

# Performance Evaluation of a Middleware Framework for CPS/IoT Ecosystem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Can Deliorman, B.Sc.**

Matrikelnummer 0927357


an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Radu Grosu

Mitwirkung: Dipl.-Ing. Haris Isakovic

Wien, 27. August 2020

  
\_\_\_\_\_  
Can Deliorman  
\_\_\_\_\_  
Radu Grosu



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Performance Evaluation of a Middleware Framework for CPS/IoT Ecosystem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**Can Deliorman, B.Sc.**

Registration Number 0927357


to the Faculty of Informatics

at the TU Wien


Advisor: Prof. Radu Grosu

Assistance: Dipl.-Ing. Haris Isakovic

Vienna, 27<sup>th</sup> August, 2020



Can Deliorman



Radu Grosu



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Erklärung zur Verfassung der Arbeit

Can Deliorman, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. August 2020



---

Can Deliorman



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

Mit der zunehmenden Anzahl von Geräten, die über das Internet verbunden sind, wird das Internet der Dinge (IoT) Realität. Diese Geräte müssen miteinander ohne menschliche Interaktion kommunizieren können. Interoperabilität, Verfügbarkeit, Zuverlässigkeit, Mobilität, Leistung, Management, Skalierbarkeit und Sicherheit sind die Hauptherausforderungen in einem IoT-Ökosystem. Als Middleware für Cyber-Physical Spaces / Internet der Dinge wurde das Arrowhead Framework entwickelt, um diese Herausforderungen zu bewältigen. Wenn die Anzahl der über das Framework verbundenen Geräte zunimmt, können Leistungsprobleme wie Fehler und / oder Abstürze auftreten. Um die Zuverlässigkeit, Verfügbarkeit, Leistung und Skalierbarkeit des Arrowhead Frameworks zu bewerten, haben wir Last- und Stresstests am Framework durchgeführt, um die Grenzen verschiedener Komponenten zu untersuchen. Da das Internet der Dinge verschiedene Hardwarekomponenten abdecken wird, haben wir unsere Tests an zwei verschiedenen Hardwarekonfigurationen wiederholt. Unser Ziel ist es, potenzielle Engpässe, Fehler und optimale Konfigurationen in verschiedenen Anwendungsfällen zu erkennen. Das Arrowhead Framework wurde in der Programmiersprache Java entwickelt und läuft auf der Java Virtual Machine (JVM). Es gibt mehrere JVMs auf dem Markt, die weniger Ressourcenverbrauch und bessere Startzeiten versprechen. Die AOT-Kompilierung (Ahead-of-Time) wird zunehmend zusammen mit den JIT-Kompilierungsmethoden (Just-in-Time) für die JVMs verwendet. OpenJ9 hat die AOT-Kompilierung in die JIT-Kompilierung und den Cache für gemeinsam genutzte Klassen integriert, um die Startzeiten zu verbessern und den Ressourcenverbrauch zu senken. Nachdem wir unsere Tests auf der HotSpot JVM ausgeführt hatten, wiederholten wir unsere Leistungstests auf OpenJ9 mit der Standardkonfiguration sowie auf OpenJ9 mit dem Cache für gemeinsam genutzte Klassen. Wir haben festgestellt, dass die Grenzwerte für gleichzeitige Requests an eine Komponente nur zehn Requests pro Sekunde betragen können. Diese Anzahl hängt jedoch stark von der Hardwarekonfiguration ab. Für Anwendungsfälle, in denen hoher Datenverkehr erwartet wird, wird eine erweiterte Hardwarekonfiguration empfohlen. Wir haben auch festgestellt, dass OpenJ9 mit Cache für gemeinsam genutzte Klassen zwar den Speicherverbrauch und die Startzeiten reduzierte, jedoch zu höheren Antwortzeiten führte als die HotSpot JVM. Dennoch kann es eine interessante Option für Geräte mit begrenzten Ressourcen und geringem Datenverkehr sein. Unsere Tests helfen Entwicklern und Benutzern des Arrowhead Framework bei der Entscheidung für die richtige Hardware- und Softwarelösung bei der Verwendung des Frameworks.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.



# Abstract

The Internet of Things (IoT) is becoming a reality, as the number of devices connected to the Internet increases rapidly. These devices need to communicate with each other without human interaction. Interoperability, availability, reliability, mobility, performance, management, scalability, and security are the main challenges in an IoT ecosystem. As a middleware for Cyber-Physical Spaces/Internet of Things, the Arrowhead Framework has been developed to tackle these challenges. As the number of devices connected through the framework grows higher, the framework may experience performance problems, such as errors and/or crashes. To evaluate the reliability, availability, performance, and scalability of the Arrowhead Framework, we have conducted load and stress tests on the framework to explore the limits of different components. Since the Internet of Things will cover various hardware, we have repeated our tests on two different hardware configurations. Our goal is to detect potential bottlenecks, errors, and optimal configurations in different use cases. The Arrowhead Framework is developed in Java programming language and runs on the Java Virtual Machine (JVM). There are multiple JVMs available on the market that promise less resource consumption and better startup times. Ahead-of-Time (AOT) compilation is also increasingly being used along with Just-in-Time (JIT) compilation methods for the JVMs. OpenJ9 has integrated the AOT compilation with the JIT compilation and shared classes cache to improve the startup times and reduce resource consumption. After executing our tests on the HotSpot JVM, we repeated our performance tests on OpenJ9 with the default configuration, as well as on OpenJ9 with shared classes cache. We have found that the limits for simultaneous requests on a component may be as low as ten requests per second. However, this number is highly dependent on the hardware configuration. For use cases where high traffic is expected, an enhanced hardware configuration is recommended. We have also discovered that even though OpenJ9 with shared classes cache reduced the memory consumption and startup times, it led to higher response times than the HotSpot JVM. Nevertheless, it can be an interesting option for devices with limited resources and low traffic. Our tests help developers and users of the Arrowhead Framework with deciding on the correct hardware and software solution when using the framework.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

<b>Kurzfassung</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>Contents</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Work</b>	<b>7</b>
<b>3 Background</b>	<b>11</b>
3.1 The Arrowhead Framework . . . . .	11
3.2 Java Virtual Machine . . . . .	16
3.3 Testing Tools . . . . .	18
<b>4 Methodologies</b>	<b>19</b>
4.1 Service Registry . . . . .	20
4.2 Authorization . . . . .	21
4.3 Orchestrator . . . . .	21
4.4 Gatekeeper . . . . .	22
4.5 Security . . . . .	22
4.6 Java Virtual Machine . . . . .	22
4.7 Tests on different hardware . . . . .	23
<b>5 Implementation</b>	<b>25</b>
5.1 Used Software and Hardware . . . . .	25
5.2 Testing Environment Configurations . . . . .	26
5.3 Generating Test Data . . . . .	26
5.4 Service Registry . . . . .	27
5.5 Authorization . . . . .	30
5.6 Orchestrator . . . . .	32
5.7 Gatling Configuration . . . . .	36
5.8 Limitations . . . . .	37
	xi

