# Modeling Cyber-Physical Systems in the Arrowhead Framework

## - Exploring the potential effects of an IoT framework in an educational environment

**Master's Thesis in Computer Science**

Michael Glomnes

May 15, 2018
Halden, Norway

Østfold University College www.hiof.no

# Abstract

The emergence of Industry 4.0 led to “Internet of Things” (IoT) becoming an increasingly growing topic both in academia and in an industrial setting. The last couple of years, an increasing number of frameworks for developing IoT-networks and system of systems have emerged.

The starting point for this thesis was the running example of a cyber-physical system named “TheRoom” used in the master course “Modeling cyber-physical systems” at Østfold University College. The goal was to explore how introducing the Arrowhead Framework would affect the design of “TheRoom”, and potentially the course. To address the questions we planned how the “TheRoom” should interact with the Arrowhead framework, and then implemented the changes to the exemplary cyber-physical system. Three experiments were conducted to evaluate how the adapted cyber-physical system performed.

The first experiment evaluates how the cyber-physical system would handle expanding the environment. The cyber-physical system was connected to ten sensors and three actuators successfully. The data from the sensors were retrieved, and the system was able to control the actuators.

The second experiment evaluates how the framework handles multiple cyber-physical systems to connect and use the same devices simultaneously. Two cyber-physical systems were started and connected to the same sensors, but different actuators. Both systems were fully able to operate using the same sensor.

The third experiments aimed to determine the resource consumption over a prolonged time. The measuring show that the proposed arrowhead compliant cyber-physical system will consume an increasing amount of resources if left running. The growing consumption of resources would be a problem for a system meant to run indefinitely.

**Keywords:** CPS, IoT, Arrowhead Framework

# Acknowledgments

I would like to express my gratitude to my supervisor Professor Øystein Haugen at Østfold University College. His patience, motivation and genuine enthusiasm for the subject helped me immensely through this thesis.

I would also like thank An Lam and Amin Shahraki for the help they gave me throughout this process.

Last, but not least, I would like to thank my friends and family for their support through it all.

Michael Glomnes, May 15, 2018

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

In 2011 the German government launched a project named “Industrie 4.0”, or industry 4.0. The name refers to the fourth industrial revolution and with the introduction of cyber-physical systems, Internet of things and cloud computing and the concept of the “smart factory”. By using cyber-physical systems monitor the manufacturing and production in a factory and make decentralized decisions, the factory becomes a network of sensors and actuators.

With industry 4.0 becoming a priority among research centers, universities, and companies, the term has become more and more blurry. To create a shared understanding of the therm Industry 4.0, Hermann, Pentek, and Otto present four design principles [6]. The principles are:

- *Interoperability:* Machines, actuators, sensors, and users are connected over a network. Wireless communication allows for ubiquitous internet access and lets the connected objects and people share information, and working in collaboration towards a common goal.
- *Information transparency:* By increasing the number of interconnected objects and people, the fusion of the physical and virtual world enables a new form of information transparency. Sensor data is linked with a digitalized copy of the physical world, creating a virtual copy of the environment.
- *Decentralized decisions:* The IoT participants perform their task as autonomous as possible. Only an exception or conflictions are tasks delegated to a higher level. Cyber-physical systems enable decentralized decisions. Cyber-Physical systems allow for monitoring and controlling the physical world autonomously.
- *Technical assistance:* The role of the User switches from an operator to a decision-maker and problem solver. Support systems must, therefore, aggregate and visualize information to ensure that the user can make informed decisions and solve urgent problems. Cyber-physical systems support humans by conducting tasks that are unpleasant, exhausting or unsafe.

The Internet of Things (IoT) frameworks presents different approaches to solve the issues presented when creating big systems of systems. In 2015, Derhamy et al.[5] surveyed some commercial IoT frameworks supported by the big players in the software industry. In this survey, they group the frameworks into three main approaches on IoT, global cloud, local cloud and peer-to-peer networks. The approach used determines its suitability for different application spaces and impact information latency, data collation, feature interdependency, module design, and network topology.

One IoT framework created as an ARTEMIS innovation project is the Arrowhead Framework. This framework uses an approach based on the concept of local clouds. The local cloud is equipped with core services capable of supporting the interaction between services, as well as handle the support functionality. While there are different approaches to interoperability, The Arrowhead Framework uses a Service-Oriented Architecture (SOA)[4]. SOA is in general about data exchange between service providing systems, and service consuming systems. The fundamental properties of SOA are Loose coupling, Late binding, and Autonomy. In the Arrowhead Framework, the core services deployed in the local cloud facilitates fundamental properties of SOA.

- *Loose coupling*: each component has little or no knowledge of the definitions of other separate components in the network. This is done by registering all service providing systems in the core service "Service Registry"
- *Late binding*: the data exchange between service providing systems and service consuming systems are created at runtime. This is done by creating rules in the core service "Orchestration Service" and retrieving information about the services to be consumed.
- *Autonomy*: Each device can act on its own regardless of other systems. It is responsible for its data and functionalities. Once a service exchange is set up between two systems, this may continue without further involvement of any supporting services.

## 1.1 Motivation

In the master course “Modeling Cyber-Physical Systems” at Østfold University College, students learn to model and develop a cyber-physical system based on a running example named “TheRoom”. This cyber-physical system was created for controlling the temperature in a room. Due to a limited amount of devices, the students are grouped and given a set of physical actuators and sensors which are exclusively by one of the group members at the time. When developing, the students are using simulated devices, and then test their system on the devices. Introducing an IoT framework will let all students have access to the sensors and actuators simultaneously, and may also facilitate more complex systems, as well as an introduction of new devices.

## 1.2 Research questions

This thesis aims to depict the consequences of introducing an IoT framework. The research questions address the requirements and caveats of such an adoption, as well as the benefits and gain. Moreover, how does implementing an IoT framework contribute to the overall experience of modeling cyber-physical systems? This leads to the following research questions:

**Research Question 1:**
*What is required for an existing Cyber-Physical system, exemplified by the running example “TheRoom”, to be ported onto the Arrowhead Framework?* We are not aiming to create a “wrapper” for the cyber-physical system, nor are we after creating a system communicating with the services in the Arrowhead framework through a translator system. We aim to deploy the cyber-physical system as an arrowhead application system registered on the network.

**Research Question 2:**
*In what ways can the Arrowhead Framework influence the course “Modeling Cyber-Physical Systems”?* With this question, we aim to investigate the positive and negative effects the arrowhead framework potentially can have on the course.

## 1.3 Thesis Structure

This section briefly summarizes the contents of each remaining chapter in the thesis.

**Chapter 2: Background**
This chapter presents the theoretical foundation of this thesis. It starts by introducing the Arrowhead Frameworks structure and functionality. It explains the mandatory core systems, the local cloud approach, and the support system availed in this thesis. The chapter then presents ThingML, the modeling language used to develop the cyber-physical system. Lastly, the chapter presents the cyber-physical system itself.

**Chapter 3: Related Works**
This chapter describes the some of the current state-of-the-art IoT frameworks and their approach.

**Chapter 4: Design**
This chapter introduces a proposed design of the communication between the cyber-physical system and the local Arrowhead cloud using the Arrowhead framework. Two different designs are presented and explained. One resembling the original cyber-physical system, and one adopting more of the functionality made available through the arrowhead framework.

**Chapter 5: Implementation**
This chapter discusses how the cyber-physical system was implemented into the local arrowhead cloud. The implementation of both designs described in Chapter 4 are explained.

**Chapter 6: Evaluation**
This chapter summarizes the results from Chapter 4 and Chapter 5. Three experiments were conducted to test the Arrowhead compliant cyber-physical systems.

**Chapter 7: Discussion**
This chapter discusses the findings from Chapter 5 and the results in Chapter 6. The research questions presented in Section 1.2 is answered in this chapter, as well as suggesting ideas and directions for future work.

**Chapter 8: Conclusion**
This chapter summarizes the thesis and briefly present the answers to the research questions.

# Chapter 2

# Background

This chapter is dedicated to elaborate on technical information about the building blocks of the technology of which the thesis is built, the Arrowhead Framework, ThingML, and the cyber-physical system “TheRoom”.

## 2.1 The Arrowhead Framework

The Arrowhead Framework is addressing IoT based automation. The approach taken is that IoT’s are abstracted to services. This to enable IoT interoperability in-between almost any IoT’s [1].

The creation of automation is based on the idea of local automation clouds and builds on the fundamental principles of SOA [2]. A local Arrowhead Framework cloud can be compared to global cloud provide improvements and guarantees regarding:

- Real-time data handling
- Data and system security
- Automation system engineering
- Scalability of automation systems

The Arrowhead framework uses the local cloud approach to IoT. The local cloud creates a contained environment that allows for real-time communication even without a connection to the Internet. In the Arrowhead framework, each cloud conducts independent tasks. However, inter-cloud communication is possible. The local Arrowhead cloud is created by deploying the Arrowhead mandatory core services on a computer connected to the local network. The mandatory core systems are:

**Service Registry**

The primary function of the service registry is to keep track of and provide storage of all active services registered within the local cloud, and hosts the service discovery service. The Service Registry system enables a service provider to publish its service instance and allows a consumer to discover what service instances it wants to consume. It provides functionality based on Domain Name Service (DNS) and Domain Name Service - Service Discovery (DNS-SD) standards. The service registry is an independent system that does not consume any other services.

**Orchestration Service**

Stores orchestration requirements and resulting orchestration rules. The orchestration service enables remote control of which service instances a consumer shall consume.

**Authorisation Service**

Provides authorization, authentication and optionally accounting for a system consuming a produced service based on a set of rules. There are two different setups defined in the Arrowhead Framework for the Authorization Service, AA (Authorisation and Authentication), and AAA (Authorization, Authentication, and Accounting). AA is better suited for systems with an abundance of resources, whereas the AAA system is suited for resource-constrained devices. It also enables a service provider to determine what consumers to accept. The authorization service is registered in the Service Registry.

**AA - Authorisation System**

The AA system implements an authorization system based on X.509 certificates, and thus require more computation power from the device. It provides two services, Authorization Management for managing the access rules, and Authorization control for controlling access to a resource or service within the cloud.

**AAA - Authorisation System**

The AAA system implements an authorisation system based on Radius tickets. In addition to the two services provided by the AA system, AAA system provides an extra service called AuthorizationID.

### 2.1.1 Arrowhead Framework local cloud

The Arrowhead framework requires the three mandatory services to be running on the local network. The arrowhead deployment can be done either as a stand-alone operating system or on a local server. In this thesis, we use the local server deployment.

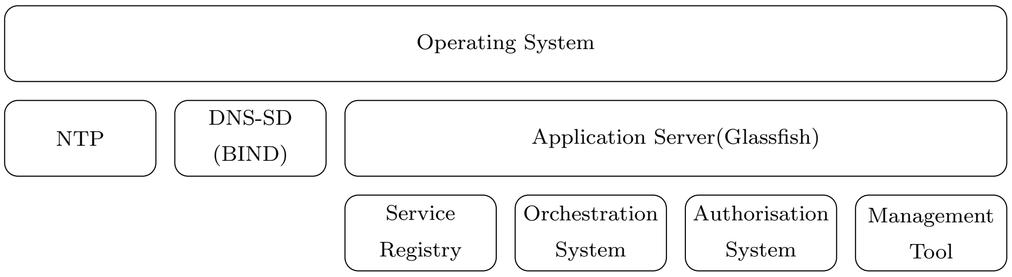
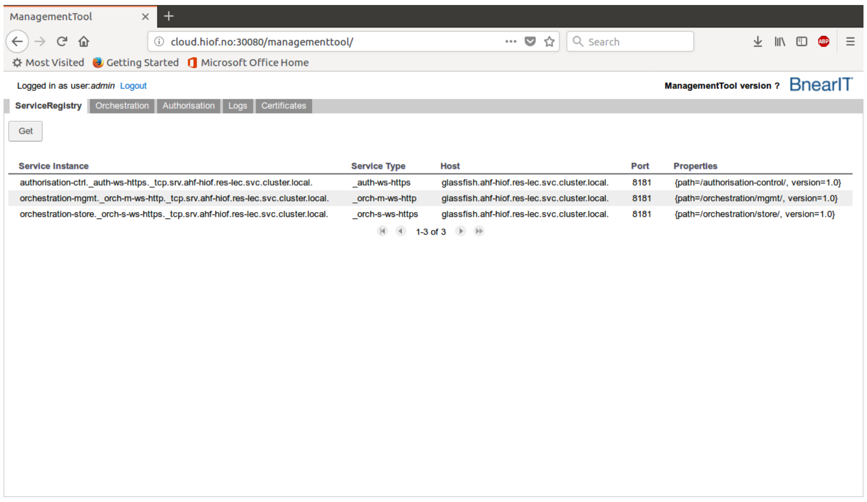


Figure 2.1: An illustration of the architecture in the Arrowhead Deployment. Three containers are deployed on the the operating systems, a network time protocol server, a domain name system service discovery and an application server. Deployed on the application server, the mandatory core system and the management tool. The figure is inspired by [4]

The reasoning behind using the local server is that the deployment is based on using Docker containers. Using Docker, the system as a whole can be deployed pre-configured. Also, the system could be implemented on a server at the University College's servers at a later time.

### 2.1.2 The Management Tool

The Arrowhead Framework provides a management tool created by BnearIT. The management tool connects to the core services and presents a graphical user interface for a user to observe the published services and create and assign orchestration and authorization rules rules[4].



Logged in as user:*admin* Logout

ManagementTool version ?

| Service Instance | Service Type | Host | Port | Properties |
|---|---|---|---|---|
| authorisation-ctrl._auth-ws-https._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _auth-ws-https | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/authorisation-control/, version=1.0} |
| orchestration-mgmt._orch-m-ws-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _orch-m-ws-http | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/orchestration/mgmt/, version=1.0} |
| orchestration-store._orch-s-ws-https._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _orch-s-ws-https | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/orchestration/store/, version=1.0} |

1-3 of 3

Figure 2.2: The UI of the Management Tool support system hosted on the application server. The three services seen are registered on the Service Registry by the other two core systems.

As seen in Figure 2.2, the management tool shows the services published in the Service Registry. Because no other services are published at this time, only the services provided by the two other mandatory core services in the Arrowhead Frameworks are shown.

### 2.1.3 Arrowhead definitions

Because we are using the arrowhead framework in this thesis, a few terms should be defined.

