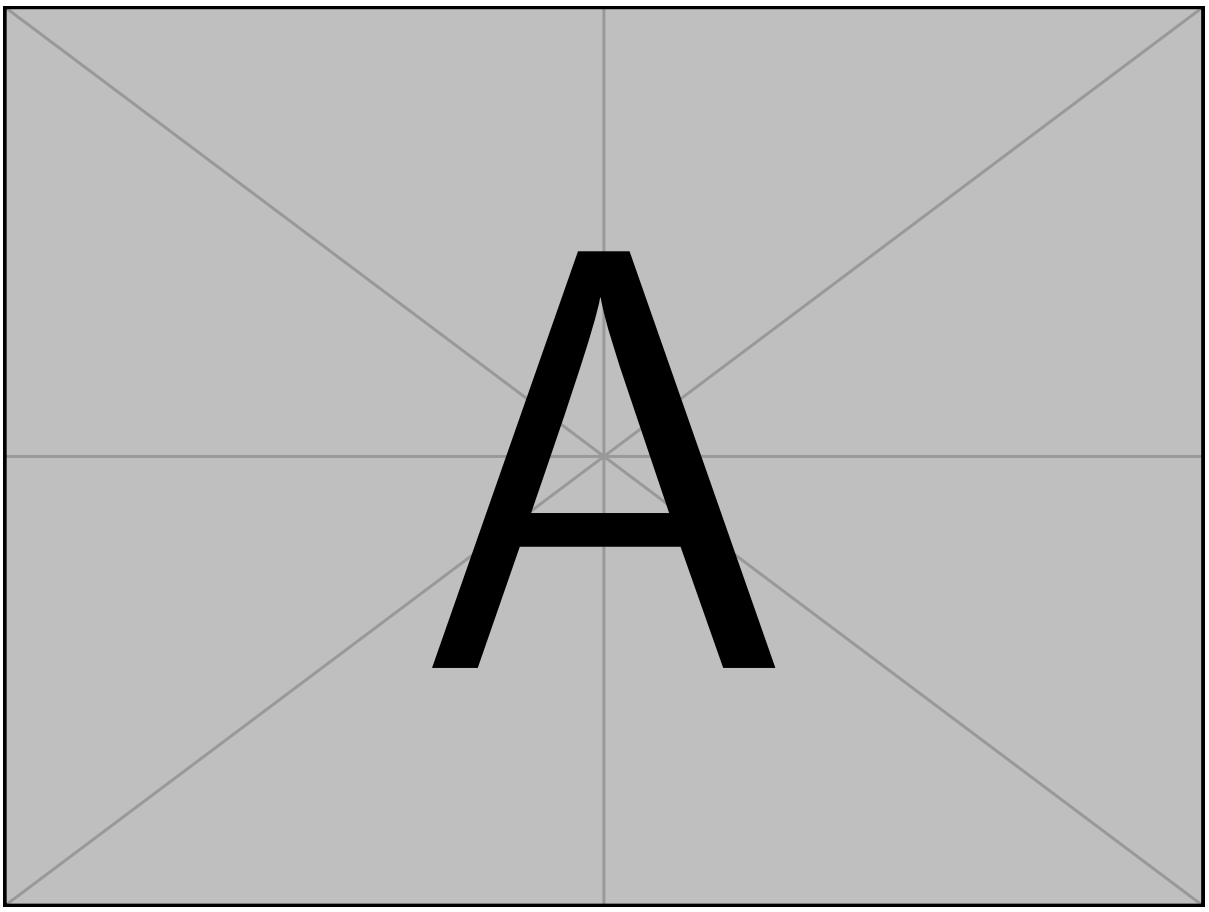
Embedded IoT for Eclipse Arrowhead



Albin Martinsson

Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisor:

Jan van Deventer

To my dad Bengt-Göran Martinsson a special thanks for proof reading is required. A special thank to my little sister Hedvig Martinsson for drawing the cover image is also required.

Abstract

This thesis will investigate the possibility of connecting an embedded device, STM32 B-L4S5I-IOT01A IoT discovery node, to a local Eclipse Arrowhead framework cloud. This thesis will also compare the benefits and limitations of using the Eclipse Arrowhead framework to commercially available solutions such as Amazon's Amazon Web Services and Microsoft Azure.
The world is on the verge of a new industrial revolution, often referred to as Industy 4.0, moving towards a more decentralized and software-oriented means of production. Incorporating System of Systems, Cyber-Physical Systems, and embedded software technologies will form the backbone and be an inherent part of every value chain.
The Eclipse Arrowhead framework contains many examples in various languages and technologies but lacks an example of a specific piece of hardware connecting to a local Eclipse Arrowhead cloud. Therefore, a project with the clear intent to showcase both the capabilities and possibilities of Cyber-Physical systems and the Eclipse Arrowhead framework is needed.
The system consists of three major parts: the stm32 board, a Python flask app, and the Eclipse Arrowhead framework. The main objective of the Eclipse Arrowhead framework is to connect the consumer and the provider in a safe and structured way. The provider is built with C/C++ using ARMs' mbed os.
The response time of the different frameworks, Eclipse Arrowhead framework and Amazon Web Services, was measured. We made a thousand attempts to form an adequate basis for an average response time. In addition to presenting the average response time, we will calculate the maximum and minimum response times to understand the different frameworks' performance further.
The thesis also examined the benefits of using the Eclipse Arrowhead framework compared to its competitors Amazon Web Services and Microsoft Azure. The thesis showed some benefits in terms of response time when running a local cloud instead of using a remote service such as Amazon Web Services, a 17.5 decrease in average response time was recorded. Maximum and minimum response times decreased by 1.9 and 134 times, respectively.

Contents

Acknowledgments

The research conducted in this thesis has been funded by the Arrowhead Tools project.

Chapter 1

Introduction

This thesis will examine the opportunity and possibility to connect an embedded IoT device to a local Eclipse Arrowhead framework cloud. This project will use the STM32 B-L4S5I-IOT01A IoT discovery node as a development board, running the Mbed-OS 6.

1.1 Background

According to Artemis-IA, the world is on the verge of a new industrial revolution, moving towards a more decentralized and software-oriented means of production.[1] This fourth new, and in some sense planned, industrial revolution is called Industry 4.0 according to Lasi.[2] Cyber-Physical systems act as a bridge between the data-rich cybernetic world and the technology-rich physical world Artemis-IA means. Artemis-IA adds that a differentiating factor between traditional embedded systems and Cyber-Physical systems is their scale, where traditional embedded systems have a more limited scale. Cyber-Physical system, on the other hand, has a much larger scale, including interconnected embedded systems, human-, and socio-technological systems as well Artemis-IA adds.[1]

For Europe to compete with the rest of the world's larger economies with initiatives, for instance, China's Made in China 2025, Europe needs to invest in the technologies mentioned above Artemis-IA adds. A shift away from proprietary solutions towards collaborative solutions is also needed means Artemis-IA in their report Embedded intelligence.[1]

1.2 Motivation

The need for a European incentive promoting Industry 4.0 is clear. According to their website, the Arrowhead Tools project aims for digitalization and automation solutions for the European industries.[3] The Arrowhead Tools project uses the open-source Eclipse Arrowhead framework, further contributing to the collaborative solutions needed for the European economy defined in the previous section.

The Eclipse Arrowhead framework contains many examples in various languages and

technologies, mainly java. Python, C#, and C++ have client libraries and example code developed for them, which you can find on the project's GitHub page.[4]

However, there is no client library or code example for a specific piece of hardware to connect to a local Arrowhead cloud fast and easy, showcasing the capabilities of this project. Therefore, a project with the clear intent to showcase both the capabilities and possibilities of Cyber-Physical systems and the Eclipse Arrowhead framework is needed.

This thesis will examine the possibilities of having a ready-made example to compile and run on a specific hardware platform that connects a local Arrowhead cloud as a proof of concept. Much like the 'Getting started with' examples from Amazon Web Services and Microsoft Azure.[5, 6]

1.3 Problem definition

This project aims to investigate the possibilities, benefits, and limitations of using the Eclipse Arrowhead framework on embedded devices in contrast to commercially available solutions such as Amazon's Amazon Web Services and Microsoft Azure.

