this figure contains the logic for communicating with the sensor and visualizing it in any useful way to the user. The reader could be located in a mobile device or a web-service. This decision has an impact, however since this is a comparison of styles we will abstract from this decision.

Functional Requirement. For this class we will only require one functionality, which is that the system should provide a visualization of measured data from a body sensor in a format that is useful to the user.

Quality Attribute Requirements:

- **Interoperability**. Interoperability is a secondary requirement for this class. This means that we can consider how interoperability with other systems could occur in the future given a certain style, however the other attributes are more important for the final verdict.
- **Evolvability**.
  - S1.New version of sensor (hardware)
  - S2.Update to data process and visualization logic
- **Performance**. Choosing the correct rate of sampling and communication pattern has an effect on latency and energy consumption. Scalability is not a primary issue.
- **Availability**. The sensors in this class are considered to be mobile. For safety critical systems the use of multiple wireless communication technologies should be used. The use of another device, such as a mobile phone, as a gateway to the Internet is also a possibility.
- **Security**. The sensors might not have the computational resources to perform advanced security algorithms. Making design decisions that remove unnecessary messages and computation can relieve the resources to be used for security purposes.
- **Privacy**. The data in this class is considered to be personal, since the data being measured points directly to a certain user.

Client-Server

The Client-Server style is a good contender because it is clear that the sensor is a data provider and the reader is a data consumer. However, the role of Client and Server can be mapped on any of the two, providing different topologies. Both can be used depending on the application to be built.

If the Reader component is a Web Server, than having the sensor play the role of the Client is the best idea, since the Web Server will be a non-terminating process and can have a static address. If the reader component is a mobile device or something similar in the proximity of the sensor, then the sensor could play the role of the Server which gets detected via Bluetooth. The Reader is then the component that initiates communication. The discussion so far has been about the two components identifying and communication with each other. However, the choice in topology also has an effect on some of the quality attributes. The two topologies will be referred to as sensor-as-client and sensor-as-server.

- **Interoperability**. Having the sensor play the role of the Server will allow other types of readers to communicate with the sensor easier, given that they have the exact specifications to initiate a communication and authorization. If the sensor is the Client then this would mean updating the logic on the sensor, which in some cases might be hard because the sensors might not have direct internet access.
- **Evolvability**.
  - S1. A change in the sensor might mean changes in the reader component if the sensor is in the Server role. If the sensor is the Client, then there is no need to make any changes.
  - S2. A change in the application and visualization logic will not require a change in the sensor in any of the two topologies, as the sensor will only continue to send data the only way it knows how or reply to requests in the only way it knows how. The application logic and visualization can change as long as the interface to the sensor remains the same.
- **Performance**. The Sensor-as-Client can save energy if it only sends a message when a certain conditions occur. However, if we only need the data occasionally, then having the Sensor-as-Server topology saves energy by having the sensor only react to requests. Both can be used to provide real-time analysis by either having the sensor periodically send its data or having the reader periodically ask for data. This symmetry is only possible because of the scale of this class, however in other classes we will see the difference this choice can make.
- **Availability**. The topologies do not make any difference with respect to availability.

- **Security**. The Sensor-as-Client can be seen as being more secure as the identity of the sensor is not known by the reader until a request is done. Also, in this topology the sensor does not accept any requests, it only initiates them which is also more secure.
- **Privacy**. The topologies do not make any difference with respect to privacy, except for confidentiality which is a part of the security discussion.

The two topologies both have their advantages and disadvantages. The Sensor as a Client should be used when new versions of sensors is likely to appear, if the messaging pattern is event or condition based and to provide more inherent security. The Sensor as a Server should be used when in anticipation of interoperability with another system and if the data is only need for specific occasions (as part of a series of computations).

Peer-to-Peer

As mentioned in the previous section, there is a clear difference between the provider of data and the consumer of the data. For this reason using a peer-to-peer style is not a good fit, as it would mean equipping the sensor with the logic contained in the reader and also the ability to ask the reader component for data, which is communication possibility that is not needed. This results in waste of memory with no gain just to stay within the constraints of the peer-to-peer style.

Pipes-and-Filters

This style can be used in the scenario where the sensor needs to send a constant stream of data, which is transformed along the way. The advantage of using this style in contrast to the Sensor-as-Client topology are the following:

- **Interoperability**: This further decreases interoperability possibility as any component that wants to communicate to this style has to understand the output from the final filter.
- **Evolvability**: The filters are independent and thus can be replaced and changed without affecting the rest of the system, which is inherent in the pipe-and-filter style.
- **Performance**: This depends on the number of filters, as there is a trade-off between performance and evolvability which is dictated by the number of filters. If there are too many filters, then this brings overhead because of the communication via pipes. However if we just view the sensor as one filter than we do not gain any evolvability over the Client-Server style.
- **Availability**: The data flow is one way, so the sensor does not get any confirmation that the messages are being received.

- **Security**: There is no effect on this attribute.
- **Privacy**: There is no effect on this attribute.

Event-Based

The event-based style can be used to send information to one or multiple recipients that are interested in a single event. In this case, the event could be the condition that the measured data exceeds a certain threshold. The Sensor would need to host an Event-Bus where readers can register to. However, this same messaging pattern can also be replicated using the Client-Server style. The only difference is that the Event-Based style provides a decoupling between the event-source and the event-processors, since the source does not know the identity of its listeners. This is good when there are multiple different types of event-processors and receiving a confirmation that the message has been received is not important.

However, in some scenarios it might be important to know for sure that an event has arrived. The difference is presented in figure 5.13.

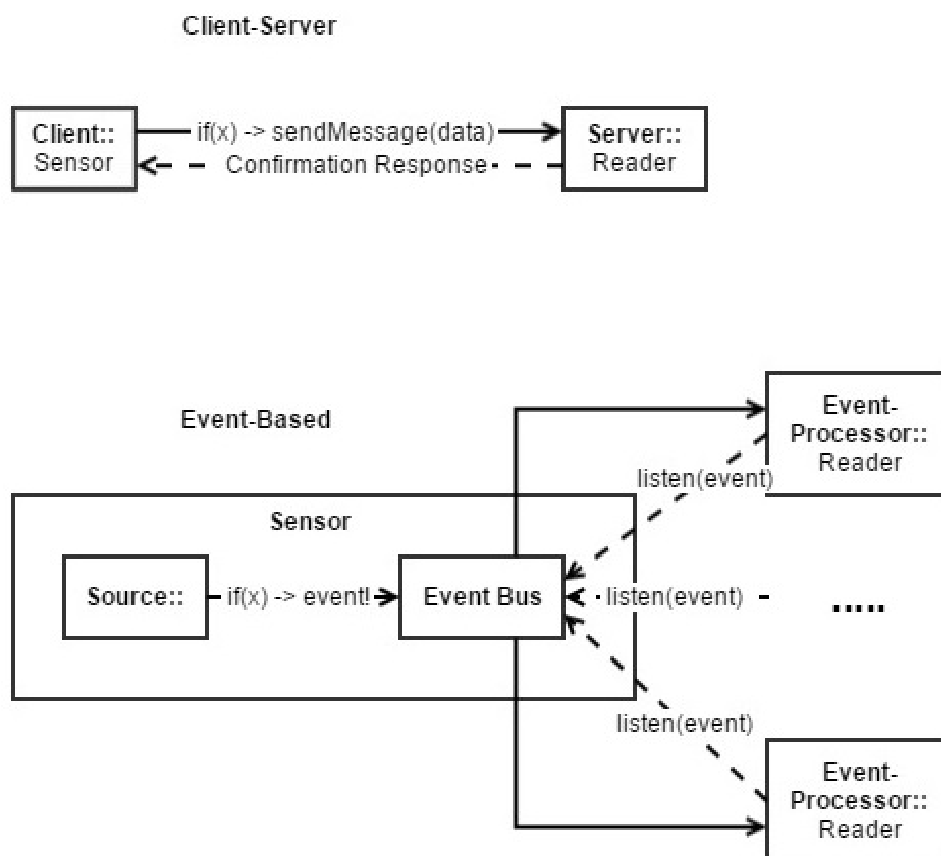

FIGURE 5.13: Comparison Client-Server and Event-Based styles for Body Sensors

Unless the sensor needs to update multiple readers, then the event-based style does not bring any advantages. In the examples we have seen so far, the sensor always communications with one application at a time. This would make having an event-bus with a registry of all listeners useless in this context and it also adds latency. Furthermore, the sensor might want to have confirmation that the message has been sent if it is a safety critical, such as a "patient is having a heart attack" event.

Publish-Subscribe

The publish-subscribe further decouples the producers and the consumers of data, which we have already seen is not desirable for this class.

Service-Oriented

The Service-Oriented style brings too many unnecessary components and constraints. Interoperability is not a primary requirement and we are not interested in dynamic lookup of services. The service registry would need to be hosted somewhere, and it would contain many descriptions of the same type of services since we only have one type of sensor per solution. Furthermore, some solutions do not even use the Internet at all.

REST

The uniform interface constraint could be used in order to facilitate communication with the device, however the REST uniform constraint also requires that services be identifiable by an URI, meaning that it can be addressed via the Web. However for this class of sensors it would not be the best solution as they are not likely to be connected to the Internet directly or have a single stable gateway.

Layered

Using the layered style will result in a minimum of three layers, a presentation layer, application layer and a sensor logic layer. The first two are located in the reader while the last is located in the sensor. If the sensor layer is put on top, then the control flow would be the same as the Sensor-as-Client topology. However, putting the presentation layer on top would mean that the user would constantly have to ask for the data, where the business logic queries the sensor, processes the data and sends it to the UI which is the same as the Sensor-as-Server topology.

Microkernel

There is no need for customizability in this class, so the disadvantage of added complexity is not a worthy trade-off.

Verdict

The analysis shows that in overall, the Client-Server style is the best choice for this class. Choosing which components take the role of the Client and the Server allows

for different types of messaging patterns that can save energy and bandwidth usage. Interoperability with these devices should occur with close-ranged wireless technologies such as Bluetooth and not by having a URI and webservice run on the device, since this costs more energy and requires a steady connection, not to mention that the IP address will constantly change since the sensor is mobile.

5.4.1 Aggregation

This category of IoT solution has the goal of collecting data from sensors, aggregating it and visualizing it for the user. These are providers of data, creating value with only sensors and not using any actuators. The sensors in one solutions are all from the same vendor. This category contains four classes, where the first two are concerned with providing information derived from all sensors in the system as a whole and the last two are concerned with the scenario that certain data belongs to specific data groups that own the sensors.

The difference is that the collective data group must gather all of the data in one point, while the second group can either provide one central point or can gather data in a more decentralized way by allowing user groups to aggregate data themselves.

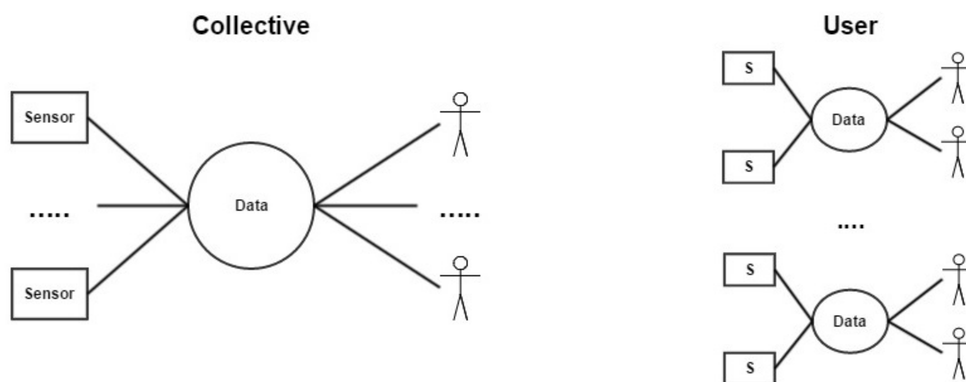

FIGURE 5.14: Collective vs. User data scenario for aggregation category

Figure 5.14 shows the difference between the collective and user data scenario. The collective data scenario generally does not have any privacy requirements, as the data is open to all users. For the user-owned data scenario, the data originating from the sensors that are owned by a user group needs to be aggregated and visualized only for their respective owners. This means that in those scenarios privacy does play a role.

The majority of the IoT solutions we have in our dataset fall into this category. This might be the case because sensing data is easier and less invasive than changing something in the world with an actuator.

5.4.1.1 Class D. Active User Collective Data Solutions

This class contains solutions where the user "actively participates" by owning one or multiple sensors that contribute to the central data storage. This data can then be used for the benefit of all users that actively participate. Examples are the WeatherCloud, where the user equips their car with multiple sensors in order to measure road conditions, or the TZOA environment sensor which measures the air around the user and updates this to a central database.

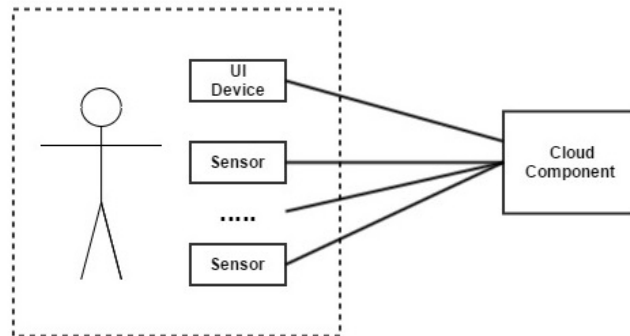

FIGURE 5.15: Active User Collective Data Solutions

Figure 5.15 shows the scenario of this class for one user. It is assumed that there must be a central Cloud component in this class because the data must be aggregated somewhere central. The dashed box indicates ownership, meaning that the user owns one ore more sensors that provide data and at least one device which is able to visualize the data.

Functional Requirement:

- User can visualize aggregated data collected by all sensors in the system.

Quality Attribute Requirements:

- **Interoperability**: Interoperability is a secondary requirement for this class. This means that we can consider how interoperability with other systems could occur in the future given a certain style, however the other attributes are more important for the final verdict.
- **Evolvability**:
  - S1. New version of sensors (hardware)
  - S2. Update to aggregation logic
  - S3. Update to user interface

  - S4. Addition of new sensor type to the system (new type of data)

- **Performance**: Scalability is an inherent issue in this class because all of the data from all sensors need to be gathered into one place for analysis.
- **Availability**: The sensors might be mobile so the connection to them may be limited. The central aggregation component is a single point of failure.
- **Security**: In this case data integrity is the most important security goal that needs to be ensured, as confidentiality is not a primary issue since the data is open. Sending data in the clear instead of encrypted can increase performance.
- **Privacy**: The data is considered to be open and privacy is not a primary issue. However, since each user still actively participates by owning the sensors, metadata could be used to infer sensitive data and behavior patterns of the user.

Client-Server

In the previous two categories we considered two topologies of the Client-Server style. For the first category we took advantage of the proximity of the devices to the hub and also that devices belong to a user. In the Body Sensors class we could provide two topologies because the scale of the system was very small. However in this scenario we potentially have many sensors and one central server component (which could be replicated and located behind a load balancer for scalability). The Server cannot know the identities of all sensors and if the sensors are mobile they will have dynamic IP addresses. From this we can derive that the sensors must know the identity of the server and not the other way around.

The Server component would contain the application logic and the data storage. The sensors would write to the data storage and the user would read from it.

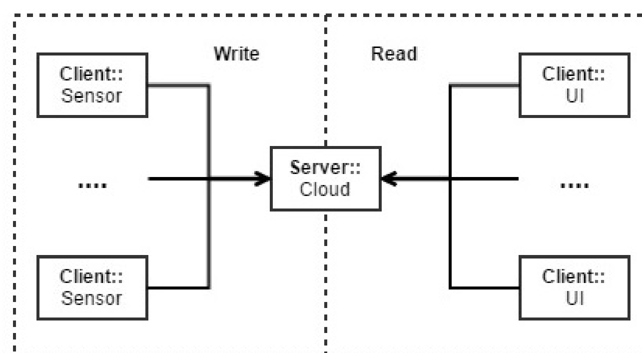

FIGURE 5.16: Client-Server Topology

Figure 5.16 shows the topology, which is separated into a write and read section. The communication between the sensor and Cloud can be seen as inefficient for this class,

especially for real-time systems, since the Cloud has to send a respond for every request containing data. It might be more efficient to use a style such as the Pipe-and-Filter style for the write part, since data flows in one way. This will be discussed later.

Note that we have not specified where the user interface logic is. It could be a native mobile, console or web application. There are IoT solutions in this data set that have all three. Having a web application increases evolvability, however there will be more workload on the central component. Of course we can assume that the central component will be split up to address the sensors and users separately, with the central point being the data storage.

- **Interoperability**: Any interoperability with this system would have to occur with the Cloud, since the sensors and UI device do not accept requests.
- **Evolvability**:
  - S1. New sensors should not affect any other components since the server does not have knowledge of the clients.
  - S2. The aggregation logic is located in one central Cloud component, which makes it easier to evolve the aggregation logic.
  - S3. It depends if there is a native/desktop application or a web application. However, in both scenarios we can assume that the UI logic is separated from the aggregation logic.
  - S4. A new type of sensor would mean updating the data model to be able to store this data. The aggregation logic might have to change to include this new data in its calculations and the UI may also want to show this data separately. In short, the entire system would need to be updated.
- **Performance**: Extra performance tactics need to be employed in order to decrease the latency and handle scalability. Prioritizing the user reads could increase the user perceived performance. Replicating the data and functional components can increase the capacity. This style keeps the edges of the network as thin as possible, which allows for sensors to be made cheaper.
- **Availability**: The central component is a single point of failure. Using replication can provide extra availability.
- **Security**: Keeping the sensors thin would allow for the free resources to be used to ensure data integrity, such as using a checksum of the hash values. The trade-off is decreased performance, however if security is the priority then this might be worth it.
- **Privacy**: Data passes through the central component, which could be considered bad for privacy. However we have decided that this is not a primary requirement for this class.

Peer-to-Peer

We have seen that there is a clear difference between the producer and the consumer of data. However, if for example a mobile device is used as a sensor and the UI in the system, then we would be within the constraints of the Peer-to-Peer style. However, since we need to perform data aggregation of all sensors in the system, there needs to be a central data storage which is not allowed in a Peer-to-Peer style.

Pipes-and-Filters

The data in this class flows from the sensors to the user, however it would not be very practical to model the entire system in the Pipe-and-Filter style, as the central converging point would need to know the identity of all user devices. This assumption cannot be made as users constantly join or leave the system. However, the Pipe-and-Filter style could be used for the sensor part of the system.

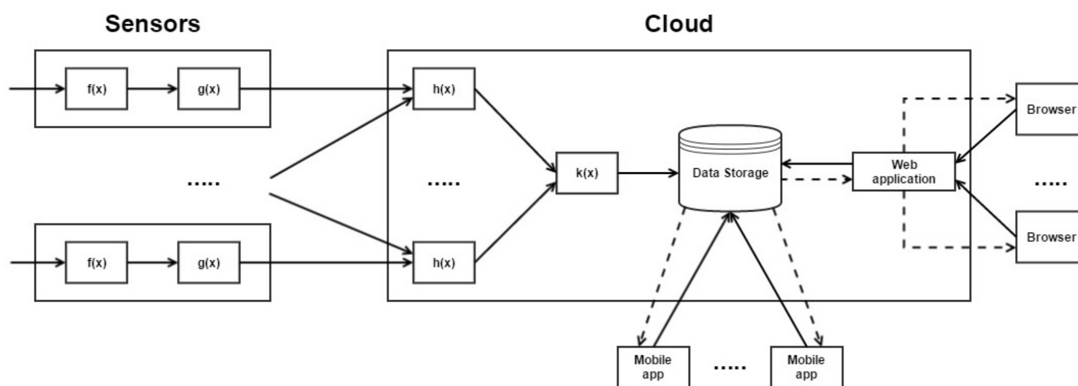

FIGURE 5.17: Client-Server and Pipes-and-Filters combination

Figure 5.17 shows how this system would look like. The sensors would be configured from the start to stream data to the Cloud Server, which has a fixed address. If partitioned correctly, the most computational expensive function could have its own Filter component and run in parallel, which is a possibility with Pipes and Filters. The image also illustrates the scenario where there is a native mobile and a web application, however this can vary within this class.

The difference between using a full Client-Server style and this combination is the trade-off between performance and availability. The combination has higher performance as data is constantly send one way and does not have to wait for a confirmation of arrival. On the other hand, there is no reassurance that the data reaches all the way to the Cloud. If the Cloud is simply computing an average and the sensors send an update every half a second, then losing some of the messages along the way might not be such a problem. However, if it a safety critical system, then it might be better to use the Client-Server style where the sensor is sure that its data arrives.

Event-Based

If the goal of the system is to alert users when a certain event happens that can be inferred from the aggregated data, than an event-based style can be used by letting users register for certain events. An example could be if the average of all sensors surpasses a certain threshold. The Cloud will only need to send a message to the users when this event happens instead of having the user constantly poll for changes. This increases bandwidth efficiency and also decreases energy consumption.

However, the trade-off is that there is no confirmation that the user receives the events. When it is important to make sure that the message has been reached, it might be better to use a request/response connector instead of an event-based.

Publish-Subscribe

In the same way as the event-based style could be used in this class, the publish-subscribe class could be used to allow the database to act as a publisher that posts updates based on certain topics. The user can subscribe to certain types of data. An example is a weather system, where the user only wants to subscribe to weather changes in their location. The effects are the same as the Event-Based style, except that the Publish-Subscribe style is considered to have inherently more overhead since the Event-Service needs to categorize the content and provide information retrieval logic, which increases latency compared to the event-based style.

The advantage of being able to have multiple heterogeneous publishers inherent in this style is not used here.

Service-Oriented There is no incentive to use this style since interoperability and dynamic lookup of services is not a requirement. It would bring unnecessary components and complexity for a system that only needs to provide data to users.

REST We have already seen that the Client-Server topology has one central server component. It could be possible to allow this component to have a REST interface in order to be used in other IoT solutions. This will increase interoperability but reduce performance as information is transferred in a standardized form. This could be mitigated by allowing the sensors and ui applications belonging to the system to still communicate in a custom protocol and only using REST for external communication. However this might also come with some trade-offs, such as having the same communication logic in two places which decreases evolvability.

Layered

I could not find a layering to model this class of system which adheres to the constraints of the layered style. This is primarily because there are two components that initiate communication and one central convergence point. In non-IoT systems, the user is usually the initiating party. This supports the claim that not every system can be modeled in a layered way [GS94].

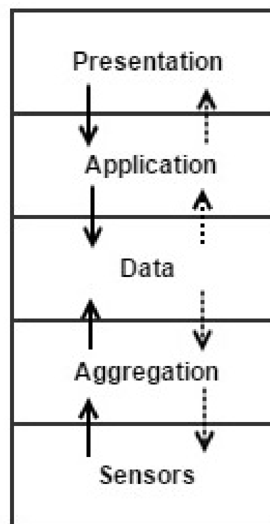

FIGURE 5.18: Layered Topology

Figure 5.18 shows an example of a layering for this class. However, this is not a valid architecture within the constraints of the layered style, since the bottom layer may not have knowledge of the layers below it. In this figure the layer sensor is using the aggregation layer and the aggregation layer is using the data layer. This is an important discovery as it can point out that the standard layered style might not be usable in the IoT since we have two points in the architecture that initiates communication, since it is not possible for the central component to know the identity of all users and devices.

Microkernel

There is no need for customization, the inherent complexity of this style would not provide any needed benefits.

Verdict

The Analysis shows that there are a few combinations possible between the Client-Server, Pipe-and-Filter and Event-Based for this class to achieve the best quality attribute effects. A REST interface can be added, converting the Client-Server into a REST style, to increase interoperability with systems in the future.

Since the goal of this class is to aggregate data from all sensors, it becomes inevitable to have a central component where all data converges. The best styles for each quality attribute will now be mentioned.

- **Interoperability**: The REST style could provide a uniform interface, which would allow other systems to use the data gathered by this class of solutions.
- **Evolvability**: For the four change scenarios analyzed, the Client-Server, Pipes-and-Filter and Event-Based style do not differ in their ability to reduce the impact

of change. However this does not mean that they all three offer the same level of evolvability, simply that in this context and proposed topologies they are the same with respect to evolvability. A Pipes-and-Filters style has independent components, however we could also model the internal architecture of the clients and servers in this way. The Event-Based allows for heterogeneous devices to listen to the same events and thus decoupling them from each other using the Event Bus, however in this class we only have one type of device that listens to the event.