<b>6</b>	<b>Results and Discussion</b>	<b>39</b>
6.1	Service Registry . . . . .	39
6.2	Authorization . . . . .	41
6.3	Orchestrator . . . . .	44
6.4	Secure mode vs. Insecure mode . . . . .	54
6.5	Startup Times . . . . .	54
6.6	Found Errors . . . . .	55
6.7	Discussion . . . . .	56
<b>7</b>	<b>Future Work</b>	<b>57</b>
<b>8</b>	<b>Conclusion</b>	<b>59</b>
	<b>List of Figures</b>	<b>60</b>
	<b>Bibliography</b>	<b>63</b>

# Introduction

Until a few years ago, most of the communication over the Internet was established between devices that were directly used by humans. This has started to change with the introduction of "smart devices". Machines can now exchange information by themselves, and the number of devices connected to the Internet is increasing rapidly. The number of devices connected to the Internet of Things (IoT) will reach 75.44 billion worldwide by 2025, which is a five-fold increase in ten years [19]. Most of the information exchange on the Internet will be between devices instead of humans. IoT can be defined as: "a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual 'Things' have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network" [53].

By integrating different devices, sensors, and communication technologies, as is the foundation of IoT, we can allow them to cooperate and communicate to fulfill common goals [56]. We can already see the initial impacts of IoT in the consumer electronics market. Approximately 225 million smart meters for electricity and 51 million for gas will have been installed within the European Union by 2024, which is when almost 77% of European households will be equipped with a smart meter for electricity [52]. Currently, consumers can already purchase a variety of smart products from refrigerators to robot vacuum cleaners. Along with the concept of Industry 4.0, IoT also becomes more important in the Industry. Industry 4.0 focuses on the digitization of the physical world and the emergence of new computing concepts such as Wireless Sensor Networks (WSN), Cyber-Physical Systems (CPS), or the Internet of Things (IoT) [45].

The interest in IoT technologies is on the rise in a wide range of industries. Several industrial IoT projects have been carried out in fields such as surveillance, agriculture, environmental monitoring, food processing, security, and so on [56]. However, connecting so many devices brought on to new challenges in their management since the majority

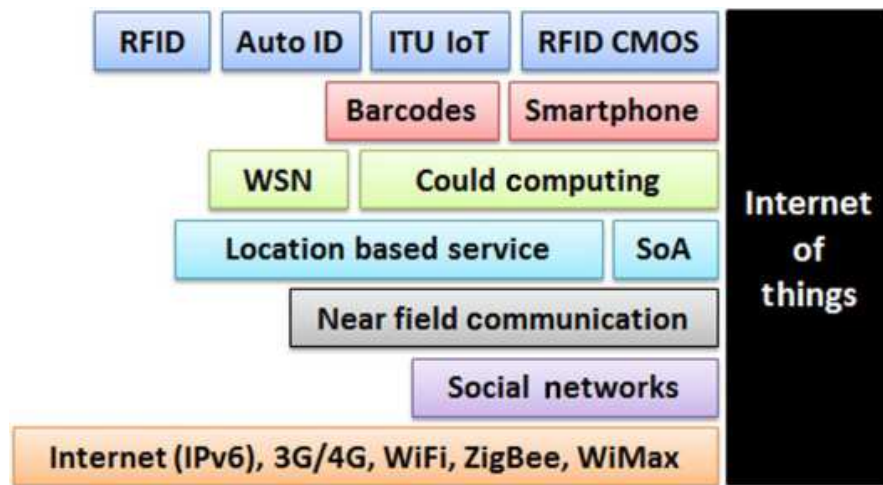


Figure 1.1: Some of the technologies associated with IoT. (copied from [56])

of these devices must communicate and be synchronized with each other. The most significant challenges are as follows:

- **Interoperability:** Interoperability can be defined as the capability of multiple systems or components for information exchange and the use of the exchanged information. Especially in the context of the Industrial Internet of Things (IIoT), the connection between sensors and actuators with the local process and the Internet must be enabled. The connectivity to other industrial networks that independently can generate added value must be provided. Moreover, we should ensure the quality of service in terms of performance and secure data transmission [45].
- **Availability:** In the software context, availability indicates that the IoT applications must be able to provide services simultaneously for everyone at different places [39].
- **Reliability:** Reliability can be described as the proper working of the system based on its specification [47]. Reliability has a close relationship with availability because it is through reliability, that the availability of information and services is guaranteed in the long term [39].
- **Mobility:** Mobile devices are a prominent portion of the Internet of Things. Especially interruptions in the connectivity of mobile devices should be taken into consideration [39].
- **Performance:** The performance of IoT services is one of the greatest challenges because it also depends on the performance of the underlying protocols and technologies, as well [39].

- 
- **Management:** The connection of billions or even trillions of devices to the Internet, makes the management of these devices a substantial challenge [39].
  - **Scalability:** By scalability we refer to the ability to add new services, devices, and functions without negatively impacting the quality of existing services. In IoT, different hardware platforms and communication protocols will be present, which makes the scalability a challenging task [39].
  - **Security and Privacy:** Heterogeneous networks and the lack of universal standard and architecture for the IoT security makes it difficult to guarantee the security and privacy of users [39].

To overcome these challenges, numerous commercial and non-commercial software frameworks have been developed. We will go into detail for some of the most common ones:

- **FIWARE:** FIWARE is an open, cloud-based infrastructure for Cyber-Physical Systems (CPS) [13]. It defines a set of standards for context data management. This allows developing smart solutions for different domains such as Smart Cities, Smart Industry, Smart Agrifood, and Smart Energy. The FIWARE Context Broker component is the core component that makes updates possible and gives access to the current state of context. The FIWARE Catalogue contains open-source platform components, which are called Generic Enablers. They can be assembled in order to create Smart Solutions [43].
- **SiteWhere:** SiteWhere is an industrial-strength, open-source IoT Application Enablement Platform. The platform works with a distributed, microservices architecture that runs on Kubernetes and is delivered with databases and MQTT brokers [29].
- **Samsung SmartThings:** SmartThings is an open smart home platform and is not supported for industrial use cases. Through the SmartThings cloud IoT devices and services can be built and integrated. Since it is primarily meant for Smart Home applications, it cannot be directly compared with most other IoT platforms [30].
- **Amazon Web Services IoT:** Amazon Web Services IoT (AWS IoT) is an IoT Platform for industries and end users, who want to build a smart home. It brings data management and analytics together. To improve the intelligence of devices, it also integrates Artificial Intelligence technologies of AWS with the IoT Platform [7].
- **Microsoft Azure IoT Hub:** Azure IoT Hub is a cloud-hosted solution backend to connect any device. It is responsible for establishing bi-directional communication between devices. It can also be integrated with other Azure products like Azure Maps, Azure IoT Edge [20].