1.4 Equality and ethics

The ability to own and control your data is becoming rarer and rarer these days, with giant corporations establishing their cloud services. As a consumer, one always takes a risk when pushing sensitive data to a cloud owned by someone else. Corporations should not infringe the right to possess your data. The Eclipse Arrowhead framework and the use of local clouds move the storage of your data from giant corporations to your own.

1.5 Sustainability

The use of small embedded devices instead of monolithic machines today provides a much-needed decrease in energy consumption for more immense industries. On a greater scale, the use of IoT devices would also enable preventive maintenance of components, reducing both the cost and materials required for maintenance later on.

1.6 Delimitations

1.6.1 Security

This thesis does not cover a solution to the numerous security risks and issues associated with IoT devices.

1.6.2 Core systems

This thesis will also only cover the three core systems of the Eclipse Arrowhead framework, which are the service registry, authorization, and orchestrator. The STM32 B-L4S5I-IOT01A IoT discovery node will not host the Arrowhead framework on the board itself since the Arrowhead framework is too resource-heavy for such a small device. Instead, the board will connect to a local Arrowhead cloud hosted by another computer in the same network.

1.6.3 Intercloud connection

It will also only cover connection within the same local Arrowhead cloud, intracloud, instead of using multiple clouds, intercloud. Intercloud connection requires two more Arrowhead systems, gateway and gatekeeper, to operate and the configuration of those systems is beyond the scope of this thesis.

1.7 Thesis structure

Chapter 2 presents related work and conducts a literature review of IoT, Industry 4.0, security, and the Eclipse Arrowhead framework. Chapter 3 covers theory, describing what technologies and scientific methods this thesis uses. Chapter 4 covers implementation, describing the design of different systems used in this thesis from a software engineering perspective. Chapter 5 presents an evaluation of the experiment conducted. Chapter 6 contains a discussion about the solution to the problem stated in chapter 1, possible alternative solutions, and how the results affect the industry. Chapter 7 presents the conclusion of the work done in this thesis. The chapter also describes how to investigate further the questions raised in this thesis. In chapter 8, there is a list of references used in this thesis.

Chapter 2

Related work

2.1 Internet of things

Cluster of European research projects regarding the Internet of Things:

> 'Things' are active participants in business, information and social processes where they are enabled to interact and communicate among themselves and with the environment by exchanging data and information sensed about the environment while reacting autonomously to real/physical world events and influencing it by running processes that trigger actions and create services with or without direct human intervention.[7]

Gubbi et al. define the Internet of Things as:

> Interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications. Achieved by seamless large scale sensing, data analytics, and information representation using cutting edge ubiquitous sensing and cloud computing.[7]

2.2 Industry 4.0

Lasi argues that the term industry 4.0 was coined beforehand as a planned fourth industrial revolution.[2] The use of internet of things devices, IoT devices from now on, and cyber-physical systems, CPS from now on is what defines the fourth industrial revolution Vadiya means.[8] See figure x for a short historic overview of previous industrial revolutions.

According to Vaidya, industry 4.0 promotes connecting sensors and devices to the internet and other sensors or devices.[8]

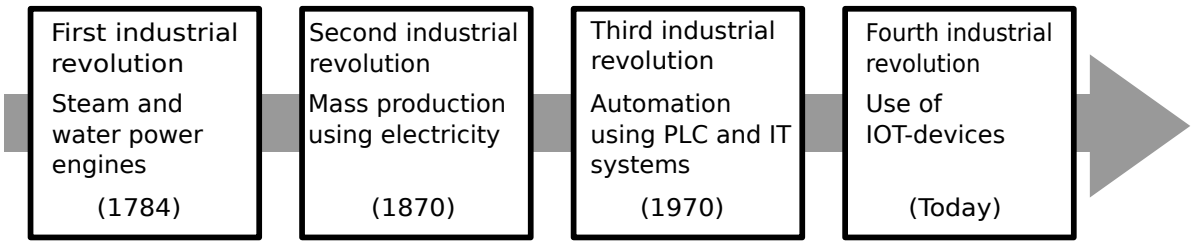


Figure 2.1: Historic overview of previous industrial revolutions

Hozdić states that a sensor is a device capable of providing an appropriate output in response to a measured value. One key feature of an intelligent sensor is that it increases information processing, and it processes the information at a logical level, Hozdić argues. An Intelligent sensor is capable of executing actions based on the measured value in contrast to regular sensors, making them easier to set up and use means Hozdić.[9]

Hozdić defines a cyber-physical system, CPS, as a new generation of a system that integrates physical and computer abilities. A cyber-physical system consists of two parts, one cybernetic and one physical. The cybernetic aspect of the system is a summation of logic and sensor units. In contrast, the physical part of the system is the summation of the actuator units, Hozdić adds. Xu et al. state that cyber-physical systems are a vital part of Industry 4.0. In contrast to the simple embedded systems of today will be exceeded due to advances in CPS that enable enhanced capability, scalability, adaptability, resiliency, safety, usability, and security.[10] Hozdić argues that the CPS can share and receive information from intelligent sensors connected to digital networks, enabling and forming an internet of things.[9]

2.3 Arrowhead framework

Delsing defines a local cloud as a self-contained network with at least the three mandatory systems deployed, more on those in a later paragraph. Delsing et al. also argue that the three mandatory core systems running a local cloud also need at least one application system deployed.[11]

To further understand what the Eclipse Arrowhead framework aims to accomplish, introducing the terms services and systems is necessary. Delsing et al. define a system as providing or consuming a service. Furthermore, Delsing et al. define a service as conveying information between a provider and a consumer.[11]

The Eclipse Arrowhead framework, Arrowhead from now on, consists of three mandatory core systems according to Delsing et. al. To fully operate a local cloud as defined in the previous section it must, according to Delsing, contain:

- Service registry system.

- Authorization system.
- Orchestration system.[11]

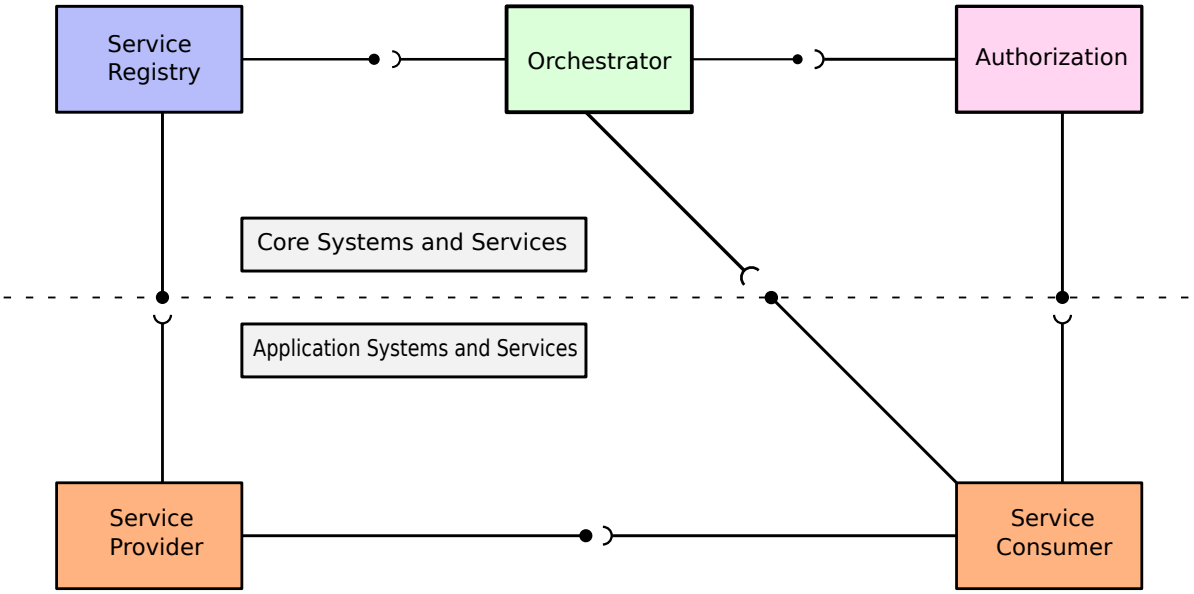


Figure 2.2: The core systems of the Eclipse Arrowhead framework

The dynamics between the consumer and provider system will follow in the theory section.