**Local cloud**
In an arrowhead framework, a local cloud is defined as a self-contained network with the mandatory core systems, and at least one system deployed.

**Service**
In the context of the arrowhead framework, service is what used to exchange information from a service producer to a service consumer. A system produces a service.

**System**
An Arrowhead system is what is producing or consuming a service. A system is not restricted to either produce or consume a single service but can produce and consume multiple services if desired.

### 2.1.4 Publishing Services

To publish a service, an instance must be created. The service instance consists of four classes.

- Producer class, containing the required information for creating and publishing a producer on the arrowhead network
- Data class containing the data to be created by the service.
- Resource class containing the different calls available to the consumer.
- Consumer class, for registering consumers of the producers.

When published, the sensor data is available on the network. By using the management tool, all services published on the local arrowhead cloud are listed. In this test environment, two temperature sensors and one actuator are published on the local cloud, registered as services in the service registry, and have manually been given addresses. When registering, the services is assigned a name, a host, a port number and a service type.

### 2.1.5 Registerring a Consumer

A service consumer can get the information needed to connect to a service producer in different ways. The Service Registry saves the information for each service, and it is possible to search for different types of producers. An application can be allocated a set of available services through the orchestration service and can use the rules to retrieve the information when needed. When an application registers as a consumer, it connects directly to the producer.

## 2.2 ThingML

ThingML is a modeling language for embedded and distributed systems. It is developed by the formerly names “Networked Systems and Services department” of SINTEF in Oslo, Norway[10]. ThingML stands for “Thing” Modeling Language as a reference to the Internet of Things. The idea behind ThingML was to develop a practical model-driven software engineering tool-chain which targets resource-constrained embedded systems such as low-power sensor and microcontroller-based devices.

ThingML is based on architecture models, state machines and imperative action language to define the behavior of components. A fundamental principle of ThingML is that it can compile into runnable applications for different platforms. The ThingML toolset compiles ThingML to C, Java, and Javascript. The compiler completely rewrites the code using model-to-text transformations, and the use of other programming languages, in this case, Java, can be used in conjunction with the ThingML language when specific functions are unavailable. This can be done by either adding existing projects as dependencies or by writing the java code directly into the source code.

Since the ThingML meta-model is lightweight, programs in ThingML have a small footprint and can run on various hardware with resource constraints [12]. The *Things* in ThingML sends messages to communicate with other Things. This way, a Thing compiled in java, can talk to a Thing compiled in C enabling co-development of complex distributed systems. ThingML also provides functionality for simulating parts of the systems by generating mock-up graphical user interfaces.

Below, some terms used in ThingML defined.

**Things**
: The Things are the main components of ThingML. A thing is mainly a software component, but can be used as a wrapper of a hardware component.

**State machines**
: A state machine is the central part of the things. A state machine is a collection of states that each has its purpose and through events transition between them. In ThingML, the state machines are referred to as *state charts* and must contain an initial state as an entry point. In a statechart, different state types can be used. In the cyber-physical system, we use two of them, standard states and composite states.

**States**
: The states are the primary building blocks in ThingML. The states can perform actions on entry, on exit, during transitions, and during internal events.

**Composite states**
: A composite state is a state that can contain other states. Like a statechart, a composite state needs an initial state as an entry point. On entry, the composite state will traverse itself until a message enables it to exit and return to another state within the state machine.

Composite states can have the history extension that enables it to remember what state it was in the last time it exited, and return to that state when entered again.

**Ports**

In ThingML, the things communicate by sending messages through ports. A state can provide ports to other things, and require ports from others. Messages sent through the ports are done as asynchronous messages. Sending messages is the main way of trigging events.

**Messages**

Messages are sent through ports and can be declared either within a thing or fragment. Messages can but is not required to, contain parameters.

**Fragment**

A thing fragment is an interface for messages allowing multiple things to include the messages.

## 2.3 TheRoom

“TheRoom” is the cyber-physical system used as a running example in the course “Modeling Cyber-Physical Systems”. “TheRoom” is a thermostat meant to control the temperature of a room. The user decides which temperature sensors and actuators the program can use, and sets the comfort temperature. Based on the input from the sensors, “TheRoom” turns on and off the temperature to regulate the room it controls.

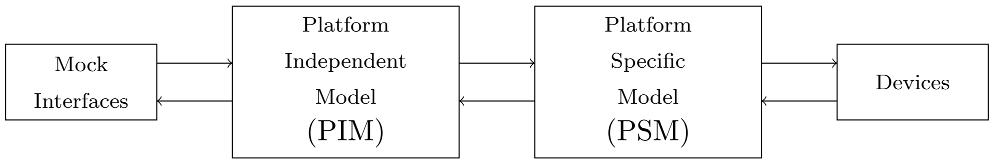

Mock Interfaces
Platform Independent Model (PIM)
Platform Specific Model (PSM)
Devices

Figure 2.3: Mock interfaces, communicates with the platform independent model to allow a User to control the system. The platform independent model sends and receives messages from the platform specific model containing data. The platform-specific model communicates and controls the devices.

“TheRoom” is developed using ObjectManagement Groups model-driven architecture [11]. The architecture of “TheRoom” can be seen in Figure 2.3. The cyber-physical system consists of two parts, a platform independent model (PIM) and a platform-specific model (PSM), each consisting of different state machines or “Things”. As the name states, the PIM is independent of the platform used to implement it. The PSM, on the other hand, is created specifically for the platform, in this case, the interaction with the telldus hardware used to communicate with the sensors and actuator.

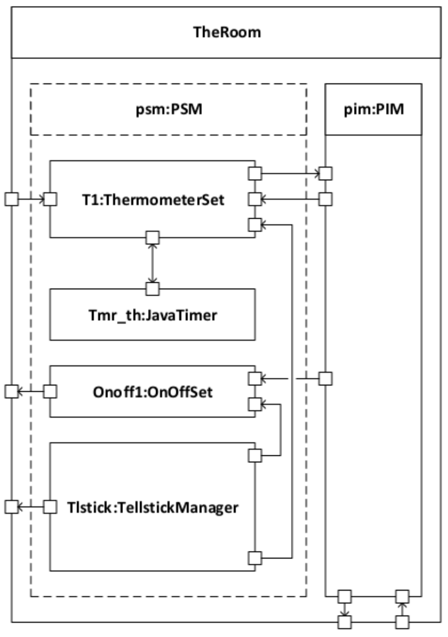


Figure 2.4: A composite structure of TheRoom. The platform-specific model is presented using dotted lines because ThingML does not allow for nested classes, resulting in it being developed as four independent Things, but presented here as a whole.

As seen in Section 2.3 the PIM consists of four independent “Things”.

**Tlstick:TellstickManager**
The tlstick:TellstickManager uses the telldus transceiver to gather information about the sensors within the range, and retrieves the actuators from the telldus support application “Telldus Center”.

**Onoff1:OnOffSet**
TheOnoff1:OnOffSet is responsible for handling the actuator, and the actuator related information. The “Thing” receives the ids from the TellstickManager

**T1:ThermometerSet**
Connected to the sensors through the tellstick transceiver.

**Timr_th:JavaTimer**

The Platform Independent Model contains one state machine, or thing, for controlling the logic. This thing receives input from the user through a mock interface for use in the setup process, and based on this input sends messages for how to control the actuator based on the parameters set by the user and temperature provided. The Platform Specific Model contains four things, one for activating the transceiver and gather information about the devices, one for handling the logic concerning adding sensors and receive data, one for adding and controlling the actuator, and one for triggering the data collection process.

### 2.3.1 TheRoom behavior

The behavior of TheRoom can be broken down into the following three phases:

**Initialization phase:**
The PSM thing “TellstickManager” gathers information about the sensors and actuators available to the application. In TheRoom, the telldus transceiver can recognize all sensors nearby and all the activated actuators.

**Setup phase:**
Prompted with the information gathered in the initialization phase, the user enters the information about what sensors and actuators to be used in the applications run phase The PIM sends the information to TemperatureSet and OnOffSet.

**Run phase:**
The application loops and the PIM sends messages to the things in the PIM for how to control the actuators based on the information based on the parameters set by the user and the temperature retrieved from the sensors.

# Chapter 3

# Related Works

This chapter presents the state-of-the-art on open source frameworks regarding the internet of things and cyber-physical systems as well as presenting challenges found during the literature review. The goal is to provide the reader with an understanding of existing techniques, tools, and approaches as well as positioning the work done in this thesis. The literature review was conducted as a mix between systematic literature review described in "*Guidelines for performing Systematic Literature Reviews in Software Engineering*" [8] and snowballing as described in *"Guidelines for snowballing in systematic literature studies and a replication in software engineering"*[13].

## 3.1 IoTivity

IoTivity is an open source IoT framework sponsored by the Open Connectivity Foundation(OCF) implementing OCF standards. The goal of IoTivity is to provide seamless peer-to-peer connectivity regardless of platform.[7]

IoTivity offers two different block stacks defining a layer between the application and the communication protocol. A rich stack allowing for using multiple communication protocols, and a lite stack only communicating through CoAP. The resources available on the device makes the deciding factor to which stack to be used. If the device has enough resources it uses the rich stack, and a device with constrained resources uses the lite stack.

The IoTivity stack blocks offers a point-to-point or point-to-multipoint network topology. This means that a single device can connect to one or more devices simultaneously. Each stack block also contains discovery functions allowing a new device deployed on the network to discover the other devices.

IoTivity offers APIs in Java, C, and C++. However the lite stack can only use an API for C. Both Telldus and ThingML supports C and Java, so this could be an alternative as a framework used in this thesis. However, IoTivity does not present any orchestration or access restriction similar to the Arrowhead Framework orchestration system.

## 3.2 LWM2M

Lightweight Machine to Machine (LWM2M) developed by Open Mobile Alliance (OMA) is a protocol developed managing devices over a network, transfer data to devices over the network, and meet the requirement for most devices.

LWM2M uses a cloud approach to IoT. This cloud can either be locally deployed or hosted elsewhere. The main feature in LWM2M is the client-server communication. A client is deployed on an end device and hosts one or more objects. An object is the "things", either sensors or actuators, connected to the client through a gateway or deployed on an LWM2M compliant device. The client sends the data to a cloud for storage, which can be accessed from an application.

A layer is defined above the CoAP communication protocol, and uses the standard CoAP to UDP binding, but can also define a binding between CoAP and SMS. LWM2M handles security by availing DTLS, but it is possible to send data without security.

Multiple open source frameworks have been developed for LWM2M, supporting C, C++, and Java. Developers of the framework do provide a test server for using the framework, however a focus point of this thesis was to not be reliant on third parties.

## 3.3 Azure IoT suite

As a part of the ever-expanding set of cloud services, Microsoft offers Azure IoT suite, a collection of services and solutions designed to facilitate the end-to-end development of IoT applications [3]. The IoT platform is hosted in the cloud and acts as a hub for communication between the cloud and the devices.

Azure allows for devices to send data either from IP-capable devices hosting an IoT client or through gateways, to the cloud. Microsoft provides Azure as a Platform as a Service (PaaS), which allows for many functionalities such as remote monitoring and device simulation, but also other Azure services such as Azure Machine Learning and Azure Logic Apps.

Azure offers libraries to build applications interact with the cloud. The supported languages include C and Java, both usable with ThingML. However, Azure IoT Suite is a commercial product and could prove expensive for the intended use.

## 3.4 SIFT

SIFT is a safety-centric programming platform for connected devices in IoT environments developed by Liang et al. [9].SWIFT is an IoT development support platform that lowers the effort and attention from non-expert users in producing safe IoT apps with automated techniques for verifying two key safety criteria. The user defines if-then rules that the IoT applications are not allowed to break. When a problem of rule conflicts or policy violations is found, SIFT informs the users to offending app rules along with model parameters.

SIFT simplifies the process of connecting new devices by connecting them to the platform indirectly through an intermediate network device working as a gateway. This gateway has the necessary radio and networking stack to communicate with nearby devices. By delegating device management to the gateway, SIFT operates only against device abstraction.

SIFT is implemented as a series of modules in C++ and Python. Although interesting, the platform is not compatible with ThingML.

Chapter 4

# Design

To answer research question 1 presented in Section 1.2 we need to implement the cyber-physical system "TheRoom" into the arrowhead framework, by adapting the areas where the application communicates with the devices. As explained in Section 2.3, each of the behavioral phases communicates with the devices differently. However, all communication from the cyber-physical system to the devices happens in the platform-specific model. To understand how the arrowhead framework should interact with TheRoom, we need to determine where the communication is performed, and what data is transferred.

This chapter presents designs for how the cyber-physical system would interact with the local arrowhead cloud is proposed for each phase. For the initialization phase, two designs are proposed, one using only the Service Discovery and one using the Orchestration Service and the Service Discovery. The reasoning behind this is that the first design bears more similarities to how TheRoom operates by itself, and the second design uses more of the functionality made available by the arrowhead framework.

## 4.1 Changes to TheRoom

To change the platform from a standalone ThingML application to a part of a framework we need to change the platform-specific model. Because of how the PSM is structured, introducing or removing “Things” was unnecessary. The “Things” however was renamed to fit the new responsibilities concerning the interaction with the Arrowhead local cloud.

**ArrowheadManager**
Similar with the tellstick manager from the original TheRoom, the ArrowheadManager is responsible for connecting to the mandatory core services and retrieve service information for the services the cyber-physical system is allowed to consume. The ArrowheadManager also initializes SensorManager and DeviceManager.

**SensorManager**
SensorManager is the formerly TermometerSet “Thing”. The SensorManager connects to the sensor services based on messages from the platform independent model. Every 10th second, an event is triggered, and the data is retrieved from the sensors and forwarded to the platform independent model

**DeviceManager**

The DeviceManager is the formerly OnOffSet “Thing”. DeviceManager connects to the actuator service based on messages from the platform independent model. Based on events triggered by messages from the platform independent model, the DeviceManager and turns the actuators on or off.

## 4.2 Behavioral phases in the new cyber-physical systems

The designs created to explain how the new systems would interact during the three behavioral phases explained in Section 2.3. The same three phases are present in the new systems. Two new systems were created, differing only in the Initialization phase. The phases are:

**Arrowhead Initialization phase:**

The PSM “Thing” “am:ArrowheadManager” gathers information about the sensors and actuators available to the application. Two designs are proposed for doing this

**Using Service Discovery:**

The first design uses only the Service Discovery to retrieve the service information about all sensors and actuators published in the local cloud.