- **IBM Watson IoT Platform:** IBM Watson IoT Platform is yet another cloud-based solution. Similar to the solutions offered by AWS and Microsoft, IBM Watson IoT Platform can also be integrated with other services of IBM Cloud like analytics and AI-based services [37].
- **Siemens MindSphere:** Siemens Mindsphere is designed for industrial applications. Siemens defines MindSphere as an "open, cloud-based IoT operating system". It has also integrated services for analytics and can run on Cloud Services like AWS and can be integrated into the services of this cloud provider [28].

There is a great number of other IoT platforms. Even though many developers use different vocabulary to describe their software frameworks, most of these frameworks are developed for similar use cases. There are countless IoT frameworks available on the market, however none of those platforms has led to successful and efficient digitization of all parts of the manufacturing industry as of yet. [45]. The Arrowhead Framework has been developed to tackle this problem and enable collaborative automation and industrial IoT. It aims to facilitate the creation of local automation clouds. That is to say, it provides "local real-time performance, and security, paired with simple and cheap engineering, while simultaneously enabling scalability through multi-cloud interaction" [41]. The objective is to enable interoperability between IoT components and build a System of Systems from individual IoT components. For achieving interoperability, the Arrowhead Framework abstracts IoT components and their functionalities into services [6]. The Service-Oriented Architecture (SOA) paradigm is a key component of the Arrowhead Framework. In other words, loose coupling, late binding, and lookup as central aspects of the SOA are also essential points in the Arrowhead Framework [41]. In order to ensure a high quality for an IoT Framework like the Arrowhead Framework, the following tests must be conducted in addition to functional tests: [49]:

- **Performance Testing:** Performance tests will assess the communication speed between the components and the computational power of the underlying software system and allow us to draw conclusions about the capabilities of the software framework.
- **Security Testing:** The privacy of users and autonomy need to be tested.
- **Compatibility Testing:** It is required to ascertain if the framework functions as expected with different devices, protocols, and software versions.
- **Exploratory Testing:** Tests from a user's perspective should be conducted without resorting to predefined testing scenarios.

An increasing number of devices that are connected through the framework may lead to performance issues in the system, which may then result in high costs or even the failure of bigger production sites. In this thesis, we have thoroughly tested the Arrowhead



---

Framework, from a performance perspective, to determine the potential bottlenecks, possible issues, and optimal configurations in a real-world application. An IoT Framework should be able to handle requests from different devices in an acceptable unit of time. To find out the limits of the Arrowhead Framework, we have conducted load and stress tests, where we sent multiple requests to different components of the Arrowhead Framework simultaneously. To execute those tests, we used a generic test automation tool specializing in performance testing through REST APIs. The Arrowhead Framework should also yield acceptable performance on devices with limited resources. We have repeated the same tests on different devices to determine how the Arrowhead Framework performs under heavy load with limited resources. These tests aim to tackle the challenges that we have defined earlier, especially the challenges of availability, reliability, performance, and scalability. A high number of simultaneous requests may render our framework unstable and even lead to crashes, which would make our framework unreliable. Through our stress tests, we can also determine the limits of the scalability of our framework.

A Java Virtual Machine (JVM) is a virtual machine that enables a computer to run Java programs and programs written in other languages that are also compiled to Java bytecode. A JVM is the core component that makes Java and other programming languages hardware and operating system independent. A JVM is like a real computing machine with its own instruction set and memory manipulations at runtime [46]. HotSpot is the standard JVM, that is released by Oracle and is used by most applications [22]. On the other hand, the Eclipse Foundation develops an alternative JVM, that is called OpenJ9 [25]. Eclipse Foundation argues that OpenJ9 provides a better performance in many aspects like startup time and RAM consumption [26]. Since the Arrowhead Framework can also run on devices with limited resources, CPU and RAM consumption may play a significant role in the framework's performance. The Arrowhead Framework is developed in Java programming language, therefore we repeated the performance tests on OpenJ9 to see whether using OpenJ9 as the JVM instead of HotSpot brings along a performance boost and/or a decrease in resource consumption in the context of the Arrowhead Framework.

This thesis has the following structure: In Chapter 2, we discuss the related work in the field of IoT and the testing of IoT Frameworks. Chapter 3 contains the background for our research, including information about the Arrowhead Framework and the Java Virtual Machine. Chapter 4 describes the methodologies employed in our testing environment. Chapter 5 presents the implementation of our testing environment, including the test scenarios. In Chapter 6, the results of our tests are presented and discussed in detail. Finally, in Chapter 7 and Chapter 8 we close by presenting our plans for future work and with the conclusions.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# CHAPTER 2

## Related Work

The testing of IoT-related technologies has been a topic of research in the recent years. Many aspects should be taken into account when testing IoT-related technologies. The purpose of the research is to create a testing environment to accelerate the development of IoT-related technologies by exploring the potential errors as early as possible and overcome the difficulties that arise from the heterogeneous nature of the IoT domain. In France, a testbed was constructed with 2728 devices. They installed over 2700 wireless sensor nodes, including 117 mobile robot nodes across six sites in France. The testbed is deployed in six sites in France, but they are interconnected and available through the same web portal. Developers can test their technologies with those devices on the platform, which provides them with a testing environment close to reality [38]. Most of the applications on an IoT Platform run on containers like Docker. The Arrowhead Framework also supports containerization. Even though containers have the advantage that they can be easily started, stopped, and moved, they may also impact the performance of the services, which was evaluated in a study by Morabito, Farris, Iera, and Taleb [48]. Despite the fact that Docker containers deteriorated the performance of the applications, the deterioration was negligible. Researchers at the São Paulo University have conducted simple performance tests on their IoT Platform, where they sent multiple requests simultaneously to the platform. They have also executed the same tests on different hardware platforms to observe the performance effect of different hardware [42]. Researchers from Ericsson Research have evaluated the performance of their IoT Framework, which they developed with the Swedish Institute of Computer Science (SICS) and Uppsala University. For the evaluation, they have sent 100,000 HTTP POST requests by 10, 100, 200, 500, 800, 1000 users and have recorded the throughput, CPU, and memory consumption with different hardware configurations. They have concluded, that the IoT Framework stays stable under heavy load but that a large number of consumers leads to a significant drop in throughput and high memory consumption, which may raise concern in a real-world application [54]. In a Master's thesis at the Luleå University of Technology, FiWare was