2.4 Amazon Web Services

According to the Amazon webpage, the AWS IoT Core connectivity services consist of the device gateway, message broker, and rules engine.

The device gateway enables devices to securely, via X.509 certificates, and efficiently communicate with AWS IoT Amazon.

Amazon states that the message broker provides a secure way for devices and AWS IoT applications to publish or receive, subscribe as known in MQTT, messages from each other. The message broker also distributes device data to devices that have subscribed to it and to other AWS IoT Core services, such as the rules engine Amazon adds.

The Rules engine connects data from the message broker to other AWS services, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and AWS Lambda, according to Amazon.[12]

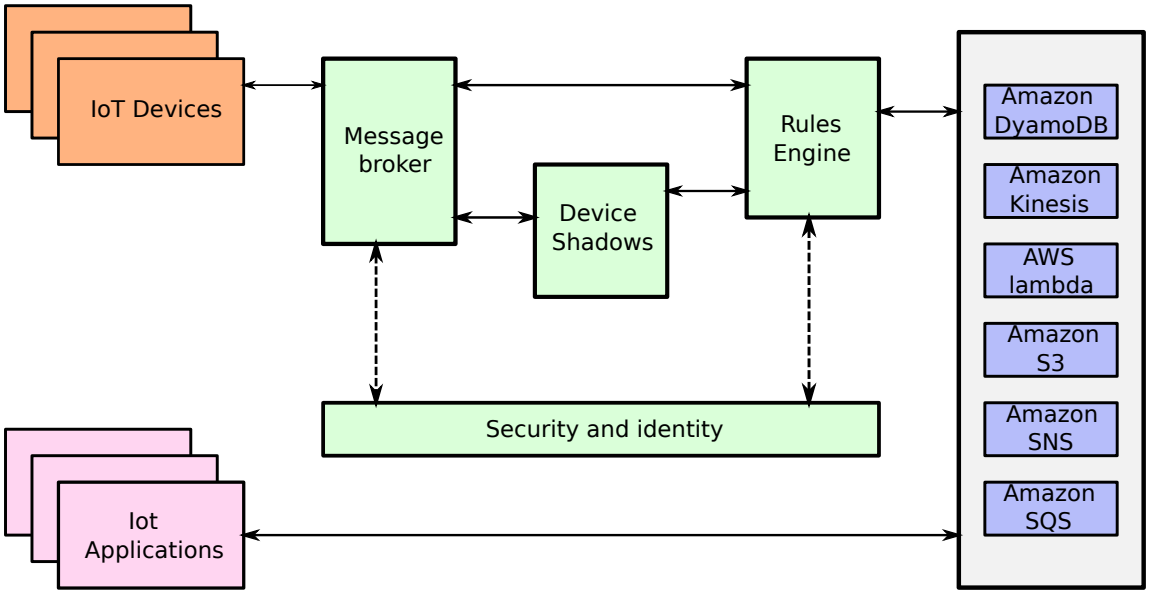


Figure 2.3: The core systems of the Amazon Web Services

2.5 Security

Meneghello et al. argue that the increasing number of IoT devices and the pervasive nature of new smart home or healthcare devices can pose a real threat to the users' integrity. Meneghello et al. define sensitive information as a video recording of the user's home, location, access to buildings, health monitoring, and industrial processes.[13]

Meneghello et al. divide the security requirements of an IoT system into three different operational levels: the information, access, and functional level. The information level should guarantee the preservation of the system's integrity, anonymity, confidentiality, and privacy. To maintain integrity, no alteration of the messages can occur during transmission. The identity of the data source and the clients' private information remains hidden. Third parties cannot read that data, Meneghello et al. argue. The access level guarantees that only legitimate users can access the network and the devices associated with that network. It also guarantees that users within the network only use resources they can use, Meneghello et al. state. The functional level should guarantee the continued functionality of a network even in the case of malfunction or a malicious attack, Meneghello et al. add.[13]

Zhang also argues that privacy is a big concern with IoT devices and suggests two solutions data collection policy and data anonymization. A policy that describes how data collected from the devices would restrict data flow, ensuring privacy preservation, Zhang

states. Data anonymization means that private information sent by the IoT devices is either encrypted or conceals the relation of the data and its owner according to Zhang.[14]

Meneghello et al. argue that one of the main aspects of security within IoT is to ensure that the data sent is the data received and that the data has not been tampered with or read during transmission. The most critical operation to guarantee is encryption, which converts the message sent in plain text to an encrypted message only readable with a decryption key, Meneghello et al. state. Meneghello et al. state that there are two mechanisms for encryption, symmetric and asymmetric. Symmetric encryption uses the same key for encryption and decryption, sharing it with both the sender and receiver. On the other hand, asymmetric encryption only shares the public key, and the sender and receiver have their private keys Meneghello et al. means.[13]

Hassija et al. state the importance of end-to-end encryption and its challenges for IoT systems. End-to-end encryption is required to ensure the confidentiality of the data. The application should not let anyone except the intended recipient read the messages sent Hassija adds.[15]

Noor, Meneghello, and Zhang state the importance of authentification in IoT systems.[16, 13, 14] Noor adds that 60% of all IoT systems use authentification to grant access to the user.[16] Zhang argues that public key cryptosystem provides more security than symmetric encryption schemes but has the drawback of having high computational overhead.[14]

Noor argues that conventional cryptographic primitive is unsuitable for IoT devices due to their lack of computational power and limited battery life and memory capacity.[16] With IoT devices lacking capabilities as background Noor, Meneghello, and Zhang all agree that a push for lightweight cryptography is required to ensure the security of these devices.[16, 13, 14]

2.6 Communication

MQTT, Message Queue Telemetry Transport, is a lightweight messaging invented by IBM suitable for IoT according to Wukkadada.[17] MQTT is a publish/subscribe protocol that requires a minimal footprint and bandwidth to connect an IoT device according to HIVEMQ.[18] MQTT consists of an MQTT broker and MQTT clients, where the broker is responsible for sending messages between the sender and its recipients.[17] On the other hand, a client publishes a message to the broker that other clients can subscribe to HIVEMQ add.[18]

HTTP, HyperText Transfer Protocol, is a request/response protocol consisting of clients and servers that communicate by exchanging individual messages. The clients are responsible for the requests, and the servers are responsible for the response Mozilla developer network clarifies. [19] In contrast to the lightweight MQTT protocol with low overhead and bandwidth, HTTP can cause serious bandwidth issues, Wukkadada adds. [17] The most significant benefits to using HTTP are that it supports the RESTful Web architec-

ture and a globally accepted web messaging standard Naik suggests. [20]

Naik argues that HTTP exceeds MQTT in message size, message overload, power consumption, resource requirements, bandwidth, and latency. All things that are considered adverse for a protocol. On the other hand, HTTP exceeds MQTT in interoperability, standardization, security, and provisioning, Naik adds.[20] All things that are considered positive for a protocol. Shariatzadeh argues that that HTTP may be expansive for many IoT devices, but it can be beneficial due to the interoperability since initially developed for the web.[21] Wukkadada also points out the lower power consumption of the MQTT protocol but adds that the more lengthy HTTP protocol can be easier for developers to understand. Wukkada drives home the point of choosing MQTT for IoT devices.[17]

Chapter 3

Theory

The theory behind the implementation in this thesis consists of performing correct SQL queries to appropriate tables in the Eclipse Arrowhead framework database with the help of Swagger UI REST API.