- **Performance**: Using the Pipes-and-Filters style for the sensor-to-Cloud communication can increase performance by removing the need for a confirmation response after every message sent. The Event-Based style can improve efficiency by only sending updates when certain events happen, if the goals of the system can be modeled in this way. Scalability will have to be provided by using tactics such as replication and layering.
- **Availability**: The Client-Server style is best for availability as it requires a response to be sent after every request.
- **Security**: The Event-Based defines an interface which the users must adhere to, which only contains the events that can be registered to. This might be considered better for security as it gives an attacker less of an opportunity to inject malicious input.
- **Privacy**: The different styles do not make a difference for privacy as the data flows through the same components.

Information Providers. This category of solutions is the first pattern I have recognized during this thesis. I noticed that there are many solutions that can be seen as information providers, i.e. they collect data from multiple sensors and provide an analysis on the data and visualize it in a way that is useful to the users of the system. In this way they create value without using actuators.

Two types of actors in the IoT. The analysis of the layered style shows an interesting side-effect of having sensors and actuators as part of the system. They can be considered actors in the system, which might simply send data periodically or when an event happens. They might also act in more autonomous ways based on complex algorithms, meaning that it can be hard to predict when they will initiate communication over the network, which is the same in most cases for human users. This makes it hard to model the system in a layered way unless you consider human users using a user interface the same as sensor and actuators. Note that this is only the case when it cannot be assumed that the identity and network addresses of all sensors and users are known.

5.4.1.2 Class E. Passive User Collective Data Solutions

This class looks very similar to the previous class, however the difference is that the number of sensors do not depend on the number of users. The effect of this is that the number of sensors is predictable and requires a less scalable system. However, in my analysis I did not find that this was a big enough difference to alter the set of best choices of architectural styles as all other requirements remain the same.

It could be argued that since the sensors are all known, we could use another client-server topology where the sensors can have the Server role. However we have no guarantee that the sensors are stationary and thus could have dynamic addresses and possibly unreliable connectivity.

Because my analysis shows similar results for this class, I will deviate from the format and move on to the next class. For the purpose of choosing software architectural styles, the classes D and E can be considered equivalent. This does not mean that they are the same type of system, as they might have different business models and varying architecture design choices moving forward.

5.4.1.3 Class F. Stationary Homogeneous Sensors

This data contains solutions where only data from sensors belonging to the same user group should be aggregated together. This data should only be viewed by the owners of this data. The sensors are stationary in this class, meaning they are assumed to have a relatively stable connection to the Internet compared to mobile sensors.

The main difference I found in my analysis of this class is that the user could run their own central server which runs the back-end software instead of having a central server for the entire system. It becomes less of a turnkey solution however, since the user will have to install and maintain a server.

The effect of choosing the decentralized version, having the user run their own server, is an increase in privacy since personal data is not shared on a Cloud server with other users, increased performance and less of a scalability requirement and overall system availability as the effect of a server going down only affects that user (group). The trade-off is that each user needs to update their server when new versions come out, there is a cost to owning a server and maintaining it.

For a solution that expects their target audience to be retailers that want to place a few sensors in their store to get insight on customer behavior such as the Scanalytics Floor Sensors solution, then the centralized turnkey solution might be the best choice. However, for a solution where there are many more sensors per user, the data is more sensitive and the user groups are companies that could likely afford or already have own data centers, than the decentralized “Do it yourself” architecture is the best choice.

The choice for styles remain the same as the previous two classes, where the Client-Server style is foundation with Pipes-and-Filters and Event-Based as possible inclusions depending on the scenario.

It remains a fact that a central component is needed in order to collect all of the data in one place to be useful for the user. Not using a central component would mean that an edge device, sensors or mobile devices, would need to host this data. This would not be a good for energy consumptions and usage of memory of edge devices, which is limited to begin with.

5.4.1.4 Class G. Mobile Homogeneous Sensors

This class is identical to the previous class except that the sensors are considered to be mobile. The effect of this is that the connectivity of the sensors to the Internet is considered to be less reliable. It is also more probable that the sensors use batteries, however this is not always the case. The mapping of styles to this class results in the same choices for the last three classes, however the less reliable connectivity needs to be addressed with design decisions moving forward. An example could be allowing the sensors to temporarily store data locally until there is a stable connection. This tactic is used by the Farmobile and DAQRI Smart Helmet solutions. The trade-off is that the sensors need to have data storage capabilities and be autonomous in the sense that they can detect when there is no connection and switch over to the local storage. This leads to higher cost of production of the sensors and makes it a more interesting target for theft. It also consumes more energy than the simple stationary sensor that just pushes its data. The battery life of the DAQRI smart helmet is estimated to a maximum of 10 hours, which is the cost of having such a smart sensor.

Another tactic that could be used is equipping the sensor with long range mobile telecommunications technology such as 4G, in case there is a scenario where there are no other access points to the Internet. This tactic is also employed by Farmobile.

A third way to do this is to make use of a mobile device as a relay of data. The Truvolo Car solution uses this tactic in order to send a steady stream of data to the Cloud regarding the diagnostic information of a car. However this is assuming that there is a user with a mobile device in the area and that there are not too many sensors per user. This is not the case for all solutions in this class.

We have seen that the same styles have come up on top for the four classes belonging to the aggregation category. This is because the goal is the collection of data, which is best handled by having a central point of convergence. Another reason is because the choice of which components are consumers and producers of data and functionality are already made for this class. It is clear that the sensors are producers of data and the

users are consumers. However, the following category introduces actuators which will bring more opportunities for multiple styles.

5.4.2 Automation

This category contains IoT solutions that have sensors and actuators, meaning that the read from and write to the physical world. The addition of actuators brings security and safety concerns, as an attacker could misuse these to cause material damage or physical harm to people. The category is split up into two classes, where the classifying attribute is whether or not there is any autonomous behavior in the system. In this context, autonomous behavior is defined as the system sending commands to the actuator without the human user explicitly issuing the command. In this way, the user receives benefits from the system without the effort of having to use it directly. The second class in this category contains solutions where the actuators are driven by explicit user commands, such as “turn the lights on” or “unlock car now”.

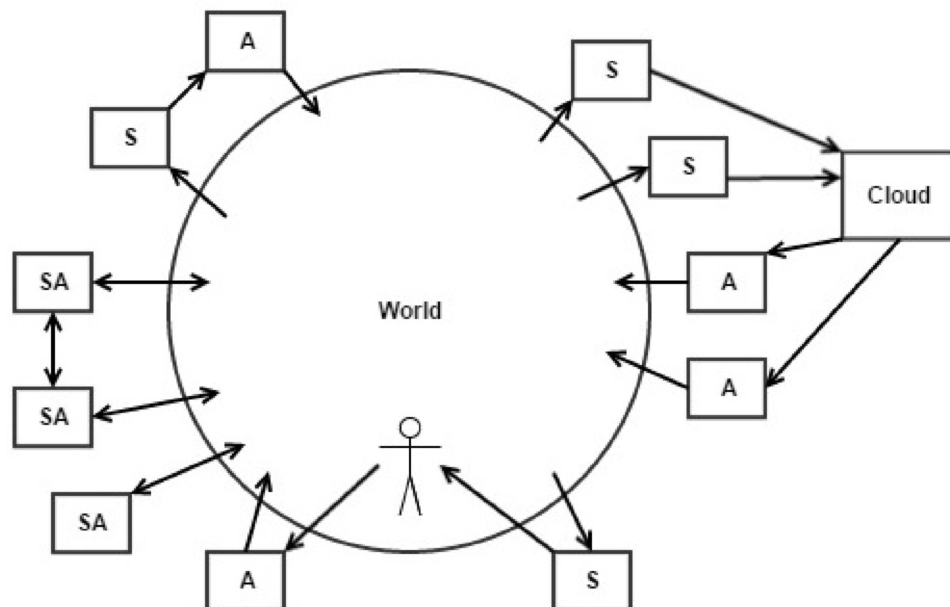

FIGURE 5.19: Automation

Figure 5.19 shows some of the different ways these solutions could be designed. The term used for these of systems are Wireless Sensor and Actuator Networks (WSAN). The boxes with S are sensors, A are actuators and SA are components that have both sensor and actuator capabilities. As we can see in the image, there can be a direct communication between a sensor and an actuator, where the actuator reacts to the data measured by the sensor. There can also be a central component between the sensor and the actuators that decide when and which actuators to send commands to. This can

be based on predefined logic by the developers of the system or user created/selected scenarios.

The system could also be dependent on a user, where a user is updated with sensor information and must decide for themselves when they want to issue commands to an actuator. And finally, there can be some solutions where the sensors and actuators are considered as one and can communicate with each other. This last example is inspired by the peer-to-peer style, we will discuss later how and why this style works.

5.4.2.1 Class H. Smart Systems

The smart systems class contains solutions that primarily rely on the system itself to decide how to use actuator depending on sensor or other data. Solutions from classes C to G can be transformed to this class if an actuator is added that reacts to the data gathered. There is a human user in this scenario, however they have a secondary role in this class of system. They can in some cases choose scenarios that actuators must react to. In most of these systems there will also be a way for the human user to override the decisions made by the system, or if the decision making logic becomes unavailable. An example is the NEST thermostat, which still allows the users to manually change the desired temperature of the room. For this class we will focus primarily on the autonomous behavior part of such systems.

Functional Requirements.

- The system controls actuators without the help of human users.
- A user can choose from a set of actions to occur given a certain condition(optional).

Note that this class is very broad and there are not many examples of such systems in the dataset of solutions used. However, this gives room for more choices of styles.

- **Interoperability**: Interoperability is a secondary requirement for this class. This means that we can consider how interoperability with other systems could occur in the future given a certain style, however the other attributes are more important for the final verdict.
- **Evolvability**:
  - S1. Changes to autonomous behavior logic
  - S2. New Type of Sensor (New type of Data)
  - S3. Addition of new actuator scenarios (Optional)

- **Performance**: This class could contain solutions of any scale, might make use of data aggregation and could have sensors and actuators connected to power outlets or that use batteries. For this reason it is hard to say up front what latency, scalability and resource efficiency mean in this context.
- **Availability**: For certain actuators it is critical that they be available even if they cannot be reached digitally. An example is the NEST thermostat, which still allows a user to change the room temperature from within the room even if there is no connection to the server.
- **Security**: The actuators bring an extra concern to security as they can be misused by attackers to cause harm or material damage.
- **Privacy**: Privacy depends on the data gathered by the sensors, for this class we have made no assumptions about this. There could be solutions in this class where actuators only react to a motion detector, in which case privacy is less of a concern than when personal data is measured and collected.

Client-Server

The most simple form of a solution in this class is a one-to-one connection between a sensor and an actuator. The sensor sends a message to the actuator to perform an action. This is a reactive system, where data is used and not saved. Three different messaging patterns in this style is illustrated in figure 5.20 for this first type of system.

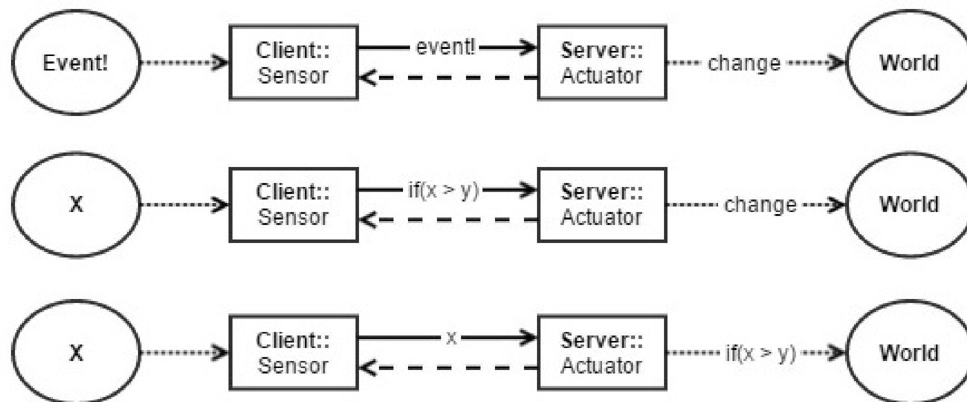

FIGURE 5.20: Client-Server one-to-one Topology

In the first pattern, the Client reacts to an event and sends this to the actuator, which proceeds to perform a change immediately to the world. An example could be an automatic door, which opens when a person stands in front of it. The client reacts to an event, however this does not mean that it is using the event-based style. It simply sends a request to an actuator, which it knows the identity of, when it detects an event in the real world.

In the second and the third messaging pattern, the sensor is measuring the attribute x of an object continuously. The client can either continue measuring until a certain event happens, such as the value of x surpassing a threshold y. However, the third option has the sensor immediately relaying data to the actuator, which must know the threshold y and only make a change to the world when this threshold is surpassed.

The first choice is good for sporadic event arrivals, which can happen at any time and cannot be predicted. The second pattern is good for systems that have sensors which periodically measure some attribute in the real world and where network efficiency is important, as it only sends a message when an event has occurred. The last option is good for when the sensor needs to be kept as simple as possible and the event condition is more complicated. This pushes this knowledge to the actuator.

This was an analysis of one-to-one communication between a sensor and an actuator using the Client-Server style. For one sensor to many actuators communication, it depends on the number of actuators which style and topology to choose. If there are only a few actuators, then the sensor can take the role of the client and know the identity all of the actuators and act in the same ways as in figure 5.20. The sensor can also act as a server and have the actuators constantly ask for the value measured, however this might result in unnecessary communication. However, for larger numbers of actuators it might be better to host a central component which handles the communication in a request-response or event-based way since the sensor might not have resources to handle all requests. This adds an intermediary in the communication and thus decreases performance as a trade-off.

If such a server is used in between the sensor and actuators, then the results of the analysis done for the aggregation category also apply here. The actuator will take the role of UI as data consumer. The same combination of Client-Server, Event-Based and Pipe-and-Filter should be used for the same reasons as previously discussed. The optional functional requirement of having a user be able to choose scenarios for the system to carry out can be added by allowing a UI to communicate to the central server. The complete topology in a Client-Server style for many sensors to many actuator communications will look like figure 5.21.

We have already discussed that the sensor to Cloud communication can also be designed in the Pipes-and-Filters style to increase efficiency at the cost of reliability. The same trade-off can be achieved for the Cloud to actuators communication by using the event-based style. The quality attribute effects of this style for this class will now be mentioned.

- **Interoperability**: This style doesn't provide any interoperability tactics, meaning it is not supported. Any system that would want to operate with the sensors, actuators or cloud server would need the exact interface specifications and authorization to do so.

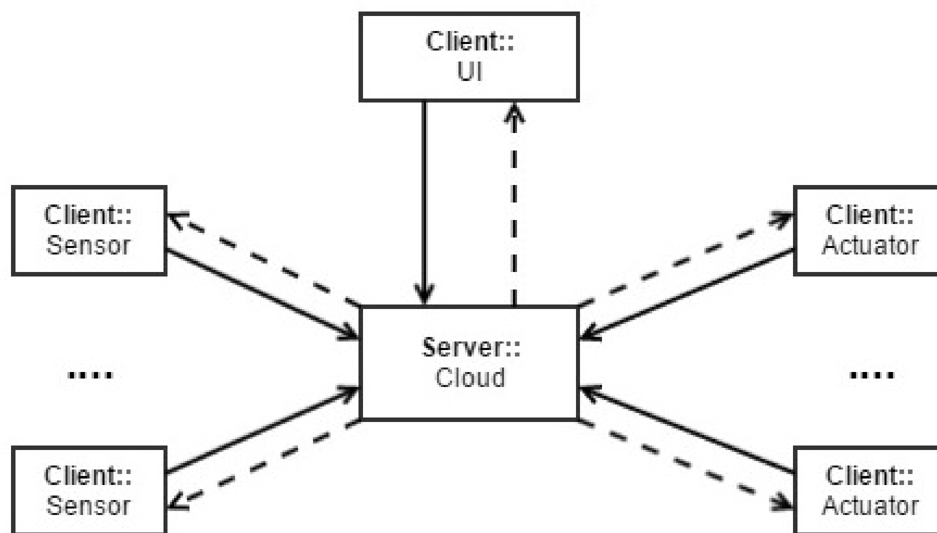

FIGURE 5.21: Client-Server many-to-many Topology

- **Evolvability**:
  - S1. For the one-to-one communication topology, the condition for the second and third pattern is considered the autonomous logic. This would need to be changed within the sensor or actuator containing this condition check. For the many-to-many scenario, the logic is isolated in the cloud component.
  - S2. This is only applicable in the many-to-many scenario. A new type of sensor would mean that the logic in the Cloud would also have to be updated to support this new data, however the change will not ripple to the actuators.
  - S3. A new scenario can be added in the Cloud component and it would not require change in other components.
- **Performance**: Having the sensor and actuators communicate directly to each other performs better than having an intermediary Cloud server in the middle. However, this would require the sensor to know all of the actuators in the system and would also require the sensors and actuators to contain the autonomous behavior logic themselves. This reduces evolvability. As for the choice of Client-Server style, the same trade-offs with reliability/availability that were discussed for the aggregation category apply here.
- **Availability**: The central component adds a single point of failure even though data aggregation is not a primary goal.
- **Security**: This type of centralized Client-Server topology has the inherent effect that the edge devices do not accept requests, making it harder for attackers to communicate with them.

- **Privacy**: In this topology all of the data passes through a main component even though data aggregation is not the primary goal. This could be avoided by allowing the sensors to communicate directly to actuators, however the trade-offs were already mentioned.

Peer-to-Peer

Up to know it has not been possible to use the peer-to-peer style as there was an inherent hierarchy of components that comes with the goals of the different categories. However, for this class of solution it is possible to model peers as a combination of sensor, actuators and decision making capabilities. These peers could communicate with each other when needed in order to perform their duties. An example could be driver-less cars that communicate with each other in order to find the best routes and not hit each other.

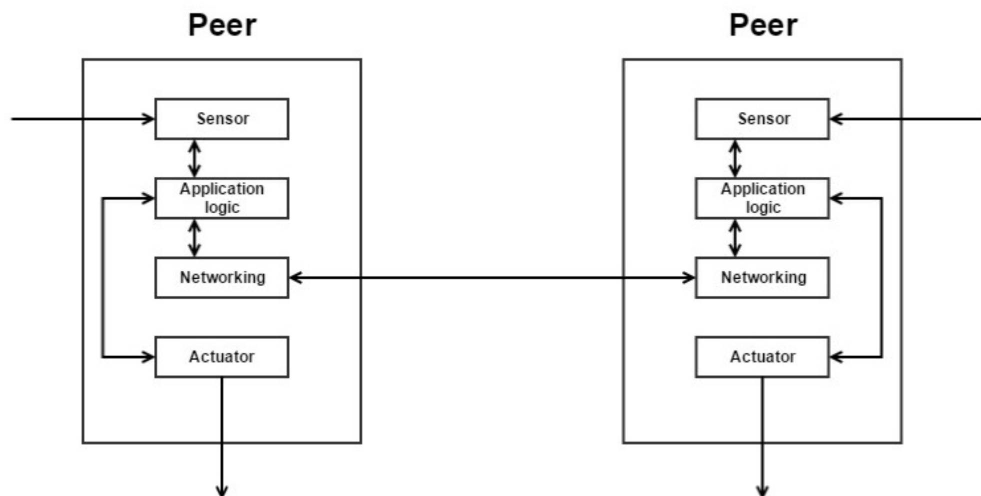

FIGURE 5.22: Peer-to-Peer Topology

A peer-to-peer style requires that all components are both producers and consumers of data and functionality. In the previous classes there was no possibility to create one component that served as both, unless the mobile device is used as a sensor and a user interface. If we think of peer-to-peer applications in the best, it required all users to run a peer node that includes all functionality. In the same way, this can be done for this class by merging sensors, actuators and decision making logic into one component. The result is shown in figure 5.22. We could also model a driver-less car solution with a centralized Client-Server style, however this will have effects on the quality attributes that the system will exhibit.

- **Interoperability**: The peer-to-peer style is inherently bad for interoperability with other systems, as communication can only occur between peers.

- **Evolvability**:Inherent in the peer-to-peer style is that any change at all to the system needs to occur in all peers. If a new type of sensor is added to one peer, than all peers must also get this addition in order to adhere to this style.
- **Performance**: This style decreases latency since peers communicate directly and not via an intermediary. It also increases scalability as every added peer not only consumes but also provides data and functionality. This however comes at the cost of having smarter edge devices that use more power.
- **Availability**: There is no single point of error in a peer-to-peer system, making it a good style for availability. In some cases, peers my take over for each other if a peer should fail.
- **Security**: Each peer needs to handle its own security.
- **Privacy**: Data does not pass through a central node, which can be considered good for privacy. However, it can also be argued that the openness of the styles to other peers could allow for unexpected and untraceable data flows.

Pipes-and-Filters

As already mentioned in the Client-Server section, the same trade-off is available in the sensor to central component section. Due to the lack of support for interactivity of this style, it is not suggested to use in any other scenario for this class.

Event-Based

The Event-Based can also be used as already described in the Client-Server section. This style cannot be combined with the peer-to-peer topology provided, as it would require either having a central event bus or having all peers run an event-bus and having peers register to each other. This would also remove the possibility that the peers could request data from each other explicitly.

The entire system could also be modeled in this style by running a central event-bus. The difference with the Client-Server topology presented in figure 5.21 is that the system can no longer provide actuator actions based on data aggregated from multiple sensors in an efficient way. However, if data aggregation is not a requirement then this style can be used as an efficient way to provide one sensor to many actuators communication if the number of actuators is large and can change. In this way the sensor does not have to know the identity of the actuators.

Publish-Subscribe

The Publish-Subscribe style can be used in the same way as the event-based style except that the actuators will be interested in a type of content instead of an event from

a specific sensor. The same performance and evolvability trade-off described before between the event-based and publish-subscribe style applies here.

Service-Oriented

The Service-Oriented Style could be used to run entire services in the sensor and actuator components. The communication with the registry would only occur to locate new sensors and actuators, however after the addresses and interfaces are recovered from the registry service the sensors and actuators could communicate directly with each other, which increases performance with regards to latency.

The trade-offs are however the same as we have seen so far when comparing centralization against decentralization. A publish-subscribe component could be used to notify sensors or actuators when a certain "type" of devices has been registered to the system, which increases decoupling and network efficiency and in this case adds some latency due to the extra layer of communication provided by the event-service, however in this scenario it might not be an issue.

The sensors and actuators can have custom interfaces for increased performance or use a uniform interface which increases interoperability.

REST The REST uniform interface can be added to increase interoperability while reducing performance, however the remaining effects remain the same as the choice for the Client-Server style. This has been the case throughout this analysis since REST is Client-Server with a uniform interface for all components that have a server role.

Layered

This class could have so many different architectures that it is possible to model some of them in the layered style by putting a sensor logic layer, then an application layer and finally an actuator logic layer. This can only be done of course if the control only flows one way, and not if the sensor and actuator layers need to communicate with the application layer.

Microkernel Because this class is still so broad, it is hard to decide if customizability is important. If it is, then this style can be used in one of the components to provide while giving up some performance and increasing complexity of the system.

Verdict

For this class, almost all styles are possible given certain additional requirements. We have seen that it is possible to have a Client-Server, Peer-to-Peer and Service-Oriented architecture, where Event-Based, Publish-Subscribe, Pipes-and-Filters and REST can play a secondary role. However, the trade-offs come down to centralization against decentralization which we have already seen in the previous classes.

5.4.2.2 Class I. User-Controlled Actuators

This class contains solutions where the actuators are controlled directly by users. However the analysis does not show any important developments for choosing an architectural style. It still depends on the same factors which style should be chosen. For this reason I will not provide the mapping for this class as it would be redundant. By now we have already discussed the most important findings of this mapping, these will be summarized in the next section.

5.5 Conclusion

This chapter has provided a mapping of software architectural styles to a set of IoT solution classes. It is clear that there is a place for all styles in the IoT, but not all styles are a good fit in every place. We will now conclude this chapter with the most important discoveries.

The Internet of Things and Software Architecture. After all of the analysis done in this thesis on Internet of Things and software architectural styles, it becomes clear that the big difference that the term Internet of Things brings to the software architecture design process is the inclusion of two logical components, which are the sensors and the actuators.

The sensor is a data provider while the actuators are data consumers. The details beyond that can be abstracted from at this level of analysis. What this means for the architecture design process is that we must consider sensors and actuators as separate components from the start, containing the minimum logic to perform their main task. The software to be designed needs to be decomposed and mapped to physical components.