tested from a performance perspective. First, they benchmarked the baseline performance of the framework by sending up to 1500 requests to the IoT agents, then they have tested the vertical and horizontal scalability of the framework. They found out that FiWare's IoT agents become unstable and crash when the request/sec. reaches 1000. They have also benchmarked the resource consumption and noted that FIWARE's IoT Agents are written in Node.js. Node.js uses a single-threaded model and cannot take advantage of the multicore capabilities of modern CPUs. Even though vertical and horizontal scaling has increased the performance of the framework, it could not prevent the crashing of FiWare IoT Agents. They also noted that the horizontal scaling proved to be more effective than scaling vertically. They additionally suggested that the IoT agents of FiWare need to be improved for future large-scale smart city scenarios [51].

Another issue when testing IoT Frameworks is the question of the comparability of results. Many researchers carry out the performance tests by applying their own scenarios with their own metrics. This makes it difficult to compare the results from a study with a framework with the results from another study. Some qualitative and quantitative metrics were defined in a study by Cruz, Rodrigues, Sangaiah, Al-Muhtadi, Korotaev [40]. They have proposed the following qualitative and quantitative metrics for comparison and performance evaluation of IoT Frameworks:

Qualitative metrics:

- **Per device authentication:** Every device should have its own credentials.
- **Different credentials to publish and consult data from the middleware:** This would allow organizations to publish device interfaces without worrying about data manipulations.
- **Devices should access other device data using their own credentials:** This allows the administrators to determine the compromised device in case of a security breach.
- **Middleware should know the device properties such as IP and MAC address:** This helps the administrators to determine the compromised device in case of a security breach.
- **Number of Standard Development Kits (SDKs):** This makes it easier for developers to use the framework and deploy code into devices.
- **Supported application protocols:** This is important for the interoperability of different devices.
- **Number of updates:** The framework should be provided with regular updates.
- **Popularity**
- **Support tiers:** Developers should offer sufficient support for the framework.

- 
- **Mobile app:** For some scenarios, mobile apps are beneficiary or even mandatory.

Quantitative metrics:

- **Packet size:** Packet size in communication is a key factor for performance and energy consumption.
- **Error percentage:** An IoT framework should be able to deal with high load without errors.
- **Response times:** The response time of a framework is an important metric, especially in high load scenarios.
- **Price:** Price is always an important decision factor for stakeholders.
- **Timeliness performance:** Performance should not be affected by the passage of time.
- **DoS and DDoS prevention:** For users' privacy and high security, it is an important aspect.

Then they made evaluations of some IoT Frameworks using these metrics to compare them. They have also conducted load tests to determine the error percentages and response times of the following IoT Frameworks: FIWARE, InatelPlat, SiteWhere, Linksmart, and Konker.

There is no academic research, that compares the performances of HotSpot and OpenJ9. Nevertheless, the Eclipse Foundation has published multiple performance tests on its blog. In one test, they compared the startup time and memory footprint of Jenkins running on HotSpot and OpenJ9 [34]. In another test, they revealed that the Java framework Quarkus retains its performance advantages while also running on an OpenJ9 JVM against the Spring Boot Framework by comparing the memory footprint and startup time [36]. The Eclipse Foundation claims that through the Shared Classes feature, we can reduce the startup time and memory footprint of applications that run on JVM [11]. To prove this, they compared the startup time of the Apache Tomcat Server [5], which runs on an OpenJ9 JVM with default settings with a Tomcat Server that runs on an OpenJ9 JVM with enabled shared classes. Both servers were running in Docker images. They concluded that the startup time was reduced by about 30 % with enabled shared classes.

There is still a great need for research in the IoT domain. There is no academic research on the Arrowhead Framework, that concentrates on the performance of the framework. This master's thesis investigates if the Arrowhead Framework is eligible for a real-world application, especially under extraordinary conditions. Even though OpenJ9 has also been on the market for a few years, there has been no academic research on the performance of OpenJ9. In this master's thesis, we evaluate the differences between HotSpot and

## 2. RELATED WORK

---

OpenJ9 from a performance perspective. We study the potential performance issues of an IoT Framework so as to see if an alternative JVM improves the performance.

# Background

## 3.1 The Arrowhead Framework

The information in this chapter is cited from [41] and [35] and is only valid for Arrowhead Framework 4.1.3. Other versions may come with other features and components.

The Arrowhead Framework aims to enable collaborative automation and industrial IoT with the creation of local automation clouds. The Local automation cloud concept was introduced by the Arrowhead project, and it foresees that specific geographically local automation tasks should be encapsulated and protected. A local cloud has all the systems that are necessary to execute the automation tasks without interference from outside activities e.g. from the open Internet. A local cloud should possess the following properties:

- **Self-contained:** The administration, orchestration, authentication, and authorization of services are located in the local cloud.
- **Automation support:** A local cloud enables information exchange based on events, information exchange audit, and supports the automation system design.
- **Security:** A local cloud should be secure against interference from external networks and support secure bootstrapping and software updates.
- **Metadata:** A local cloud should support devices, systems, and service metadata.
- **Transparency:** A local cloud should support protocol, and semantics transparency.
- **Administration and data exchange:** Secure administration and data exchange with external networks should be supported.

The Arrowhead Framework combines the local cloud concept with the Service-Oriented Architecture (SOA) paradigm, which enables the services to be exchanged between systems. In order to achieve all these goals of the local cloud concept, the Arrowhead Framework defines three types of services:

- **Mandatory core services:** The mandatory core systems and their services provide the fundamental services, that facilitate the exchange of services between a service provider and a service consumer, with the corresponding level of security and autonomy. Following mandatory core services are defined:
  - ServiceRegistry
  - Authorization
  - Orchestration
- **Automation support core services:** These services aim to facilitate the automation application design, engineering, and operation. They implement the housekeeping within the local cloud, security, inter-cloud service exchange and interoperability of systems and services. Following automation support core systems were implemented by the Arrowhead Framework 4.1.3:
  - EventHandler
  - Gatekeeper
  - Gateway
- **Application services:** The application services within a local cloud implement application functionalities. They may also consume and/or produce other services from the local or external clouds.