3.1 Eclipse Arrowhead framework

As mentioned in the related work section, the Eclipse Arrowhead framework consists of three mandatory core systems

- Service registry system.
- Authorization system.
- Orchestration system.

3.1.1 Service registry system

The service registry system is responsible for enabling discovery and registring services, Delsing et al. state. According to the Eclipse Arrowhead projects own GitHub page, the service registry system provides the database which stores the offered services in the local cloud.[22] The Github page also states the three main objectives of the service registry system are:

- To allow the application system to register available services to the database.
- Remove or update available services from the database.
- Allow application system to use the lookup functionality of the registry.

3.1.2 Authorization system

According to the projects Github page, the Authorization system contains two databases for keeping track of which system can consume services from which other systems, de-

pending on whether the Application system is in the same cloud or not. The GitHub documentation also states that if the authorization happens within the same cloud, it is called intra-cloud authorization, and if it happens across two local clouds, it is called inter-cloud authorization.[22]

3.1.3 Orchestrator system

The Orchestration system is responsible for pairing and finding service providers and consumers, Delsing et al. declare. Delsing et al. continue to state that the orchestrator also stores the orchestration requirements and the resulting orchestration rules.[11] The project's documentation argues that the main objective of the orchestrator system is to find an appropriate provider for the requesting consumer system.[22]

The documentation also states that there are two types of orchestration, store orchestration and dynamic orchestration. Store orchestration uses the database orchestration store to find predefined orchestration rules. On the other hand, dynamic orchestration searches the entire local cloud, or even other clouds, to find the matching provider.[22]

3.2 Arrowhead database

One can view the Eclipse Arrowhead framework as a series of database tables to connect. One table relevant to all the core systems is the service_interface table. It correlates a connection interface, i.e., HTTP-INSECURE-JSON, HTTP-SECURE-JSON, and HTTPS-SECURE-JSON, to a specific ID used later on.

3.2.1 Service registry system tables

The tables in the Arrowhead database relevant to the service registry system are

- system_ keeps the information about consumers and providers.
- service_registry contains information about the different services.
- service_definition, stores service definition name, and ID.
- service_registry_interface_connection correlates a service registry ID to an interface ID.

3.2.2 Authorization system tables

The tables in the Arrowhead database relevant to the authorization system are

- authorization_intra_cloud, adds authorization rules for the provider, consumer, and service definition. Dictates which consumers are allowed to connect to which providers. It also dictates the services the consumer is allowed to use.

- authorization_intra_cloud_interface_connection correlates an intracloud authorization ID to an interface ID.

3.2.3 Orchestration system tables

The tables in the Arrowhead database relevant to the orchestrator system are

- orchestrator_store, contains orchestrator store entry, with information about which endpoints the consumers can use.

3.3 Swagger UI

The Eclipse Arrowhead framework has integrated Swagger UI into their core services. Meaning that the databases can be accessed and altered using HTTP methods instead of SQL queries. Each core system has its swagger UI page, which serves as a visual, user-friendly REST API. A REST API, Representational state transfer, is a way to structure an API and consists of the following main principles according to Wikipedia.[23]

- Each resource has its own URI.
- The resources have a common interface for commands between client and server. These are
  - POST creates a new state for a resource.
  - GET requests the state of a resource.
  - PUT replaces the state of an already created state.
  - DELETE deletes a resource state.

Instead of sending SQL-queries, the Swagger UI uses JSON, JavaScript Object Notation, and how we should construct these will be covered in the implementation section.

3.4 ARM Mbed

The STM 32 board needs to be connected to the internet to use the Swagger UI. The ARM mbed OS 6 provides the functionality needed for this.

Mbed OS 6 uses a hardware abstraction layer, HAL, to enable the microcontroller's most essential parts, such as timers. When compiling your code, libraries, drivers, and support for standard peripherals for microcontrollers are imported when using hardware supported by the Mbed OS 6. This foundation enables the user to write applications against a shared set of APIs. Another advantage of using supported hardware is the retargeting layer and boot process integration, so application development feels similar to C or C++ development.

ARM Mbed OS enables the user to use a range of connectivity tools and suits. This thesis uses Wi-Fi, but there is support for Bluetooth, NFC, and much more on the STM32 B-L4S5I-IOT01A board. Mbed OS connectivity and network stack offer a stable core of existing connectivity technologies, perfect for even the most demanding and versatile IoT applications.

The Mbed online compiler allows the user to develop their code online, independent of the platform. The online compiler also supports the importation of code from GitHub, making it easier for the users to import this code and genuinely use this as a 'Getting started with the Eclipse Arrowhead framework' example.

3.5 Performance measurement

Response time is measured to compare the performance of the three different IoT frameworks. Due to the vastly different implementation of the three frameworks, Amazon Web Services, Microsoft Azure, and Eclipse Arrowhead framework, the only thing worth measuring is the response time when pinging the different frameworks.

The Test-Connection command in power-shell sends four 32-bit packets to the specified destination. Iterating the Test-Connection 250 times gives us 1000 samples, a sufficient amount of attempts to produce an average response time. The average is calculated with $\frac{\sum \text{response times}}{\text{number of response times}}$. In addition to presenting the average response time, we will calculate the maximum and minimum response times to understand the different frameworks' performance further. The results of these calculations one can read in detail in Chapter 5.

Chapter 4

Implementation

The system consists of three major parts: the stm32 board, a Python flask app, and the Eclipse Arrowhead framework. The main objective of the Eclipse Arrowhead framework is to connect the consumer and the provider in a safe and structured way. The provider is built with C/C++ using ARMs' mbed os, and mainly the mbed-http library.

4.1 System architecture

4.1.1 Services

The provider offers two services to the consumer. The first one is sending the temperature from the LPS22HB temperature and pressure sensor. The service URI of this service is /temperature, which will send the temperature reading as an integer. A sequence diagram illustrating the implementation of the temperature service.

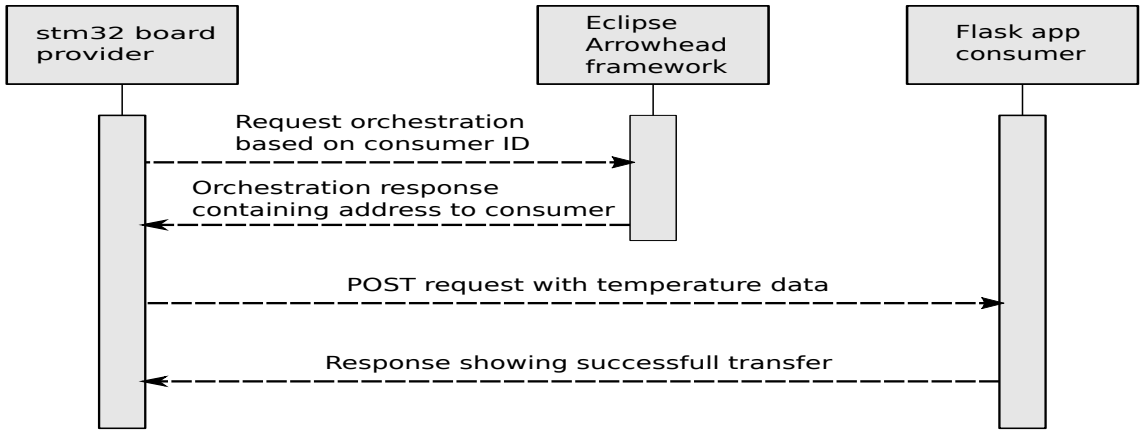


Figure 4.1: Sequence diagram of the process of connecting the consumer and provider through the temperature service.

The second service is turning on and off a LED based on the state of that LED. The service URI of this service is /LED, and the desired state of the LED can be ON or OFF. The service will send the current state of the LED to the consumer, and the consumer inverts the state and returns it to the provider. The provider acts on that action, turning the LED on or off. The sequence diagram below shows how the implementation of this service.