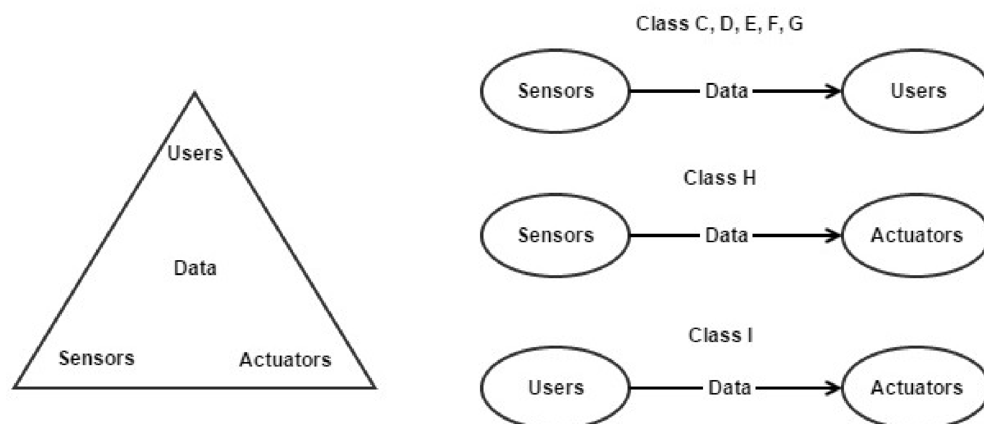

Figure 5.23: The Internet of Things effect on the software architecture design process

Figure 5.23 shows a picture of what I consider to the "Internet of Things trinity". The user in this case can be a system administrator that configures the system once and

allows the autonomous logic to perform its tasks, or it can be an someone that actively uses the system. At the center of the trinity is data, which is used as communication between the three parts of the trinity. Sensors provide data either to users or actuators. Actuators receives data either from sensors or from users. The question of how this transmission of data is done is what we try to do when designing a software architecture and thus choosing a starting style.

This trinity alone is not enough information to suggest an adequate architecture. The designer needs to know what the goal of the system is and what the most important quality attributes are. If data aggregation is the goal, then it might be better to use a Client-Server or Service-Oriented style. However, if the goal is to create independent components with low-latency communication, then a Peer-to-Peer or decentralized Service-Oriented style is the best choice.

The image does not give a depiction of classes A and B, since these are solutions that do not provide sensors and actuators but instead allows interoperability to occur between solutions or devices.

Interoperability. Interoperability between solutions in the IoT can occur by either speaking the same language or using a mediator. The mediator can be hosted by one of the two parties trying to communicated or by a third party. Classes A and B deal with the scenario where a third party is used as intermediary.

Direct communication is best for performance, however since solutions can be from different vendors it might be the case that this is not feasible depending on the design decisions made for both parties. The second scenario is where one of the parties hosts a mediator, typically the party that wants to "use" the other party, meaning that it depends on the response. The last case is by using a third party mediator that provides an infrastructure for communication between multiple parties.

For this case, the best style is the Service-Oriented style as it provides a way for the parties to announce themselves as services and describe how they should be invoked. They parties have to agree to use the same communication protocol and have to describe their services in the language required by the third party solution, however once this is done the parties remain completely decoupled from one another. The third party mediator can use an orchestration service to script how the multiple parties should interact. The Service-Oriented style was made for interoperability [Bas] and it shows in this analysis.

The REST style can be used to provide a uniform interface in anticipation of interoperability with another system. A good example is to put a REST interface to one of the aggregation classes, allowing for the data to be used by other systems.

Security and Privacy. It was a difficult task to reason about security and privacy during this analysis. For security, it comes down to freeing resources at the edge of the

network so that they could be used for security reasons. Another indicator could be the number of entry points an attacker has. For privacy, it came down to centralization versus decentralization. If it was possible to keep data locally instead of sending it to a Cloud, then the architecture is inherently friendly.

There are no architectural models for security[Bas], i.e. there is no real model for reasoning about security at the software architecture level. A paper that describes a reference architecture for achieving security and privacy in the IoT[Add14] concludes by saying that the security and privacy of any system is primarily dependent on the implementation.

The design tactics for security all have a lot to do with implementation, such as authorization, encryption, data integrity checks and availability checks. For this reason I must conclude that the analysis done in this thesis is at a high level where security cannot be reasoned about properly.

Consumer and Producer. When designing software, we are used to thinking in terms of consumers and producers of data and functionality. For a long time it has been that the users of the system are the only consumers of functionality and the back-end of the system is the provider. However, in the Internet of Things we now have sensors and actuators, which also have a role to play. The reason that the Client-Server is suggested as the best style in most cases in this analysis is because the sensors, actuators and user interface tend to be on different physical components. This creates a hierarchy of consumers that use producers, or clients that use servers.

We have seen that if the sensor, actuator (and ui if present) of a system can be located in one physical component, then it is possible to use the Peer-to-Peer style, which brings many benefits. This is because, as defined in the style description, a peer must be both consumer and producer. However, if this is not the case, then it is more likely that we will choose a more centralized style.

Pipes-and-Filter and Event-Based. The analysis shows that the Pipes-and-Filters style can be used to send data from the sensors to its destination points, increasing performance, network efficiency and scalability while reducing reliability by constantly sending messages without getting a confirmation that the message has been received. This means that the main component has less of a workload since it does not need to send confirmation messages, meaning less use of resources.

The event-based style can be used to get the same trade-off when sending data to UI or actuators. The added bonus is that this decouples the event-source from any of the event-processors, which increases evolvability. It also removes the constant need for UI or actuators to have to poll the server to see if any changes has occur, instead they receive a notification when this has happened. This increases scalability as the main component does not have to process the constant polling messages.

In practice, this would be like choosing between TCP and UDP. We choose the Client-Server style when we want to be sure that all messages reach their destination, by receiving a confirmation from the destination for every message sent. This is analogous to TCP connections. If we don't care about this and just want to send messages faster, we use UDP. UPD does not check if all messages arrive at their destination, which is analogous to the messaging in Pipes-and-Filters and Event-Based.

Centralization Versus Decentralization. We have seen that in some scenarios there is no choice but to have a central Cloud component for all users and devices. However in most cases there is some choice to be made between centralization and decentralization. Decentralization, meaning moving logic and data to the edge of the network, has the following effects:

- Increase in performance due to less latency, less throughput requirement, Scalability as central components are less taxed (if there is a central component).
- Increase in availability, since any central components become less of a single point of failure.
- Increase in privacy, since data can remain at the edge of the network.
- Increase in bandwidth efficiency, since the edge devices can perform more tasks without having to ask a central component, meaning less communication over the network.
- Decrease in evolvability, since updates have to be pushed to edges instead of changed in one location.
- Decrease in energy efficiency, since the edges have to perform more computations.

Microkernel. The Microkernel style is used to provide customizability to an application. However, in most cases this is not useful and only brings more complexities without any benefits. For class A it is a good fit, since not every user will have every compatible device in their smart home or environment, meaning they can only install the plugins they need. For the other classes I could not think of any reason to use this class.

Layered style. The layered style can be useful to illustrate the decomposition of a system into cohesive layers. However, not all systems can be modeled in a layered way [Ric15]. I find that this is the case for most IoT topologies, as we now have two entry points into the system, i.e. the users and the devices. This makes it hard to model the system in the constraints of the layered style, unless you considers users and devices to be in the same layer, which would not be cohesive.

Sensors as initiators, actuators as reactors. Allowing sensors to be the initiating party in communication allows for more choices when deciding on messaging patterns.

We could choose periodic or sporadic messaging patterns, which allow for better optimization to the goal of the system. For real-time systems we would want to send periodic messages while for event-based systems such as a motion detector we want to send a message only when it matters.

Having sensors as reactors can also be an option, in the case that the initiating party knows the identity of the sensor and only needs the value being measured at the moment of initiating communication.

Actuators are data consumers, so in most cases it is better to address them when needed instead of having them constantly poll for changes. The choices are either to send messages directly to them or using an event-based or publish-subscribe mediator.

Software Architectural Styles in the Internet of Things. The results of this analysis shows that even with the decomposition of the IoT into multiple classes, there are still a class where multiple styles could be chosen depending on the goal of quality attribute requirements. What this means is that the term “Internet of Things” does not provide enough information to choose even a starting software architectural style, which makes it even harder to provide a reference architecture for the entire IoT. This does not mean that the reference architectures provided so far are not useful, however there are more than enough scenarios where they are not applicable.

This entire analysis is based on a set of IoT solutions that span multiple application domains. However, we might find even more IoT classes and different architectures if we expand this set to include newer solutions. What is noticeable in this set of IoT solutions is that there are many sensor-only solutions, where the goal is to provide information to the user. This makes it that centralized styles provide better quality attribute effects on average for this dataset. As we move towards including more actuators into solutions, I believe we will start to develop the need for more decentralized architectures in certain areas.

In chapter 2 I proposed to look as the Internet of Things as an event, or at least an attribute of a system and not as a type of system itself. If a system claims to be an IoT solution, then they are only stating that their system includes sensors and/or actuators. When people speak of the Internet of Things, they should refer to it as an event that is happening and not as a type of system. This does not mean that the analysis done in this thesis or by other literature has been in vain, as it gives a view of how things are now for a subset of the solution space and how we can potentially move forward. However, trying to provide statements or architectures that will hold for the entire Internet of Things as a single domain is a difficult thing to do, thanks to the dynamic nature of the concept.

Which software architectural styles best fit each of the IoT classes and why? We have seen that for the integration category, the best styles are Service-Oriented and REST. For the observation category, the Client-Server style provided the best overall quality

attributes. For data aggregation, the best style was also Client-Server where certain quality attribute requirements would suggest a Pipes-and-Filter or Event-Based style. However, for the smart system's class I provided both a Client-Server and a Peer-to-Peer topology. These two styles are the opposite, meaning using either one as a starting style for an architecture would result in completely different quality attribute effects. The research question cannot be answered any further than this with the dataset of solutions used.

Can the set of best styles be merged into a single reference architecture? Given the answer to the previous question, it is not possible to merge all of these different styles together to provide a single reference architecture for the Internet of Things. This would have only been possible if we had the same "best style(s)" for each of the classes. However, as anticipated in the introduction, it was unlikely that such a reference architecture could exist with the wide variety of applications that carry the term Internet of Things. This analysis has now shown that it is indeed not possible.

Chapter 6

Conclusion

This section is the conclusion of this thesis, which contains a summary of the most important findings and suggestions for future research in this area.

6.1 The Internet of Things

The concept of the Internet of Things is shrouded in ambiguity. First of all, there is a difference between the definitions provided in literature and the solutions that are available right now. The result of this is that the IoT in literature has many quality attribute requirements that are not important to all solutions that are branded as IoT. To further increase the confusion, the analysis done on a set of Internet of Things solutions that exist at the moment reveals a great variety of goals, requirements and implementations.

A classification of IoT solutions provided at least four categories of solutions with different goals. However, increasing the size of the dataset might also increase this number.

What is *the* Internet of Things? Based on the analysis done in this thesis, I must conclude with the following definition:

"The Internet of Things is the continuous increase of connectivity between the physical world and the digital world by connecting physical entities to a network"

Which defines the IoT as an event that is happening instead of as a network, which is how it is defined in literature. When an IoT solution is developed, it is contributing to the Internet of Things.

However, what has been focused on in this thesis are Internet of Things solutions, which can be referred to as *an* Internet of Things. The realization that there are multiple

Internet of Things is the important message in this thesis, where the common denominator in the entire solution space is the connecting of the physical world to the digital world by use of sensors and actuators that come in many forms.

6.2 Software Architectural Styles

There is many literature available on software architecture and in most cases they turn out to be consistent with each other. There is an agreement that at least the quality factors of a system is for the most part decided by the architecture. The literature present on software architectural styles however, is very limited. Most papers have a small section about them and refer to the paper written in 1994 by David Garlan and Mary Shaw[GS94]. It also doesn't help that some references refer to them as software architectural patterns.

I have chosen the styles that are most common, in the sense that they appear in various literature where their definitions are consistent with each other to some point. I also used informations from web blogs and videos to confirm this.

However, I believe that there needs to be a standard reference catalog of styles with a clear definition of what the constraints are. There are some styles I did not cover in this thesis simply because their constraints were not clear and well documented. This needs to become a goal for the architecture community if styles/patterns are ever to be taken seriously in computer science moving forward.

6.3 Quality Attributes Evaluation

The methods that exist for evaluating architecture for their fulfillment of quality attribute requirements operate at a lower level than is possible in this thesis. They require far more context information in order to make a verdict. However, they did provide some examples of how to reason about interoperability, evolvability, performance and availability. This was harder to do for security and privacy, however this has been known to be difficult even in later stages of architecture design.

6.4 Software Architectural Styles in the Internet of Things

The analysis of software architectural styles in the context of the IoT classes has various results. For some classes it is clear which styles should be chosen in order to get the best quality attribute effects. However, for the class "Smart Systems", almost all styles could be used since that class does not give enough context information. This is a consequence

of the variety of solutions in the IoT. There will most likely always be a “miscellaneous” class which contains all solutions that do not fit in the other classes.

What we have seen is that the Client-Server style seems to be applicable to receive on average good results for each class, however I believe that this says something about the dataset used and not for the entire IoT. The solutions used as examples for the classification in this thesis lean towards a sensor-oriented and centralized solutions. There are some exceptions of course, however the imbalance creates a specific type of classification tree which is biased to some extent. However, it is not possible for me to use a dataset which representative of all types of solutions in the IoT, as such a dataset is not likely to exist due to the huge variety and constant evolution.

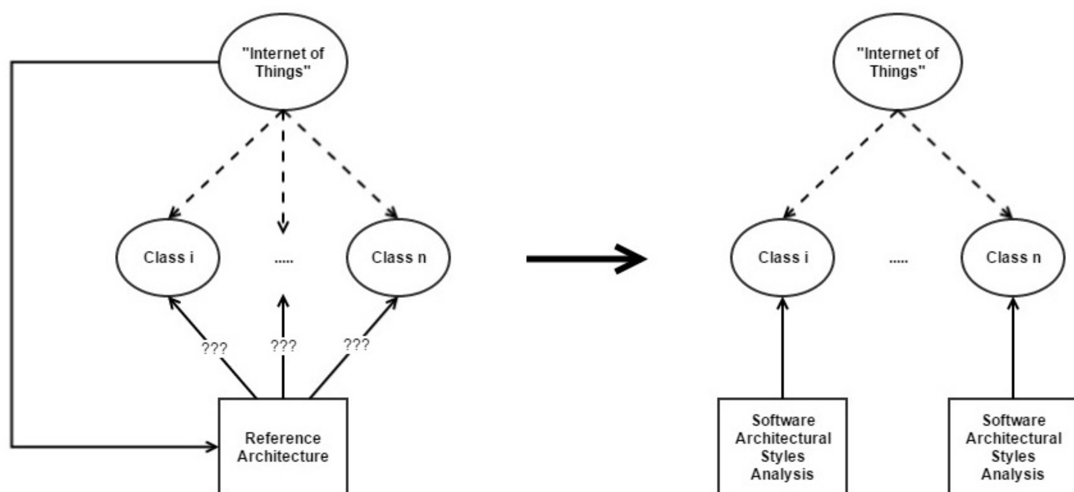

FIGURE 6.1: The problem scenario vs. the thesis scenario

Figure 6.1 shows the problem scenario we began with, where reference architectures were inferred form the Internet of Things as a single domain. However this thesis argues that these reference architectures are not applicable in reality due to the great variety of applications. This thesis has provided some examples classes of applications and has performed an analysis of software architectural styles on these classes.

The classification provides the first steps into achieving more specific details about the IoT solution space. The result of this thesis is that there is a place for all styles in the Internet of Things, however not all styles can be used in all places. There are clearly better styles for different classes, which shows that a single reference architecture for the Internet of Things as a single system cannot exist. The main research question is answered with these results.

6.5 Future research

This thesis has provided the first step of recovery in the field of software architecture in the Internet of Things. The problem of using the term Internet of Things as context for software architecture is a serious one as it will continue to bring confusion to the discipline. Future research could be done by using a larger dataset of Internet of Things solutions, preferably one which takes into account the number of solutions that have actuators and smarter systems. In this way we can provide a better taxonomy of Internet of Things solutions, to which we can start to provide reference architectures for with the correct labels. This will result in designers being able to pick from a list of reference architectures designed for "Class x" in the Internet of Things instead of a general "one size fits all" architecture.

Another direction we could go into is to provide a formal and universal definition for the Internet of Things, with which we can truly classify systems as being IoT or non-IoT. In this thesis I have derived a definition based on what is considered IoT at the moment. However, it could also be possible to come up with a new and more concrete definition so that the IoT does not simply contain "any system that measures and changes the real world".

A direction that should be avoided is the proposals for more reference architectures for the Internet of Things as a single domain.

Appendix A

IoT Definition List

This appendix contains a list of definitions for the IoT found in various sources. All the following definitions are direct quotes from the sources, which are all mentioned.

ID	Definition	Source
1	The Internet of Things is the network of physical objects or "things" embedded with electronics, software, sensors and connectivity to enable it to achieve greater value and service by exchanging data with the manufacturer, operator and/or connected devices. Each thing is uniquely identifiable through its embedded computing system but is able to interoperate within the existing Internet infrastructure	Wikipedia[1]
2	The Internet of Things is a computing concept that describes a future where everyday physical objects will be connected to the Internet and be able to identify themselves to other devices	Techopedia[2]
3	The Internet of Things is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment	Gartner[3]

ID	Definition	Source
4	The Internet of Things is a network that makes all devices collaborate at all times, converse spontaneously among themselves and with the rest of the world, and all together make up a kind of single virtual computer - the sum of their respective intelligence, knowledge and know how.	Vision and challenges for realising the Internet of Things [SW10]
5	The Internet of Things is a new actualization of subject-object relationships. Me and my surroundings, objects, clothes, mobility, whatever, will have an added component, a digital potentiality that is potentially outside of my control	Vision and challenges for realising the Internet of Things [SW10]
6	The Internet of Things is a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communications protocols where physical and virtual things have identities, physical attributes, virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network	Vision and challenges for realising the Internet of Things [SW10]
7	The IoT contains things that are expected to become active participants in business, information and social processes where they are enabled to interact and communicate amongst themselves and with the environment by exchanging data and information sensed about the environment, while reacting autonomously to the real/-physical world events and influencing it by running processes that trigger actions and create services with or without direct human intervention	Vision and challenges for realising the Internet of Things [SW10]
8	The Internet of Things allows people and things to be connected anytime, anyplace, with anything and anyone, ideally using any path/network and any service	Vision and challenges for realising the Internet of Things [SW10]

ID	Definition	Source
9	IoT refers to the networked connections of everyday objects, which are often equipped with ubiquitous intelligence	Internet of Things: A vision, architectural elements, and future directions[GP13]
10	The worldwide network of interconnected objects uniquely addressable based on standard communication protocols	Internet of Things: A vision, architectural elements, and future directions[GP13]
11	The IoT is an interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications. This is achieved by seamless ubiquitous sensing, data analytics and information representation with Cloud computing as the unifying framework	Internet of Things: A vision, architectural elements, and future directions[GP13]
12	The basic idea of this concept is the pervasive presence around us of a variety of things or objects - such as Radio-Frequency Identification tags, sensors, actuators, mobile phones, etc. - which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbours to reach a common goal	The Internet of Things: A Survey [AM10]
13	The Internet of Things can be described as an amalgamation of multiple heterogeneous digital devices, people, and services working collaboratively to solve a problem across technology boundaries, with the ability to seamlessly interact and share data about themselves and their environment.	Reference Architectures for Privacy Preservation in Cloud-Based IoT Applications [Add14]
14	Based on the traditional information carriers including the Internet, telecommunication network and so on, the Internet of Things is a network that interconnects ordinary physical objects with the identifiable addresses so that provides intelligent services	Internet of Things: Objectives and Scientific Challenges [Ma11]

ID	Definition	Source
15	A world-wide network of interconnected objects uniquely addressable, based on standard communication protocols	Internet of Things - Applications and Challenges in Technology and Standardization [BS11]
16	The Internet of Things is an emerging global Internet-based information architecture facilitating the exchange of goods and services in global supply chain networks	Internet of Things - New security and privacy challenges [Web10]
17	The notion of an Internet of Things refers to the possibility of endowing every day objects with the ability to identify themselves, communicate with other objects, and possibly compute	The Software Fabric for the Internet of Things [RA08]
18	Extending the Internet to physical objects - the Internet of Things - promises humans to live in a smart, highly networked world, which allows for a wide range of interactions with this environment	Object Recognition for the Internet of Things [QvG08]

Appendix B

IoT Solutions Analysis

This appendix contains an analysis done on a selected list of IoT Solutions.

B.1 Methodology

Firstly the IoT Solutions will be explained and explored to illustrate the business goal. The topology of the key components are extracted from any information available about the solution. These components are presented in two diagrams, one of which illustrates the cardinality of the association between the components.The second diagram shows how the data flows between the components. The vocabulary used when discussing software architectural styles contain components and connectors, hence the choice of identifying this for each solution and displaying it in diagrams.

The first diagram displays the cardinalities of communication between the components (Cardinality diagram). A one-to-many connection indicates that one instance of a component will communicate with one or more instances of another component. For each of these relationships we need to identify the scale, as it can reveal if scalability is important for the solution and thus which software architectural styles can be used. Also included in the diagram is the location of the data storage and the application logic.

The second diagram contains arrows indicating the flow of messages between components (message flow diagram). In this diagram we are interested in either the flow of data that is measured from the sensors or command messages that are sent to actuators. Note that not all messages between all components are illustrated, as this is out of scope. Both diagram also indicate which components contain sensors and/or actuators.

The solutions are grouped by their respective IoT Domains. The remainder of this appendix contains an analysis of the solutions.

B.2 Connected Home

B.2.1 Nest Thermostat

The Nest thermostat is a smart thermostat which attempts to learn your daily routine for temperature changes in your house. When it learns this, it will apply these changes automatically without the user having to think about it. The user can also change the temperature in their home via the Nest app or via the thermostat directly. Information regarding the Nest Thermostat was found on the Nest Home page Nest[1], the technical specification page[2] and the NEST developer documentation page[3].

FIGURE B.1: Nest Thermostat

FIGURE B.2: Nest Thermostat cardinality diagram

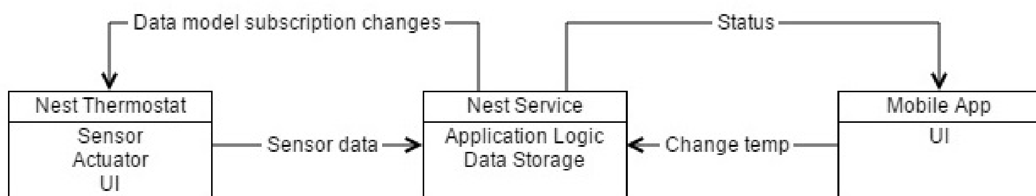

FIGURE B.3: Nest Thermostat message flow diagram

[1] https://nest.com/
[2] https://nest.com/support/article/2nd-generation-Nest-Learning-Thermostat-technical-specifications
[3] https://developer.nest.com/documentation

Characteristic	**Value**
Physical Entity	Home (Environment), Boiler.
Attibute(s)	Temperature, Humidity, Proximity, Far-Field Activity, Near-Field Activity, Ambient light.
IoT Connector	Sensor and Actuator. The Thermostat acts as both a sensor for the room and an actuator that changes the temperature in the room by operating the boiler or other heating system present.
Application logic location	Nest Service (Cloud).
Data storage location	Nest Service (Cloud).
User Interaction	Nest Mobile App (Mobile device), LCD display and rotating ring to manually change temperature (Thermostat).
Scale of system	The number of requests to the Nest service is potentially always growing.
Internet-Dependency	Internet-Dependent. Even though the temperature can always be manually changed by rotating the ring on the thermostat, the functionality of using the mobile app is lost. Also, the model that predicts what temperature the room should be at resides in the Nest service, thus the "smartness" is also absent without an Internet connection.
Type of invocation	Explicit and Implicit. The user can explicitly change the temperature. However, the thermostat listens to changes in the data model located in the Nest service. If such a change (event) should occur, it triggers the actuator which changes the temperature in the room.
Interoperability	With the Nest API, an outside system can build clients to access Nest data. The client can subscribe to changes.
Battery life	Built-in rechargeable lithium ion battery.
Autonomous behavior	Nest service can learn the behavior of the user and can predict which temperature the room should be in. The Nest service can change the temperature of the room without human interaction at this point.

.

B.2.2 Homeseer

The Homeseer is a complete package for home automation. Rather than focusing on one particular aspect of the connected home domain, homeseer offers automation for lights, temperature, locks, security, water, energy, audio and video. A big selling point for this solution is that they promise not to store personal user information in the cloud. Information regarding the Homeseer was found on the home page[4] and the developer support page[5].