We will go into details for some of these services.

#### 3.1.1 Service Registry

ServiceRegistry system provides storage of all registered services within a local cloud and enables the discovery of them. The ServiceRegistry is an independent system that does not consume any other services. We can differentiate between three types of endpoints that are offered by the Service Registry:

- **Client endpoints:** These endpoints are used by the client applications that register and unregister their services that they offer, and they can query the applications of the core services.
- **Private endpoints:** These endpoints are only accessible by other core services. Other core services can query desired services from other providers with parameters through these endpoints.



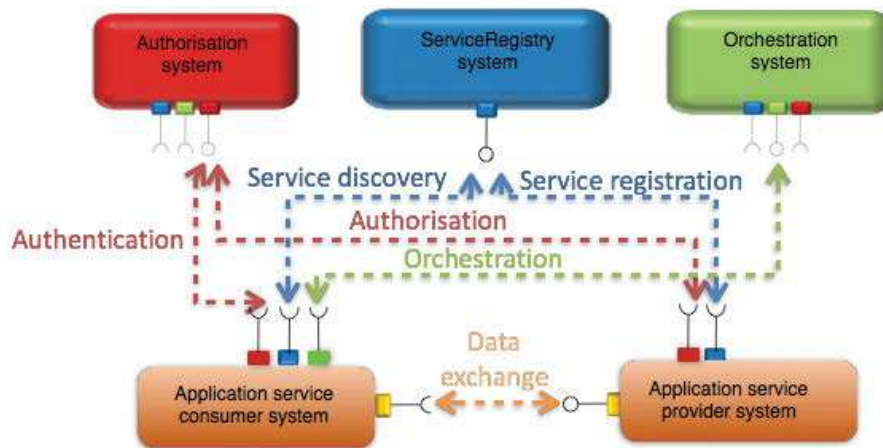


Figure 3.1: Mandatory core services with interactions enabling the service exchange between two application systems. (copied from [41])

- **Management endpoints:** These endpoints are used by management tools and cloud administrators to manage the local cloud. Administrators can get a list of registered services and details, as well as edit or delete these services.

In the ServiceRegistry component, the endpoint to register services is one of the most susceptible to performance issues. Multiple providers may try to register their services simultaneously. ServiceRegistry should check if this service is already registered and if the service provider already exists. This means that it requires a high computing power on the side of the ServiceRegistry and the database behind it.

### 3.1.2 Authorization

The Authorization core service manages the authorization rules with a database. It differentiates between provider and consumer applications, and stores in which consumer applications can access which provider applications. There are two types of authorization rules defined in the Authorization service. The first type is intra-cloud access rules. These set the access rules within the local cloud. The second type of rule is inter-cloud access rules. These define the access rules between different local clouds. As in Service Registry, we can differentiate between three types of endpoints, that are offered by the Authorization service:

- **Client endpoints:** These endpoints are used by the client applications. The only functionality is accessing the public key of the Authorization service.
- **Private endpoints:** These endpoints are only accessible by other core services. They can use these services to check intra-cloud, and inter-cloud access rules, and

generate an access token for a consumer system so that it can access a provider system.

- **Management endpoints:** These endpoints are used by management tools and cloud administrators to manage the local cloud. Administrators can get a list of all the access rules, filter them by ID, or delete them.

The authorization service is mostly used by the Orchestrator service to determine the applications that have access to a specific service. In the case of an enabled token generation for security, the authorization service requires high computing power, to check the access rules and generate a token. This may cause performance problems in the event of a high demand from the Orchestrator.

#### 3.1.3 Orchestrator

Orchestrator is a key component of the Arrowhead Framework. It allows service reusability, service discoverability, and service composability by finding and pairing service consumers and providers. It provides the consumers with the necessary information to establish a connection with the provider, including the accessibility information such as the network address, port, and a security token, if tokens are enabled. The orchestration process may unfold in two ways. In the first option, the orchestration rules are stored in the database to spot the appropriate providers for the consumer. This is called "Store Orchestration". The second option is "Dynamic Orchestration", where the Orchestrator searches the local cloud and neighboring clouds for the requested service. In the Orchestrator service, we have two types of endpoints:

- **Client endpoints:** These endpoints are used by the client applications. Application services may use these endpoints to initiate an orchestration process.
- **Management endpoints:** These endpoints are used by management tools and cloud administrators to manage the local cloud. Administrators can add orchestration rules for store orchestration, edit, delete, and prioritize them.

The dynamic orchestration process is especially resource-intensive because it needs to establish multiple connections with the service registry and authorization services to find the requested service.

#### 3.1.4 Gatekeeper

Gatekeeper provides inter-cloud servicing capabilities. Local clouds can search other local clouds through Gatekeeper for services. If an eligible service is found in another cloud, Gatekeeper may also initialize the negotiation with the Gatekeeper from the other cloud for consuming the service by obtaining the necessary information like tokens. For communication with other local clouds, Gatekeeper uses a Relay system, such as ActiveMQ [2]. In the Gatekeeper service, we have two types of endpoints:

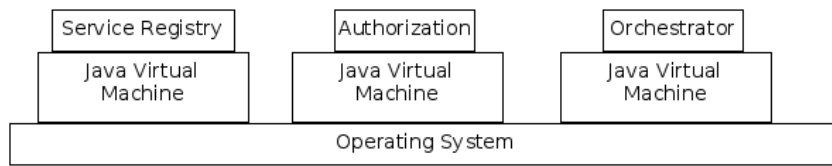


Figure 3.2: Arrowhead Framework Core services deployed

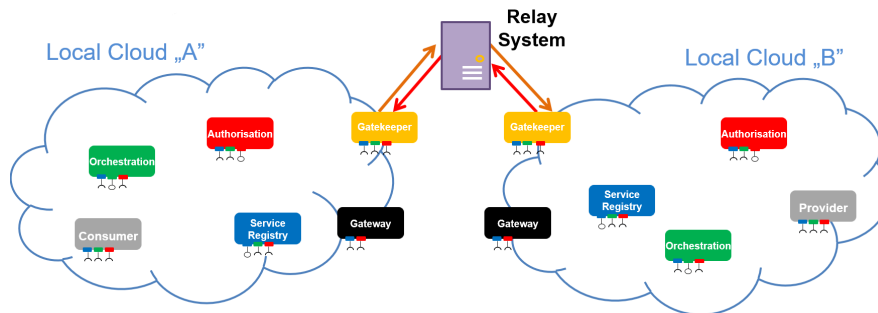


Figure 3.3: Inter-cloud communication over Gatekeeper (copied from [35])

- **Private endpoints:** These endpoints are only accessible by other core services. Through these endpoints, The Orchestrator service can initiate the search for specific services, and in case of success, it can obtain the necessary information for the connection from the other cloud.
- **Management endpoints:** These endpoints are used by management tools and cloud administrators to manage the local cloud. Administrators can add neighboring clouds to the database, add the relays, and edit and delete this data over these endpoints.