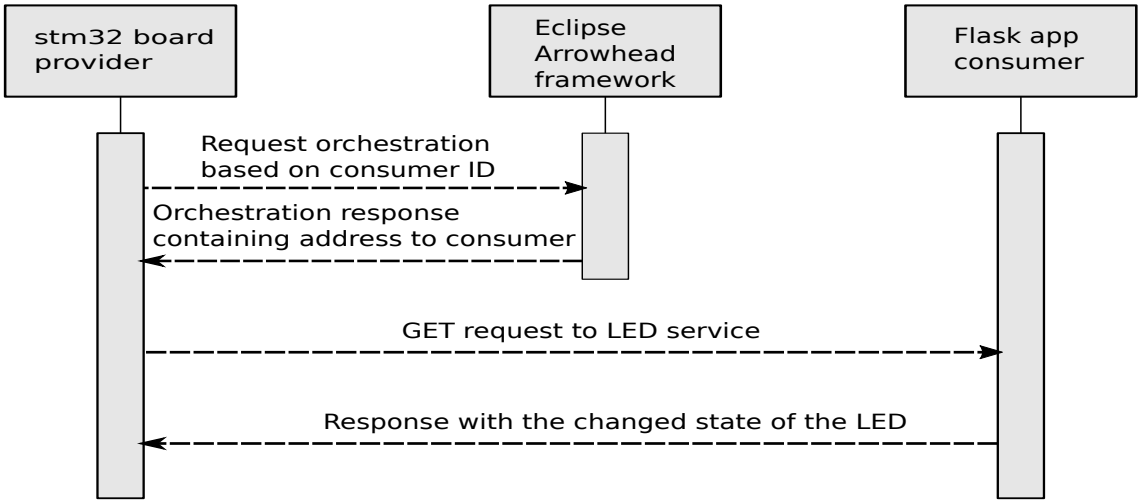


Figure 4.2: Sequence diagram of the process of connecting the consumer and provider through the LED service.

4.1.2 Sequence of execution

Since the core systems are dependent on each other, the order of the queries to the database matters a lot. The board must follow this order to successfully interact with the Eclipse Arrowhead framework local cloud core systems.

- Register provider.
- Register consumer.
- Register a service definition.
- Add intracloud authorization rules.
- Create an orchestration store entry.
- Recieve orchestration information based on consumer ID.

A sequence diagram visualizing the order of execution in the core systems.

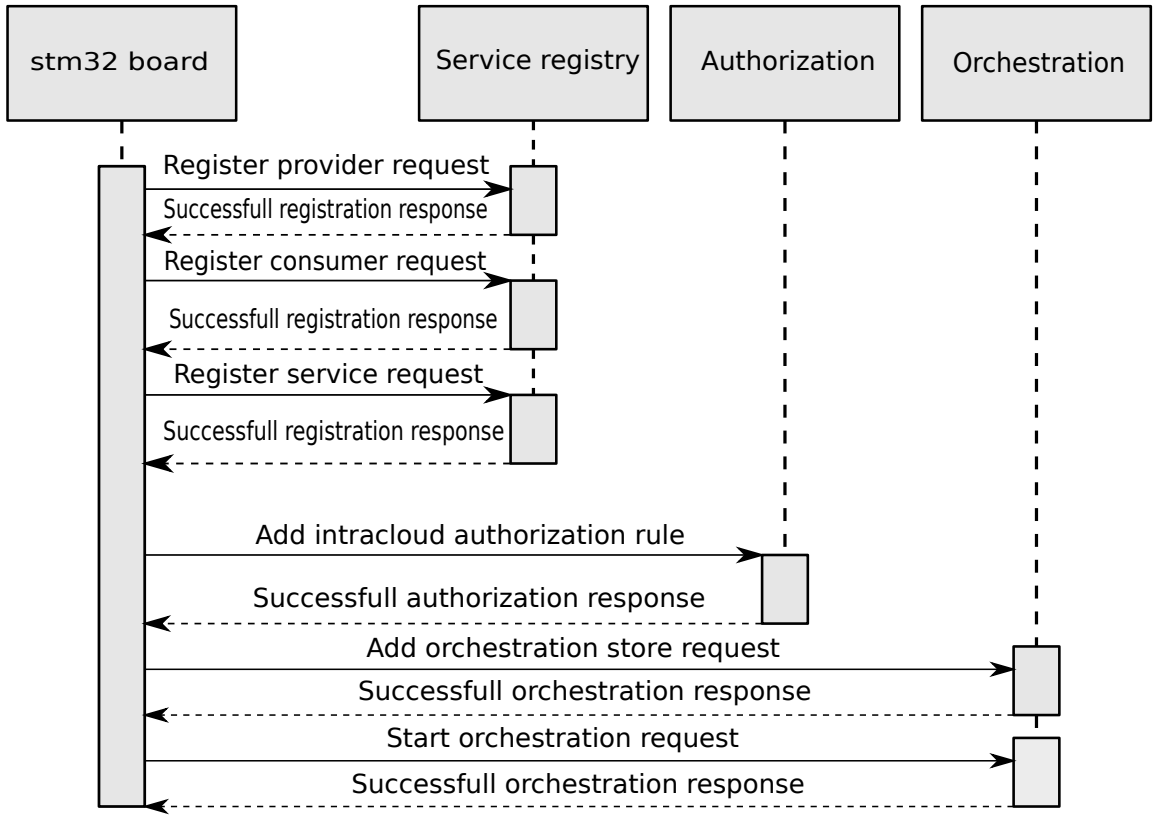


Figure 4.3: Sequence diagram of the process of using the Eclipse Arrowhead framework.

4.2 System components

Based on the architecture described in the previous section, it is clear that the software components have two main tasks

- Send and receive information using HTTP POST and GET methods.
- Constructing correct JSON strings to act as payload in the POST and GET methods.

4.2.1 HTTP post using the mbeb-HTTP library

The first function performs an HTTP post with a constructed JSON body. It does that with the help of the network interface object defined in the setup and sends that post to the appropriate URL.

```
1  std::string http_post_request_with_response(NetworkInterface* _net, std
   ::string URL, std::string body)
2  {
3      HttpRequest *post_request = new HttpRequest(_net, HTTP_POST, url.
   c_str());
4      post_request->set_header("Content-Type", "application/json");
5      HttpResponse *post_response = post_request->send(body.c_str(),
   strlen(body.c_str()));
6      if (!post_response) {
7          printf("HttpRequest failed (error code %d)\n", post_request->
   get_error());
8          return std::to_string(post_request->get_error());
9      }
10     printf("\n------ HTTP POST response ------\n");
11     std::string response_body = post_response->get_body_as_string();
12     delete post_response;
13     return response_body;
14 }
```

4.2.2 HTTP get using the mbeb-HTTP library

The second function is similar to the first one, with the main difference that it performs an HTTP get with a JSON payload. It also uses the network interface to send it to the appropriate URL. It does that with the help of the network interface object defined in the setup and sends that post to the appropriate URL.

```
1  std::string http_get_request_with_response(NetworkInterface* _net, std::
   string URL)
2  {
3      HttpRequest *get_request= new HttpRequest(_net, HTTP_GET, url.c_str());
4
5      HttpResponse *get_request_response = get_request->send();
6
7      if (!get_request_response) {
8          printf("HttpRequest failed (error code %d)\n", get_request->
   get_error());
9          return std::to_string(get_request->get_error());
10     }
11     printf("\n------ HTTP GET response ------\n");
12     std::string response_body = get_request_response->get_body_as_string();
13     delete get_request_response;
14     return response_body;
15 }
```

4.2.3 Constructing appropriate JSON strings

To use the POST and GET function defined in the previous section, a correct JSON payload, or HTTP body, has to be created. To register a system, consumer, or provider, a body similar to the one defined underneath should be used.

```
std::string register_system_body = "{\"address\": \"192.168.0.101\", \"
authenticationInfo\": \"\", \"port\": 1234, \"systemName\": \"
system_name\"}";
```

The next operation is to register a service, and just as when registering a system, a correct JSON payload is required. The previously defined provider system is passed as a parameter here an interface, has to be defined as well.

```
std::string register_service_body = "{\"serviceDefinition\": \"
service_definition\", \"providerSystem\": {\"systemName\": \"
system_name\", \"address\": \"192.168.0.101\", \"port\": 1234, \"
authenticationInfo\": \"\" }, \"interfaces\": [\"HTTP-INSECURE-JSON\"],
 \"serviceUri\": \"temperature\"}\r\n";
```

The board has to make the three above calls to the service registry core system.