**Using Orchestration:**

The second design uses the Orchestration system to retrieve the service identities of the services the application is allowed to use and uses the identities to retrieve the service information from the Service Directory.

When “am:ArrowheadManager” has retrieved the service information, the sensor and actuator services are given a sensor or actuator identification respectively. The sensor identification and sensor service information is sent to “sm:SensorManager” along with the initialization message. The “dm:DeviceManager” receives the actuator identification number and service information for the corresponding actuator services. The services and their identification was presented to the User through a UI.

**Setup phase:**

The PIM remains unchanged, so the setup phase is identical from the users perspective. In the setup phase, the User decides which sensor and actuator “TheRoom” will use by entering the identification number for the service. The identification is sent to sm:SensorManager for sensors and dm:DeviceManager for actuators, which again connects to the corresponding service.

**Run phase:**

Similar to the Setup phase, the run phase is mostly controlled by the PIM and remains unchanged for the user. The sm:SensorManager retrieves the temperature from the sensor service, and forward it to the PIM. Based on the temperature the PIM either sends a message to turn the actuator on, off or sends no message at all.

### 4.2.1 Arrowhead Initialization Using Service Discovery

Reminiscent of how TheRoom works, TheRoom using the only the Service Discovery (TheRoomSD) communicates with the arrowhead network in a way that might prove problematic for larger systems; by retrieving the Service Information about all services with a service type of either sensor or actuator.

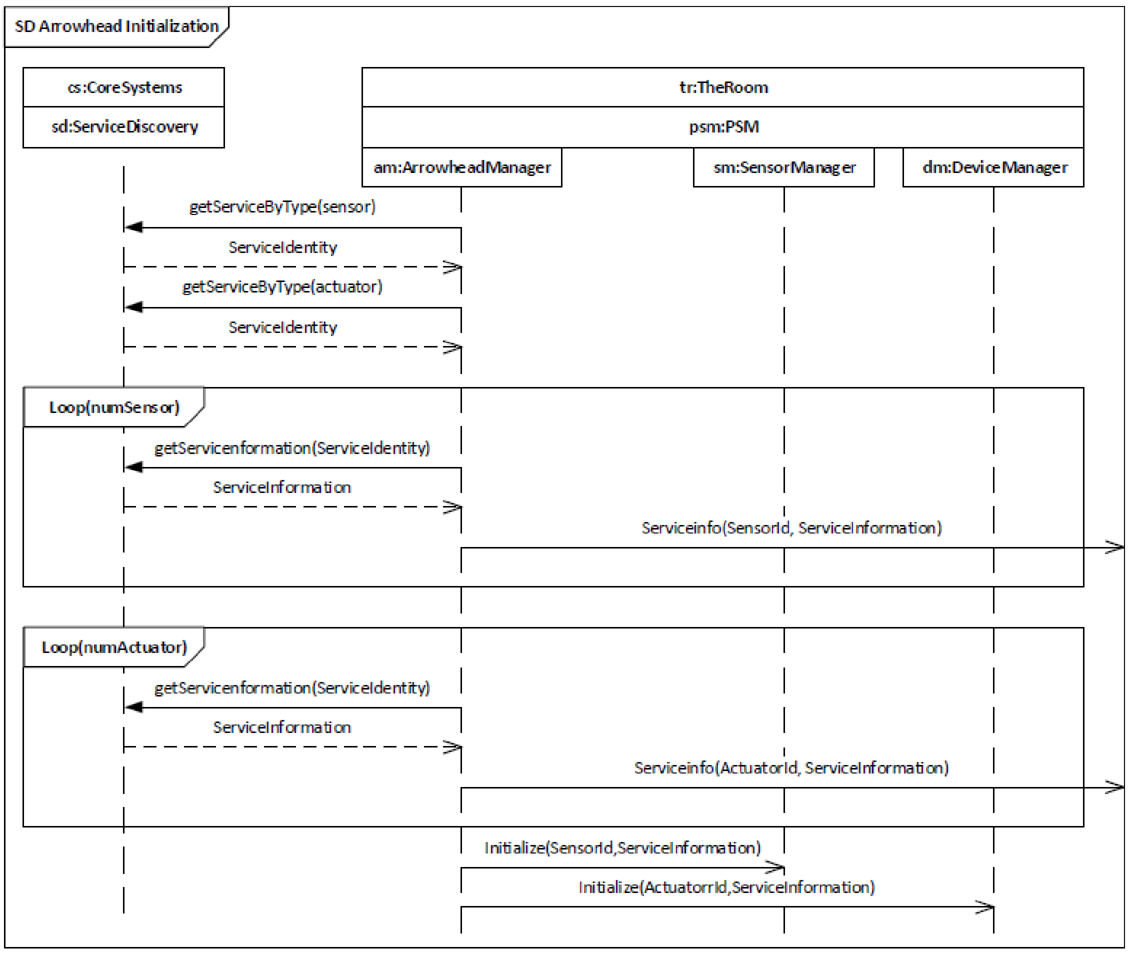

SD Arrowhead Initialization
cs:CoreSystems
sd:ServiceDiscovery
tr:TheRoom
psm:PSM
am:ArrowheadManager
sm:SensorManager
dm:DeviceManager
getServiceByType(sensor)
ServiceIdentity
getServiceByType(actuator)
ServiceIdentity
Loop(numSensor)
getServicenformation(ServiceIdentity)
ServiceInformation
Serviceinfo(SensorId, ServiceInformation)
Loop(numActuator)
getServicenformation(ServiceIdentity)
ServiceInformation
Serviceinfo(ActuatorId, ServiceInformation)
Initialize(SensorId,ServiceInformation)
Initialize(ActuatorrId,ServiceInformation)

Figure 4.1: Sequence Diagram of the Arrowhead Initialization phase. The cyber-physical system uses only the Service Discovery to retrieve the services and initializing the SensorManager and the DeviceManager

Figure 4.1 shows how TheRoomSD communicates with the Arrowhead local cloud in the initialization phase. The communication can be broken down as follows:

1. am:ArrowheadManager connects to sd:ServiceDiscovery and asks for all services with the service type ”sensor”.
2. sd:ServiceDiscovery sends the service identity of each service published that fulfill this requirement to am:ArrowheadManager.
3. am:ArrowheadManager connects to the sd:ServiceDiscovery again, and ask for all services service type ”actuator”.

4. sd:ServiceDiscovery sends the service identity of each service published that fulfill this new requirement to the am:ArrowheadManager.
5. A loop is initiated to retrieve the service information for each sensor.
   (a) For each of the sensors, am:ArrowheadManager ask sd:ServiceDiscovery for the service information.
   (b) sd:ServiceDiscovery sends the service information to the corresponding services to am:ArrowheadManager.
   (c) Each sensor is assigned with an id, and both sensor id and service information is sent to a user interface and presented for the user.
6. A second loop is initiated to retrieve the service information for each actuator.
   (a) For each of the actuators, the ArrowheadManager ask the Service Discovery for the service information.
   (b) The Service Discovery sends the service information to the corresponding service.
   (c) Each actuator is assigned with an id, and both actuator id and service information is sent to the same graphical user interface and presented for the user.
7. am:ArrowheadManager sends all the service ids and service information to, and initializes, sm:SensorManager.
8. am:ArrowheadManager sends all the actuator ids and service information to, and initializes, dm:DeviceManager.

### 4.2.2 Arrowhead Initialization using the Orchestration Service

In the Orchestration Service, rules can be set for which services an application can avail, so TheRoom using the Orchestration Service (TheRoomO) will only retrieve information about these services.

For TheRoomO, the following design for communication is proposed:

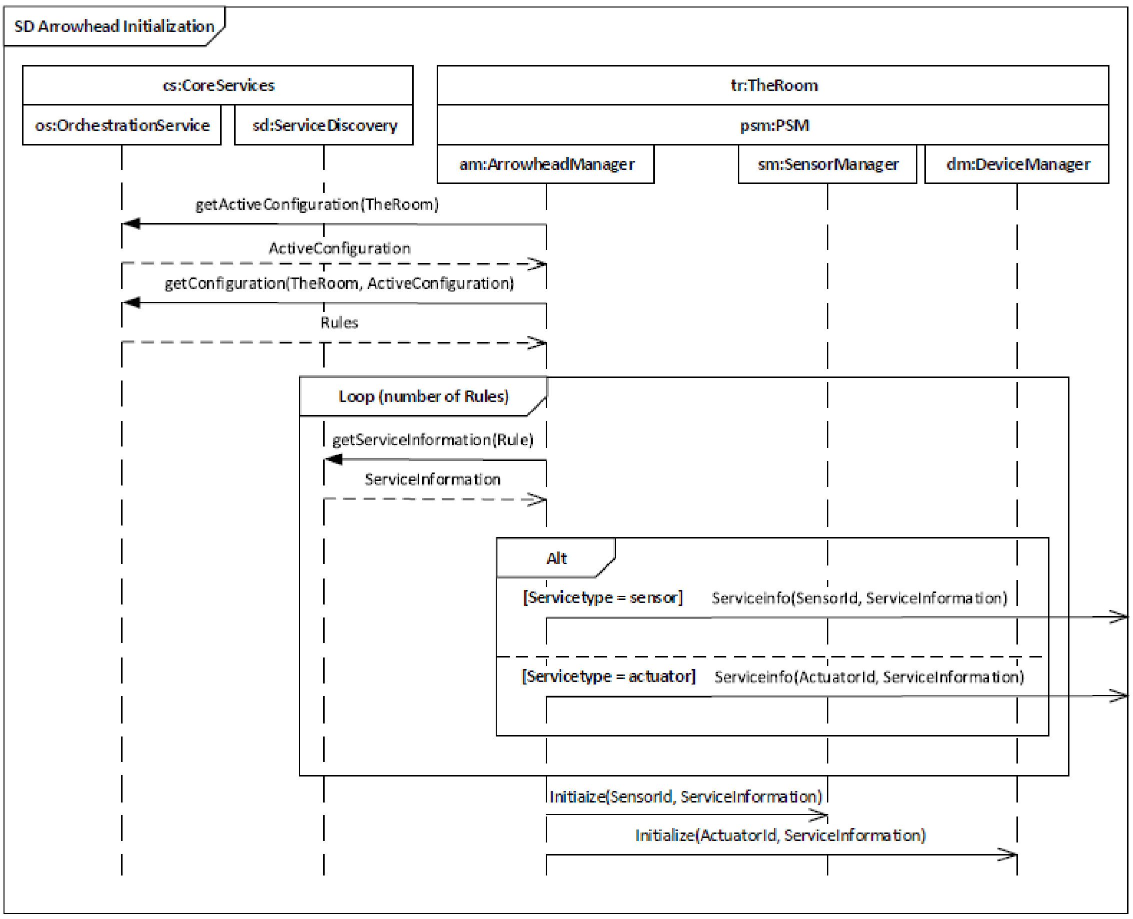

SD Arrowhead Initialization
cs:CoreServices
os:OrchestrationService
sd:ServiceDiscovery
tr:TheRoom
psm:PSM
am:ArrowheadManager
sm:SensorManager
dm:DeviceManager
getActiveConfiguration(TheRoom)
ActiveConfiguration
getConfiguration(TheRoom, ActiveConfiguration)
Rules
Loop (number of Rules)
getServiceInformation(Rule)
ServiceInformation
Alt
[Servicetype = sensor]
Serviceinfo(SensorId, ServiceInformation)
[Servicetype = actuator]
Serviceinfo(ActuatorId, ServiceInformation)
Initiaize(SensorId, ServiceInformation)
Initialize(ActuatorId, ServiceInformation)

Figure 4.2: Sequence Diagram of the Arrowhead Initialization phase. The cyber-physical system uses the Orchestration System to retrieve the rules of what services the cyber-physical system is allowed to consume, and then the Service Discovery to retrieve the service information. Finally, the ArrowheadManager initializes the SensorManager and the DeviceManager

A sequence diagram of the initial interaction of TheRoomO's initial interaction with the arrowhead cloud using only the Service Discovery can be seen in Figure 4.2. In this scenario, the communication can be broken down as follows.

1. am:ArrowheadManager connects to the os:OchestrationService asking for the name of the active configuration for the application.
2. os:OrchestrationService sends the active configuration back to the am:ArrowheadManager.
3. am:ArrowheadManager connects to os:OrchestrationService again to ask for the rules using the application name and the name of the active configuration.

4. os:OrchestrationService sends a set of Rules with the name of the services providers the application is allowed to consume to am:ArrowheadManager.

5. A loop is initiated to retrieve the service information for each rule and presenting it to the user.

   (a) am:ArrowheadManager connects to sd:ServiceDiscovery to ask for the service information for each of the rules.

   (b) sd:ServiceDiscovery sends the service information for the corresponding rule back to am:ArrowheadManager.

   (c) depending on the service type, one out of two things can happen:

      - If the service type is "sensor", the service is given a sensor identification number, and both sensor id and service information is sent to a UI, and presented for the user.
      - If the service type is "actuator", the service is given an actuator identification number, and both actuator id and service information is sent to a UI, and presented for the user.

6. am:ArrowheadManager sends all the ids and service information along with an inistialization message to sm:ServiceManager.

7. am:ArrowheadManager sends all the ids and service information aalong with an inistialization message to dm:DeviceManager.

### 4.2.3 Setup Phase

In the setup phase the PIM, SensorManager, and DeviceManager are running simultaneously. Prompted with the sensor ID and actuator ID as well as the service information the user enters the required information in the setup phase.