The Gatekeeper service is also resource-intensive. The communication with the relay and other core services and the orchestration process require a high computing ability in the local cloud and may cause performance issues.

### 3.1.5 Security

Security is an essential quality in the development of the Arrowhead Framework. The first aspect of security is the authorization in the local cloud. In order to ensure the authorization, all services in the local cloud must have an Arrowhead compliant X.509 identity certificate. These certificates make sure that the service is correctly bootstrapped into the Local Cloud, that they belong to the Cloud, and that they can register their services in the Service Registry. On the other side, the Authorization service ensures that

only the consumers that have access rights can consume the provider services and generate access tokens if enabled. Another essential security feature of the Arrowhead Framework is the encrypted communication between all the components. For the encryption, the HTTPS protocol is used. Even though these security features are essential, they may reduce the local cloud's performance and/or increase the resource consumption.

## 3.2 Java Virtual Machine

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. JVM is the core component that makes Java and other programming languages hardware and operating system independent. JVM is like a real computing machine with its own instruction set and memory manipulations at runtime [46].

When programming, developers write their code in Java programming language or others that can run on a JVM. This code is then compiled to a `class` file, that contains the Java bytecode. This bytecode is a binary format that is independent of hardware and operating system. All programming languages that can be compiled to Java bytecode can run on a JVM. In order to run the bytecode on a JVM efficiently, it needs to be recompiled to native/machine code. The binary code can also be interpreted by the JVM directly, but it is inherently slow. For the compilation process to native code, there are two main different approaches. Code compilation can occur at load-time or runtime. Load-time compilation is called ahead-of-time (AOT) compilation, and execution time compilation is called just-in-time (JIT) compilation. Both models have their advantages and disadvantages. In AOT compilation, the compilation happens only once, that is before the execution, and not on each execution of the program. This may increase the startup times. The JIT compilation happens at runtime and can increase the overall program performance by collecting profiling data and utilize this data for profile-guided code optimizations (PGO) to customize the native code for each input [55]. The JVM may also apply aggressive and potentially unsafe code optimizations speculatively, which then can be taken back if the speculation is invalidated later on. Depending on the implementation, a JVM can also combine these compilation methods.

The HotSpot JVM released by Oracle has introduced a so-called Tiered Compilation with Java Development Kit 7 (JDK 7). This type of compilation is also based on JIT compilation. However, with the introduction of different levels, it aims to improve the application's performance by using the profiling data. They have also released an AOT based compilation method, but Tiered Compilation is the default compilation method. The AOT compiler of HotSpot is still in an experimental phase [21].

The OpenJ9 JVM released by the Eclipse Foundation also uses a JIT based compilation method as default. They have introduced different optimization levels for compilation. Depending on the consumed time of a method, the compilation of the methods is increasingly optimized. This optimization process means a higher memory and CPU consumption during optimization, but the overall improved performance of the application

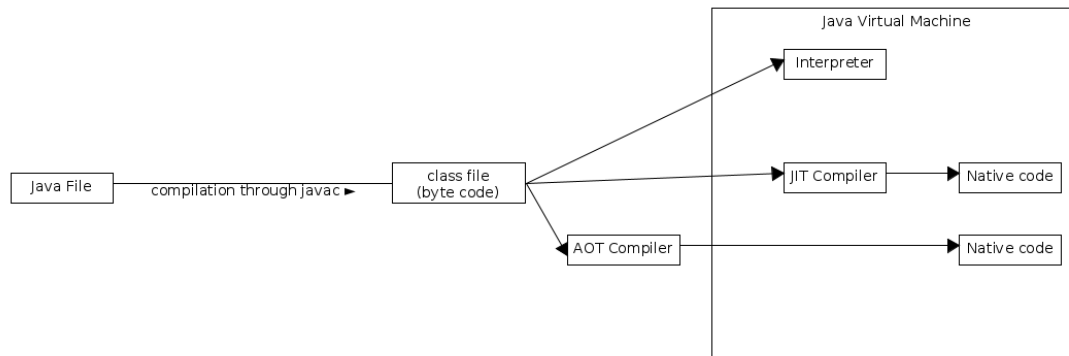


Figure 3.4: Compilation process of a Java class file

may make the tradeoff worthwhile [23]. The AOT compilation is also supported in OpenJ9, but needs to be enabled and is used mostly with class data sharing.

Class data sharing is a feature that may improve the startup times of applications and reduce the memory footprint. The class data that should be loaded into a JVM is loaded first to a cache. During subsequent JVM invocations, the shared cache is memory-mapped to allow sharing of class data for these classes among multiple JVM processes. Otherwise, the class data would need to be replicated in each JVM instance, which would increase the startup time of the application. This is especially useful if multiple JVM instances have to run on a single machine. Oracle has integrated this feature into HotSpot JVM in JDK 5 already. But until the release of JDK 10, the class data sharing was only limited to Java Runtime Environment (JRE) classes. With JDK 10, Oracle has introduced Application Class-Data Sharing (ApsCDS) to include selected classes from the application classpath [18]. OpenJ9 has implemented the class data sharing feature, as well. They also implemented the class data sharing for application classes before HotSpot, which has led to a more mature implementation [8]. OpenJ9 combines the class data sharing with AOT compiling to increase startup times. When class data sharing, which is also known as shared classes cache, is enabled, the AOT compiler is automatically activated. The AOT compiler compiles Java classes into native code for subsequent executions of the program. When an application runs in shared classes mode, the AOT compiler generates native code dynamically and saves this native code in the shared classes cache. Subsequent JVMs can load and use the AOT-compiled native code from the shared cache. This leads to a faster startup than running a JIT-compiled application [33]. When a cached AOT method is executed, it may also be optimized further by the JIT compiler [3]. Shared class cache is enabled by default for bootstrap classes. For application classes, it needs to be enabled explicitly.

The Arrowhead Framework is programmed in Java and runs on a JVM. The Arrowhead Framework's different components are based on the Spring Boot Framework [31], which

runs on an Apache Tomcat server. This means multiple instances of Spring Boot classes and other common libraries run on the same host machine. In this case, class data sharing may lead to better startup time and a decrease in memory footprint because of the high number of common libraries between different Arrowhead Framework components.