FIGURE B.4: Homeseer

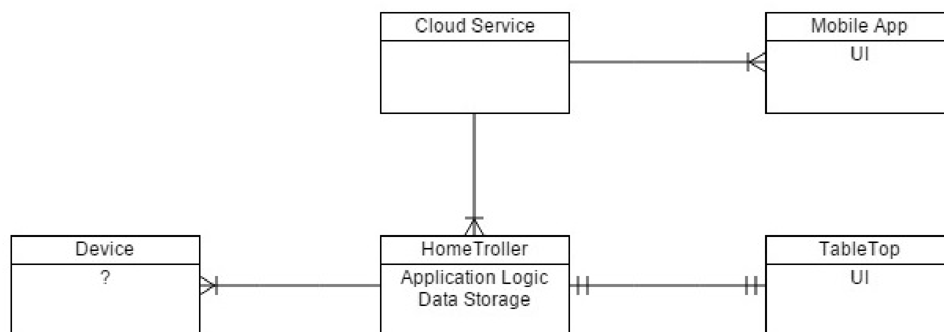

FIGURE B.5: Homeseer cardinality diagram

[4]http://www.homeseer.com/
[5]http://homeseer.com/support/homeseer/HS3/SDK

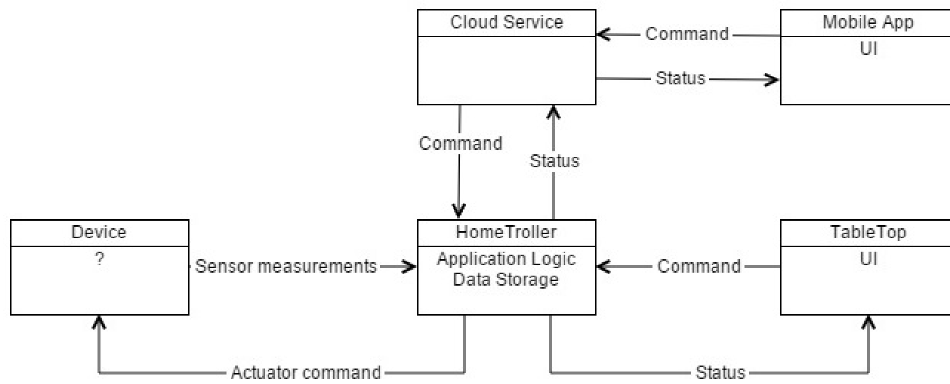

FIGURE B.6: Homeseer message flow diagram

Characteristic	**Value**
Physical Entity	Home (Environment) and potentially all connected things inside a connected home
Attibute(s)	Light, Temperature, Access (door, locks), Power, Water, Energy, Occupancy, Audio/Video and more
IoT Connector	Sensor and Actuators. Many devices are supported
Application logic location	Homeseer Hometroller and also in devices depending on which are used. Some third party devices, which require plugins to use, have their own application logic.
Data storage location	Homeseer HomeTroller Hub
User Interaction	Devices in the connected home can be controlled by a tabletop android-powered touchscreen or through a mobile app.
Scale of system	The scale of the communication between the HomeTroller and the TableTop is bound to the number of devices of a single connected home, which is relatively small in scale. The Cloud service will have to scale, but since this is simply used as a router/gateway for communications for the Mobile App, the resources needed could also be relatively small.
Internet-Dependency	Internet-Dependent. While they advertise that they are internet-independent, the functionality of being able to control your home remotely via the mobile app is impossible without an internet connection.
Type of invocation	Explicit and Implicit. The user can explicitly control their home. Plugins can be installed in order to react to events.
Interoperability	Plugins can be installed on the HomeTroller, which allows it to work with various technologies and devices. Most plugins need to be bought. Interoperability is also reached by communicating with the IFTTT web service. Plugins cannot communicate with each other.
Battery life	The HomeTrollers connect directly to power outlets. The devices inside the house vary in battery life.
Autonomous Behavior	Most interaction is done by the user choosing to change the state of the home. Developers are able to write their own plugins using the plugins SDK to allow custom triggers, actions and conditions to events. It is also possible to program some smartness into the HomeSeer by using the IFTTT webservice.

B.2.3 SmartThings

Smart Things also offers a smart home environment. The main nodes are the SmartThings Hub and the SmartThings app. Similar to the Homeseer, the scale of the solution can be as big as the user wants it to be. The biggest difference being that SmartThings has a cloud service that contains the data gathered by the SmartThings system and also contains application logic to make the system work. Another difference is that it primarily depends on third party devices as the number of SmartThings sensors and actuators are limited. Information regarding the SmartThings solution was found on the home page[6] and the Developer Documentation page[7].

FIGURE B.7: SmartThings

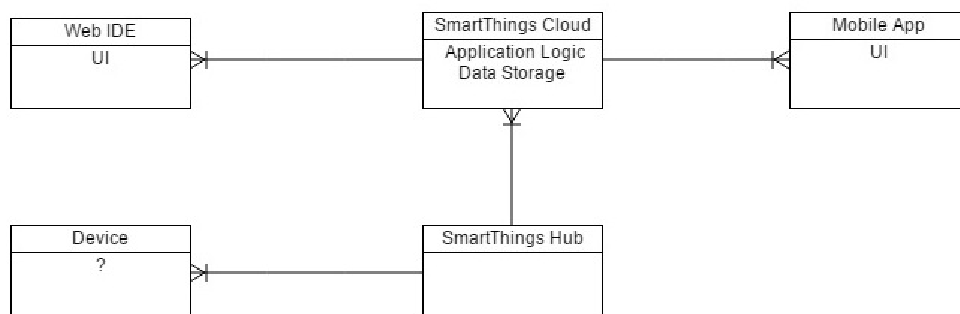

FIGURE B.8: SmartThings cardinality diagram

[6]http://www.smartthings.com/
[7]http://docs.smartthings.com/en/latest/

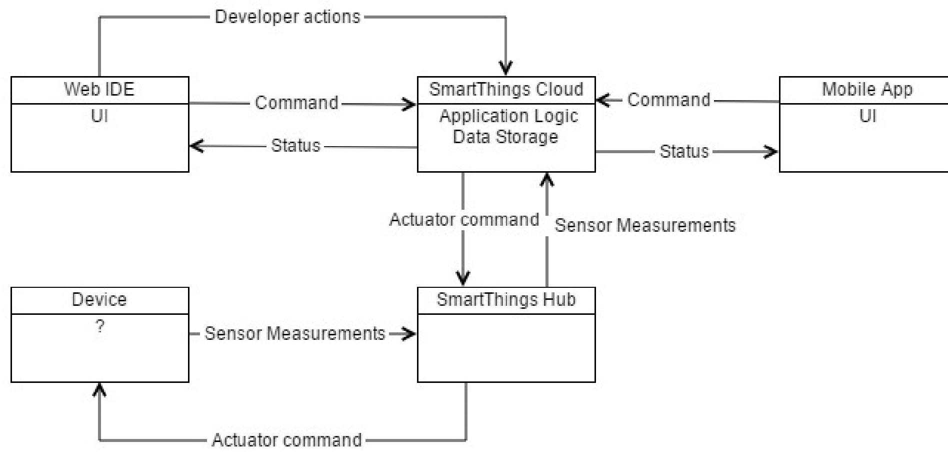

Figure B.9: SmartThings message flow diagram

Characteristic	**Value**
Physical Entity	Home (Environment) and potentially all connected things inside a connected home
Attibute(s)	Light, Temperature, Access (door, locks), Power, Water, Energy, Occupancy, Audio/Video and more
IoT Connector	Sensor and Actuators. Many devices are supported
Application logic location	All intelligence is located in the SmartApps and SmartServices layer in the SmartThings Cloud
Data storage location	A model called the "Open Physical Graph" is located in the Cloud service which is a representation of the real world. The model and the real world are meant to be kept in sync.
User Interaction	Mobile application and Web IDE
Scale of system	The scale of the workload at the edge of the network is very low, because it is limited to the number of devices of a single home and the Hub is simply a gateway to the Cloud. The Cloud service on the other hand should be very scalable as it acts as a service for all SmartThings users. The number of users is potentially always increasing
Internet-Dependency	Internet-Dependent. Due to the "Cloud first" philosophy of SmartThings, there is a seperation of "Intelligence" from edge devices. This means that if there is no Internet, there is no functionality.
Type of invocation	The user can explicitly control their home. SmartApps can be connected to the SmartThings Hub in order to react to events.
Interoperability	The SmartThings Hub is updated constantly to interoperate on a syntactic level. The semantic interoperability happens in the SmartApps layer in the Cloud
Battery life	The SmartThings Hub connect directly to power outlets. The devices inside the house vary in battery life.
Autonomous Behavior	Initially there is no autonomous behavior. SmartApps can be developed and connected to the SmartThings hub to induce autonomous behavior. SmartThings also have a channel to communicate with the IFTTT webservice.

B.3 Connected Body

B.3.1 Angel Wristband

The Angel Wristband has three sensors that can monitor the health of the user. It works with an array of fitness and health tracker mobile applications. There is an SDK to develop for the Angel Wristband and extract the health metrics, device info, the Raw Waveform and to configure the alarm clock which is also present in the band. Information regarding the Angel Wristband was found on the home page[8] and the SDK GitHub page[9].

FIGURE B.10: Angel Wristband

FIGURE B.11: Angel Wristband cardinality diagram

FIGURE B.12: Angel Wristband message flow diagram

[8] http://www.angelsensor.com/
[9] https://github.com/SeraphimSense/angel-sdk

Characteristic	**Value**
Physical Entity	Human Body
Attibute(s)	Heart rate, temperature, movement, pulse
IoT Connector	Sensor
Application logic location	The Wristband has just enough logic to understand a request for the value of a certain sensor and send it via a Bluetooth connection. The Applications will contain the logic to use this data in an intelligent way.
Data storage location	Applications. The applications that connect with the Angel wristband could either store the values locally or upload them to a server.
User Interaction	Applications. The Wristband itself does not have a user interface or gives the user feedback in any way.
Scale of system	The scale of the solution is very small. There is no central cloud server. Instead the wristband will be synced to mobile apps via Bluetooth. There is no specification on how many requests the wristband can take at a time.
Internet-Dependency	Internet-Independent. The communication between the wristband and devices occur via Bluetooth communication.
Type of invocation	Explicit. The mobile app has to query the wristband for the measurements.
Interoperability	The SDK allows for full interoperability with the wristband, allowing the apps to access all of the measurements, an alarm clock, device info and raw waveform (for research). The SDK is currently supported for Android and iOS.
Battery life	There is a non replaceable rechargeable lithium polymer battery in the wristband. The estimation is 7 days battery life per charge. Charge time is 1 hour.
Autonomous Behavior	There is no autonomous behavior in the Angel Wristband itself. It could be possible that the apps could be programmed in a way to work without human interaction, however this is out of the scope for this analysis.

B.3.2 Nymi Wristband

The Nymi band uses the user's unique heart signature to authenticate them to a system or device. An example of a possible application is logging into a pc or gaining access

to an area using the Nymi band. The company offers a developer kit for the platform. Information regarding the Angel Wristband was found on the home page[10], the Nymi SDK page[11] and a whitepaper[12].

FIGURE B.13: Nymi Wristband

FIGURE B.14: Nymi Wristband cardinality diagram

FIGURE B.15: Nymi Wristband message flow diagram

[10] https://www.nymi.com/
[11] http://www.nymi.com/dev/documentation/nymi_band.html
[12] https://www.nymi.com/wp-content/uploads/2013/11/NymiWhitePaper-1.pdf

Characteristic	**Value**
Physical Entity	Human Body
Attibute(s)	Heart signature
IoT Connector	Sensor
Application logic location	The Nymi band has application logic that allows it to enroll the user and keep the identity available on the band until it is taken off. The Nymi band is also secure by running crypto.
Data storage location	The user identity is stored while the once the user has enrolled themselves by wearing the band and putting their finger on the sensor. This data is stored on the band. There is also a local storage of security keys.
User Interaction	The user receives feedback in the form of vibration and a LED lights. The user has to authenticate themselves once everytime they put on the Nymi band, but remain authenticated until the band is taken off.
Scale of system	The scale is fixed to a one-to-one connection, as it only communicates with applications to authenticate the user. The connection is broken once the user is authenticated.
Internet-Dependency	Internet-Independent. The communication between the wristband and devices occur via Bluetooth communication.
Type of invocation	Explicit. Enrolling the identity when wearing the band is an explicit action. Authenticating to an application is also done explicitly.
Interoperability	Applications can communicate with the Nymi band by importing the Nymi Communications Library (NCL). Currently Android, iOS, Mac and Windows are supported.
Battery life	The estimated battery life is 5 days and takes up to two hours to charge.
Autonomous Behavior	There is no autonomous behavior. The main goal is authentication by heart signature.

B.3.3 Zebra Motionworks

The Zebra motionworks solution uses a set of Zebra receivers (RFID) to follow the movements of NFL players that each have an RFID transmitter tag placed inside their shoulder pads. The system measures precise location measurements of the players to

offer real-time stats. The coaches can use the motion data to change their strategies and algorithms can aggregate player's stats and display them real time for fans to create a deeper experience. Information regarding the Zebra Motionworks was found on the homepage[13].

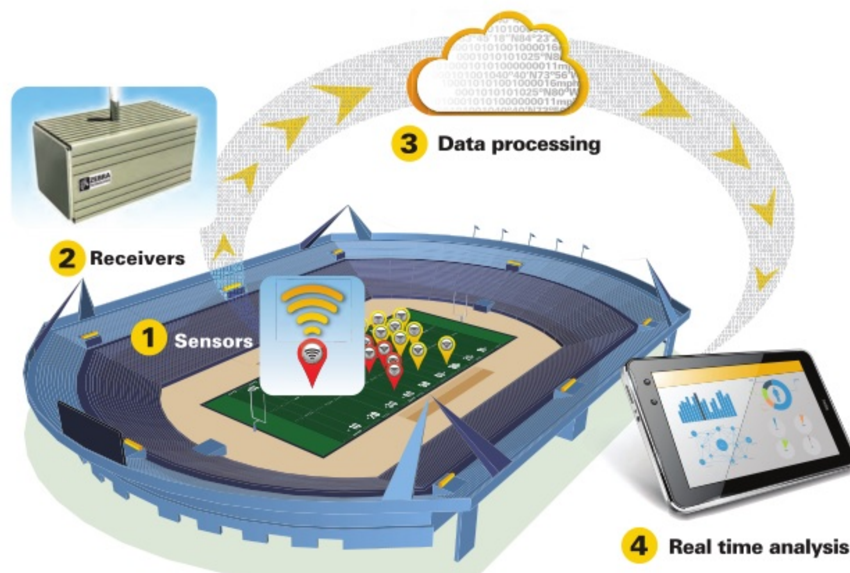

FIGURE B.16: Zebra Motionworks

FIGURE B.17: Zebra Motionworks cardinality diagram

FIGURE B.18: Zebra Motionworks message flow diagram

[13] https://www.zebra.com/us/en/nfl.html

Characteristic	**Value**
Physical Entity	Human Body
Attibute(s)	Location
IoT Connector	Sensor reading RFID tag
Application logic location	Cloud server
Data storage location	Cloud server
User Interaction	Tablet or mobile phone UI to view the data analysis
Scale of system	The number of tags and receivers per stadium is constant, so those nodes do not need to scale. The number of users using the mobile app and need to have real time data can potentially always increase, so the Cloud service should be scalable.
Internet-Dependency	Internet-Dependent. The information gathered needs to be uploaded to the Cloud for analysis and visualization.
Type of invocation	Implicit. The RFID receivers are constantly updating the data model. The UI's get updated based on the changes to this model.
Interoperability	There is no mention of interoperability with any type of system.
Battery life	The battery life of an active RFID tag is estimated at around 3 to 8 years depending on the broadcast rate. The receivers placed around the stadium are plugged into a power outlet.
Autonomous Behavior	The system is able to record movement, push it to the server, analyze and visualize it without human intervention.

B.4 Connected Retail

B.4.1 Scanalytics floor sensors

Scanalytics floor sensors extends the floor with sensors that measure how people walk on the floor. This data can be used to analyse customer behaviour in a store. The store can then change the arrangements of the items for sale to better emphasize the most popular items. One of the selling points of the scanalytics floor sensor is that it does not gather personal information. People contribute to the data anonymously. Information

regarding the Zebra Motionworks was found on the homepage[14]. I tried to gain access to the SDK but I have not received a reply from Scanalytics.

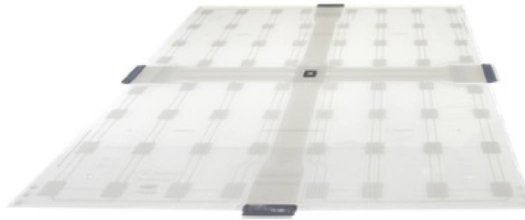

FIGURE B.19: Scanalytics Floor Sensor

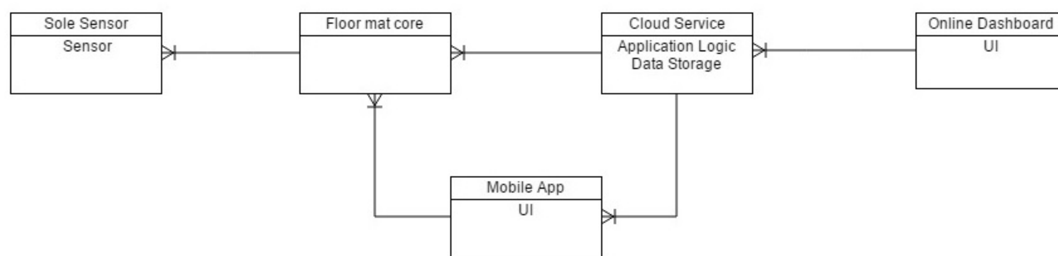

FIGURE B.20: Scanalytics cardinality diagram

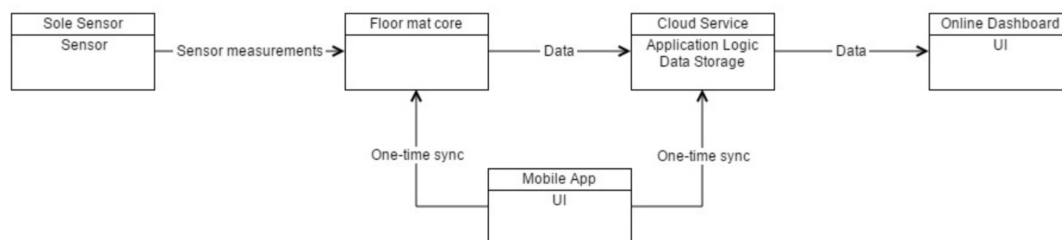

FIGURE B.21: Scanalytics message flow diagram

[14] http://www.scanalyticsinc.com/

Characteristic	**Value**
Physical Entity	Floor
Attibute(s)	Occupancy
IoT Connector	Sensor
Application logic location	Cloud server. The Floor mat is only used as a gateway to the Internet for the sensors.
Data storage location	Cloud server
User Interaction	The user of the system can access an online dashboard to view the data.
Scale of system	The number of sensors per mat is fixed at 64. The communication between the mobile app, the floor mat and the cloud service is a one-time synchronisation of the mats to the cloud service, so it is assumed that this does not increase the workload. The Cloud service needs to be scalable because the number of floor mats and connections from online dashboards can be potentially always increasing.
Internet-Dependency	Internet-Dependent. The Floor mat needs to have an internet connection to push the data.
Type of invocation	Implicit. The moment there is a change in the occupancy this is pushed to the server.
Interoperability	There is an SDK to develop on this platform, but I could not gain access to this so no more information is known.
Battery life	Connects to a power outlet.
Autonomous Behavior	The system can push data do the server and visualize it without human intervention.

B.4.2 S5 Electronic Shelf labels

The S5 Electronic Shelf labels can be placed in stores in order to keep prices of products up to date. Transmitters called T3 are placed around the store to form a network in order to push the prices from the management software to the S5 shelf labels. Information regarding the S5 Electronic Shelf labels was found on the home page[15].

[15]urlhttp://www.s5tech.com/en/

Figure B.22: S5 Electronic Shelf labels

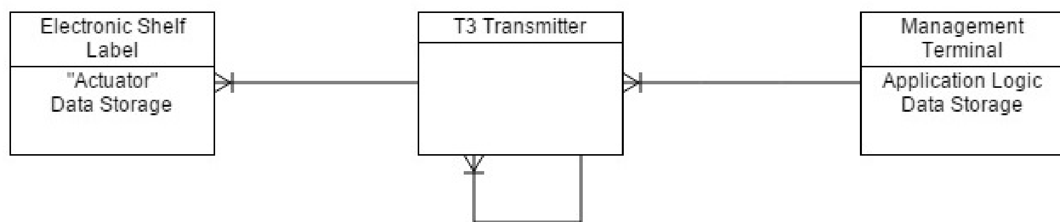

Figure B.23: S5 Electronic Shelf labels cardinality diagram

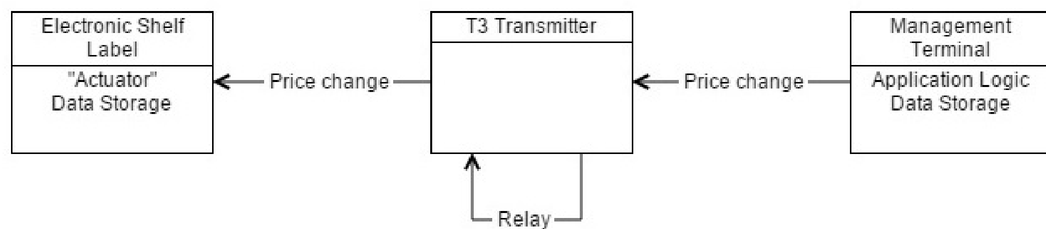

Figure B.24: S5 Electronic Shelf labels message flow diagram

Characteristic	**Value**
Physical Entity	Shelf labels
Attibute(s)	State
IoT Connector	Actuator
Application logic location	Server
Data storage location	The server contains the model with all the prices. The electronic shelf labels also store a history of their past states.
User Interaction	It depends on the type of UI that communicates with the IT backend of the store, which contains the prices to be pushed to the shelf labels. This is not provided by S5, instead the S5 system can be integrated with an existing backend.
Scale of system	It depends on the type of store, however the number of products should not increase or decrease with a huge amount. For this scenario we assume that there is a local central server for each store. It could be possible that there is a cloud server that contains the prices for a chain of stores, but this is out of scope for this analysis.
Internet-Dependency	The S5 Electronic Shelf label system itself is Internet independent, the T3 transmitters create a local network.
Type of invocation	Implicit. When a prices change, these are pushed to the correct shelf labels.
Interoperability	There is an ability to integrate with the store's backend system, but this has to be done by S5. There is no further mention of interoperability.
Battery life	The T3 transmitters are plugged into a power outlet. No information about the battery life of the shelf labels themselves could be found.
Autonomous Behavior	The T3 transmitters can decide for themselves to which other T3 transmitter the data should be pushed in order to reach its destination. The shelf labels send a message back to confirm the price has been changed, if this is not received the price is pushed again.

B.4.3 Nomi Brickstream live

Figure B.25: Nomi Brickstream live

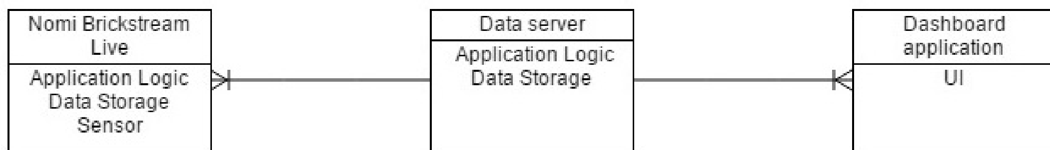

Figure B.26: Nomi Brickstream live cardinality diagram

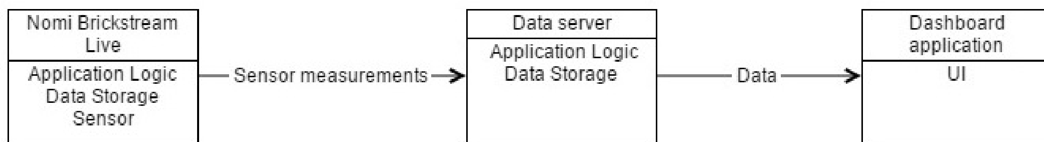

Figure B.27: Nomi Brickstream live message flow diagram

Characteristic	**Value**
Physical Entity	Room (Environment)
Attibute(s)	Occupancy
IoT Connector	Sensor
Application logic location	Nomi Brickstream live contains all the application logic to analyze its own data and detect faces. A central server is used only to aggregate the data from multiple sensors.
Data storage location	The Nomi Brickstream live sensor can store up to 30 days of full resolution video
User Interaction	The user can view the analysis of the aggregated data on a customized online dashboard.
Scale of system	Each store or chain of stores run their own central server to which the sensors report to. The system is based on edge processing, where the nodes at the edge contain relatively high computing capabilities, making it easier to add more nodes since it has less effect on the central server.
Internet-Dependency	Internet-Independent. The sensor can keep collecting data during a network outage. When the network is restored, the sensor uploads the data to the central.
Type of invocation	Explicit. The sensor is continuously recording.
Interoperability	There is no mention of interoperability with other solutions except for an integration with the data backend of the store.
Battery life	The Nomi Brickstream live is powered over ethernet (POE).
Autonomous Behavior	The Nomi Brickstream records video, detects faces and pushes the data to a server without human intervention.