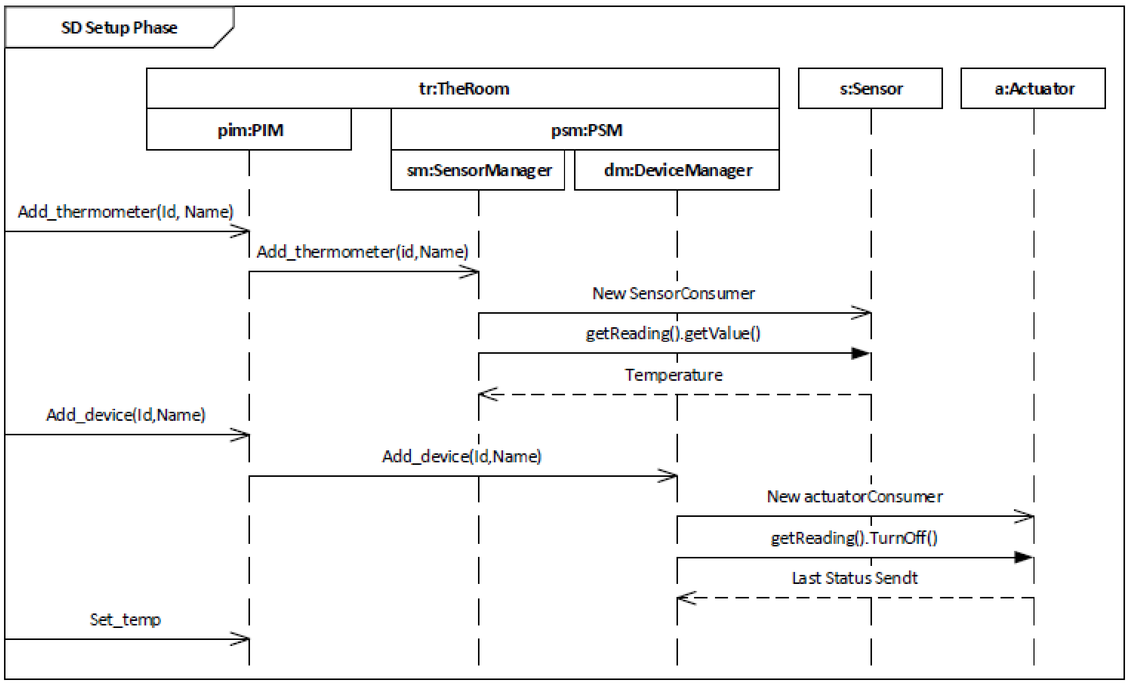


Figure 4.3: Sequence Diagram of the Setup Phase. The User enters the identification numbers assigned in the Initialization phase to pim:PIM, which forwards the information to either sm:SensorManager or dm:DeviceManager depending on the service type. connects to the corresponding sensor system

A sequence diagram of the setup phase of both rooms can be seen in Figure 4.3. The communication can be broken down as follows.

1. pim:PIM receives a sensor id and a name from the user through UI.
2. pim:PIM forwards the information to sm:SensorManager.
3. sm:SensorManager connects to s:Sensor and register itself as a consumer.
4. sm:SensorManager connects to s:sensor and asks or the temperature.
5. s:sensor sends the temperature to sm:SensorManager.
6. pim:PIM receives an actuator id and a name from the user through the UI.
7. The PIM forwards the information to dm:DeviceManager.
8. dm:DeviceManager connects to a:Actuator and registers itself as a consumer.
9. dm:DeviceManager connects to a:Actuator and turns it off.

10. a:Actuator sends the status back to dm:DeviceManager.
11. pim:PIM receives the desired comfort temperature from the user.

### 4.2.4 Run Phase

The run phase is the autonomous part of TheRoomSD and TheRoomO.

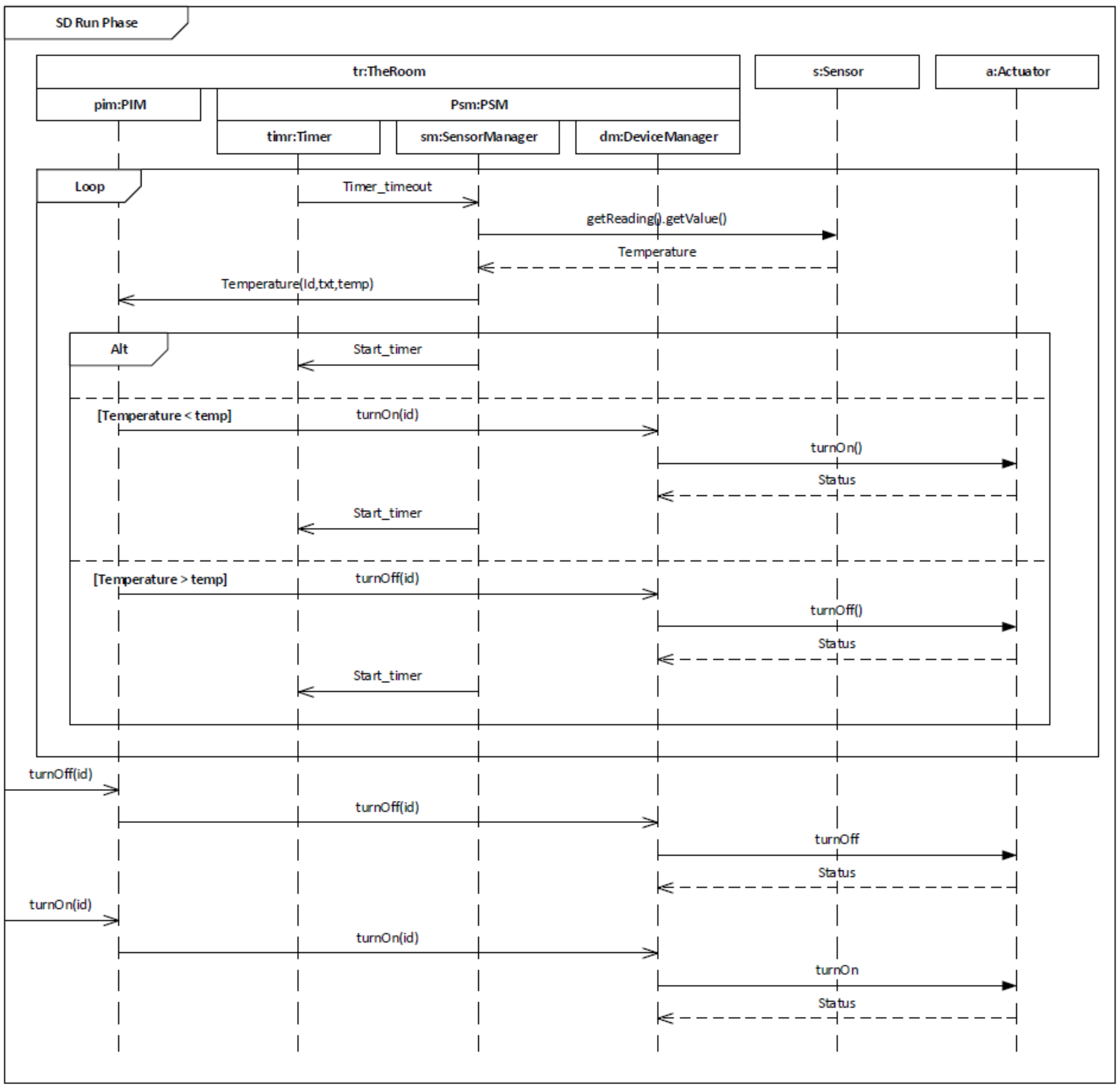


Figure 4.4: Sequence Diagram of the Run Phase. The tmr:Timer times out and triggers an event in sm:SensorManager to retrieve the temperature from the sensor, and sends a message to pim:PIM with the sensor identification number and the temperature. Based on this information The pim:PIM decides the course of action. The Loop can be broken by the User, by sending either a turn on or turn off command.

1. The run phase mainly consists of a loop
   (a) timr:Timer times out, and triggers an event in sm:SensorManager.
   (b) sm:SensorManager asks s:Sensor for the temperature.
   (c) s:Sensor sends the temperature to sm:SensorManager.
   (d) sm:SensorManager sends the temperature to pim:PIM.

(e) Depending on the message from sm:SensorManager one of three things happens:
   - If the temperature is satisfactory nothing happens and the timer is restarted.
   - If the temperature is too low, the PIM sends a message to DeviceManager to turn on the actuator. The timer is restarted.
   - If the temperature is too high, the PIM sends a message to DeviceManager to turn off the actuator. The timer is restarted.

2. pim:PIM receives a turnOff command from the user containing an actuator id.
3. pim:PIM forwards the turnOff message to dm:DeviceManger.
4. dm:DeviceManager sends a turnOff command to a:Actuator.
5. pim:PIM receives a turnOn command from the user containing an actuator id.
6. pim:PIM forwards the turnOn message to dm:DeviceManger.
7. dm:DeviceManager sends a turnOn command to a:Actuator.

# Chapter 5

# Implementation

This chapter explains in more detail how "TheRoom" is imlpemented in the Arrowhead framework based on the designs presented in Chapter 4. First the architecture of the network and applications are explained, followed by the service application that publishes the services as explained in Section 2.1.4, and the changes made to the room. Secondly, the first implementation is presented followed by TheRoomSD, and finally TheRoomO.

## 5.1 Architecture

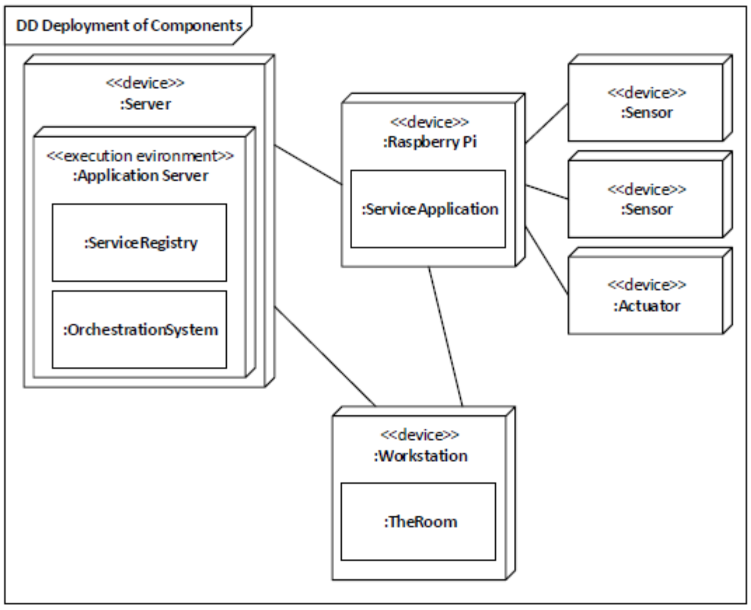


Figure 5.1: Architecture of the environment. The Arrowhead core services is deployed on the local server, the service application is deployed on a Raspberry Pi, and "TheRoom" is deployed on a workstation.

Figure 5.1 shows the deployment diagram of the local cloud during the initial testing and implementation. The service application was deployed on a Raspberry Pi to simulate a real life scenario, where the consumer and producers of services where not deployed on the same device, and for testing that the services were available on the network.

## 5.2 Service Application

The service application was a wrapper functioning as a service system for both the sensors and the actuator. The service system connected to the service registry and published the services to the arrowhead local cloud.

**SensorService**

A SensorService contains the functionality of retrieving data from a sensor. The sensors used in this experiment provides temperature, humidity and a time stamp.

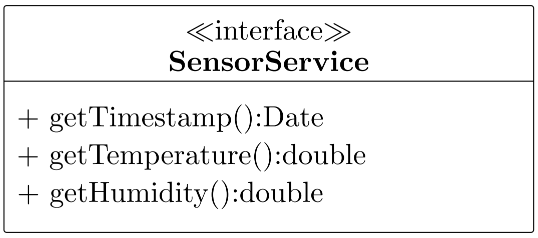


Figure 5.2: Interface for SensorService, with functionality and datatypes.

Figure 5.2 shows the class diagram for the sensor services. Functionality for retrieving the information is added as produced services.

Listing 5.1: getReading service

```
@Path(”/reading”)
@GET
@Produces(MediaType.APPLICATION_XML)
public Response getReading(@Context HttpServletRequest request) {
 Response response;
 sensorData.setTemperature(ts.getSensor(id).getTemperature());
 sensorData.setHumidity(ts.getSensor(id).getHumidity());
 Timestamp time = new Timestamp(ts.getSensor(id).getTimeStamp());
 Date date = new Date(time.getTime());
 sensorData.setTimestamp(date);
 response = Response.ok(sensorData).build();
 return response;
}
```

Listing 5.6 shows the reading call for the SensorService interface

**ActuatorService**

An ActuatorService contains the functionality of controlling the actuator as well as providing data about the last value sent to the actuator.

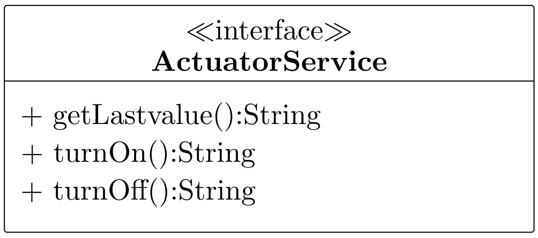


Figure 5.3: Interface for ActuatorService, with functionality and datatypes.

Figure 5.3 shows the class diagram for the actuator services. Functionality for controlling the actuator is added as produced services. The data of the last value sent is added to both the services and given its own functionality.

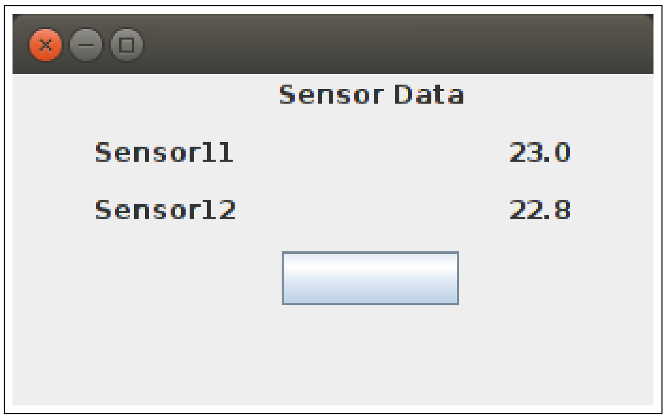
Listing 5.2: TurnOn service

```
public Response turnOn(@Context HttpServletRequest request) {
 Response response;
 ts.sendCmd(id, "ON");
 actuatorData.setStatus(ts.getLastCmd(id));
 response = Response.ok(actuatorData).build();
return response;
}
```

Listing 5.2 shows the turnOn call for the ActuatorService interface

**TestProgram**

A test application was created to test the connection to the arrowhead local cloud.



```
                 Sensor Data

   Sensor11                         23.0

   Sensor12                         22.8
```

Figure 5.4: The Gui from the test application. The data was retrieved from the services published on the network. The button turned the actuator on or off depending on last command sent

Seen in Figure 5.4 is the GUI of the test application developed consume the services through the arrowhead local cloud. The interface was simplistic and its only function was to display the data retrieved from the sensors and turn the actuator on and off.