### 3.3 Testing Tools

For the execution of performance tests, we need a tool that can send multiple requests to different Arrowhead Framework components. The following specifics need to be supported by a testing tool to be eligible for our performance tests:

- **Count of maximum requests:** The testing tool should be able to send up to 10,000 requests to a component simultaneously.
- **REST interface:** The testing should support REST interfaces, since the Arrowhead Framework components communicate through REST interfaces.
- **Detailed analysis of results:** The testing tool should record the results in detail and ideally present the results with graphics.
- **HTTPS Support:** The Arrowhead Framework supports secured connection over HTTPS, which should be tested.
- **High Configurability:** Different parameters like a key store for HTTPS connection or timeout limits, should be easily configurable.
- **Costs:** The necessary features should be available free of charge.
- **Resource Consumption:** The testing should have an acceptable resource consumption.

There are numerous tools available on the market for performance tests. Apache JMeter [4] and Gatling [15] are currently the most common tools on the market, that are freely available. Both of them support the features that are listed above. The main difference is that the performance tests on JMeter are configured through a Graphical User Interface (GUI), and the performance tests on Gatling are programmed with Scala programming language. There are also some benchmarks on the Internet demonstrating that both testing tools have a comparable performance. However, Gatling requires less memory consumption [16]. All in all, we decided on Gatling as our testing tool, since it comes with all the necessary features in its free version, and consumes less memory.

# Methodologies

Before implementing the testing environment, we defined some objectives that our testing environment should achieve:

- **Repeatability of tests:** All performance tests should be repeatable. This is necessary so that other researchers can also repeat the tests to comprehend and retrace the results. It is also important to have an identical testing environment for every test for the repeatability of tests. That is to say, all tests should be automated.
- **Performance and resource observability:** In testing scenarios, multiple components of the Arrowhead Framework may interact with each other. For a better understanding of results, performance, and resource consumption of the components should be observable.
- **Load generation:** In order to understand the limitations of the framework, a remarkably high number of requests should be generated and sent to different components.
- **Adherence to specifications:** The testing environment should be implemented according to the Arrowhead Framework's specifications and standards.

For the repeatability of tests, it is essential that the Arrowhead Framework, as our test object, runs in a stable environment where software updates or other software running on the machine do not affect the Arrowhead Framework. Software updates or other software may have unexpected impacts on the results of our performance tests. We decided to utilize a Virtual Machine to protect our test object from abrupt alterations. A virtual machine allows us to execute the tests in an identical state of the machine for every test. Performance and resource observability can be ensured through our testing tool and

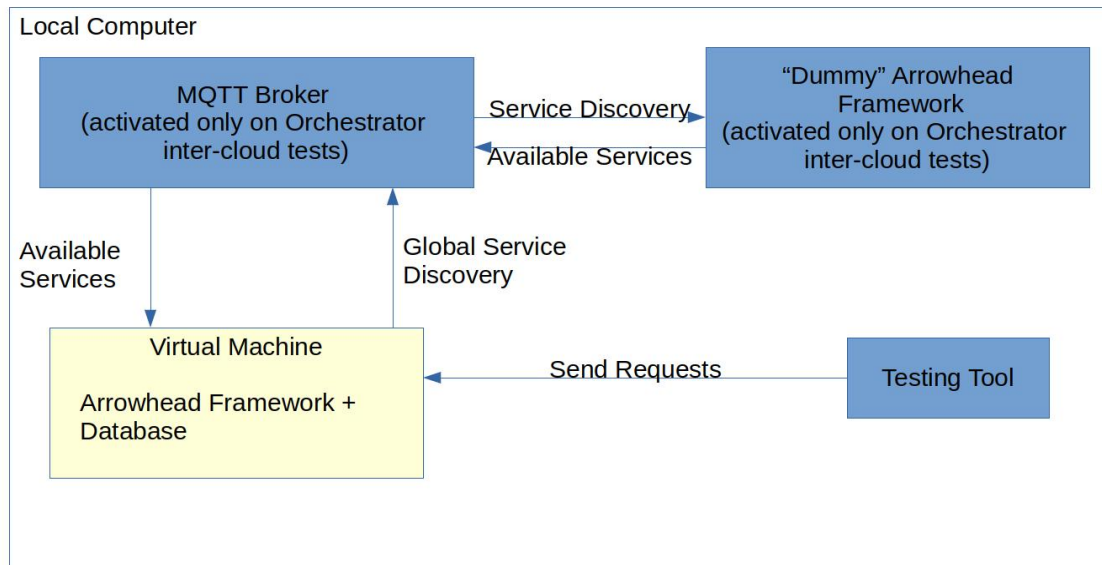


Figure 4.1: Testing Environment

tools that observe the resource consumption of Java applications. The load generation is provided by our testing tool. Our testing tool uses our test data generated earlier to send multiple requests to the components simultaneously.

One of the goals of the tests, besides finding potential errors, is to find out the count of simultaneous requests, where we have an acceptable response time. For response times, we can differentiate between 3 limits. If the response time is less than 0.1 second, the user feels that the system has reacted instantaneously, which requires special feedback for the user in addition to showing the result. If the response time is less than 1 second, the user's flow of thought remains uninterrupted, but the user feels the delay. If the response exceeds 10 seconds, the user loses their focus on the application. In this case, special feedback about the process should be given to the user about the progress of the task [50]. For this reason, our goal was to find out the limit, where we can guarantee a response time under 10 seconds.

The following components of the Arrowhead Framework were tested.

## 4.1 Service Registry

As discussed in the Background chapter, Service Registry does not have any dependency on other services. It is mainly responsible for the registration and management of the systems and their services. Multiple systems may try to register their services simultaneously at the Service Registry. With our performance tests, we want to determine the upper



limit of simultaneous registration requests. The upper limit may change depending on the hardware configurations and the Java Virtual Machine (JVM) that is being used. In order to understand the upper limits and potential problems, we send multiple service registration requests simultaneously. The only unit tests on the Service Registry are conducted for the registration service. Other core services also use Service Registry for different reasons, e.g. finding eligible services by the Orchestrator. These functionalities of Service Registry are also tested when testing the other components and are detailed in their respective sections.