To create intracloud rules, provider, consumer, and service definition ids must be passed as parameters. One implemented two helper functions to achieve this. The first one parses the response from registering a system, finds the substring containing the systems id, and returns that as a character pointer. The second one parses the response from registering a service, finds the substring containing the service definition id and returns that as a character pointer. The field interfaceIds can be looked up in the table system_interface in the Arrowhead database and should correlate with the selected interface.

```
+----+----------------------+---------------------+---------------------+
| id | interface_name       | created_at          | updated_at          |
+----+----------------------+---------------------+---------------------+
|  1 | HTTP-SECURE-JSON     | 2021-02-03 11:56:35 | 2021-02-03 11:56:35 |
|  2 | HTTP-INSECURE-JSON   | 2021-02-03 11:56:35 | 2021-02-03 11:56:35 |
|  3 | HTTPS-SECURE-JSON    | 2021-02-16 17:45:57 | 2021-02-16 17:45:57 |
+----+----------------------+---------------------+---------------------+
```

The correct JSON can now be constructed and posted to the authorization core system.

```
std::string add_intracloud_authorization_body = "{\"consumerId\": " +
std::to_string(consumer_id) + ",\"interfaceIds\": [3], \"providerIds\":
 [" + std::to_string(provider_id) + "], \"serviceDefinitionIds\": [" +
std::to_string(service_id) + "]}\r\n";
```

The request to create an orchestration store rule must contain information about the previously defined provider system and the consumer systems id. Information about the operating cloud and interface has to be defined, and posted to the orchestrator core system.

```
std::string create_orchestration_store_body = "[{ \"serviceDefinitionName
\": \"service_definition\", \"consumerSystemId\": " + std::to_string(
consumer_id) + ", \"providerSystem\": { \"systemName\": \"system_name
\", \"address\": \"192.168.0.101\", \"port\": 1234, \"
authenticationInfo\": \"\"}, \"cloud\": { \"operator\": \"aitia\", \"
name\": \"testcloud2\" }, \"serviceInterfaceName\": \"HTTP-INSECURE-
JSON\", \"priority\": 1}]\r\n";
```

A get request is sent to the orchestrator with the consumer's id as a parameter. The response from the orchestrator contains information about the address, port, and service URI of the provider the consumer wants to create a connection to. A helper function parsed the response from the orchestrator. The helper function takes the response as a parameter and returns the address, port, and service URI as a string. If every step is successful, the consumer can connect to the provider.

4.3 Error handling

The sequence diagram, 4.3, above shows that each subsequent operation's success depends on the success of the previous one. Without adequately defined and registered systems, the board can not register a service definition. If one command fails, The commands will still execute but return an appropriate error code and error message describing the error.

For instance, a provider system with the same name as a previous one in the database is trying to register itself to the service registry. The service registry will throw an error stating that such a system already exists and not return the system id, which will cause the command to add intracloud rules to fail since the id of the provider system is required. The orchestrator can not perform the orchestration process if the systems are not authorized correctly. In that case, the GET method to the orchestrator will return a blank response and make the connection to the provider impossible.

Chapter 5

Evaluation

The objective of this thesis was to investigate the possibilities, benefits, and limitations of using the Eclipse Arrowhead framework on embedded devices in contrast to commercially available solutions such as Amazon's Amazon Web Services and Microsoft Azure. The previous chapter showed that it was possible to connect the STM32 B-L4S5I-IOT01A to an Eclipse Arrowhead framework local cloud. Tests to figure out which IoT framework had the shortest response time carried out, and the results of those will be presented later on in this chapter.

5.1 Incorporating an Eclipse Arrowhead framework local cloud

We can deem the implementation successful if it manages to connect the system and services in agreement with the Eclipse Arrowhead framework. It has to send the temperature data from the STM32 B-L4S5I-IOT01A board to the flask app running on another device. The implementation should be easy to use, which due to the use of an online compiler and the ability to import the code, requiring minimal setup for the end-user, proved to be the case. The implementation presented in the previous section managed to do that successfully.

Based on the previous section, it seems possible to successfully use an Eclipse Arrowhead local cloud and an embedded device connected to the internet. The subsequent section will present the test results, showcasing the benefits of using the Eclipse Arrowhead framework. The next chapter will also cover the limitations of using the Eclipse Arrowhead framework in the next chapter.

5.2 Performance test result

As described in the theory section, the test consisted of pinging the different IoT framework 1000 times and calculated the average, minimum and maximum response times.

Line diagrams of the response time and average response time are presented below for the different IoT frameworks are shown below.