## 5.3 The first prototype: TheArrowheadRoom

The initial experiment adapting ThingML to the Arrowhead Framework was based on the original room. The prototype bypasses the mandatory core services and connect directly to the service producers. This prototype was developed on TheRoomV1. TheRoomV1 contains the same platform specific model as the later versions of TheRoom, but a more simplistic platform independent model that allows for registering multiple sensors and actuators and pulling temperatures without the use of a timer. The ArrowheadManager was used only for initializing the DeviceManager and the SensorManager, and not for communication with the cs:CoreSystems.

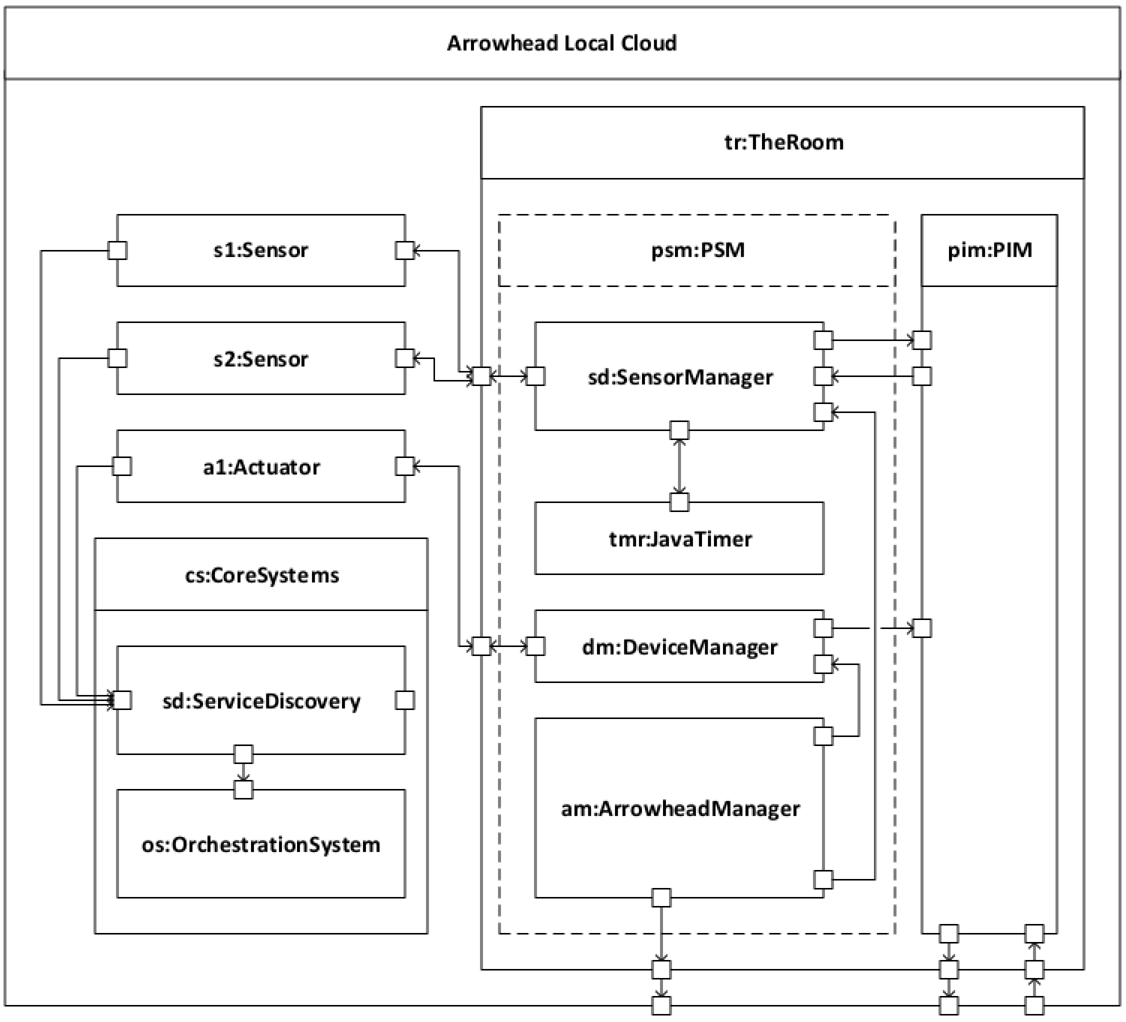


Figure 5.5: The composite structure of the first prototype. The services were published on sd:ServiceDiscovery, however, dm:DeviceManager and sm:SensorManager were connected directly to the sensors and actuators by hardcoding the connection.

Figure 5.5 shows a composite structure of how the first arrowhead compliant version of TheRoom was created. However the data is now collected from services published on the arrowhead local cloud. While the service producers was registered in the Service Discovery, there was no communication with TheRoom.

### 5.3.1 The first prototype and Arrowhead

The prototype connected to the service producers to register itself as a consumer. Because the ArrowheadManager did not communicate with the core systems, the information needed to connect to the sensor and actuator service producers was coded directly into the SensorManager and DeviceManager respectively. It is possible to create custom datatypes in ThingML, and for easier interaction with the arrowhead framework four datatypes was created presented in Listing 5.3.

Listing 5.3: PSM datatypes definitions

```
1  object Sensor
2  @java_type "tellstickSensors.SensorConsumer";

4  object Device
5  @java_type "tellstickActuators.ActuatorConsumer";

7  object Endpoint
8  @java_type "se.bnearit.arrowhead.common.core.service.discovery.endpoint.
       HttpEndpoint";

10 object REST
11 @java_type "se.bnearit.arrowhead.common.service.ws.rest.
       ClientFactoryREST_WS";
```

The datatypes we defined are the necessary classes from the dependencies. Seen in Listing 5.3 lines 1-2 and 4-5 defines the service interfaces from the service application that allows the use of the services. Lines 7-8 and 10-11 defines the classes in the core utilities needed to connect to the service on the network.

Listing 5.4: Using the defined datatypes in ThingML

```
portNumber = 15000 + id_s
host = '"http://localhost:"+'&portNumber&'+"/sensor/"'
ep = 'se.bnearit.arrowhead.common.core.service.discovery.endpoint.
    HttpEndpoint.createFromString('&string&')'
rest = 'new se.bnearit.arrowhead.common.service.ws.rest.
    ClientFactoryREST_WS("./cacerts.jks", "changeit","./tester.jks", "
    changeit")'
sensors[last_thermo] = 'new tellstickSensors.TellstickConsumer("Sensor",
    '&ep&','&rest&' )'
```

Listing 5.4 shows an excerpt from the add thermometer state in the SensorManager thing. The variables are explained below

**portNumber** Integer: the port range used for sensors in this example + id_s

**id_s** Integer: sensorid sent from the PIM

**host** String: the full address to the service

**ep** Endpoint: the custom datatype created using the host variable

**rest** REST: custom datatype needed to use rest services in arrowhead

**sensors** Sensor: custom datatype for registering as a consumer of a service

## 5.4 TheRoom using Service Discovery

The first approach adapting TheRoom to the Arrowhead Framework was built as a system resembling the original TheRoom. It keeps the principles of the previous approach: That sm:SensorManager registers and retrieves data from the sensors and the dm:DeviceManager registers and controls the actuators. However, the collection of the information about the services was moved to the am:ArrowheadManager, and sent to the two *Things* with the initialization message. The devices was given an identification to let the user distinguish which device the cyber physical system should use during run time.

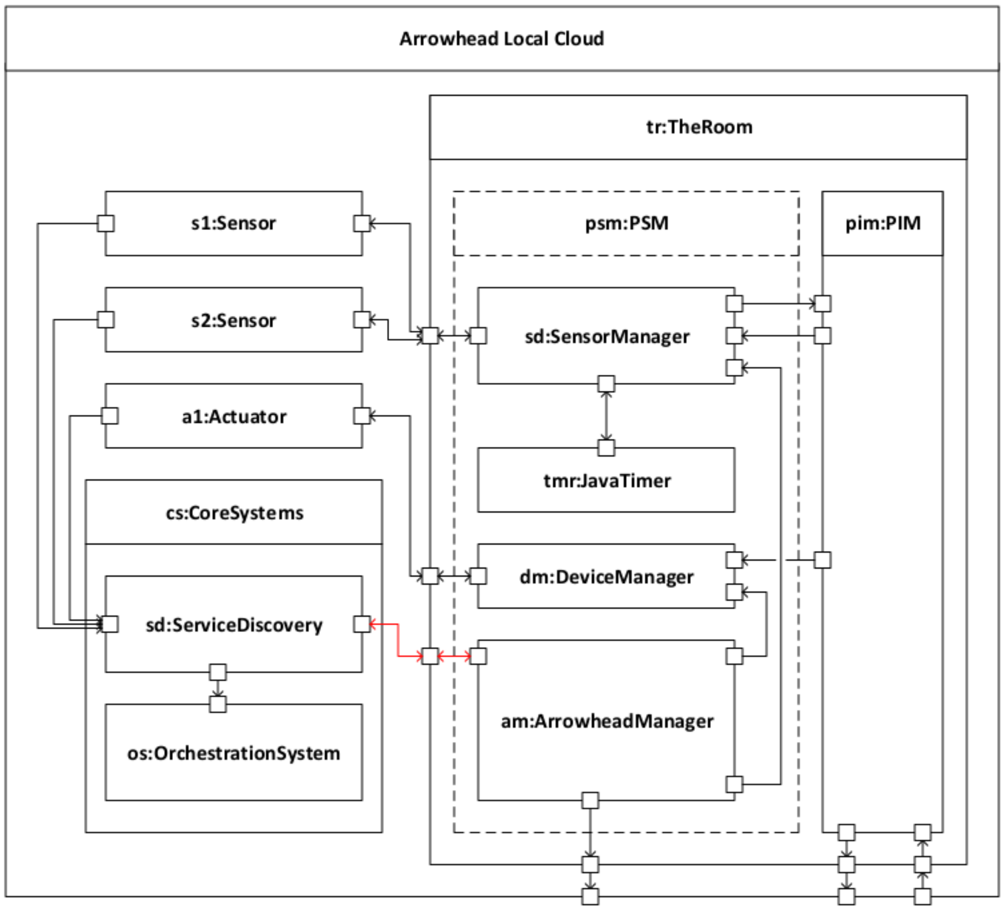


Figure 5.6: The composite structure of "TheRoomSD". The "Thing" am:ArrowheadManager is connected to the sd:ServiceDiscovery and uses this connection to retrieve the service information needed to connect to the sensors and actuators.

Figure 5.6 shows the composite structure of TheRoomSD. The new connection from sd:ServiceDiscovery to am:ArrowheadManager allows the cyber-physical system to retrieve the service information from all services published in the registry.

### 5.4.1 TheRoomSD and Arrowhead

Using the service discovery three additional custom datatypes was defined as presented in Listing 5.5

Listing 5.5: Added PSM datatypes in ThingML

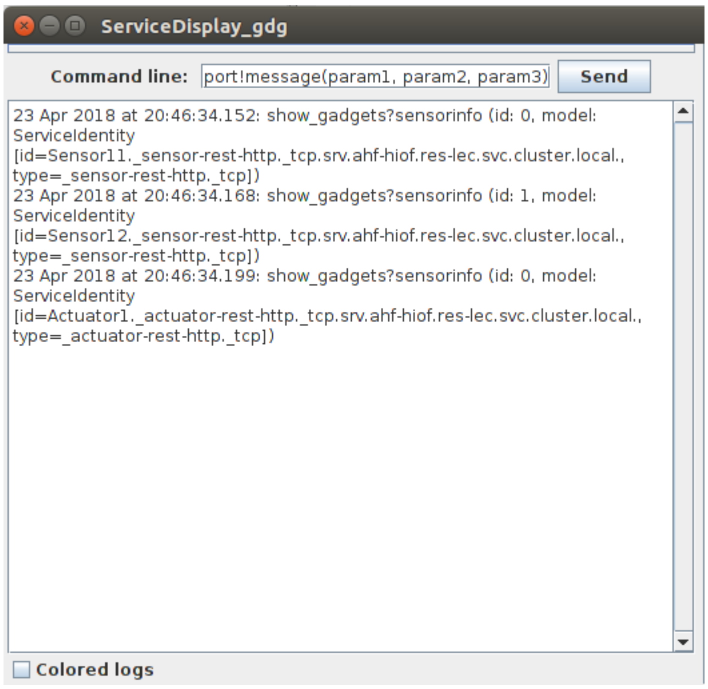
```
object ServiceDiscovery
@java_type "se.bnearit.arrowhead.common.core.service.discovery.ws.rest.
    ServiceDiscoveryREST_WS";

object ServiceIdentity
@java_type "se.bnearit.arrowhead.common.service.ServiceIdentity"

object ServiceInformation
@java_type "se.bnearit.arrowhead.common.service.ServiceInformation"
```

The new datatypes allows for the gathering of the information about the services TheRoomSD can consume. The ServiceIdentity is the name given to the service producer when registered in the service discovery, and using this name the SrviceInformation is collected. The service information contains all information needed to connect to the service producer, such as endpoint, host and service type.