B.5 Connected Transporation

B.5.1 WeatherCloud

In an attempt to provide better weather and road condition information, the WeatherCloud solution places multiple sensors inside and outside of a car. These sensors measure precipitation rate and type, ambient lighting, dew point, ambient temperature, pavement temperature, road conditions, slip/grip of tires and vehicle dynamics. This data is then aggregated by an on-board Smart Hub which pushes it to the Cloud. The resulting data on the Cloud is sent to driver's cell phones or navigation screens or can

be accessed through the WeatherCloud desktop application. Information regarding the Weather Cloud was found on the homepage[16].

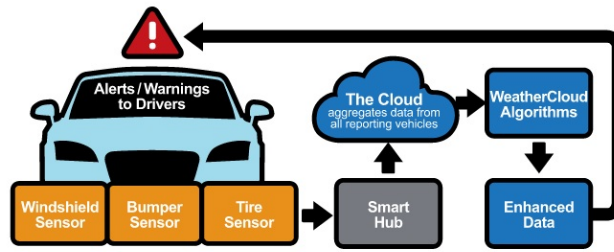

FIGURE B.28: WeatherCloud

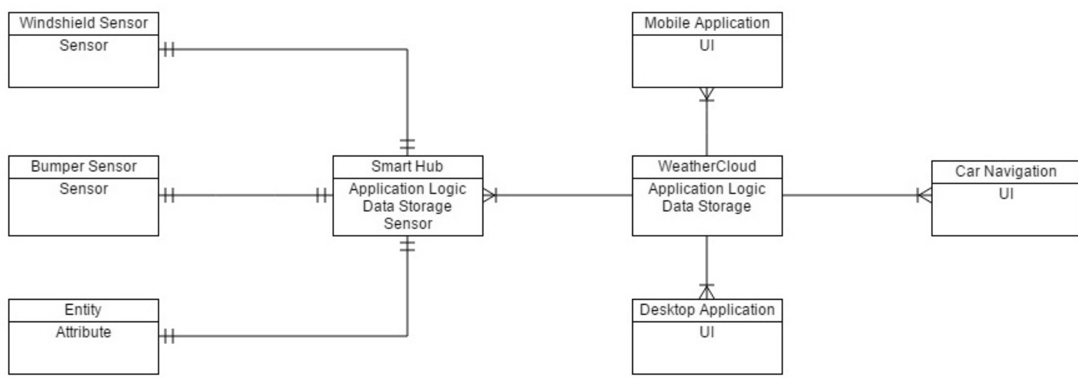

FIGURE B.29: WeatherCloud cardinality diagram

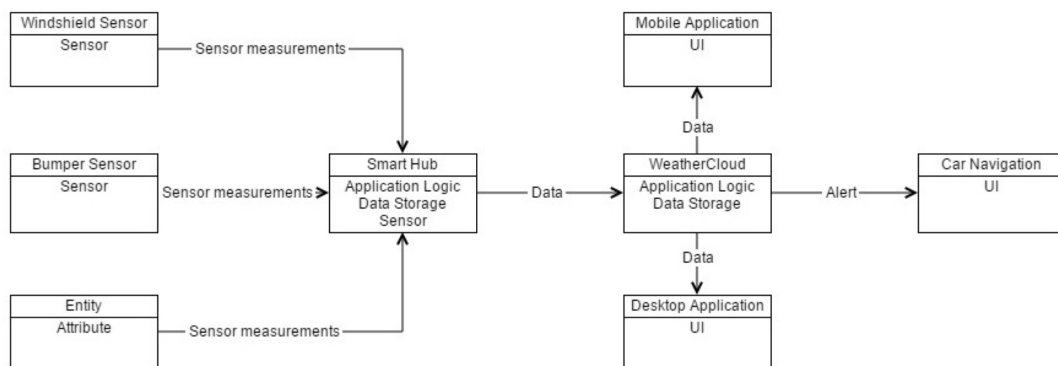

FIGURE B.30: WeatherCloud message flow diagram

[16] http://weathercloud.co/

Characteristic	**Value**
Physical Entity	Car
Attibute(s)	Temperature , wind direction , velocity , precipitation rate , Tire slip and grip, Pavement temperature, Road condition , Humidity and Visibility.
IoT Connector	Sensor
Application logic location	The Smart Hub contains some application logic to aggregate the data from the three sensors and compress them in such a way that they can be sent to the Cloud. The Cloud contains the rest of the application logic to make the entire solution work.
Data storage location	Data for an individual car is stored in the corresponding Smart Hub, while the aggregated data for all WeatherCloud enabled cars is stored in the Cloud.
User Interaction	The user can view the data on on a desktop app or gets a notification of important alerts inside his car navigation system or mobile phone.
Scale of system	The number of nodes inside the car is fixed to 4. The Cloud server needs to be able to scale since the number of users (connected cars) can increase. Each connected car has an effect on the demand of resources for the Cloud server.
Internet-Dependency	Internet-Dependent. The Cloud is needed to complete the solution process flow.
Type of invocation	Explicit. The data is pushed from time to time to a smart hub. Users explicitly use the desktop app to query the current weather conditions.
Interoperability	There is no mention of interoperability with any type of system.
Battery life	Two of the three sensors run on solar energy and the third one generates energy from the accelerations of the tire. The Smart Hub is powered by the car.
Autonomous Behavior	The system will send important alerts to drivers without human interaction.

B.5.2 Truvolo Connected Car Solution

The truvolo solution gives a car connectivity by plugging a small device (Truvolo Drive) into the On-board diagnostics(OBD) port of the car, which relays data about the car via a mobile phone to the Cloud which can then be accessed by an online dashboard. The data available on the dashboard shows how much gas is used, how safe the car is

being driven and monitors the status of the car and gives an alert if the car needs to be maintained. All cars of a household can be managed by the dashboard. Information regarding the Truvolo Connected Car Solution was found on the home page[17].

FIGURE B.31: Truvolo

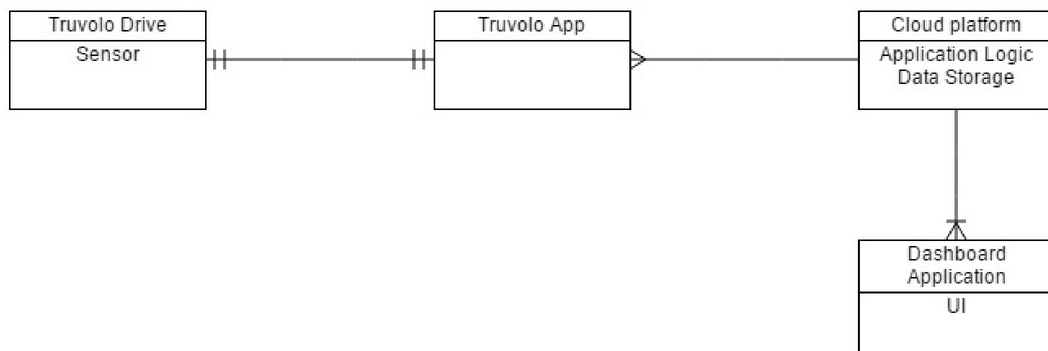

FIGURE B.32: Truvolo cardinality diagram

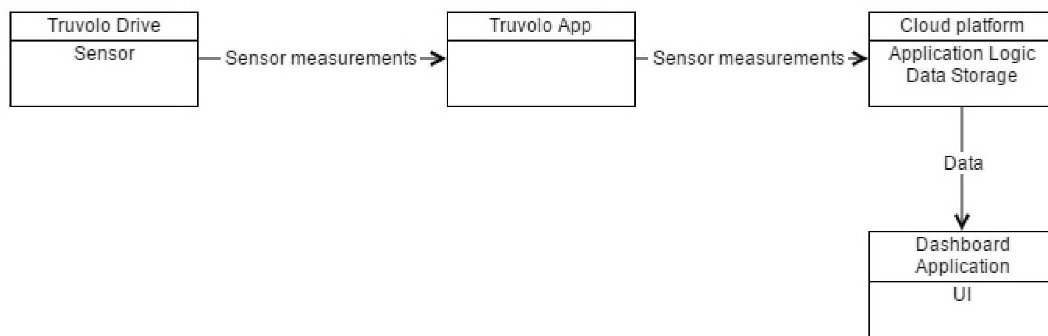

FIGURE B.33: Truvolo message flow diagram

[17] http://www.truvolo.com/

Characteristic	**Value**
Physical Entity	Car
Attibute(s)	All information available from an On-board diagnostic
IoT Connector	Sensor
Application logic location	Cloud server. The Truvolo Drive and the Truvolo App simply relay the data to the Cloud which actually uses the data.
Data storage location	Cloud server.
User Interaction	The user can access the data via an online dashboard.
Scale of system	The Cloud service needs to scale since all Truvolo users communicate with it and the number of users can always increase.
Internet-Dependency	Internet-Dependent. The solution can not function without a connection to the Cloud.
Type of invocation	Explicit. The Truvolo app queries the Truvolo Drive device on a regular interval and pushes the data to the Cloud.
Interoperability	There is no mention of interoperability with other solutions.
Battery life	The Truvolo Drive is powered by the car through the OBD.
Autonomous Behavior	Data is simply collected and analyzed.

B.5.3 Veniam Vehicular Networking

Marketed as "The Internet of Moving Things", this IoT solution turns vehicles into WiFi hotspots. This network can be seen as a way to provide pervasive Internet and thus help the IoT by providing access to the Internet for devices. The reason why Veniam is also considered an IoT Solution is because it also gathers information about the location of the vehicle. The connected vehicles act as a sensor for the city, providing valuable information to improve fleet management, operations and security. Information regarding the Veniam Vehicular Networking was found on the home page[18].

[18] https://veniam.com/

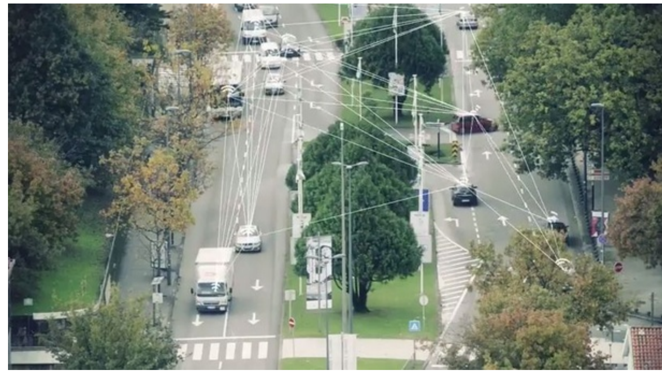

Figure B.34: Veniam Vehicular Networking

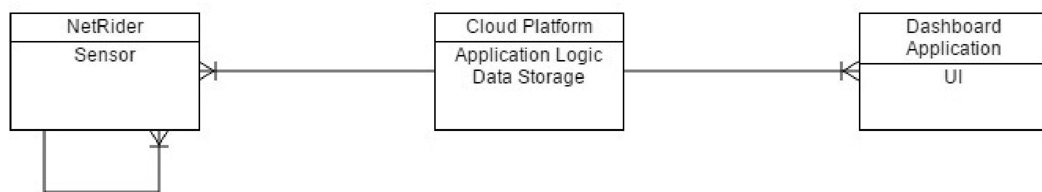

Figure B.35: Veniam Vehicular Networking cardinality diagram

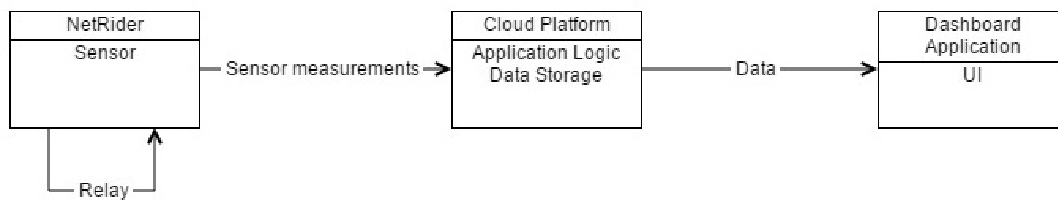

Figure B.36: Veniam Vehicular Networking message flow diagram

Characteristic	**Value**
Physical Entity	Vehicle
Attibute(s)	Location
IoT Connector	Sensor
Application logic location	Cloud Platform
Data storage location	Cloud Platform
User Interaction	The user can access real-time monitoring of their fleet on an online dashboard.
Scale of system	The number of connected vehicles can increase rapidly. The Cloud server needs to scale since it collects data from all Veniam connected vehicles.
Internet-Dependency	Internet-Dependent. The entire system is a network of WiFi repeaters.
Type of invocation	Explicit. The user can view the data on an online dashboard.
Interoperability	There is no mention of interoperability with other solutions.
Battery life	The NetRider device is powered by the vehicle's battery.
Autonomous Behavior	There is no autonomous behavior.

B.6 Smart City

B.6.1 Bitlock Bycicle Lock

Bitlock is a bycicle lock that unlocks when a smartphone containing the appropriate key is in proximity to it. The system also remembers the last location of interaction with Bitlock and can show a history of use. Access to the bycicle can also be shared with other users of the app. The digital key is shared between the mobile phones, so the Bitlock itself only needs the right key to be opened. It does not need to be connected to the Internet. Information regarding the Bitlock was found on the home page[19].

[19] http://bitlock.co/

FIGURE B.37: Bitlock Bycicle Lock

FIGURE B.38: Bitlock cardinality diagram

FIGURE B.39: Bitlock message flow diagram

Characteristic	**Value**
Physical Entity	Bycicle lock
Attibute(s)	Access
IoT Connector	Actuator
Application logic location	The application logic is located in both the Bitlock and the mobile app. The Cloud server acts as a storage for user information and as a router between apps, but also has some application logic to manage keys.
Data storage location	The key is stored on the mobile app and the bitlock. All other data regarding the history, usage, keys and user accounts are located in the cloud.
User Interaction	The user can interact with a mobile app to update the history of use and share access to bycicles. The Bitlock itself has a buttons in order to unlock the Bitlock in case the battery of the phone dies.
Scale of system	The connection between the Bitlock and the mobile app is a one-to-one connection via Bluetooth. The Cloud Service should be scalable since it is central to the system and the number of users could always increase.
Internet-Dependency	Internet-Dependent. The Bitlock can be unlocked with a 4 digit combination in case the phone dies. The Bitlock communicates via Bluetooth with the mobile device, so in this sense it is Internet-independent. However, the complete solution which includes access sharing needs the mobile app to be connected to the Internet.
Type of invocation	Explicit. The user has to not only be in proximity of the lock, but also has to press a button on the Bitlock in order to unlock it.
Interoperability	There is no mention of Interoperability with other solutions.
Battery life	The estimated battery life of a Bitlock is 5 years.
Autonomous Behavior	There is no autonomous behavior.

B.6.2 Array of Things

The Array of Things (AoT) is a smart city IoT solution where many sensors are placed around the city. The sensors around Chicago at the moment collect real-time data about the city and this data is published on an open database for researchers and the public to use. Things like temperature, humidity, light, carbon monoxide, nitrogen dioxide

and vibrations to name a few. The solution promises not to collect any personal data. Information regarding the Array of Things was found on the home page[20] and the Waggle home page[21], which is the platform the AoT is based on.

FIGURE B.40: Array of Things

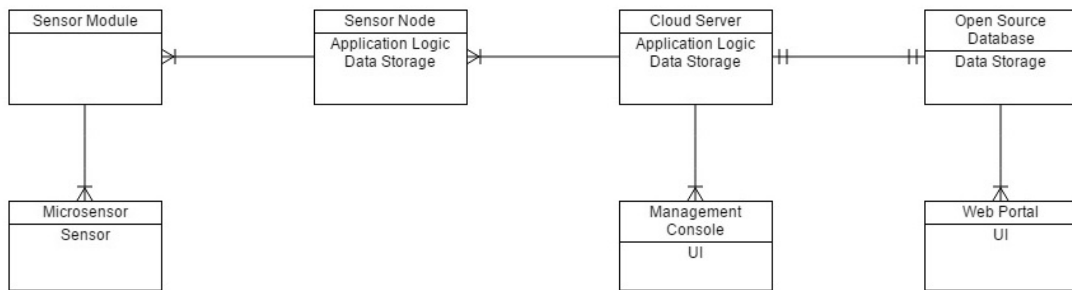

FIGURE B.41: Array of Things cardinality diagram

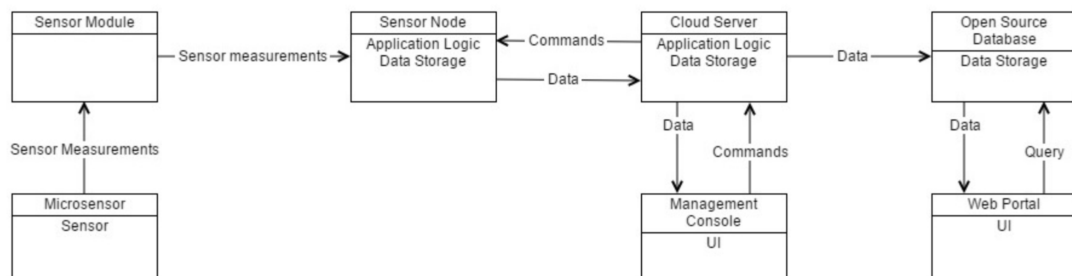

FIGURE B.42: Array of Things message flow diagram

[20] https://arrayofthings.github.io/
[21] http://wa8.gl/

Characteristic	**Value**
Physical Entity	City (Environment)
Attibute(s)	Temperature, Humidity, Light, Carbon-Monoxide, Nitrogen-Dioxide, Vibrations and more
IoT Connector	Sensor
Application logic location	The Waggle Node is considered an "Intelligent" sensor because it contains logic to provide smart power management, monitor the vitals of microsensors in its environment, cache data, process the data from the microsensors and contains a security module. The Cloud contains application logic to analyse the aggregated data from all sensor nodes.
Data storage location	Data collected from the microsensors are kept on the corresponding sensor node. The Cloud stores the aggregated data from all sensor nodes. After analysis and "cleaning", the data is also stored in a public database which can be accessed by a web portal.
User Interaction	The Cloud service and Sensor nodes can be managed by a management consoles. Users that want to access the public data published on the open database can use the public web portal.
Scale of system	It depends on the size of the city, however each city will host their own Cloud database with their own Public data portal, so the scale is not global and should not spontaneously increase since the users that manage the Array of Things can decide themselves when to add new sensors.
Internet-Dependency	Internet-Dependent. The microsensors are connected to the Sensor node via USB, I2C or Ethernet, but the pushing of the data to the Cloud is done via Internet.
Type of invocation	Explicit. The microsensors will stream real-time to the sensor nodes as they are connected by wire. The pushing of the data to the Cloud is done in regular intervals, meaning that it pushes the data explicitly and not event-based for example.
Interoperability	Interoperability can occur by listening to changes to the open database. There is no interoperability with the devices themselves.
Battery life	The Sensor nodes are connected to an external power outlet and distributes the power to all sensor modules connected to it.
Autonomous Behavior	The AoT does not exhibit any autonomous behavior.

B.6.3 Enevo Waste Collection

The Enevo waste collection solution also connects the city by measuring how full garbage containers are so that garbage trucks can optimize their routes. The solution promises to provide up to 50 percent in direct cost savings. It will also make the city look and feel cleaner, as garbage containers are less likely to be full and optimized routes can lead to less garbage trucks on the road. Information regarding Enevo Waste Collection was found on the homepage[22].

FIGURE B.43: Enevo Waste Collection

FIGURE B.44: Enevo Waste Collection cardinality diagram

FIGURE B.45: Enevo Waste Collection message flow diagram

[22] http://www.enevo.com/

Characteristic	**Value**
Physical Entity	Waste containers
Attibute(s)	Fill level
IoT Connector	Sensor
Application logic location	The sensor applies "advanced algorithms" to determine the fill level of a container, even if the surface is unevenly shaped. The Cloud server contains the application logic to aggregate, analyze and predict.
Data storage location	The data is stored in the Cloud server.
User Interaction	The user can access the data collected from the Eveno ONe web portal.
Scale of system	The Enevo ONe Cloud server needs to be scalable as it is central for all Enevo users and sensors.
Internet-Dependency	Since all data is pushed to the Cloud, a connection is needed. The Enevo sensors connect using wireless GSM communication.
Type of invocation	Explicit. Data is pushed on regular intervals. User can access the data from the Enevo ONe portal.
Interoperability	The Enevo ONe service provides a REST based API for accessing and managing data, allowing for integration with other solutions.
Battery life	Estimated at 10 years.
Autonomous Behavior	There is no autonomous behavior.

B.7 Industrial Application

B.7.1 Farmobile Fleet Management

The Farmobile Simplicity PUC is a fleet management system for farms. It collects data to create a digital farm record and shows information about the movement of farm vehicles in a web or mobile app. The devices called PUC are installed on a farm vehicle like a tractor and connects to the Cloud. This data is gathered for archive but also for analysis. Farmobile also manages a cellular data plan for this solution. Information regarding the Farmobile PUC Fleet Management was found on the home page[23].

[23] https://www.farmobile.com/

Figure B.46: Farmobile Fleet Management

Figure B.47: Farmobile cardinality diagram

Figure B.48: Farmobile message flow diagram

Characteristic	**Value**
Physical Entity	Farm Vehicle
Attibute(s)	Position, Speed
IoT Connector	Sensor
Application logic location	The PUC only collects data and sends it to the Cloud. The Cloud contains the logic to process the data.
Data storage location	The PUC have storage capacity of 64 gigabytes. The data is stored locally and pushed to the Electronic Farm Record whenever there is a cellular connection.
User Interaction	The user can access the Electronic Farm Record via a web or an iOS application.
Scale of system	The Cloud server needs to be scalable since it acts as a central components connected to all devices and UI.
Internet-Dependency	Internet-Dependent. The PUC can continue to gather information while there is no connection, but the connection is needed to sync the information to the Cloud.
Type of invocation	Explicit. The data is pushed to the Cloud on while there is a connection to the Cloud. The data can be accessed via an app.
Interoperability	There is no mention of Interoperability.
Battery life	The device is powered by the farm vehicle.
Autonomous Behavior	There is no autonomous behavior

B.7.2 Condeco Workspace Occupancy Sensors

The Condeco Workspace Occupancy Sensor is a sensor that reports to the Cloud how a workspace is occupied. Based on this data, the company can choose how to better use the workspace. The occupancy sensors push their data to a Cloud service. This data can be accessed via a web or mobile application. The application allows for many ways to view the data. The users of the system can then decide themselves how to better use workspaces, there are no actuators in this system. Information regarding the Condeco Workspace Occupancy Sensors was found on the home page[24].

FIGURE B.49: Condeco Workspace Occupancy cardinality diagram

[24] http://www.condecosoftware.com/products/condeco-sense/

FIGURE B.50: Condeco Workspace Occupancy cardinality diagram

Characteristic	**Value**
Physical Entity	(Work)space (Environment)
Attibute(s)	Occupancy
IoT Connector	Sensor
Application logic location	The sensor only senses and pushes the data immediately every 15 seconds. The logic to be applied to the data is in the Cloud.
Data storage location	The data is stored in the Cloud.
User Interaction	The data can be accessed via a web application.
Scale of system	The Condeco Cloud needs to scale since it acts as a central component for all sensors and UI.
Internet-Dependency	Internet-Dependent. The sensors need to be connected to the Cloud.
Type of invocation	Explicit. The data is pushed every 15 seconds and accessed via a UI.
Interoperability	There is no mention of interoperability with other systems.
Battery life	Estimated at 2 years.
Autonomous Behavior	There is no autonomous behavior.

B.7.3 DAQRI Smart Helmet

Designed specifically for industrial applications, the DAQRI smart helmet uses an array of sensors, cameras and augmented reality to connect your work environment simply by being there. The cameras recognize the devices in your work environment and uploads the state of the devices online. Other users that are also wearing the helmet in this environment can use this data. The entire landscape of your work environment is collaboratively mapped by all smart helmets. The Helmet also has a screen which shows the wearer what his work objectives are for the day, which are issued by a central server running the DAQRI application. Information regarding the DAQRI Smart Helmet was found on the home page[25].