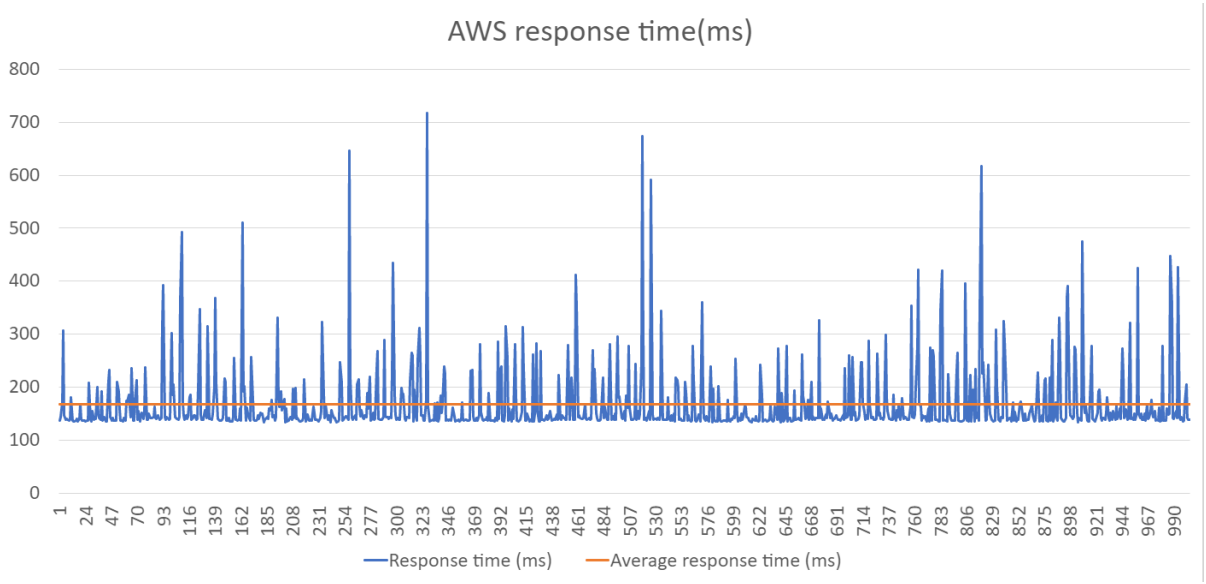


Figure 5.1: Response time and average of AWS

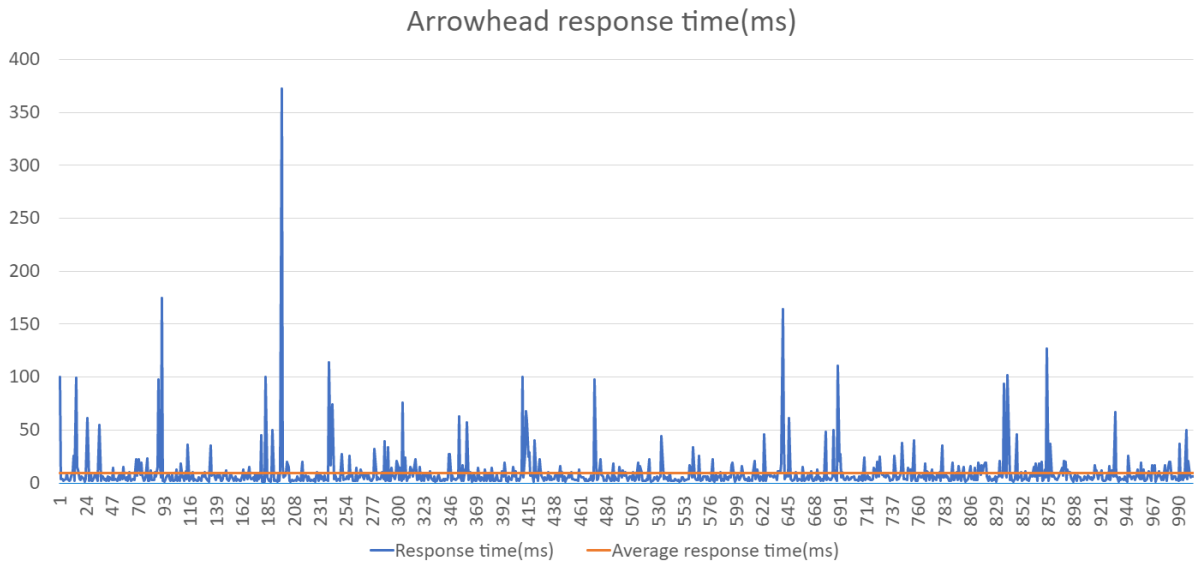


Figure 5.2: Response time and average of Arrowhead framework

To give a clearer picture of the results in the test te following table was constructed.

Parameters	AWS	Arrowhead	Azure
Average response time(ms)	166.94	9.5105	0
Max response time(ms)	717	372	0
Min response time(ms)	134	1	0

The table above shows that the Eclipse Arrowhead framework has the shortest average response time, 17.5 times faster than Amazon Web Services. The table also shows that Amazon Web Services has a max response time 1.9 times greater than the Eclipse Arrowhead framework. We can see the most considerable difference occurs in the minimum response time, where the Eclipse Arrowhead framework is 134 times faster than Amazon Web Services. The subsequent chapter will cover the advantages of having a quicker response time.

Chapter 6

Discussion

This thesis aimed to investigate whether it was possible to use an embedded device in conjunction with the Eclipse Arrowhead framework. If that was possible, this thesis also aimed to show the advantages and limitations of using the Eclipse Arrowhead framework on embedded devices. Another goal of this thesis was to show that using the Arrowhead framework on embedded devices provides a ready-to-make example for people to try out the framework. To fill a void in the Arrowhead project GitHub with a complete example ready to compile and using an online compiler.

6.1 Choice of development environment

The IDE, integrated development environment, chosen for this thesis stood between the Mbeb online compiler and the STM CUBE IDE. The STM cube IDE requires setup on the local machine, installing C and Cmake, but provides shorter compilation time. The Mbed online compiler requires no setup on the local machine but takes much longer to save and compile the code.

Both compilation time and an easy setup are favorable attributes for an IDE, and it all depends on the end goal. If the goal were to try out multiple different iterations of the same source code, then STM cube IDE would be the best choice. On the other hand, if the goal was to provide an example of a framework or concept and get the users started as quickly and painless as possible, the Mbed online compiler is the obvious choice. This thesis uses the latter because it fits the project's goals better.

Both IDEs provide ready to run example that features different aspects of the development boards, and both contain an example of connecting the board to the internet. The Mbed online compiler, in conjunction with arm Mbed OS, provides an easy and intuitive way to get started on desired projects. The Mbed online compiler has its drawbacks as well. The Mbed compiler pushes the users into using libraries integrated into the Mbed OS 6. It is possible to use outside libraries but more often than not, resulting in a rabbit hole of including header files. On the other hand, STM Cube IDE works similarly to a

local C-program. If the header files are in the include folder or installed locally, they can use them without much effort.

The STM Cude IDE also supports code generation from Cube MX, letting the user choose which peripherals to be included and adding initiation code before developing a project. Cube MX also supports importing supported libraries, generating the required include folders and header files for the user beforehand. When using the Mbed online compiler, all the peripherals, sensors, and libraries have to be initialized by the user. It will result in a trade-off between ease of use, performance, and customizability.

One disadvantage of using the STM cube IDE is that all applications will become platform-dependent. STM cube IDE requires the user to have a Windows computer. Mbed online compiler, on the other hand, builds everything in the browser. When the build finishes, the Mbed online compiler prompts the user to download a .bin file that can be dragged to the STM32 B-L4S5I-IOT01A board, as if storing a file on a flash drive. The STM32 B-L4S5I-IOT01A board compiles the .bin file and outputs the result to the user, no need to install Cmake or other toolchains.

6.2 Provider arcitechture

The implementation in this thesis follows the publish/subscribe model, as described in the related work section. The STM32 B-L4S5I-IOT01A board is the publisher, and the remote computer will act as the subscriber. The board has to find where to send the temperature data and sends it in predefined intervals after that. The implementation made in this thesis lacks the functionality of acting as a server, serving a request from a consumer.

In the Eclipse Arrowhead system, a provider system will typically be passive, reacting to requests. When speaking of it in an HTTP sense, the provider will act as a server. A consumer will usually act as a client and find the provider's address to request services from the provider. In this implementation, the provider will find the consumer's address and send the temperature information. The service it provides is sending the reading from the temperature sensor in specific time intervals. It is important not to get hanged upon the terminology as both a provider and a consumer are considered systems in the Eclipse Arrowhead framework. In essence, they are the same thing, and we can use the terms interchangeably; it all depends on how the service is defined. The Eclipse Arrowhead framework uses the term prosumer, a combination of the words provider and consumer, mitigating the confusion caused by these terms.

Depending on the application, it might be more efficient to send the data when the consumer demands it. It is possible to imagine the opposite as well. If a system relies on past information, it might be more efficient to send the data at a specified rate. If the system has to process the data and perhaps depends on a lot of data, it may be more efficient to use the subscribe/publish model. Negating the need for the embedded device to respond to requests can increase energy efficiency by having it boot at specific times

to send the data. If a system relies on real-time data creating a REST API to handle these requests would be the next logical step for developing this example. The future work section in the upcoming chapter covers implementing a REST API in more detail.

6.3 Comparing different frameworks

The previous chapters showed that it was possible to use the Eclipse Arrowhead framework with embedded devices. It also showed the advantages of using the framework. The main advantage was the response time, having on average 17.5 times faster response time than its competitors. A faster response time could be a great advantage when dealing with real-time applications when an immediate response is as essential as a correct one.

One disadvantage of using the Eclipse Arrowhead could be the lack of supported hardware. This thesis was the first example to show that it was possible to connect to the arrowhead framework on embedded devices. In contrast, Amazon Web Services have many examples of using different hardware. Amazon Web Services also has examples showcasing the different sensors and connection possibilities of their supported boards. Creating an open-source example of using the Eclipse Arrowhead framework could inspire others to start developing embedded Arrowhead applications, expanding the functionality of this implementation.

The eclipse Arrowhead Framework requires hardware to run on, a service the customer has to pay for when using Amazon Web Services. It is probably cheaper to purchase hardware and run the Eclipse Arrowhead on a larger scale, but that requires an upfront cost. In smaller-scale applications or when trying out a proof of concept, the prices offered by Amazon Web Services are hard to beat. When using Amazon Web Services, the user relies on that their hardware and network runs continuously, an issue which magnitude showed its face this fall.

One significant advantage of using Amazon Web Services is its services offered out of the box. Lambda functions, S3 buckets, and all the functionality that follows with them is something the user will have to implement on their own if choosing the Eclipse Arrowhead framework. On the other hand, if the user wants to do something not supported by Amazon Web Services, they will be constrained. As with the choice of IDE, it will result in a trade-off between ease of use and customizability.