Listing 5.6: Initiating arrowhead

```
REST = 'new se.bnearit.arrowhead.common.service.ws.rest.
    ClientFactoryREST_WS("./cacerts.jks", "changeit", "./tester.jks", "
    changeit")'
SEPstring = "http://cloud.hiof.no:30045/servicediscovery"
SEP = 'se.bnearit.arrowhead.common.core.service.discovery.endpoint.
    HttpEndpoint.createFromString(&SEPstring&)'
SD = 'new se.bnearit.arrowhead.common.core.service.discovery.ws.rest.
    ServiceDiscoveryREST_WS ('&SEP&', '&REST&', "srv.docker.ahf")'
'List<se.bnearit.arrowhead.common.service.ServiceIdentity> sensors = '&SD
    &'.getServicesByType("_sensor-rest-http._tcp");
'List<se.bnearit.arrowhead.common.service.ServiceIdentity> actuators = '&
    SD&'.getServicesByType("_actuator-rest-http._tcp");
```

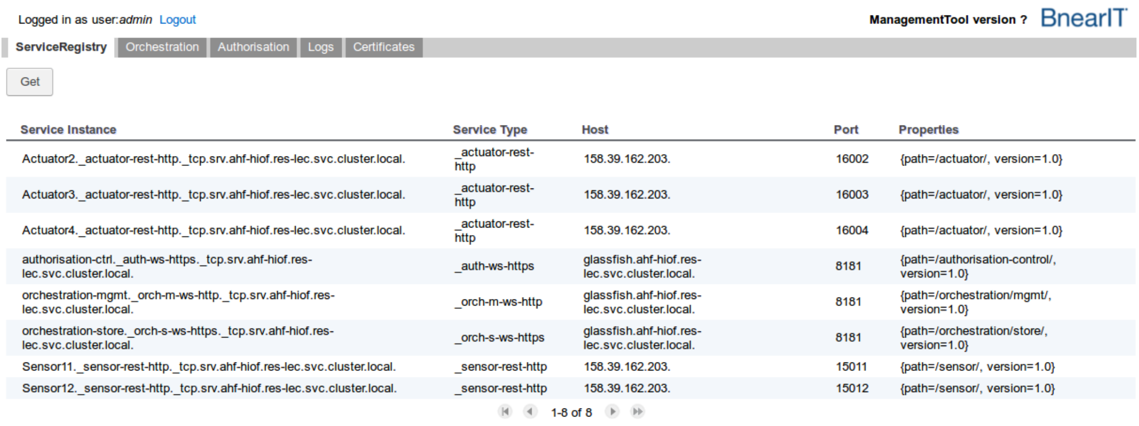
Listing 5.6 contains the necessary code for gathering the ServiceIdentity for each sensor and actuator. Because ThingML did not have the simplistic looping functionality java does, lists were used. Each list was lopped, and the service information was gathered and put into arrays in ThingML. The services was also given a sensorid and actuatorid depending on the service type, and presented for the user as seen in Figure 5.11.



```
23 Apr 2018 at 20:46:34.152: show_gadgets?sensorinfo (id: 0, model:
ServiceIdentity
[id=Sensor11._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local.,
type=_sensor-rest-http._tcp])
23 Apr 2018 at 20:46:34.168: show_gadgets?sensorinfo (id: 1, model:
ServiceIdentity
[id=Sensor12._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local.,
type=_sensor-rest-http._tcp])
23 Apr 2018 at 20:46:34.199: show_gadgets?sensorinfo (id: 0, model:
ServiceIdentity
[id=Actuator1._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local.,
type=_actuator-rest-http._tcp])
```

Figure 5.7: The mock interface ServiceDisplay for TheRoomSD showing the services published in the ServiceRegistry with the ”sensor” or ”actuator” service types.

Figure 5.7 shows the output from the ArrowheadManager. The ServiceDisplay presents all service producers published on the network with either an actuator or sensor service type. The available services are given an id for reference. Seen in Figure 5.8 we see all service producers published but this time through the management tool-



| Service Instance | Service Type | Host | Port | Properties |
|---|---|---|---|---|
| Actuator2._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _actuator-rest-http | 158.39.162.203. | 16002 | {path=/actuator/, version=1.0} |
| Actuator3._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _actuator-rest-http | 158.39.162.203. | 16003 | {path=/actuator/, version=1.0} |
| Actuator4._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _actuator-rest-http | 158.39.162.203. | 16004 | {path=/actuator/, version=1.0} |
| authorisation-ctrl._auth-ws-https._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _auth-ws-https | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/authorisation-control/, version=1.0} |
| orchestration-mgmt._orch-m-ws-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _orch-m-ws-http | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/orchestration/mgmt/, version=1.0} |
| orchestration-store._orch-s-ws-https._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _orch-s-ws-https | glassfish.ahf-hiof.res-lec.svc.cluster.local. | 8181 | {path=/orchestration/store/, version=1.0} |
| Sensor11._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | 158.39.162.203. | 15011 | {path=/sensor/, version=1.0} |
| Sensor12._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | 158.39.162.203. | 15012 | {path=/sensor/, version=1.0} |

Figure 5.8: The management tool showing the services published in the ServiceRegistry.

## 5.5 TheRoom using Orchestration

The second approach of adapting TheRoom to the Arrowhead framework was built to avail more of the functionality that the Arrowhead Framework has to offer. Similar to TheRoomSD, the principles of the first approach was kept. However this version was connected to the orchestration service, and used this to gather rules about the services it was allowed to consume. The ArrowheadManager used the rules given by the orchestration to retrieve the service information about the services, and based on the service type, they were given an sensorid or actuatorid to let the user distinguish them.

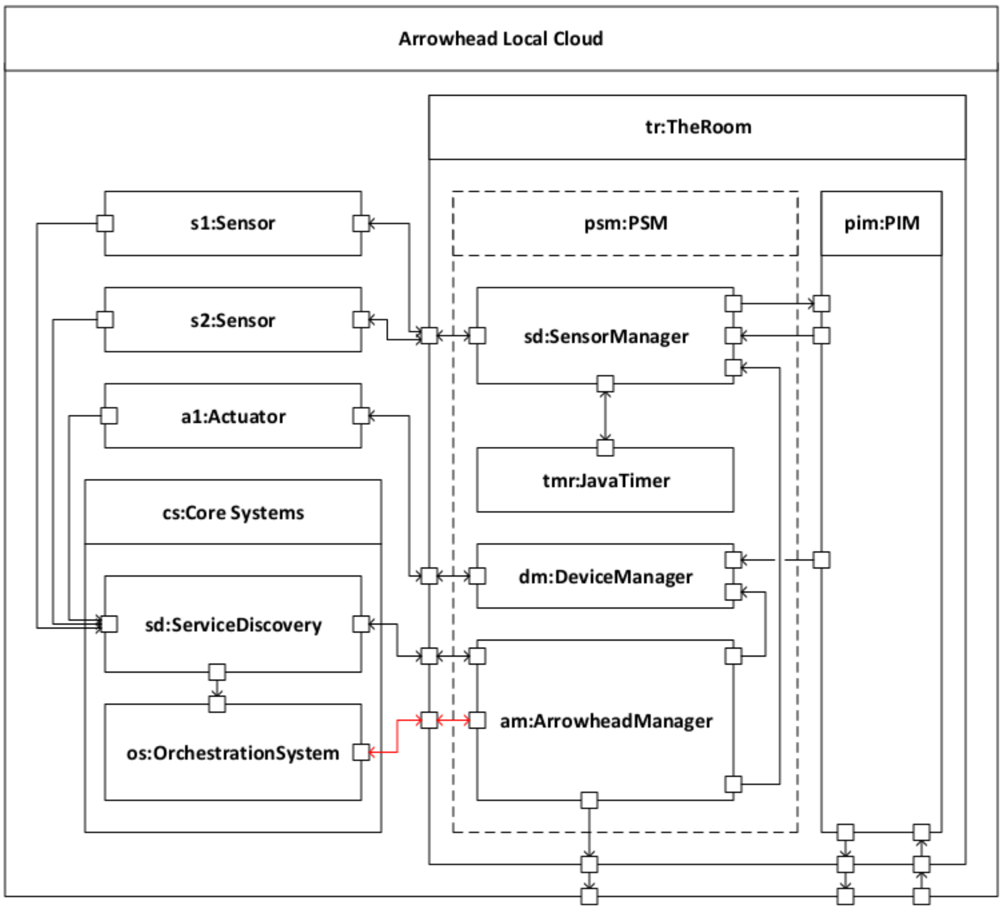


Figure 5.9: The composite structure of “TheRoomO”. The “Thing” am:ArrowheadManager is connected to the sd:ServiceDiscovery and os:OrchestrationSystem. The service information is retrieved from sd:ServiceDiscovery using the service identities retrieved from os:Orchestrationsystem

Figure 5.9 shows the composite structure of the environment when using TheRoomO. This structure differs from TheRoomSD by the connection to os:OrchestrationSystem.

### 5.5.1 TheRoomO and arrowhead

For communicating with the orchestration system we must add two additional datatypes for the PSM.

Listing 5.7: Added datatypes for orchestration

```
object Orchestraton
@java_type "se.bnearit.arrowhead.common.core.service.orchestration.
    OrchestrationManagement"

object Oconfig
@java_type "se.bnearit.arrowhead.common.core.service.orchestration.data.
    OrchestrationConfig"
```

Listing 5.7 shows the two additional defined datatypes for TheRoomO. The Orchestration datatype is used for connection to the orchestration service, and the Oconfig datatype is used for gathering the rules.

Listing 5.8: Initial interaction with the orchestrtation system

```
REST = 'new se.bnearit.arrowhead.common.service.ws.rest.
    ClientFactoryREST_WS("./cacerts.jks", "changeit", "./tester.jks", "
    changeit")'
OEPstring = "https://cloud.hiof.no:30081/orchestration/mgmt"
OEP = 'se.bnearit.arrowhead.common.core.service.discovery.endpoint.
    HttpEndpoint.createFromString(&OEPstring&)'
''&REST&'.setAcceptMismatchedHostnames(true);'
ORC = 'new se.bnearit.arrowhead.common.core.service.orchestration.ws.rest.
    OrchestrationMgmtConsumerREST_WS('&OEP&','&REST&')'
AConf = ''&ORC&'.getActiveConfiguration("TheRoom")'
Conf = ''&ORC&'.getConfiguration("TheRoom",'&confA&')'
'List<String> Rules = '&Conf&'.getRules();
```

Listing 5.8 shows the code necessary to execute the first four lines of communication shown in Figure 4.2. The rules are “Strings” and contains the serviceidentity of the services allowed for consumption by the cyber-physical system.

### 5.5.2 Orchestration rules

TheRoomO retrieves the active configuration from the orchestration service during the initialization phase, and uses the rules to acquire the service information from the service discovery. Creating and updating the configuration can be done effortlessly using the management tool. To illustrate the use of orchestration systems, two examples is presented.

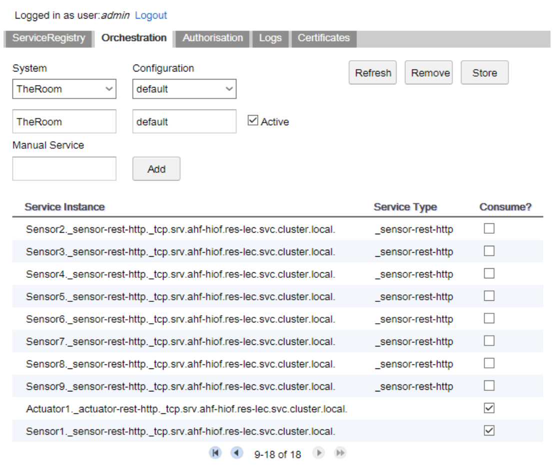
Table 5.1: Services made available for TheRoomO

| Example | Sensor1 | Sensor 2 | Actuator1 | Actuator2 | Actuator3 |
|---|---|---|---|---|---|
| 1 | x | | x | | |
| 2 | | x | | x | x |

Using the management tool, a system can be assigned services for use. Each system can have multiple configurations, but only one can be active at a time. The active configuration is gathered by the systems as seen in line 6 in Listing 5.8.

#### Example 1: one sensor one actuator

In the first example, we create an active configuration named default. This configuration are assigned sensor1 and actuator1.


Logged in as user:*admin* Logout

System | Configuration
TheRoom | default

TheRoom | default | ✓ Active

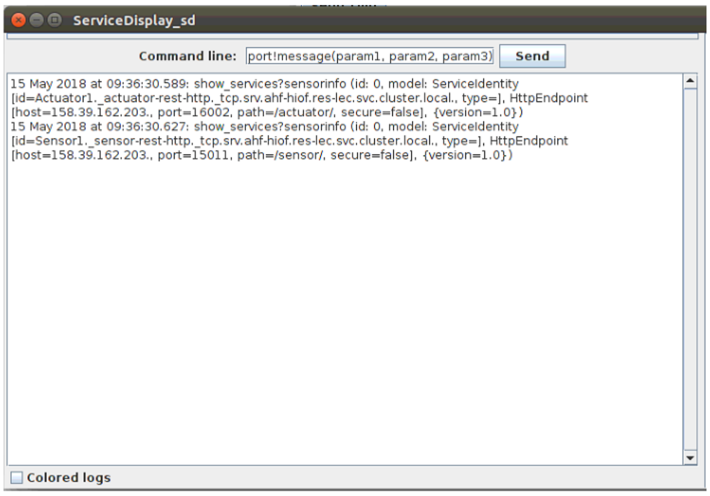
Manual Service

| Service Instance | Service Type | Consume? |
|---|---|---|
| Sensor2._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor3._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor4._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor5._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor6._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor7._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor8._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor9._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Actuator1._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | | ✓ |
| Sensor1._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | | ✓ |

9-18 of 18

Figure 5.10: Active configuration created in the management tool.

Figure 5.10 shows the creation of the orchestration rules for TheRoom. The configuration is named default, and it is set as active. Using TheRoomO, only the two services in the room is considered available for the application.



port!message(param1, param2, param3)

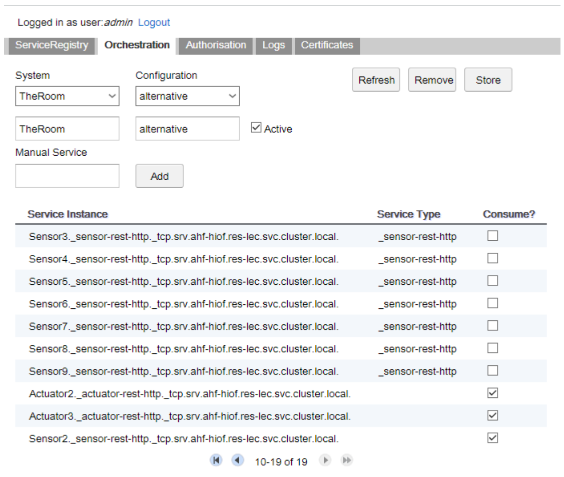
```
15 May 2018 at 09:36:30.589: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Actuator1._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=16002, path=/actuator/, secure=false], {version=1.0})
15 May 2018 at 09:36:30.627: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Sensor1._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
```

Figure 5.11: ServiceDisplay for TheRoomO using the “default” orchestration configuration.

As seen in Figure 5.11, actuator1 and sensor1 are available as services.

## Example 2: one sensor two actuators

In this second example, a new configuration is created and set as the active configuration. The assigned services are sensor2, actuator 2 and actuator 3.