[25] http://hardware.daqri.com/smarthelmet/

Figure B.51: DAQRI Smart Helmet

Figure B.52: DAQRI cardinality diagram

Figure B.53: DAQRI message flow diagram

Characteristic	**Value**
Physical Entity	Industrial workplace
Attibute(s)	State (Combination of attributes)
IoT Connector	Sensor
Application logic location	The Smart Helmet contains logic to process the data and even runs android apps.
Data storage location	Data captured is stored on the helmet and also pushed to a central server.
User Interaction	Wearer of the helmets receiver work objectives that are viewed on the screen. Work objects and sensed data can be handled in the Industrial 4D studio application.
Scale of system	Each company hosts their own Central Server and is thus capable of deciding themselves when to add more Helmets. There is no central DAQRI server that collects data from all Helmets in the world.
Internet-Dependency	Internet-Dependent. The Smart Helmet still needs to be connected to push the data measured and receiver work objectives that can change during the day.
Type of invocation	Not specified.
Interoperability	There is an SDK to develop Android apps that run in the Helmet, which could lead to interoperability with other solutions.
Battery life	Estimated to last "An entire shift".
Autonomous Behavior	The Helmet learns the environment with the help of its own sensors and data collected by other helmets that are present in the central sever.

B.8 Quality Attributes analysis

This section provides an analysis quality attributes in the Internet of Things based on the solutions explored in appendix B. It will also be shown that not all of these quality attributes are a priority in the reality, since there is such a huge variety of solutions.

Interoperability. The solutions that were explored exhibit various levels of interoperability with other solutions. The only real example of interoperability between different solutions in the dataset examined is in the connected home domain. The solutions in other domains offer the possibility of interoperability for the future. There is also a group of solutions that currently do not mention the ability to communicate with other solutions at all.

For the connected home domain, two example of systems were analyzed which offer interoperability between devices inside a smart home. The Homeseer solution allows

users to buy or develop plugins for the system in order to operate with more devices. The SmartThings solution achieves interoperability by having SmartApps run in the Cloud which understands the different types of third party devices. Users can also develop their own SmartApps.

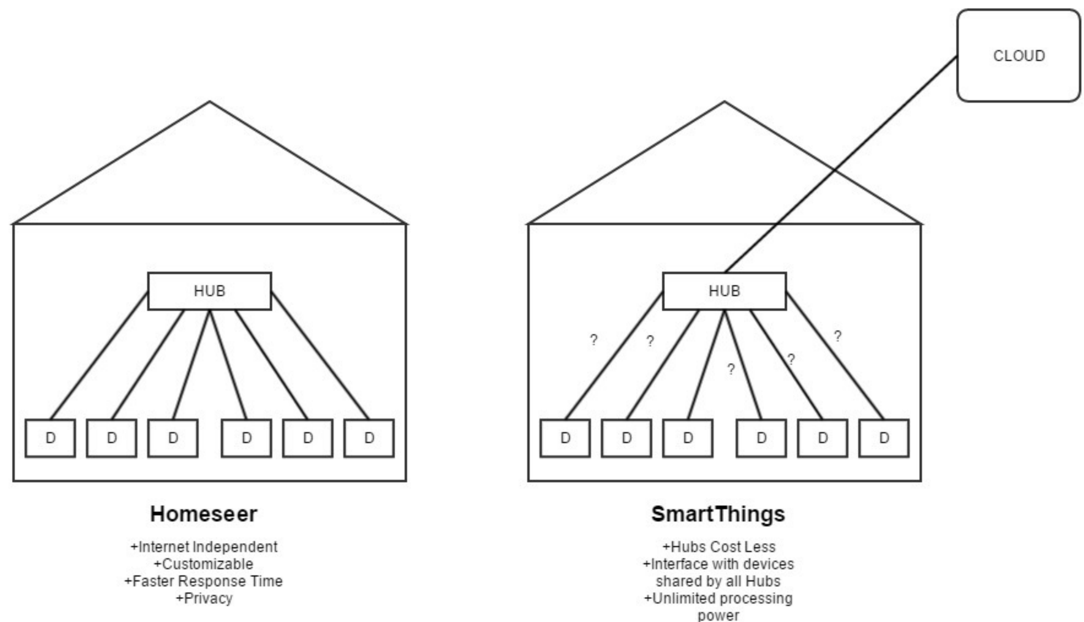

Figure B.54: Homeseer VS. SmartThings Interoperability

Both solutions offer a "Hub" which acts as a central communication point between the devices inside the home environment. Figure B.54 illustrates these two systems. The difference between these two solutions is the location of the logic needed to understand the communication with the devices. This logic is needed to reach semantic interoperability between the multiple devices inside a smart home. For the Homeseer, the logic is located in the Hub itself, while for the SmartThings solution this logic is located in the SmartServices layer in the SmartThings Cloud. The SmartThings Hub merely relays the communication to the Cloud.

This design choice has effect on the system on different levels. The interoperability between devices in the Homeseer solution is Internet-independent, the hub is customizable (as many plugins as the user needs), can respond faster (no need to connect to the Cloud) and is more privacy friendly.

The SmartThings Hub on the other hand shares the intelligence with all other hubs connected to the Cloud, which makes it easier to add new intelligence to interface to devices to all hubs at once. Because the logic is located in the Cloud, the computation power is not restricted by the hardware on the hub. Finally, as the hub only acts as a gateway between devices and the Cloud, it can be assumed that the SmartThings hub costs less to make than the Homeseer hub.

Both of these connected home solutions offer interoperability via the IFTTT[26] webservice. IFTTT is a webservice that allows users to link up two services by a conditional statement called recipes. If a certain condition occurs, for example a SmartThings motion sensor picks up some movement, then the IFTTT webservice can send a message to a Phillips Hue[27] bridge to turn all the lights on. These "If This Then That" recipes can be used for interoperability between solutions, as long as the IFTTT is given permission to communicate with them. A reason why this is a nice feature is because it allows people that are not programmers to create recipes that allow for interoperability and also autonomous behavior.

What is interesting about the IFTTT and the connected home domain is that it illustrates another way of providing interoperability between solutions, namely using a third party webservice as a mediator. This allows for the devices at the edge of the network to be less interoperable and autonomous, leaving the logic in the Cloud. This leads to Internet dependency and an increase in latency, but reduces the cost of production and power consumption of nodes at the edge of the network.

For solutions outside of the connected home domain, interoperability comes in the form of providing an SDK to develop apps that can communicate to the device or by providing an API to access the system. Both types of solutions implement a sensors-as-a-service pattern, i.e. they provide a way to collect data and access to it, however the usage of that data depends entirely on the system that is accessing the data.

An example is the Angel Wristband, which is a wristband that can measure vital signs of a person. How this data is used depends entirely on the app that is developed using the SDK. Another interesting example is the Array of Things. This is a Smart City solution where sensors are placed around a city that measure various attributes of the city. This data is published on a public database which can be accessed through an online portal. The use of this data depends on businesses, researchers and the general public.

Finally, there is a group of solutions that do not interoperate with any other solution. This supports the statement that *interoperability with other solutions is not important for all IoT Solutions*. There are some solutions that might not want to be open because of security, privacy and safety reasons. They might form an interconnected network of things with a select few solutions in the future, but not in such an open way as other IoT solutions.

An example is the DAQRI smart helmet, which is an augmented reality based helmet that can map an industrial environment while receiving work objectives for the worker wearing the helmet for the day. The helmet contains many sensors and processing power to understand its surroundings and help the workers with their daily routines while keeping the state of the machines in an industrial workplace synced with a data model

[26] `https://ifttt.com/recipes`
[27] `http://www2.meethue.com/nl-nl/`

on a central server. Some possible reasons why interoperability is not important for this solution is because it handles a lot of private information and the data it measures is very complex, making it hard for another system to understand. It could also be argued that such a system would likely evolve itself if more functionality is needed instead of relying on other solutions, because it is such a specific and complex solution.

To summarize, the following levels of interoperability between solutions were identified for current IoT Solutions:

- **Hub/Gateway**: The Hub/Gateway middleware node contains the logic to achieve semantic interoperability between devices.
- **Cloud**: The Cloud component contains the logic to achieve semantic interoperability between devices.
- **Third Party Service**: A Third party service is used as a mediator between two IoT solutions.
- **SDK**: An SDK is available to develop for the platform.
- **Open Database**: The data measured by the IoT Solution is available to other systems.
- **No Interoperability**: The solution currently does not support interoperability with other systems.

Evolvability. Although not a lot of information is available about the modularity of the software in IoT Solutions, one thing that is important for the IoT with respect to evolvability is the ability to push software updates to a node via the network.

Some solutions have very thin edges, which include just sensors that push their data to the cloud. All of the application logic for these types of solutions are this located in the Cloud. What this means for evolvability is that the software only needs to be updated in one location in order to evolve the system. In this way, the evolution of the system is more easily controlled, which is also one of the characteristics we identified for evolvability. If the knowledge is distributed amongst nodes at the edge of the network, these updates have to be propagated to all of these nodes. Futhermore, if the edges are less smart, they are more easily replaceable with respect to cost in case of damage or theft. This means that the exchangeability is increased in such a scenario, allowing for new versions of sensors to be deployed more easily. Figure B.55 illustrates this difference.

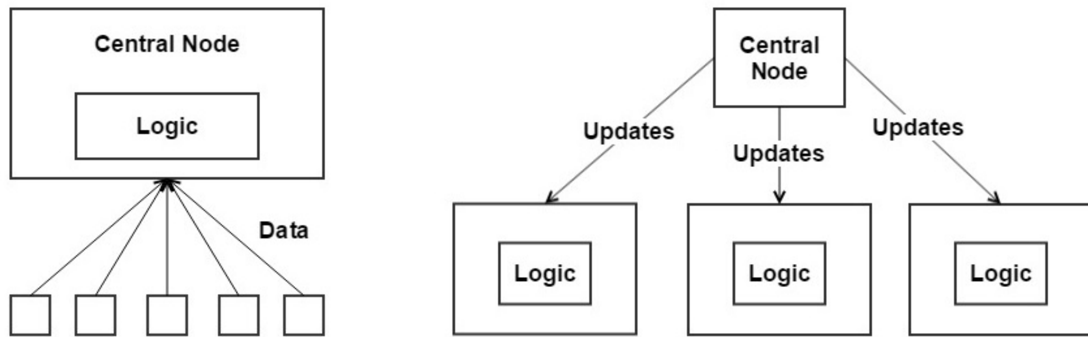

FIGURE B.55: Centralization vs. Decentralization for Evolvability

Having a centralized intelligence also has an effect on evolvability with respect to interoperability. If there is an environment where the number of heterogeneous devices is likely to increase at a fast rate, having a centralized intelligence means only updating one node in the network with the logic to communicate with these new devices. If the edge of the network is responsible for the interoperability, these updates will have to be pushed to these nodes. These nodes are also bound by their hardware capacity, which is in most cases very limited.

The choice for which type of solution to develop, centralized or decentralized with respect to evolvability, depends on the anticipated rate of change and number of nodes at the edge. If both are high, a centralized solution can be the best choice. However, if this is not the case and updates are infrequent, then having more logic at the edge of the network also brings advantages such as performance and availability.

Performance. As already mentioned, decentralization of the system can lead to an increase in performance due to low-latency. Half of the solutions examined rely completely on a centralized intelligence which is usually a Cloud component. This could mean that for these solutions, low-latency is not the highest priority. For many of these systems, the concept of having data "eventually consistent" , i.e. measured data from the real world is eventually synced to its digital counterpart or the change to the digital counterpart is eventually propagated to the edge node and changed in the real world, is good enough.

In an environment such as a connected home, there is a reliable Internet connection and the devices are stationary. This combination makes it possible to achieve relatively good performance even though all of the logic is in the Cloud. Take another environment where the connectivity might be unstable and the nodes are constantly in motion, and suddenly having logic closer to the edge can have a big impact on performance, especially if the device can sense, process and give feedback to the user without even needing a connection.

The DAQRI smart helmet is an example of a smart device at the edge of the network. It is able to completely analyze its environment while giving feedback to the user. The

connection to a central node is only used to obtain instructions and share the data measured with other helmets. The choice of having the intelligence in the smart helmet itself leads to the ability of always giving an analysis of the data instead of only when a connection is available.

The Array of Things also has some intelligence in their sensor nodes. In this case it is not to provide low-latency but instead to limit the number of operations the central component has to perform. The sensor nodes can pre-process the data so that the Cloud does not have to process all the raw data from the sensors. Instead it only has to aggregate the data from the various sensor nodes.

Scalability. For the majority of the systems analyzed (11 out of 18), the solutions have a Cloud component which all devices and users communicate with. For these systems nothing more can be said other than the "Cloud servers need to be scalable". The fact that the Cloud is scalable is a safe assumption to make, however having an architecture that requires less scaling decreases costs.

By my estimation, the Cloud server that would need to be most scalable out of the systems evaluated is the SmartThings Cloud. Each device inside a household will communicate with this Cloud component, albeit indirectly via the SmartThings Hub. A reason why this system is huge is because it is an example of a centralized interoperability-enabling solution. For IoT Solutions in other domains that do not currently support interoperability, the number of users and devices is not relatively big compared to some of the bigger IT systems that exist today with millions of users. Scalability in the IoT becomes more of a unique issue as services start to show up which offer interoperability between individual IoT Solutions, thus creating a network of many devices, where each new user brings a multitude of new sensors and thus more connections to central components.

Some solutions have less of a scalability problem if either the number of devices or number of users is fixed. For the Zebra Motionworks example, the number of devices is fixed, because there is a fixed number of NFL athletes allowed to play on the field at the same time. The number of RFID readers placed around the stadium is also fixed. The S5 Electronic Shelf solution is an example where the number of users is fixed. The number of shelf labels might increase dramatically if the store expands or adds a new branch, but there is always a fixed (or rather small) number of administrators that manages the system.

There are a number of solutions where scalability does not play a role. Take for example the Angel or the Nymi wristband. They communicate with one to a few applications at a time via Bluetooth. In this case both the number of devices and users is fixed.

The Nomi Brickstream live team argues that their system is more scalable because intelligence is located in the edge device, which is a camera. This makes it easier to add many nodes as each of them do their own computations and do not cause much

strain on the central server. This is especially important when the central server is not necessarily in the Cloud but perhaps a private data server.

The following statements can be made regarding scalability:

- Decentralization is good for scalability, especially if the central node is not a Cloud component, as the logic in the edge of the network decreases the workload for the central node.
- Some systems have a fixed number of devices or fixed number of users, making scalability less of a priority.
- Some solutions do not need to scale at all.
- Scalability in the IoT becomes more interesting when interoperability between solutions come into play.

Availability. One of the reasons a solution might become unavailable in the IoT is if the battery of the devices at the edge run out. Most of the solutions analyzed either plug in to a power outlet or have pretty long battery life. The shortest battery life among the solutions is that of the DAQRI smart helmet, which is estimated at one day.

The availability of solutions in the IoT can also be affected by connectivity issues. One way this is handled is by allowing the device to store its measurements locally and upload them when a reliable connection has been established. This is how the Farmobile solution handles this problem, where connectivity in the farm lands can be limited. This connectivity problem is also handled by the DAQRI smart helmet by allowing the device itself to process its data locally and store it, allowing for the majority of the functionality to remain available.

Resiliency. Most of the solutions have a central node, but it is the location of the logic and data (centralization) that determine how big of a single point of failure this central node is.

Take for example the DAQRI Smart Helmet, which receives its objectives and sends its data to a central server. Should this server fall, some functionalities will be temporarily unavailable. The Smart Helmet has enough logic, processing power and data storage to continue functioning while the central server has time to recover.

On the other hand, a system like the SmartThings environment is entirely dependent on the central SmartThings Cloud. Should this ever completely fail, the entire solution would become inoperable. However, since we are talking about the Cloud, downtime can be limited by providing active backup servers.

Decentralization can remove a single point of failure scenario, increasing overall resiliency of the solution.

Another way to increase resiliency in an IoT Solution is by adding multiple nodes that measure the same thing, this is referred to as replication. Take for example the WeatherCloud solution, which equips cars with sensors in order to measure the weather and road conditions. Should one or more of these cars fail, assuming there are many WeatherCloud enabled cars, an approximation of the weather can still be done with the data from the remaining cars. This is different for a Nest Thermostat solution, where if the thermostat fails the entire solution for that particular house is inoperable. The difference between those two solutions of course is the stakeholder to which the data is relevant.

Security. It is hard to say anything about security measures that the IoT Solutions use, because they are usually not open about this. This is to be expected as making the security protocols used public will make it much easier for an attacker to initiate an attack. You usually don't find much more than a statement saying "we use security algorithms present in online banking and large-scale web-apps". For this attribute not much more can be said.

Privacy.The IoT solutions exhibit various ways to provide privacy. One way is through data minimization, where the solutions claim only to collect data that is needed. An example is the Nomi Brickstream live, which is a camera that can identify faces so that a store can count how many people walk in and out of the store. However, they claim not to actually save the faces that are caught on camera. The Scanalytics floor sensors have the same goal as the Nomi, except that they only measure movement on the floor, which is guaranteed to be anonymous.

While both systems probably provide the same level of privacy for the same goal, the Scanalytics floor sensors provide a sense of trustability with regards to privacy due to the technology used.

Another example is the Array of Things, which also makes claims not to gather personal data of people in the city. They do however measure mobile device signals to provide an estimation of how busy certain parts of the city is.

One interesting IoT Solution with regards to privacy is the Homeseer. At the start of this thesis, I visited the Homeseer website. The main selling point of this system at that point was that it was privacy friendly by being Internet-independent. The Homeseer did not store any personal information in any Cloud service because it did not have one. This did come at a cost, because users could only control their home from inside. They could not control the home remotely via a mobile device, which is what the SmartThings environment provided.

However, roughly a month after I visited the website, it had been updated to include new features. The first feature was the possibility to control the Homeseer remotely via a mobile app. The second included the ability to communicate with the IFTTT webservice, allowing interoperability with other systems. This is an example of how the guarantee of privacy must be given up in order to achieve more functionality in the

IoT. This does not mean that the system does not provide privacy, it only means that the inherent trustability of the system due to the technology and networking used is no longer present.

The Homeseer remains the privacy-friendly alternative to the SmartThings, simply because the personal data is still not stored in the Cloud. The Cloud is simply used as a router between the mobile app and the Homeseer Hub. From this we can conclude that decentralizing data storage can provide more privacy, since most of the data is kept on a local network. There is no guarantee however that no data at all is kept on the Homeseer Cloud router, which makes privacy such a difficult issue. However, trusting companies with data without having a full guarantee what is done with it has always been a privacy issue even before the IoT.

An example of an IoT solution were privacy plays no role is the Zebra motionworks. The data that is being gathered is the game behavior of NFL players on the field. This is information that is already available to all people attending the game or watching at home.

Taking look at some of the privacy statements made by solutions, one that stood out is the Nymi band. For this solution, Privacy by Design[28] was used. Privacy by Design contains 7 objectives in order to ensure privacy in systems. These are the following:

- **Proactive not Reactive**: Anticipate and prevent privacy-invasive events before they happen.
- **Privacy as the Default Setting**: Maximum degree of privacy should be the defualt setting. Users don't have to do anything to be ensured their privacy.
- **Privacy Embedded into Design**: Privacy should be an integral part of the architecture and not an after-the-fact add-on.
- **Full Functionality**: Instead of considering trade-offs such as privacy vs. functionality, the system should be designed in a win-win manner where both can be achieved.
- **End-to-End Security**: Privacy by Design extends throughout the lifecycle of the data collected, where at the end the data is securely destroyed.
- **Visibility and Transparency**: Privacy is individually verifiable to all stakeholders.
- **Respect for User Privacy**: Design of a system should be user-centric with the privacy of the user always in mind.

[28] http://www.privacybydesign.ca

While it is possible to achieve more functionality while keeping privacy in the IoT, it is harder to ensure users that the situation remains privacy-friendly after such a connection has been made with another system. Take for example the Homeseer once more, where previously it was not possible for the system to be privacy-unfriendly because of the technology and design choices used, namely Internet-dependent and no Cloud component. The new version of the Homeseer could still function in a privacy-friendly way, however it is harder for users to trust this. This is where visibility and transparency play a role, allowing individual users to verify for themselves.

What we have learned about Privacy in the IoT Reality is the following:

- Decentralization of data storage is deemed privacy friendly.
- Privacy is promised by assuring the users that only necessary data is gathered and not personal data (data minimization).
- There is a trade-off between Privacy and Functionality in the IoT.
- There are some solutions where privacy does not play a role at all because of the nature of the data being gathered and processed.

Appendix C

Software Architectural Styles

This section contains a definition for and a list of software architectural styles that will be used for this thesis. The styles will be explained in short, since they are not new, and will be illustrated in a diagram. Some of the trade-offs mentioned in literature will be listed.

C.1 Introduction

The term software architecture refers to the process of making design decisions for a specific system to fulfill a certain business goal. The decisions are trade-offs between qualities that the system to be developed will have based on the requirements for this system. The decisions made on this abstract level of design primarily have an impact on the quality attributes[1] that the system will exhibit.

The Institute of Electrical and Electronics Engineers (IEEE) defines software architecture as being "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and the principles guiding its design and evolution". [CB11].

When talking about the architecture of a specific system, we abstract from implementation details and focus on the decisions that need to be made in order to achieve certain qualities for that system. As stated in the definition above, these design decisions relate to how the system is decomposed into components and how they interact with each other. When the same decisions are continuously made for the same type of systems to achieve similar goals with success, patterns start to become visible. These patterns are known as software architectural styles.

The software architecture is an abstraction of a system, however it remains specific for that particular system. A software architectural style is an abstraction of software

[1] Quality attributes are non-functional requirements.

architecture, i.e. it abstracts from a specific system and instead looks at patterns of design decisions that could be used in various types of systems in different domains. A software architecture can be an instantiation of one or multiple software architectural styles. The remainder of this chapter gives a definition of software architectural styles and provides a list of the styles that will be used in this thesis.

C.2 Definition

Software architectural styles can be seen as guidelines for making architectural design decisions regarding the decomposition of a system and how the components may interact. They are based on years of experience in the field of software architecture, extracted from successful systems.

Another definition for a software architectural style is a "coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style" [Fie00].

From this definition we can derive that software architectural styles contain restrictions on what an element in the architecture can do but also which other elements it can interact with.

The decomposition of a system should begin based on design decisions that are likely to change [Par72]. Inherent in all styles is the increase of evolvability of the system due to the decoupling of components in different ways. Decomposing the system into components and having them not be entirely dependent on each other makes it possible for certain components to be changed without affecting the entire system.

A Software Architectural Style is a labeled collection of architectural design decisions that are applicable to a particular context, constraining development within that context, and yielding beneficial qualities [Tay13]. These styles can be used to design the software architecture of a specific system. Software architectures are unique to the system they represent. There are an infinite amount of unique software architectures (as many as there are systems) while there are a finite amount of commonly used and well-defined software architectural styles. Figure C.1 shows the design process using an architectural style.