Another significant advantage of Amazon Web Services is the support for and enforcement of using HTTPS. When deploying a 'Thing' to the AWS IoT Core, it generates a certificate, and the user has to include it in their source code. The security issues raised in the related work section illustrate the need for HTTPS and other security measures for IoT devices in production. When developing a proof of concept, as this thesis has been doing, the requirement of HTTPS might be overstated. It becomes a trade-off between usability, development time, and security. If the intent is to demonstrate that the Eclipse Arrowhead framework works on embedded devices, HTTPS and all the extra steps for implementing it might do more harm than good.

The Eclipse Arrowhead framework supports HTTPS and encourages its developers to use it. There are guides for generating Arrowhead compliant certificates and support for it in the Java implementation of a client. However, it proved to be challenging to implement and beyond the scope of this thesis. Implementing HTTPS will be covered more in the future work section in the next chapter.

Chapter 7

Conclusions and future work

This chapter summarizes the thesis by outlining its accomplishments and remaining work left to do. Section 7.1 contains the conclusion, and in section 7.2. the reader can find a description of work to be done in the future.

7.1 Conclusion

This thesis examined the possibility of using an Eclipse Arrowhead framework local cloud on an embedded device, namely the STM32 B-L4S5I-IOT01A board. It was possible to implement functionality for the Eclipse Arrowhead framework on the STM32 B-L4S5I-IOT01A board, having the board send temperature readings to a computer within a local network.

Another goal of this thesis was to provide an easy-to-run example of the Eclipse Arrowhead framework that anyone could compile by anyone wanting to give the framework a try, a missing feature right now for embedded devices. The implementation was done with usability in mind, making it easy for users to try the code. The desired usability was achieved with the Mbed online compiler's code importation and minimum setup development environment, allowing users to try out the Eclipse Arrowhead framework on the STM32 B-L4S5I-IOT01A board.

The thesis also examined the benefits of using the Eclipse Arrowhead framework compared to its competitors Amazon Web Services and Microsoft Azure. The thesis showed some benefits in terms of response time when running a local cloud instead of using a remote service such as Amazon Web Services, a 17.5 decrease in average response time was recorded. Maximum and minimum response times decreased by 1.9 and 134 times, respectively.

7.2 Future work

The research presented in this thesis can take many different directions moving forward. Two main issues were raised during this thesis, security and having the STM32 B-L4S5I-IOT01A board act as a server. The following sections will address these two issues.

7.2.1 Security

We have to address the security issues raised in the previous chapters if the implementation done in this thesis ever is to be used by the industry in production. This thesis made several attempts at implementing HTTPS; the Mbed-HTTP library has support for HTTPS, and STM cube IDE has support for wolfSSL.

The problem to be solved before implementing HTTPS is how the Eclipse Arrowhead framework handles certificates in languages other than Java. Both Mbed and STM cube has examples using HTTPS that works, and getting started with Amazon Web Services uses HTTPS with a user-generated certificate. One of the main issues is that the certificates are self-signed, meaning no trusted certificate authority has signed them. The self-signed certificates proved to be the main obstacle for implementing HTTPS using C. None of the libraries, Mbed-HTTP or wolfSSL, could trust the certificate from the Eclipse Arrowhead framework. There is a need to conduct further research using the certificates generated by the Eclipse Arrowhead framework on embedded devices. One area of research could be to move away from the .pk12 format, generally used by java applications, and include more support for the .pem format used by C and many other languages. Another area of research needed is lightweight cryptography and possible ways to move away from the idea that an IoT device has its certificate. A concept that quickly becomes unbearable when dealing with thousands of devices in one network.

7.2.2 Server implementation

Before being appliable to the industry, one would have to solve the matter of the board's inability to react to requests. The issue of reacting to requests is of utmost importance. Both the Mbed online compiler and the STM Cube IDE have a working example of an HTTP server that can request the temperature data from a generated webpage. Those examples use pure HTTP requests and responses, leading to very lengthy and challenging messages to parse. Future research that promotes the same usability as Mbed-HTTP and responding to the request would greatly benefit the Eclipse Arrowhead framework.

A server implementation on the STM32 B-L4S5I-IOT01A board could also have great educational potential by using it in courses for young adults or aspiring engineers. With the number of IoT devices connecting to the internet only increasing, understanding connected embedded devices is crucial for future engineers. Introducing concepts like IoT and embedded system programming early in an engineering degree and real-life examples could enhance knowledge and spike interest for those subjects, making aspiring engineers ready for the future.

References

[1] Artemis-IA News Embedded Intelligence: Trends & Challenges book release. https://artemis-ia.eu/news/embedded-intelligence-tsrends-challenges-book-releas html, accessed 2021-04-21.

[2] Heiner Lasi, Peter Fettke, Hans Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business and Information Systems Engineering*, 6:239–242, 8 2014.

[3] Arrowhead tools. https://arrowhead.eu/arrowheadtools/, accessed 2021-04-22.

[4] Arrowhead consortia. https://github.com/arrowhead-f, accessed 2021-03-10.

[5] User Guide and Freertos Qualification. User manual Getting started with X-CUBE-AWS STM32Cube Expansion Package for Amazon Web Services ® IoT. (September):1–31, 2020.

[6] Getting started with the STMicroelectronics B-L475E-IOT01A / B-L4S5I-IOTOA1 Discovery https://docs.microsoft.com/en-us/samples/azure-rtos/getting-started/getting-started-with-the-stmicroelectronics-b-l475e-iot01a--b-l4s5i-iotoa1-dis accessed 2021-04-22.

[7] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29:1645–1660, 9 2013.

[8] Saurabh Vaidya, Prashant Ambad, and Santosh Bhosle. Industry 4.0 - a glimpse. volume 20, pages 233–238. Elsevier B.V., 1 2018.

[9] Elvis Hozdić. Smart factory for industry 4.0: A review, 2015.

[10] Li Da Xu, Eric L. Xu, and Ling Li. Industry 4.0: State of the art and future trends. *International Journal of Production Research*, 56:2941–2962, 2018.

[11] Delsing Jerker. Iot automation: Arrowhead framework - 1st edition - jerker delsing -. 2017.

[12] How AWS IoT works - AWS IoT Core. https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html,accessedon2021-04-24.

[13] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. Iot: Internet of threats? a survey of practical security vulnerabilities in real iot devices. *IEEE Internet of Things Journal*, 6:8182–8201, 10 2019.

[14] Zhi Kai Zhang, Michael Cheng Yi Cho, Chia Wei Wang, Chia Wei Hsu, Chong Kuan Chen, and Shiuhpyng Shieh. Iot security: Ongoing challenges and research opportunities. pages 230–234. Institute of Electrical and Electronics Engineers Inc., 12 2014.

[15] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. A survey on iot security: Application areas, security threats, and solution architectures, 2019.

[16] Mardiana binti Mohamad Noor and Wan Haslina Hassan. Current research on internet of things (iot) security: A survey. *Computer Networks*, 148:283–294, 1 2019.

[17] Bharati Wukkadada, Kirti Wankhede, Ramith Nambiar, and Amala Nair. Comparison with http and mqtt in internet of things (iot). pages 249–253. Institute of Electrical and Electronics Engineers Inc., 12 2018.

[18] Getting Started with MQTT. `https://www.hivemq.com/blog/how-to-get-started-with-mqtt`, accessed 2021-03-18.

[19] An overview of HTTP. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview`, accessed 2021-03-10.

[20] Nitin Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. Institute of Electrical and Electronics Engineers Inc., 10 2017.

[21] N Shariatzadeh, T Lundholm, L Lindberg, G Sivard Procedia Cirp, and undefined 2016. Integration of digital factory with smart factory based on internet of things. *Elsevier*.

[22] eclipse-arrowhead/core-java-spring. `https://github.com/eclipse-arrowhead/core-java-spring`, accessed 2021-03-10.

[23] Representational state transfer - Wikipedia. `https://en.wikipedia.org/wiki/Representational_state_transfer` accessed on 2021-04-29.