System TheRoom
Configuration alternative

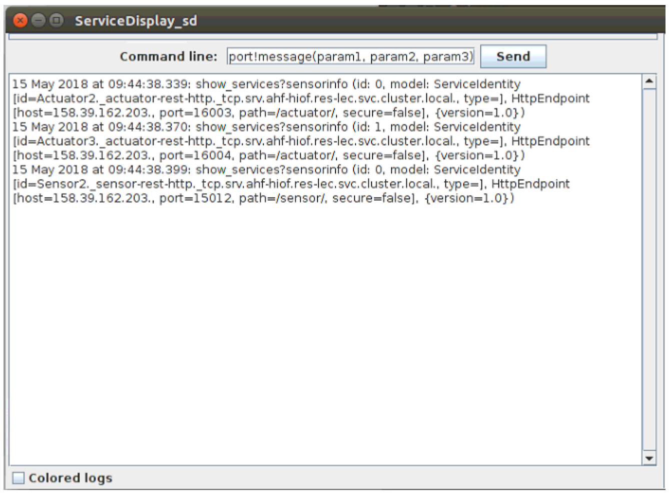
TheRoom
alternative
[✓ Active]

Manual Service

| Service Instance | Service Type | Consume? |
|---|---|---|
| Sensor3._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor4._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor5._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor6._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor7._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor8._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Sensor9._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. | _sensor-rest-http | ☐ |
| Actuator2._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. |  | ✓ |
| Actuator3._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. |  | ✓ |
| Sensor2._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local. |  | ✓ |

Figure 5.12: Management tool showing the “alternative” configuration.

Figure 5.12 shows the management tool where the new active configuration was created.



```
15 May 2018 at 09:44:38.339: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Actuator2._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=16003, path=/actuator/, secure=false], {version=1.0})
15 May 2018 at 09:44:38.370: show_services?sensorinfo (id: 1, model: ServiceIdentity
[id=Actuator3._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=16004, path=/actuator/, secure=false], {version=1.0})
15 May 2018 at 09:44:38.399: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Sensor2._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
```

Figure 5.13: ServiceDisplay showing the services maid available through the “Alternative” orchestration configuration

# Chapter 6

# Evaluation

This chapter presents the results from the implementation explained in Chapter 5 of the designs proposed in Chapter 4. First some empirical data from the three versions of the room will be presented, followed by some findings from using the two arrowhead compliant cyber-physical systems and the arrowhead framework. The evaluation presented in this chapter will be further discussed in Chapter 7.

## 6.1 Comparison of TheRoom, TheRoomSD and TheRoomO

| Aspect | TheRoom | TheRoomSD | TheRoomO |
|---|---|---|---|
| Lines of code | | | |
| Number of things in PSM | 4 | 4 | 4 |
| Number of dependencies | 1 | 2(4) | 2(4) |
| Size of deployable jar file: (in KB) | 132.1 | 137.6 | 142.5 |

Table 6.1: Statistical data about the original and new versions of TheRoom

While the change in size of the deployable jar files is minor, the two new version requires more maven dependencies to function. The dependencies are an existing project, in this case the service application containing the service interfaces and the telldus API. The PSM uses two dependencies, which again has one dependency each. This means that the cyber-physical system requires more available resources, however the the system can now run on a different location than the devices it is connected to, thus removing the problem.

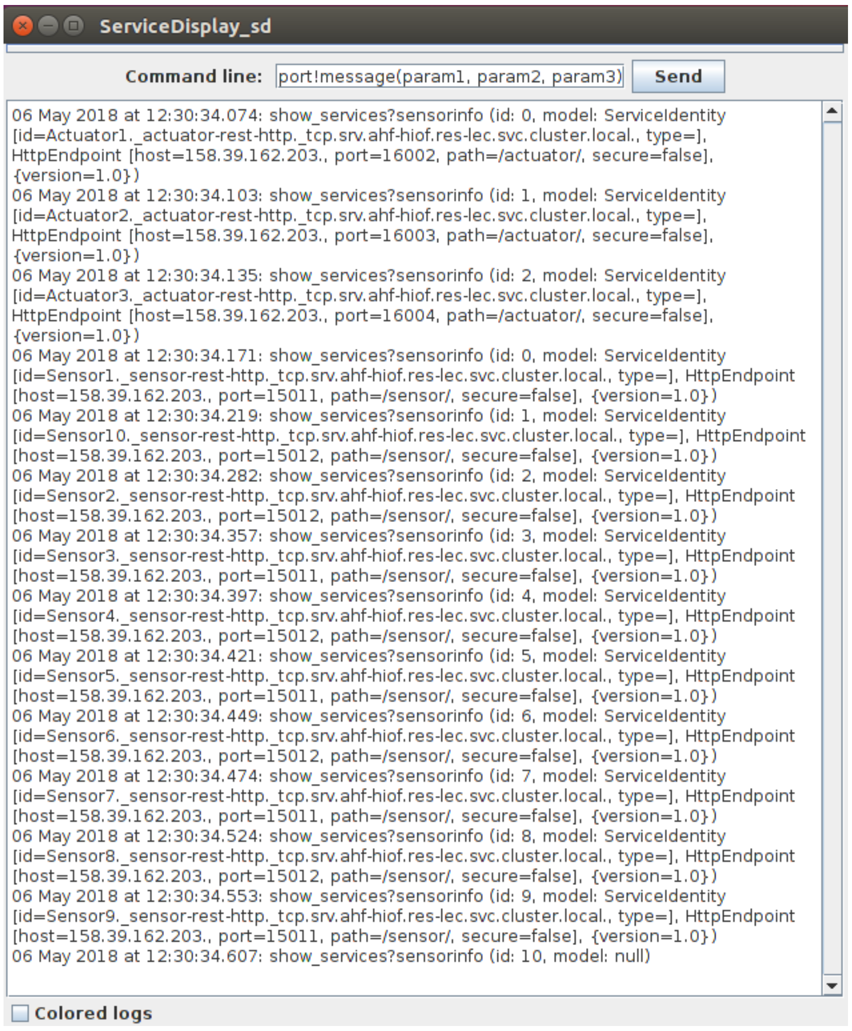
## 6.2 Experiments

To get insight in how the cyber-physical system and the arrowhead framework functions in unison is, three experiments were conducted. The first experiment aimed to see how an expansion of the environment would impact the environment. The second experiment aimed to see how multiple system would work when consuming the same service simultaneously. The third experiment aimed to map the resource consumption of running a cyber-physical system over a prolonged time.

### 6.2.1 Experiment 1: Environmental expansion

In the first experiment, the cyber-physical system was allowed to consume multiple sensor services and actuator services. The two sensor service producers were duplicated and published five times to simulate an expansion of the environment. In addition two more actuators where physically added and published totaling ten sensor service producers and three actuator service producers.

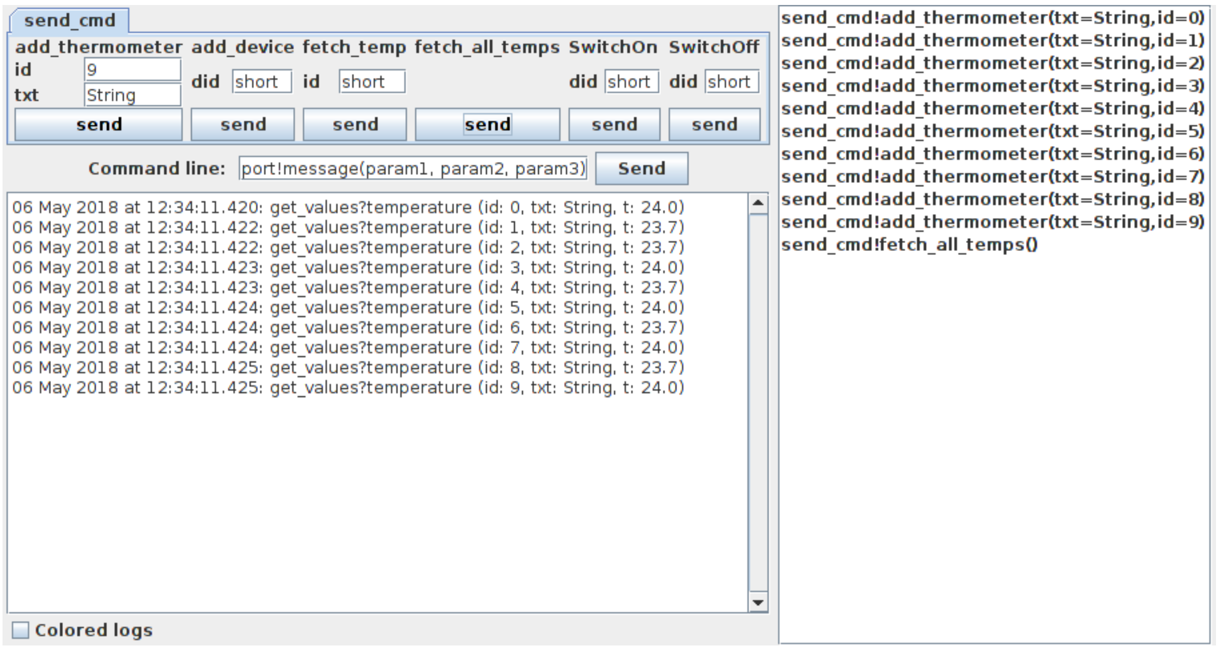
For TheRoom to work using multiple sensors and actuator the PSM from TheRoomO was inserted into TheRoomX1, creating an Arrowhead Compliant version of TheRoomX1 similar to the one created in the first prototype (Section 5.3.1). Unlike the first prototype however, the this verson of TheRoomX1 uses orchestration. Using the management tool, a new System and configuration was created for this.



```
06 May 2018 at 12:30:34.074: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Actuator1._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=],
HttpEndpoint [host=158.39.162.203., port=16002, path=/actuator/, secure=false],
{version=1.0})
06 May 2018 at 12:30:34.103: show_services?sensorinfo (id: 1, model: ServiceIdentity
[id=Actuator2._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=],
HttpEndpoint [host=158.39.162.203., port=16003, path=/actuator/, secure=false],
{version=1.0})
06 May 2018 at 12:30:34.135: show_services?sensorinfo (id: 2, model: ServiceIdentity
[id=Actuator3._actuator-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=],
HttpEndpoint [host=158.39.162.203., port=16004, path=/actuator/, secure=false],
{version=1.0})
06 May 2018 at 12:30:34.171: show_services?sensorinfo (id: 0, model: ServiceIdentity
[id=Sensor1._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.219: show_services?sensorinfo (id: 1, model: ServiceIdentity
[id=Sensor10._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.282: show_services?sensorinfo (id: 2, model: ServiceIdentity
[id=Sensor2._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.357: show_services?sensorinfo (id: 3, model: ServiceIdentity
[id=Sensor3._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.397: show_services?sensorinfo (id: 4, model: ServiceIdentity
[id=Sensor4._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.421: show_services?sensorinfo (id: 5, model: ServiceIdentity
[id=Sensor5._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.449: show_services?sensorinfo (id: 6, model: ServiceIdentity
[id=Sensor6._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.474: show_services?sensorinfo (id: 7, model: ServiceIdentity
[id=Sensor7._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.524: show_services?sensorinfo (id: 8, model: ServiceIdentity
[id=Sensor8._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15012, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.553: show_services?sensorinfo (id: 9, model: ServiceIdentity
[id=Sensor9._sensor-rest-http._tcp.srv.ahf-hiof.res-lec.svc.cluster.local., type=], HttpEndpoint
[host=158.39.162.203., port=15011, path=/sensor/, secure=false], {version=1.0})
06 May 2018 at 12:30:34.607: show_services?sensorinfo (id: 10, model: null)
```

Figure 6.1: The Service Display for the Arrowhead compliant “TheRoomX1” showing the services the application is allowed to consume

As seen in Figure 6.1 the service display shows the services made available for the cyber-physical system. Entering the id numbers in the PIM's mock interface, the ten sensor services was added to the cyber-physical system.



```
06 May 2018 at 12:34:11.420: get_values?temperature (id: 0, txt: String, t: 24.0)
06 May 2018 at 12:34:11.422: get_values?temperature (id: 1, txt: String, t: 23.7)
06 May 2018 at 12:34:11.422: get_values?temperature (id: 2, txt: String, t: 23.7)
06 May 2018 at 12:34:11.423: get_values?temperature (id: 3, txt: String, t: 24.0)
06 May 2018 at 12:34:11.423: get_values?temperature (id: 4, txt: String, t: 23.7)
06 May 2018 at 12:34:11.424: get_values?temperature (id: 5, txt: String, t: 24.0)
06 May 2018 at 12:34:11.424: get_values?temperature (id: 6, txt: String, t: 23.7)
06 May 2018 at 12:34:11.424: get_values?temperature (id: 7, txt: String, t: 24.0)
06 May 2018 at 12:34:11.425: get_values?temperature (id: 8, txt: String, t: 23.7)
06 May 2018 at 12:34:11.425: get_values?temperature (id: 9, txt: String, t: 24.0)
```

```
send_cmd!add_thermometer(txt=String,id=0)
send_cmd!add_thermometer(txt=String,id=1)
send_cmd!add_thermometer(txt=String,id=2)
send_cmd!add_thermometer(txt=String,id=3)
send_cmd!add_thermometer(txt=String,id=4)
send_cmd!add_thermometer(txt=String,id=5)
send_cmd!add_thermometer(txt=String,id=6)
send_cmd!add_thermometer(txt=String,id=7)
send_cmd!add_thermometer(txt=String,id=8)
send_cmd!add_thermometer(txt=String,id=9)
send_cmd!fetch_all_temps()
```

Figure 6.2: The Mock Interface for "TheRoomX1" shwoing the commands sent to the PIM on the right, and the retrieved temperatures, from using the "fetch_all" command, in the main window.

After registering the sensor in the mock interface, the PIM sends a message to the Service-Manager to start consuming the service. The SensorManager calls the service to retrieve the temperature when registering as a consumer of the service, and sends the temperature to the PIM. Usin the "fetch_all" command in the UI, the sensor id given by the ArrowheadManager as well as the temperature measured is presented as seen in .

## Results from experiment 1

In experiment 1 we used the PIM from TheRoomX1 because this was developed for using multiple sensors and actuators. The PSM from TheRoomO was implemented and needed no change to communicate with the new PSM.

The service application was modified to publish the two sensors multiple times to simulate an expansion of the environment. Two additional actuators were also introduced to the environment.