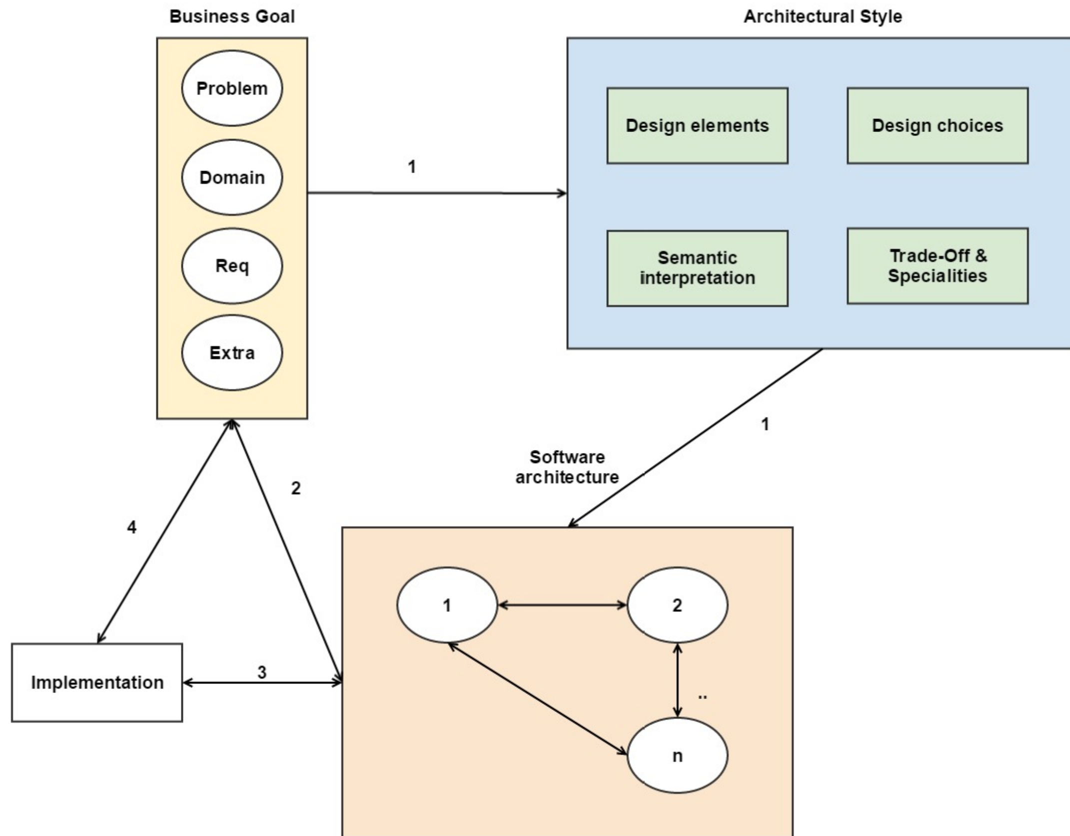

FIGURE C.1: Software Architecture Design Process Life Cycle

On the top left side we have a block containing the current business goal. This business goal is usually provided by a client. At the heart of the business goal is a problem. A problem is defined as the difference between the current state and a desired state[CB11]. Some problems are not completely new. There might already be some knowledge on how to solve some parts of the problem. This knowledge can be considered domain knowledge. Every software development situation has its set of requirements made by the client. Besides these three properties mentioned above, there are also other factors that should be considered when analysing a situation such as the stakeholders and the market. Who are the people that are going to be using this product? What is the current trend on the market? There are many other factors you could think of, but this is out of the scope of this thesis. For now, these will be represented as “Extra”. Each situation is unique and will therefore have to be analyzed in order to choose the right architectural style to use.

An architectural style consists of four parts. The first part is the choice of Design Elements. It is possible to describe software architecture as a set of components and connectors according to [GS94]. The choices for which components and connectors are used in a style are the design elements. An example is a Client-Server style, where the components are a Client and a Server and the connectors are requests and responses.

However, for some styles it is not always clear what the components or connectors are, so it is much easier to simply refer to them as design elements or the vocabulary of a style.

The second part are the design choices. The design choices are a set of configuration rules, topological constraints that determine allowed composition of design elements. An example of such a constraint is that a component may be connected to at most two other components.

The third part is semantic interpretation, which means that the composition of design elements must have well-defined meanings. What does it mean for a class to implement an interface? What does it mean for a particular component to be connected to another specific component? Does it mean the first component uses the second component?

The last part contains the trade-offs and specialties. Each Architectural style has its advantages, disadvantages and which types of systems are recommended to use this style. These trade-offs relate to the quality attributes gained and lost when using a particular style. It is possible to use different styles to create the same functionality, however the measure in which quality attributes such as evolvability, interoperability and performance efficiency are provided depend on the software architecture and thus by the styles used (or not used).

Finally, when the first version of the software architecture is complete, an implementation based on the architecture can be developed. The implementation can then be tested against the requirements present in the business goal. If it solves the problems and conforms to the requirements, then the process is done. If this is not the case, the implementation, business goal and architecture can be iteratively reviewed and changed to reach a situation that is desirable for the client.

The resulting software architecture of this process adheres to a software architectural style. This software architecture can be documented in many ways and on many views such as the logical, process, development and physical view [Kru]. Often software architecture is documented as a line and box diagram, however it is not always clear what the lines and boxes represent. The views are used to make this more clear and the documentation of a system can contain a diagram for each view. Software architectural styles live in the logical view, however in this thesis the physical view will also be considered as the IoT is distributed and the mapping of software components to hardware is essential in achieving certain quality attributes.

While there are not that many well defined styles to begin with, choosing one over the other can have many consequences that will remain inherent in the architecture of the system. Choosing the right starting point for the architecture of a system is the goal of software architectural styles.

C.3 Styles

As mentioned in the previous section, there is no ISO or IEEE standard for software architectural styles. The most referenced to paper describing this topic is "An Introduction to Software Architecture" by David Garlan and Mary Shaw [GS94], however this was written in 1994. The book "Software Architecture in Practice: Third Edition" is a more recently written book that covers all of the styles mentioned by Shaw and Garlan and even includes newer styles [Bas]. This book refers to styles as architectural patterns, both of which are usually used synonymously in literature. These two references will serve as the main sources for the architectural style description, however other papers will be used as support.

It is important to note that this list is not complete. It can be argued that simply by adding or removing a constraint, a new style can be formed. Instead the focus will be on the most commonly used and mentioned styles in literature. An example of a style that is left out is Microservices[2], as it is relatively new and not yet well documented. Furthermore, it is my opinion that this style can be instantiated using the Service-Oriented style. There are probably more styles, however in this paper we will focus on the following styles:

- Client-Server
- Peer-to-Peer
- Pipes and Filters
- Event-Based
- Publish-Subscribe
- Service-Oriented
- REST
- Layered
- Microkernel

This list of styles contains enough variety in order to illustrate the importance of making the first architectural design step for IoT solutions, namely choosing a style. The choice of style will remain embedded into the architecture as it evolves, meaning that making the wrong decision early in the design phase can lead to unwanted side-effects in the implementation of the system.

The styles will be cataloged using the following template:

[2] http://martinfowler.com/articles/microservices.html

Item	**Description**
Name(s) and Abbreviation	The name and abbreviation of the style. Some styles might have multiple labels referencing to it
Reference(s)	The references used to describe this style
Overview	A general description of the style
Design Elements	The vocabulary used in this style. The names used for the types of components and connectors.
Constraints	The type of constraints, such as how components are connected and what they are allowed to do.
Trade-offs	The general effect the style has on quality attributes of a system implementing that style. Note that not all effects on the quality attributes we are interested in are documented. This will have to be analyzed separately. Furthermore, there is no guarantee that these trade-offs will hold, as these depend on the context of the application being designed.

C.3.1 Client-Server (CS)

References:

- "An introduction to Software Architecture" [GS94]
- "Software Architecture in Practice" [Bas]
- "Software Architectural Styles: An Introduction" [MJ]"
- "Architectural Styles and the Design of Network-based Software Architectures" [Fie00]

The Client-Server architectural style is a very common and well known pattern for network-based applications . One or multiple Client components initiate a request to a Server, which then performs some computations and responds to the Clients. This is different from the Event-Based and Streaming invocations we have seen so far, as the Client decides itself when to initiate a request.

One common example of a Client-Server application is a website, where the web browser acts as a client and the website is itself is hosted on a server. The Browser initiates a request for a website to the Server, which responds with the data necessary to illustrate the website in the browser.

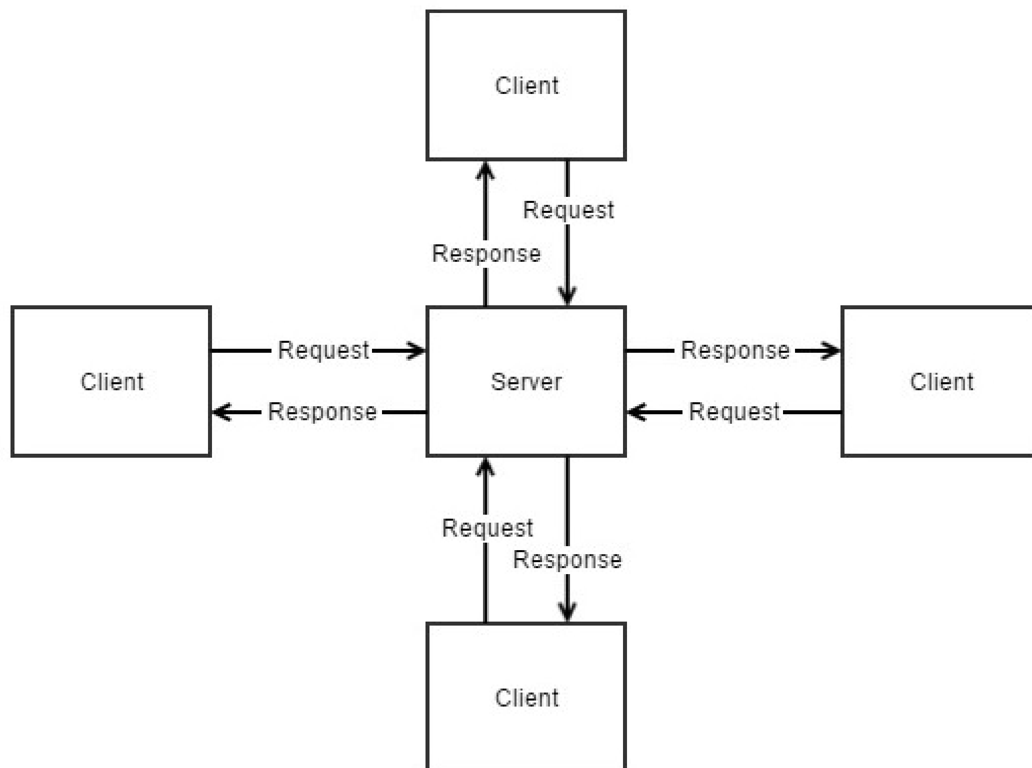

FIGURE C.2: Client-Server Architectural Style

Design Elements

- **Client**: A client is a component that invokes services of a server component.
- **Server**: The server component provides services to clients.

Constraints

- Only clients can initiate communication, servers only respond to requests from clients.
- Server components can be clients to other servers.
- Clients cannot communicate to each other.

Advantages

- Evolvability: Common services only need to be modified in a single location.
- Availability/Scalability: Centralizing control of resources allows for distribution among physical servers.

Disadvantages

- Performance: The server can be a bottleneck, since all communications go through it.
- Availability: The server is a single point of failure.
- Complexity: For some applications it can be hard to decide where to put functionality.

C.3.2 Peer-To-Peer(P2P)

References:

- "An introduction to Software Architecture" [GS94]
- "Software Architecture in Practice" [Bas]
- "Software Architectural Styles: An Introduction" [MJ]"

Considered to be the opposite of the Client-Server style, the Peer-to-Peer (P2P) style contains components which all have equivalent capabilities and responsibilities. All peers are producers and consumers of data and functionality.

Design Elements. The only design element in this style are peers. Peers are independent components running on a network.

Constraints

- Peers connect to each other directly, no component that processes data in between.
- Peers are both providers and consumers of data and processing power.
- The primary content is provided by peers, there are no central components providing content. There may however be a central component providing information regarding peers (Centralized peer-to-peer).
- Peers are autonomous and control their own activities. They are not controlled by a central component.
- Peers can be added and removed from the system at any time.

Advantages:

- Performance: Response-time only depends on connection to peer instead of connection of both peers to a central server.

- Availability: If a peer should fall, the system would still continue to function although the data on that peer would be gone.
- Availability; If a peer is not available, another peer can pick up the workload.
- Scalability; The system can be scaled simply by adding more peers. The supply and the demand of resources is increased by adding peers.

Disadvantage

- Managing Security and Availability is much more complex.
- The performance and availability gains depend if the network is large enough.

C.3.3 Pipes and Filters

References:

- "An introduction to Software Architecture" [GS94]
- "Software Architecture in Practice" [Bas]
- "Architectural Styles and the Design of Network-based Software Architectures" [Fie00]

A Pipes and Filters style contains components that are called filters because they transform the data received from another filter, and output this transformed data to the following filter in the chain [GS94]. Filters are connected to each other by Pipes, which are routers for the data stream. Multiple instances of a filter can be run if a certain filter has more workload. Filters can be reused in other systems where applicable as illustrated in figure C.3.

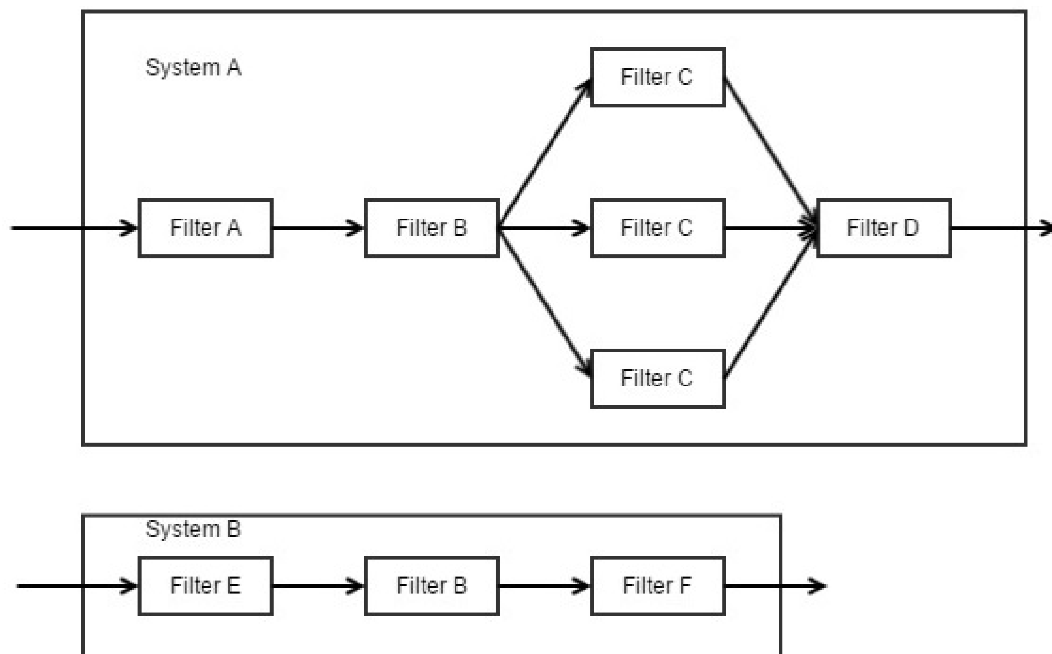

FIGURE C.3: Pipes and Filters

Design Elements

- **Filter**: A Filter is a component that transforms the data received from its input port to its output port.
- **Pipes**: A Pipe is the connector that transports data from a filter's output port to another filter's input port. The Pipe does not transform the data.

Constraints

- Filters connected by a pipe must agree on the type of data being passed along that pipe.
- Filters are independent of each other and do not share a state.
- Filters can execute in parallel.
- Filters start to producing output as soon as they start processing input.

Advantages:

- Understandability: The entire system can be considered a composition of the behaviors of the individual filters.

- Reusability: Any two filters can be hooked together given that they agree on the data being transmitted. A filter can also be reused in another system completely.
- Evolvability: New filters can be added andold filters can be replaced since they are all independent.
- Correctness: Throughput and Deadlock analysis is made possible.
- Scalability: It is possible to individually scale certain filters by running more instances in parallel.

Disadvantages:

- Interactivity: Due to the batch organization of processing, interactivity cannot be handled well.
- Performance: System may have to wait for parallel instances of a filter to finish processing the data.
- Performance: Certain implementations may force a lowest common denominator on the transmission of the data, which may result in more work for some filters.
- Performance: Each extra filter adds to overall latency.
- Availability: If a filter in the chain of filters fail, it usually leads to failure of the system (unless there are multiple instances of the filter running).

C.3.4 Event-Based, Event-Driven, Event-Oriented

- "An introduction to Software Architecture" [GS94]
- "Software Architecture in Practice" [Bas]
- "Software Architectural Styles: An Introduction" [MJ]"
- "Architectural Styles and the Design of Network-based Software Architectures" [Fie00]

The Event-Based software architectural style is based on implicit invocation. As opposed to explicit invocation, where procedures or methods are called, in implicit invocation they are induced when a certain event takes place. The Event-Based architectural styles have evolved through the years, with the earliest version specialized for applications running on a single node. The components in this version were an Event Source and Event processors, where each processor would register an interest in a specific event which was produced by a source[GS94]. The common uses were GUI programs, where the system reacted to clicks and performed asynchronous tasks, or debuggers that invoked methods

such as scrolling to the source line of the breakpoint and showing the values of the variables.

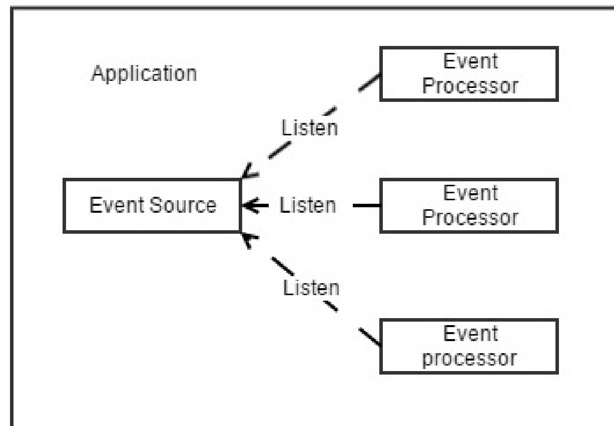

FIGURE C.4: Early Event-Based Style

Figure C.4 illustrates the first version of Event-based style. The Event source and processors are all running in one application, when an event takes place in the source, it triggers functions located inside the event processors.

The Event-Based style has since been adapted to be used in distributed systems. The connector between component has been changed, now called the Event-Bus which is in charge of sending the event to all listeners over the network [MJ]. This becomes necessary as nodes that are interested in a certain event are no longer running on the same hardware, meaning that some administration and networking has to occur.

This type of Event-Based style has been recommended for real-time systems that need to react to external interaction and have distributed sensors. Some examples include activity monitoring and early warning systems.

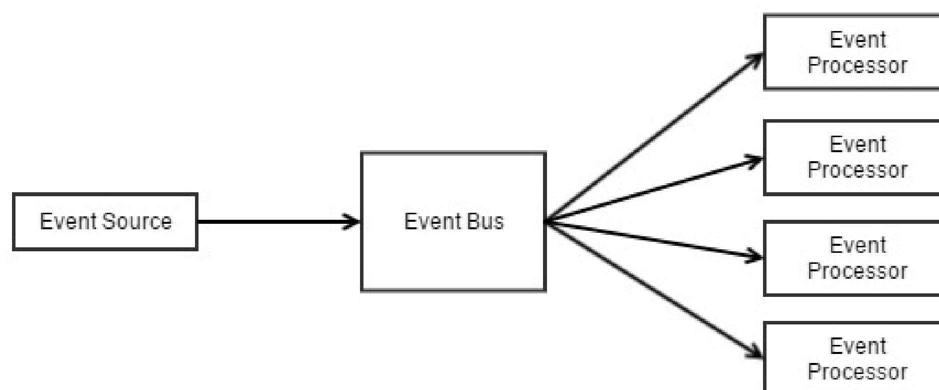

FIGURE C.5: Event Based Style with Event Bus

Figure C.5 illustrates an example of an event-based system with an Event Bus. The Event Bus and Event Source can run in the same application, however it is also possible to have an Event-Bus run on middleware. Because the nodes are distributed across a network, the event is pushed to the Event Processors rather than immediately reacting to the event. Furthermore, the distributiveness makes brings the necessity to sometimes also send data along with the event. Previously the event processors and the source shared a state, which is not possible in a distributed system unless they all share a database for example.

This type of event-based style comes close to the Publish-Subscribe style, however in that style the consumer also does not know the identity of the source of the event and is instead only interested in a type of event.

Design Elements

- **Event Source**: This component posts an event to the Event-Bus.
- **Event Bus**: The Event Bus is responsible for keeping track of all listeners of the event and notifying them when that event has happened.
- **Event Processor**: Registers to the Event bus and receives a notification when this event happens.

Constraints

- The event source does not know the identity of the event processors.
- Event processors are independent of each other.

Advantages

- Reusability: A component can be used in any other system by registering to its event.
- Evolvability: Components can evolve or be replaced without affecting the event-source or other event-processors.
- Performance: The ability to perform tasks in parallel lend to performance.
- Scalability: Each event processor can be scaled independently. The Event Bus only needs to relay data and notify listeners. However, the number of connections possible depends on the Event-Bus.
- Efficiency: For applications dominated by data monitoring rather than retrieval, this style can increase efficiency by removing the need for polling.

Disadvantages

- Testability: It is generally harder to recreate some events and the asynchronous characteristics also make testing harder.
- Correctness: Due to the asynchronous nature, it is hard to reason about termination and correctness of the end state.
- Performance: The Event Bus is an intermediary and adds to overall latency. For one-to-one communication this style is not useful.
- Availability: The Event Service is a single point of failure.

C.3.5 Publish-Subscribe

- "Software Architecture in Practice" [Bas]
- "The many faces of publish/subscribe" [EK03].

The Publish-Subscribe software architectural style contains publishers that publish data to a central Event Service. At the same time, interested parties called Subscribers can subscribe to events via this Event Service. The Event Service is in charge of matching the published data with the corresponding need for it. There are three variants of this style. These are Topic-based, Content-based and Type-based.

In Topic-based Publish-Subscribe, events are grouped into topics that are identified by keywords. This variant might not be flexible enough for some types of systems. The Content-based variant allows events to be classified based on their properties and not by some predefined grouping. In this variant a subscription language is provided in order to allow subscribers to express in detail which events they would like to subscribe to. Type-based requires events published to have a well defined type., which allows type safety at compile-time.

While it shares some similarities with the Event-Based architectural style, there is a big difference. The subscribers are all interested in a type of event happening without knowing the publisher of the event. In Event-based, the subscribers are interested in a particular source of an event.

What this means is that a subscriber might get multiple notifications for one request, coming from multiple publishers if the implementation allows it.

The Publish-Subscribe style is developed to provide decoupling between event sources and event consumers in systems that are large in scale and have heterogeneous systems. In this style, it is possible to show an interest in the weather at my location for example

while not really being interested who provides this information. Figure C.6 illustrates the Publish-Subscribe style.

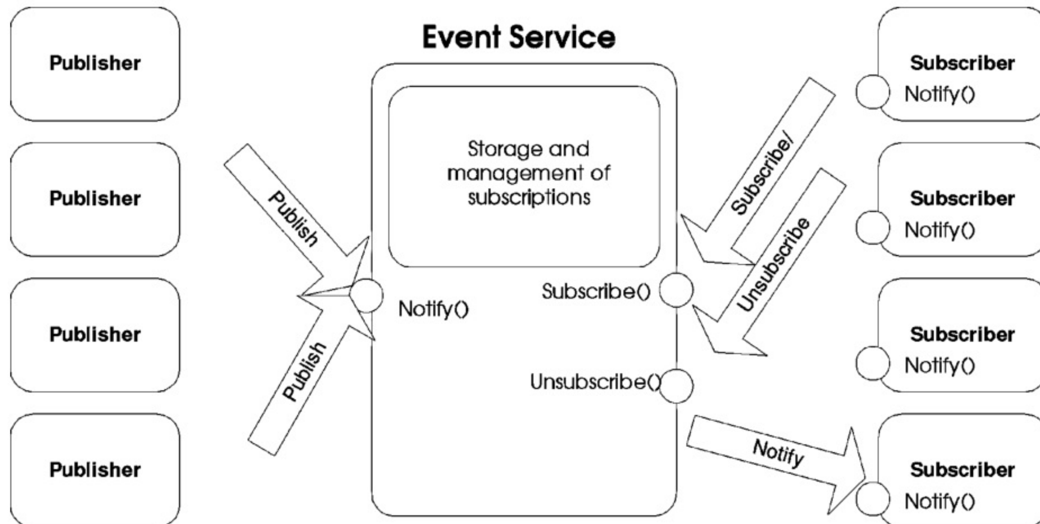

FIGURE C.6: Publish-Subscribe software architectural style[EK03]

Design Elements

- Publisher: The producer of data. It pushes its data to the event service.
- Subscribe: The consumer of data. It registers its interest in a particular type of data and receives a notification from the event service when this type of data arrives.
- Event Service: Contains the storage and management logic for subscriptions. It is responsible for notifying subscribers when an data of interest arrives.

Constraints

- All publishers and subscribers are connected to the event service.
- A component can be both a publisher and a subscriber.

Advantages:

- Evolvability: Publishers and Subscribers are loosely coupled and thus allow for independent evolution.

Disadvantages:

- Performance: The event service is an extra layer and adds to overall latency. The Event Service increases latency more than the Event Bus for the Event-Based style since it also has to be able to classify data and understand complex queries.
- Correctness: Due to the asynchronous nature, it is hard to reason about termination and correctness of the end state.
- Availability: The Event Service is a single point of failure.

C.3.6 Service-Oriented

- "Software Architecture in Practice" [Bas]

The Service-Oriented Architecture (SOA) style contains services that are decoupled and have defined interfaces which can be combined to form distributed applications. Services can communicate to each other regardless of the platform due to the platform-independent interfaces. This makes this style very suitable for an environment containing heterogeneous subsystems.

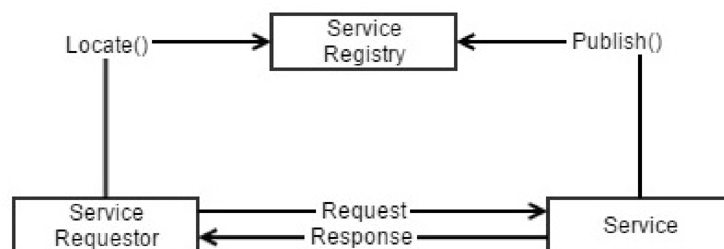

Figure C.7: Service-Oriented Architectural Style

Figure C.7 illustrates the types of components in Service-Oriented Architecture. First of all, there is the service which provides a set of functionalities. A service publishes its description to the service registry. The service registry keeps a list of all services, their locations and their functionality. When a service requestor is in need of a service, it first consults the registry returns the information needed to locate and communicate with the service. From this point on the service requestor and the service can communicate, regardless of the platforms they are running on, because the communication is standardized.

The service requestor can also be a service itself. In this case the service uses the registry to find one or more services to complement its own functionality.

Design Elements

- Service Registry: Keeps a record of all registered service containing their descriptions as well as their interfaces.
- Service Consumer/Requestor: The component in need of functionality from another service.
- Service Provider: A component that provides functionality.

Constraints

- All services are independent, meaning they do not share a state and do not have to know the identity of each other.
- Service providers need to register their services to the service registry.

Advantages:

- Interoperability: The registry component allows for lookup of services and also knowledge about their interfaces. Orchestration services can be used to script and translate interactions between multiple services.
- Evolvability: Services are completely decoupled.

Disadvantages:

- Complexity: Service-Oriented Architectures are typically complex to build.
- Loss of control: The evolution of independent services cannot be controlled.

C.3.7 REST

- "Architectural styles and the design of network-based software architectures" [Fie00]

The REST architectural style is basically the same as the Client-Server style except that the components in the server role have a uniform interface, which contains the verbs GET, SET, POST and DELETE and a uniform addressing scheme. The trade-off between using this style over the Client-Server style is that the uniform interface increases evolvability by decoupling the components and interoperability as the interface is known throughout the system. The disadvantage is decreased performance as the uniform interface forces information to be changed into a standardized form.

The constraints of caching and stateless communication is considered as constraints for this style, however for this thesis I have chosen to view this as implementation details and not inherent in the style itself.

C.3.8 Layered

- "An introduction to Software Architecture" [GS94]
- "Software Architecture in Practice" [Bas]
- "Architectural Styles and the Design of Network-based Software Architectures" [Fie00]

The Layered software architectural style allows the developer to partition a system into layers which can be developed and scaled independently [GS94]. Generally, a layer only communicates with adjacent layers. Each layer provides an interface for the layer "above it" (or "below it", it varies). As long as this interface remains constant, the implementation of the layer can continue to evolve without causing much change to the entire system. In some layered systems it is allowed to skip certain layers. This is usually due to performance reasons.

The layered style has been used for communication protocols, database systems, operating systems and Java EE applications. An interesting characteristic of a layered style is that it can be used in combination with many other styles, which is not a statement that holds for all styles. The layered style is also popular because it allows a mapping of layers to organizational structure, where specific team of developers work on layers that correspond to their expertise.

One consideration to make when using the layered architectural style is to avoid the "sinkhole anti-pattern", which is when a majority of the requests simply pass through all of the layers without any processing being done in the layers[Ric15]. All systems will have some processes that fall into this anti-pattern, a good rule is to have less than 20 percent of the requests be a simple pass through.

Figure C.8 shows an example of a common implementation of the layered architectural style, where the system is decomposed into a presentation layer for UI, a Business layer that handles the business logic, a Presentation layer which interfaces with the database and the database layer that contains the database.

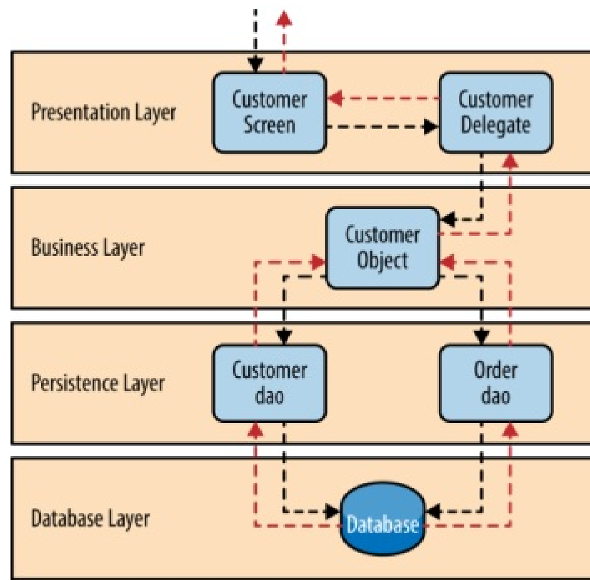

FIGURE C.8: Layered Architecture example [Ric15]

Design Elements. The only design elements are layers, which can contain multiple components.

Constraints

- Every piece of software is part of exactly one layer.
- There are at least two layers.
- A lower layer cannot "use" an upper layer.

Advantages:

- Evolvability: Layers chan change as long as their interfaces remain the same.
- Reusability: Layers can be reused in other systems.
- Security: Layers can be put in place to authenticate connections before allowing anything to change in sensitive areas of the system.
- Testability: Other layers can be mocked in order to test specific layers.

Disadvantages:

- Performance: Multiple layers need to be traversed in order to fulfill a business request.
- Suitability: Not all systems can be modeled in a layered fashion.
- Resiliency: Should one layer fail, the entire system becomes inoperabl

C.3.9 Microkernel

- "Software Architecture Patterns: Understanding common architecture patterns and when to use them" [Ric15]

The Microkernel architectural style is commonly used for product-based applications. A product-based application is packaged and made available for download in versions. The main goal is to provide a set of basic functionalities, located in the microkernel component and a set of added functionalities in plugins. Figure C.9 illustrates this style.

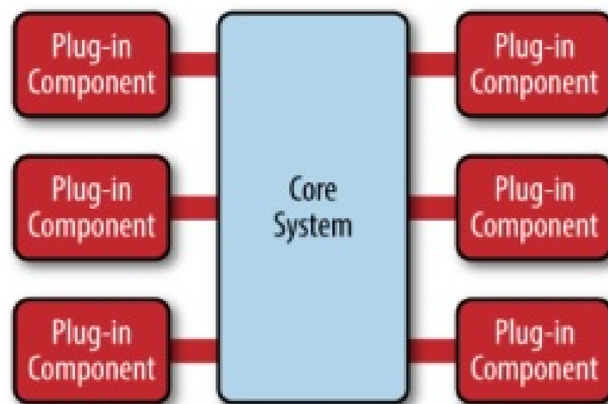

FIGURE C.9: Microkernel Style [Ric15]

The advantage of using this style is allowing every user of the product to only download the plugins they will be using instead of providing a complete set of functionalities from the very start. An example is the Eclipse IDE, where users can choose to download whichever plugins they want to add functionality that they need.

Design Elements.

- Microkernel: The component containing the minimum functionality needed to run the system.
- Plugin: A set of functionalities that is called by the microkernel.

Constraints

- Plugins are independent and cannot communicate to each other.

Advantages:

- Evolvability: Plugins can evolve independently without affecting each other.

- Performance: With respect to memory used, since the user can choose to only download and install plugins that they need.

Disadvantages:

- Complexity: It is for some cases hard to decide what the bare minimum functionality is and to predict which types of plugins will come in the future.
- Performance: Depending on the context, the plugins might run on other processors or even other components in the network, increasing latency.

Appendix D

Software Architecture Evaluation Methods

In order to reason about software architecture for the different IoT classes discerned, there must be a way to evaluate the architecture that results when using a certain architectural style. This chapter discusses software architecture evaluation methods and to which degree they can be used in this thesis.

D.1 Evaluation Methods

Software architecture evaluation methods provide a way of analyzing architectures for their quality attribute fulfillment. They can focus on one attribute or provide a general method for analyzing multiple attributes. These methods have been collected and compared in a paper [Mau10], providing a summary of all the methods. Half of these methods require a high-level implementation or prototype to be used. These were omitted because we only have abstract classes and lack the information that is needed in order to build such a model.

Three examples of methods are briefly discussed, however these cannot be directly used for this thesis but are used as inspiration for the mapping and analysis moving forward.

Attribute-Based Architectural Styles (ABAS). Attribute-Based Architectural Styles are extension of architectural styles based on a specific need for one particular quality attribute [KL]. With this concept they try to move away from the ad-hoc method for choosing styles, such as "use the pipe and filter style when reuse is desired and performance is not a top priority" to an approach that is based on reason. The name of this method seemed promising, however what it does is give a model to reason about further development of a style given a quality attribute as a priority. An example is

how to reason about performance in a Pipes-and-Filter style, where latency is measurable response. However, only limited examples are given and does not make clear how you could do this for other styles. The Pipes-and-Filter style is commonly used as an example to illustrate such techniques because one of its inherent advantages is that it provides the ability to reason about the behavior of the system as a composition of functions (filters). Furthermore, we are interested in the difference between choosing styles as a starting point and not in how to further develop styles.

Architecture Trade-off Analysis Method (ATAM). This evaluation method is an evolution of the Software Architecture Analysis Method (SAAM) which contained five steps, by adding a final step that shows which components in an architecture influence quality attributes[KC]. The paper introducing this method argues that quality attributes are typically evaluated in isolation, while in reality they interact with each other. The examples of trade-offs given are between performance and modifiability, availability and safety, and security and performance. The ATAM can be used as inspiration for asking the right questions as well as an example of how to measure performance an availability of software architectures, however it is developed to compare competing architectures that closely resemble each other. The only example given is where a Client-Server style is already chosen and they reason about the effects of adding an extra server.

Architectural-Level Modifiability Analysis (ALMA). Studies have shown that about fifty to seventy percent of the lifecycle cost for a software system is spent on evolving the system. The aim of this method is to address modifiability at the design phase [BvV]. This method provides a straightforward way of reasoning about modifiability of a system, by describing the changes that are likely to occur and determining which components need to make a change. The goal is to encapsulate change to one component or at least reduce the ripple effect of changes. These ripple effects can occur when:

- If components need to know about the existence and names of other components
- If assumptions are made about the type of data and their relationship. A change might cause a failure in other components that make these assumptions.
- If assumptions are made about the meaning of data. An example if is the temperature data received is in Fahrenheit or Celsius.
- Internal representation of data. The example given is an array versus a linked list, however I believe the first three points to be more relevant for this thesis.

ALMA can be used for this thesis as it is relatively simple compared to the other solutions and allow for a comparison to be done between architectural styles with respect to modifiability. However, in the paper they mention that the hardest part is the collection of change scenarios, which is usually done by interviewing many stakeholders. ALMA

is used in this thesis by providing a set of likely changes per class and evaluating how this change will affect the entire system.

D.2 Scenario-based modeling and architectural design tactics

One way of evaluating how an architecture can fulfill certain quality attribute needs is to use a scenario model, which is used in all of the above mentioned methods. In this model there are three main concepts. The first is the *stimulus*, which is an event to which the software architecture must provide a *response* to, which is the second concept. The response must be measureable. Performance is measured in time (response time) for example. Mediating between the stimulus and the response are *architectural decisions*, which have an effect on the response in some way. These architectural decisions are also referred to as design tactics. Each quality attribute has one or more general scenarios. However, upon closer inspection of the examples given it becomes clear that some of these tactics and responses take place on a lower level of abstraction than we are trying to analyze in this thesis, i.e. the initial choice of software architectural style. The choice of scheduling policy for processes can be done after any initial software architectural style has been chosen for example.

A list of general tactics is provided in [Bas], however I found only a subset of these tactics to be useful for this analysis. This because they are present in some of the styles or that the choice of style might restrict the use of certain tactics. The tactics I found relevant are:

- **Interoperability**:
  - Discovery Component: A component that provides the ability of discovering services in the system and providing a description of how to interact with them.
  - Orchestration Component: A component that scripts the coordination between multiple other components which can be based on a task that needs to be fulfilled.
  - Uniform Interface: Require all participating systems to have the same structure for their interfaces.
- **Evolvability**:
  - Reduce large components into smaller ones: This can lead to a reduce cost per change. This is inherent in all styles through the initial decomposition into design elements.

- Increase semantic coherence: If two responsibilities in the same component serve different purposes, then they should not be in the same component.
- Encapsulation: Provide an explicit interface to a component (API). Interfaces should abstract from implementation that is likely to change, in order to reduce coupling.
- Use an intermediary: Intermediate components can decouple components from each other, such as the event-service in a publish subscribe style, a shared data repository or a lookup service in SOA.
- Abstract common services: If two components provide very similar functionality, it might be better to move this into a single component.

- **Performance**:
  - Manage the sampling rate: There is a trade-off between having high fidelity data and more predictable levels of latency. By choosing to have a reduced sampling rate it is possible to lose less messages along the way, at the cost of less precise data. This is a decision that usually has to be made in signal processing.
  - Prioritize Events: Certain incoming events might have a higher priority due to several reasons, one example being safety. In this case it might be better to process the high priority tasks first and ignore lower ones if there is not enough processing power to perform all tasks. A Pipe-and-Filter style has a built in FIFO queue, which means that the first message that comes in is the first that gets processed.
  - Reduce Overhead: It is possible to reduce overhead by removing intermediaries, co-locating resources, perform periodic cleanup of resources and running single-threaded servers and split workload across them. The first choice is an example of the trade-off between modifiability and performance.
  - Introduce Concurrency: Process requests in parallel on different threads to reduce blocked time.
  - Maintain Multiple Copies of Computations: An example is to have multiple servers in a client-server system to reduce contention. A load balancer is used to assign work to the different servers. In a Peer-to-Peer system, all nodes are copies of computations.
  - Maintain Multiple Copies of Data: An example is caching, which keeps a copy of data on storage with different access speeds.Data replication is when multiple copies of data to reduce contention, however it becomes a responsibility of the system to keep these copies consistent.
  - Schedule Resources: When there is contention for a resource, this needs to be scheduled. This is true for processors, buffers and networks. Each need to be individually analyzed to choose the correct scheduling policy.

- **Availability**:
  - Monitor component: A component that monitors the state of health of various parts of the system. In the IoT this could be done for sensors and actuators connected to the network.
  - Voting: When having redundant components that all receive the same input but do not give the same output, the system must have a voting mechanism such as majority rules. The components can be exact clones, functional redundancy (must give the same output to input but are differently designed) and analytically redundant (components have the same goal but perform calculations in different ways).
  - Predictive modeling: Use a predictive model when certain information sources are not available by looking at the history of data. This can be done in the case of sensor failure in the IoT.
- **Security**: For security I did not find any of the tactics useful enough to mention. The standard "identify, authenticate, authorization levels, use encryption, hash and checksums" are details that can be added after any choice of style. However it should be noted that there is a trade-off between security and performance. The usage of security algorithms and tactics increases latency as the complexity of the tasks increase.

The IoT-A also lists a set of tactics specifically meant for the IoT, however there are no mayor changes to what is already present for the general case [BM]. In some cases they mention very vague tactics such as "make design compromises", which is the entire point of designing a system.

While these tactics give a good example of what to look for when reasoning about quality attributes, they will not be strictly used for the remainder of this analysis. They will be referred to in specific situations when it helps illustrate a point trying to be made.

The software architecture evaluation methods reviewed all show how it is possible to measure an architecture for performance, modifiability, availability and to some extent security. However, in order to do this completely as proposed by these methods, you would need more information about the system to be made and the architecture suggested. Because we are dealing with classes of solutions and architectural styles, it is not a complete fit. However, the scenario based method and the quality attribute models can be used as reference for reasoning about the effect of architectural styles[Bas].

What these methods have in common is that they try to compare architectures that only differ in a small way. An example is a Client-Server architecture where one variant only has one server and the other has multiple servers. What this thesis is trying to show is the difference between architectural styles, which is the starting point of the architecture. These styles differ so much from each other, that we do not need absolute metrics in

order to reason about their applicability. Instead we just need a good understanding of what software architecture is, what the constraints of the styles are, a definition of IoT classes and what quality attributes are and how they can be measured.

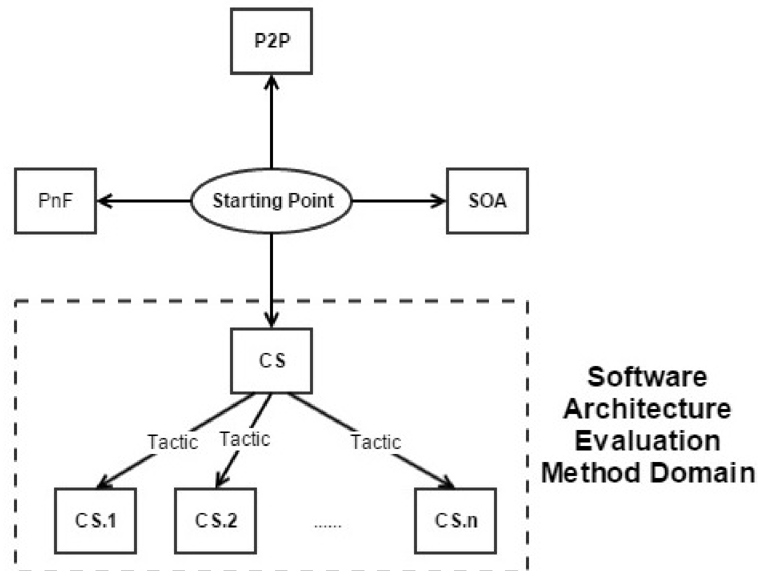

FIGURE D.1: Thesis domain vs. Software Evaluation Methods Domain

Figure D.1 illustrates the different between the domain of this thesis and that of software architecture evaluation methods. The starting point indicates the phase where no decision regarding the architecture has been made. From that point on, a style can be chosen to build to architecture upon. In this example four styles are illustrated. Regardless of the style chosen, many design decisions (tactics) still need to be made before reaching the first instantiation of the software architecture. It is at this level that software architecture evaluation methods play a role. However, for this thesis we are interested in showing which path should be taken from the starting point in the context of several classes of IoT solutions.

Appendix E

Discussion Panel

On the 18th of May 2015 I held a panel to discuss my project. The meeting consisted of a presentation of the project so far, followed by an interactive session where we discussed several IoT solutions and styles. Attendees included Geert Schuring, Hans Gringhuis, Angelo van der Sijpt and myself. The meeting went on for two hours.

The presentation contained the following information:

- My background
- The research question
- The motivation behind the project
- An explanation of the IoT Vision (Definition and requirements)
- An explanation of the IoT Reality
- The difference between the IoT Vision and Reality
- An explanation of the IoT Illusion
- The scope of the project (Which styles and IoT solutions)
- The goals of the project (Expected results)

The following points were discussed:

- The participants think that the topic is interesting and relevant. It is important to have this view on the Internet of Things.
- Scalability requirement for solutions should be estimated for individual components and not entire system. Furthermore, something can be said for scalability by checking to see if the number of users or number of devices is fixed.
- Decentralization means less bandwidth use, more autonomous behavior

- Centralization means less energy consumption, devices cost less
- Think about how sensors transmit data, streaming vs iteratively vs event-based.
- Styles can be combined, so look at styles for individual communications instead of the entire system.
- Gather more information on sensors, how much they cost, battery life etc.

Bibliography

[Add14] Ahamed S.I. Yau S.S. Buduru A. Addo, I.D. Reference architectures for privacy preservation in cloud-based iot applications. *International Journal of Services Computing*, 2:65–78, 2014.

[AG95] Allen R. Abowd, G. D. and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology(TOSEM)*, 4:319–364, 1995.

[AM10] Iera A. Atzori, L. and G. Morabito. The internet of things: A survey. *Computer networks*, 54:2787–2805, 2010.

[Axe09] W. Axelrod. Investing in software resiliency. *The Journal of Defense Software Engineering*, pages 20–25, 2009.

[BA12] Milito R. Zhu J. Bonomi, F. and S. Addepalli. Fog computing and its role in the internet of things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[Bas] Clements P. Kazman R. Bass, L. *Software Architecture in Practice: Third Edition.*

[BM] Bauer M. Fiedler M. Kramp T. Van Kranenburg R. Lange S. Bassi, A. and S. Messner. *Enabling things to talk.*

[BS11] D. Bandyopadhyay and J. Sen. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, 58:46–69, 2011.

[BvV] Lassing N. Bosch J. Bengtsson, P. and H. van Vliet. Architecture-level modifiability analysis (alma). *Journal of Systems and Software.*

[CB06] S. Ciraci and P. Broek. Evolvability as a quality attribute of software architectures. 2006.

[CB11] J. O. Coplien and G Bjørnvig. *Lean architecture: for agile software development.* John Wiley & Sons, 2011.

[EK03] Felber P. A. Guerraoui R. Eugster, P. T. and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35:114–131, 2003.

[Fie00] R. T. Fielding. Architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California, Irvine*, 2000.

[GP13] Buyya R. Marusic S. Gubbi, J. and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29:1645–1660, 2013.

[GS94] D. Garlan and M Shaw. *An introduction to software architecture.* 1994.

[Hoe12] J. H. Hoepman. In things we trust? towards trustability in the internet of things. *Constructing Ambient Intelligence*, pages 287–295, 2012.

[KC] Klein M. Barbacci M. Longstaff T. Lipson H. Kazman, R. and J. Carriere. The architecture tradeoff analysis method. *Engineering of Complex Computer Systems, Fourth IEEE International Conference.*

[KL] Kazman R. Bass L. Carriere J. Barbacci M. Klein, M. H. and H. Lipson. *Attribute-based architectural styles.*

[Kru] P. B. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE 12.6.*

[KS] Cimander R. Kubicek, H. and H. J. Scholl. *Organizational interoperability in e-government: lessons from 77 european good-practice cases.*

[LD] Kacem A. H. Jmaiel M. Loulou, I. and K. Drira. Formal design of structural and dynamic features of publish/subscribe architectural styles. *Software, IEEE 12.6.*

[Ma11] H. D. Ma. Internet of things: Objectives and scientific challenges. *Journal of Computer science and technology*, 26:919–924, 2011.

[Man] Chui M. Bisson P. Woetzel J. Dobbs R. Bughin J. Aharon D. Manyika, J. *The Internet of Things: Mapping the value beyond the hype.*

[Mau10] L. S. Maurya. Comparison of software architecture evaluation methods for software quality attributes. *Journal of Global Research in Computer Science*, 1, 2010.

[MJ] Mullen M. Wong K. Midwinter, A. and M Jones. Software architectural styles: An introduction.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[PH] Giesecke S. Pahl, C. and W. Hasselbring. An ontology-based approach for modelling architectural styles. *Springer Berlin Heidelberg.*

[PW01] Wegmann A. Preiss, O. and J. Wong. On quality attribute based software engineering. *Euromicro Conference, 2001*, pages 114–120, 2001.

[QvG08] Bay H. Quack, T. and L. van Gool. Object recognition for the internet of things. *The Internet of Things: First International Conference, IOT 2008*, pages 230 – 246, 2008.

[RA08] Duller M. Gilmer K. Maragkos D. Papageorgiou D. Rellermeyer, J. S. and G. Alonso. The software fabric for the internet of things. *The Internet of Things: First International Conference, IOT 2008*, pages 87–104, 2008.

[Ric15] M Richards. *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them.* O'Reilly Media, 2015.

[SR14] Nambi S. N. Prasad R. V. Sarkar, C. and A. Rahim. A scalable distributed architecture towards unifying iot applications. *Internet of Things (WF-IoT), 2014 IEEE World Form on*, pages 508–513, 2014.

[Sti08] V. Stirbu. Towards a restful plug and play experience in the web of things. *Semantic Computing*, pages 512–517, 2008.

[SW10] Guillemin P. Friess P. Sundmaeker, H. and S. Woelfflé. *Vision and challenges for realising the Internet of Things.* 2010.

[Tay13] R. N. Taylor. The role of architectural styles in successful software ecosystems. *Proceedings of the 17th International Software Product Line Conference*, pages 2–4, 2013.

[Web10] R. H. Weber. Internet of things-new security and privacy challenges. *Computer Law and Security Review*, 26:23–30, 2010.

[Wei06] Goodenough J. B. Weinstock, C. B. On system scalability. *Carnegie-Mellon University Pittsburgh PA software engineering Institute Technical Note*, 2006.