The Arrowhead compliant version of TheRoomX1 using the Orchestration System was fully able to connect to multiple services, retrieve the temperatures, and present them to the user through the mock interface.

### 6.2.2 Experiment 2: Simultaneous consumption

In the second experiment, deploy two cyber-physical systems in the same environment. The two cyber-physical systems are simultaneously consuming the same sensor service producer. The aim is to evaluate how the service providers would handle being consumed from multiple systems.

For multiple rooms to be deployed, we create a copy of TheRoomO and create two new systems with configurations in the management tool. The first TheRoomO will be set to collect the configuration for “Group1”, and the second collects the configuration for “Group2”. Unlike example 2 in Section 5.5.2, the two rooms are running simultaneously using different orchestration rules.

Table 6.2: Setup for experiment

| | available services | | |
|---|---|---|---|
| Configuration | sensor1 | actuator1 | actuator2 |
| Group1 | x | x | |
| Group2 | x | | x |

The PIM in TheRoomO only allows for one sensor and one actuator, so the groups will be assigned devices as seen in Table 6.2. Both configurations are assigned sensor1. Group 1 and Group2 are assigned actuator1 and actuator2 respectively.

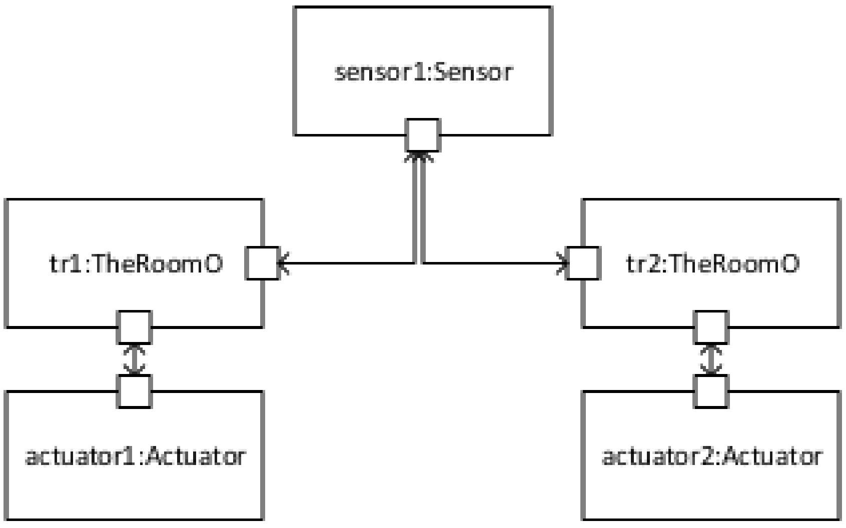


Figure 6.3: An illustration of which systems are connected during experiment 2

Figure 6.3 shows the setup for this experiment. The Group1 orchestration let tr1 connect to sensor 1 and actuator1, and the Group2 orchestration let tr2 connect to sensor 1 and actuator2.

#### Results from experiment 2

Experiment two aimed to investigate how two TheRooms would work when both systems consumed the same sensor service, but different actuators.

The experiment showed that two cyber-physical systems had no problem sharing services. The cyber-physical systems should be able to use the same actuator services as well, however, the platform-independent model will not allow the systems to function properly with the current setup.

### 6.2.3 Experiment 3: resource usage

During the initial testing it was observed an increasing resource usage by the cyber-physical system. To measure the consumption, the service application and TheRoomO was left running for an extended period of time.

For this experiment, both the service application and TheRoomO was restarted. At first, the memory usage was measured at 0,5,10,15 and 30 minutes, and every hour after that. This experiment lasted 13 hours. During the experiment, the comfort temperature of the cyber-physical system was set to 20°C, and the actuator would be turned on and off multiple times.

The experiment started at 08:00 and concluded at 21:00.

#### Results from experiment 3

Table 6.3 shows the measures made during experiment 3. The measurements shows that the new system uses an increasing amount of resources. This will be a problem for a system meant to run continuously. The service application did not increase in resource consumption. Both the service application and the cyber-physical system used minimal CPU-power, so this will not be measured.

Table 6.3: Comsumption of memory in MiB

| system | start | 5min | 30min | 1h | 2h | 3h | 4h | 5h |
|---|---|---|---|---|---|---|---|---|
| TheRoomO | 354 | 354.4 | 441.5 | 441.5 | 705 | 739.1 | 840 | 1000 |
| Service Application | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 |
| system | 6h | 7h | 8h | 9h | 10h | 11h | 12h | 13h |
| TheRoomO | 1000 | 1200 | 1200 | 1200 | 1300 | 1400 | 1500 | 1600 |
| Service Application | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 | 4.1 |

The experiment showed that “TheRoomO”
The increasing resource consumption seemed related to changes in temperature, so the experiment was redone at a shorter period, but with shorter periods between measurements, and a manipulation of the sensor.

#### Comparing the new and old “TheRoom”

The second resource consumption experiment was done over an hour, and measured every 5 minutes. During the run, the temperature the thermometers sent varied from 15°C to

35°C. The comfort temperature was set at 20°C. The same experiment was done on the original “TheRoom” to have a basis of comparison.

Table 6.4: Comsumption of memory in MiB

| system | start | 5min | 10min | 15min | 20min | 25min | 30min | 35min | 40min | 45min | 50min | 55min | 60min |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TheRoomO | 347.4 | 417.5 | 416.9 | 445.8 | 481.9 | 485.1 | 500.1 | 542.8 | 544 | 576.1 | 576.6 | 577.8 | 579.2 |
| TheRoom | 326.4 | 325.9 | 326.8 | 327.7 | 328 | 329.2 | 329.6 | 330.7 | 330.9 | 331.6 | 331.6 | 331.9 | 332.5 |

The experiment showed that both “TheRoom” and “TheRoomO” did increase the resource consumption during the course of one hour. However, whilst the original “TheRoom” had an increase of 1.9% “TheRoomO” had an increase of 66.5%

# Chapter 7

# Discussion

This chapter starts with a discussion on the requirements for porting an existing cyber-physical system to a framework based on the work done in Chapter 4 and Chapter 5. Followed by a discussion on how implementing a framework could influence the master course “Modeling Cyber-Physical System” based on the evaluation done in Chapter 6. This chapter is concluded with a section presenting future work.

## 7.1 RQ1: What is required for an existing Cyber-Physical system, exemplified by the running example “TheRoom”, to be ported onto the Arrowhead Framework?

To answer research question one, the PSM was studied to understand how the cyber-physical system should interact with the Arrowhead Framework. Based on this study, two designs featuring different ways to retrieve the information about the sensors where proposed and implemented. This section discusses the findings from this process.

### 7.1.1 Deploying the framework

Using the framework requires the mandatory core services to be deployed to create a local Arrowhead cloud. The core services were first deployed locally on a workstation using docker deployment and later deployed on the servers on the premises of Østfold University College.

### 7.1.2 Implementing the framework to the Cyber-Physical System

Implementing the framework into the platform-specific model was possible, but tedious, by using ThingML. The necessary classes needed to communicate with the service registry, orchestration system and services were made available by defining them as custom datatypes in ThingML. The functions made available through the classes where called by writing directly into the source code. Two functioning Arrowhead compliant cyber-physical systems was developed.

#### TheRoomSD

“TheRoomSD” uses the service registry to retrieve the service identity to all services published with the service types “sensor” and “actuator”, and then used the service identities to gather the service information. This resembles the tellstick initiation from the original “TheRoom”.

#### TheRoomO

“TheRoomO” uses the orchestration system to retrieve a set of rules. The rules functions as service identities of the services assigned to the system. The service information is retrieved from the service registry and presented to the user. The orchestration assigns services to the system, which limits the services presented to the user.

## 7.2 RQ2: In what ways can the Arrowhead Framework influence the course “Modeling Cyber-Physical Systems”?

To answer research question 2, experiments were designed to test some situations related to the course to understand how implementing a framework could benefit it.

### 7.2.1 Making the development more flexible

Implementing a framework introduces possibilities during the development of the cyber-physical systems that the course only had while simulating the devices. Letting all students in a group develop their system simultaneously. Using a PSM adapted to a framework makes the physical devices available without the necessity of dongles or simulations.

The experiment showed that it is possible for multiple cyber-physical systems to connect to multiple services simultaneously. This is an important functionality because as a part of the course, one of the tasks is to develop a platform-independent model that lets the cyber-physical system control different actuators based on input from multiple sensors. Now, this can be done using real sensors and actuators.

Using orchestration rules, a set of services can be assigned to each group. The services are not limited to one user, and a single service can be assigned to multiple groups if wanted. This will allow each group to be granted access to a specific actuator and a shared pool of sensors.

By deploying the framework on a local server, the services are accessible from within the network, and from the outside by the use of VPN.

### 7.2.2 Scaling the environment

By introducing an IoT network of devices, the environment can be scaled up, not only in the number of devices on the network, but also allowing the sensors to cover a larger area beyond the range of a single transceiver, and still be available for a cyber-physical system.

The number of devices can also be scaled up. The devices used in the course communicates over the rf 433MHz protocol. During the course, it was stated that this protocol was outdated, but changing to new devices could prove expensive. By availing a framework, new devices using different protocols can be introduced into the environment and be used in conjunction with each other without changing the cyber-physical system. This would limit the number of new devices needed to be bought, and by creating similar service interfaces, the system can get data from the different services without knowing what protocol the device is using.

### 7.2.3 Coping with interference

During the course one major problem with using physical devices was interference. Having multiple transceivers sending and receiving signals to and from the devices caused a slow data transfer if any. Letting the data be transferred over the network will remove this problem, letting the students develop the cyber-physical system without relocating themselves.

### 7.2.4 Centralizing the resources

Centralizing the resources lets an administrator manage the network through the management tool. This can be used to create and manage orchestration rules. However, the framework is dependent on the core services being functional, and deploying the framework on a server, can lead to new potential weak points. Keeping the server running and well maintained will be important.

## 7.3 Future work

There is a number of ways the cyber-physical systems presented in this thesis can be improved further. This section discusses some of the possible improvements to give an idea of how the systems can be developed further.

### 7.3.1 Implementing the Authorization System

The Arrowhead Framework offers two-way authentication through the authorization system. This was not implemented during this thesis. Without the authentication, it is possible to access the devices by a web browser. Introducing the authorization system will lead to a more secure and robust IoT-network.

### 7.3.2 Update service list during runtime

The service information is retrieved during the Arrowhead Initialization Phase. At the end of this phase, the service information is sent along with the initialization message. By allowing the cyber-physical system to update this list during runtime will make it more flexible. If, for example, an actuator fails, or a sensor run out of power, implementing this functionality makes it possible to assign a new service using the orchestration system and retrieve it without restarting the entire application.

### 7.3.3 The resource requirements

Implementing the framework lead to the cyber-physical system using an increasing amount of memory during runtime. In experiment 3, it was concluded that the resource usage increased by 66% over 13 hours. Finding a way to limit the usage, or release the resources, will be necessary for creating a system that can run indefinitely.

### 7.3.4 Improving the Service Application

Today, the IP-address to the device running the service application is hardcoded into the source code. Relocating or turning of the device for a prolonged time may lead the device to receive a new IP-address. By implementing functionality to update the address during startup, will make the service application easier to distribute, and deploying new devices much faster.

# Chapter 8

# Conclusion

In this Thesis, I presented two prototypes of Arrowhead Compliant Cyber-Physical Systems, “TheRoomSD”, and “TheRoomO”. The first prototype, “TheRoomSD” was only using the Service Discovery to retrieve information about the services published on the network. The second prototype, “TheRoomO”, used the orchestration system to retrieve information from the services assigned to the cyber-physical system. To evaluate how implementing the framework could influence the masters course “Modeling Cyber-Physical Systems”, three experiments were conducted.

To understand how the new cyber-physical systems would work in an educational setting, three experiments were conducted to evaluate the prototypes. The first experiment tested the cyber-physical systems ability to handle multiple services simultaneously. The second experiment tested if the services could handle being consumed by multiple cyber-physical systems at once. The third measured the resource consumption of the prototype.

The framework could influence the master course in different areas. Making the development more flexible by using the orchestration to assign a set of devices to each group, and making them available from within the local cloud without adding hardware to their machines. Making it easier to scale the environment, not only by making it possible to add devices using different protocols and making them interoperable but also by making it possible to deploy devices over a larger area. And finally, solving the interference problem from having multiple transceivers nearby.

# Bibliography

[1] *Arrowhead*. http://www.arrowhead.eu/. Accessed: 2018-05-10.

[2] *Arrowhead Wiki*. https://forge.soa4d.org/plugins/mediawiki/wiki/arrowhead-f/index.php/Main_Page. Accessed: 2018-05-10.

[3] *Azure IoT Fundamentals*. https://docs.microsoft.com/en-us/azure/iot-fundamentals/. Accessed: 2018-05-10.

[4] Jerker Delsing. *IoT Automation: Arrowhead Framework*. CRC Press, Feb. 9, 2017. 366 pp. ISBN: 978-1-315-35086-8.

[5] H. Derhamy et al. “A survey of commercial frameworks for the Internet of Things”. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. 2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA). Sept. 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301661.

[6] M. Hermann, T. Pentek, and B. Otto. “Design Principles for Industrie 4.0 Scenarios”. In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. 2016, pp. 3928–3937. DOI: 10.1109/HICSS.2016.488.

[7] *IoTivity: Home*. https://www.iotivity.org/. Accessed: 2018-05-10.

[8] Barbara Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007. URL: http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf.

[9] Chieh-Jan Mike Liang et al. “SIFT: Building an Internet of Safe Things”. In: *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. IPSN ’15. Seattle, Washington: ACM, 2015, pp. 298–309. ISBN: 978-1-4503-3475-4. DOI: 10.1145/2737095.2737115. URL: http://doi.acm.org/10.1145/2737095.2737115.

[10] *ThingML*. http://thingml.org. Accessed: 2018-05-10.

[11] Frank Truyen. *The Fast Guide to Model Driven Architecture*. Cephas Conslting Corp. 2006.

[12] A. Vasilevskiy et al. “Agile development of home automation system with ThingML”. In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. 2016, pp. 337–344. DOI: 10.1109/INDIN.2016.7819183.

[13] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: ACM, 2014, 38:1–38:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268. URL: http://doi.acm.org/10.1145/2601248.2601268.