## 4.2 Authorization

The Authorization service is responsible for the management of the access rights in inter-cloud and intra-cloud communication. First, these access rights need to be entered by an administrator. In order to understand the upper limits and potential problems, we simultaneously send multiple requests to the Authorization service to save the authorization rules. When a new authorization rule is added, the Authorization service contacts Service Registry to get the corresponding services. Therefore, this test is also an integration test, since it requires the communication of two elementary components. The Authorization service is also used by other core services, mainly by the Orchestrator. When searching for a service, the Orchestrator checks the authorization rules and generates the necessary authorization information for the consumer. Thus, other functionalities of Authorization are also tested as part of the tests of other components.

## 4.3 Orchestrator

Consumers contact the Orchestrator when they need a service. The Orchestrator searches the intra-cloud and inter-cloud for available services. The search process in intra-cloud and inter-cloud is different. When the Orchestrator receives a request for intra-cloud service discovery, it contacts the Service Registry to spot the eligible services for the requestor. If it finds any eligible service, it contacts the Authorization service to check if the requestor has access rights for this service. If access rights are given, and token-based security is enabled, a token is generated and returned to the requestor. For our first test on the Orchestrator, we sent multiple requests to the Orchestrator simultaneously for an intra-cloud dynamic discovery process so as to determine the upper limits and potential problems. It is important to keep in mind that all tests on the Orchestrator are also integration tests since the Orchestrator is dependent on other services.

The second type of service discovery takes place in inter-cloud. In the inter-cloud service discovery, the Orchestrator searches for the requested service in the neighboring clouds, which are also based on the Arrowhead Framework. Through an MQTT Broker, different local clouds may communicate with each other. In order to test an inter-cloud service discovery process, we need a realistic testing environment, including a second local cloud. The Arrowhead Framework local cloud that is deployed in the Virtual Machine is our test

object. The other local cloud is a modified version of the Arrowhead Framework with limited functionalities. These modifications reduce resource consumption by the second Arrowhead Framework so that our tests' results are not affected. Since both instances of the Arrowhead Framework run on the same physical machine (see Figure 4.1 ), it is important to limit the resource consumption of the second "dummy" instance.

Our test object, the Arrowhead Framework in the VM, is tested in two aspects for the inter-cloud search process. First, the performance is tested when an inter-cloud search request is sent to the local cloud in the VM. When the Orchestrator receives a request, it begins a search in the neighboring cloud, in our case the "dummy" Arrowhead Framework on our local machine. As in other tests, multiple requests are sent simultaneously to determine the upper limits and potential problems. In this use case, the most noteworthy part of the computation occurs in the "dummy" Arrowhead Framework. It needs to search the requested service in its local cloud and return the results with access rules and authorization information. As mentioned earlier, this search process is simplified and modified in the "dummy" Arrowhead Framework to reduce the resource consumption, so that our test results are not affected. The second aspect of the inter-cloud search is when the request is received by the "dummy" local cloud and the "dummy" local cloud searches in its neighboring clouds for the requested service (see Figure 4.2 ). In this case, the "dummy" cloud searches for the service in the local cloud, the Arrowhead Framework in the VM. In this case, most of the computation occurs in the VM.

We tested the dynamic orchestration process, since it requires a higher computational power and may lead to problems.

### 4.4 Gatekeeper

The Gatekeeper is responsible for the inter-cloud communication. It communicates with the MQTT Broker to initialize the service discovery with other local clouds, and acquire the information of the requested service in the event of success. The Gatekeeper cannot be used directly by application services. The functionalities of the Gatekeeper are tested as a part of the Orchestrator tests for inter-cloud communication.

### 4.5 Security

Security functions like encryption and token generation require a high computational power. In order to understand the impacts of the security functions on the performance, tests for service registration, and storing authorization rules are also repeated in an insecure mode. In the insecure mode, HTTPS and token-based security are disabled.

### 4.6 Java Virtual Machine

For the comparison of HotSpot and OpenJ9, we compared the performance of both JVMs according to their response times and memory consumption. In order to better evaluate

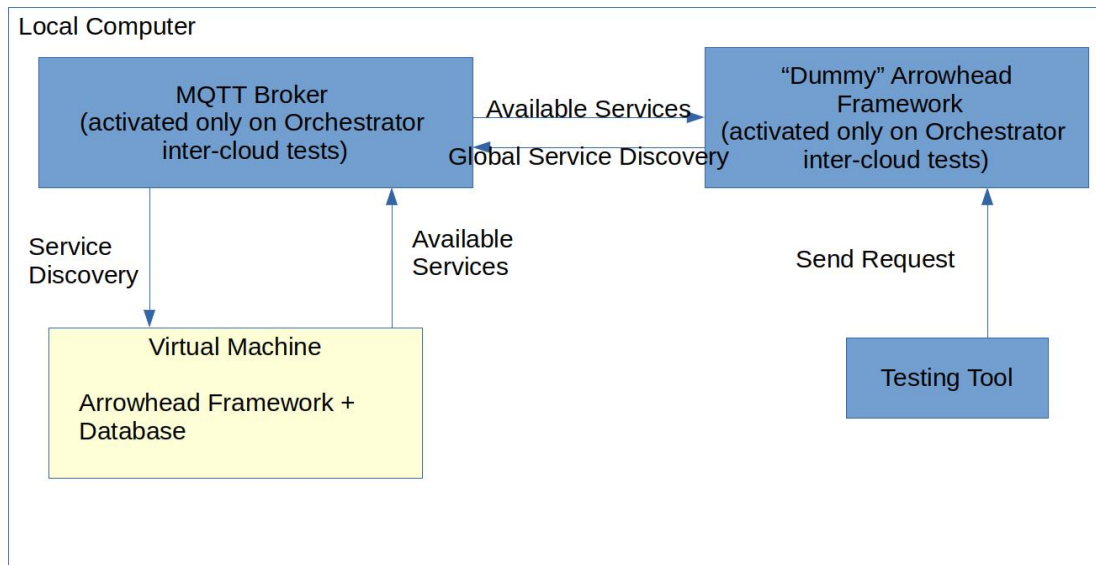


Figure 4.2: Testing Environment for the second version of Orchestrator tests

the performance of the JVMs, the tests were also repeated on OpenJ9, which ran on both JVMs without errors. Performance tests with an extremely high number of simultaneous requests, which led to timeouts and/or errors on HotSpot, were not repeated on OpenJ9.

The tests on OpenJ9 were repeated in two different modes. First, we used the default mode of OpenJ9, where we did not change any configurations. We repeated the same tests with shared classes cache to observe if this mode has any impact on resource consumption or performance.

## 4.7 Tests on different hardware

We executed the performance tests on two different hardware configurations. x86 based CPUs, and ARM-based CPUs are widely used in the industry. Especially in recent years, ARM-based devices have gained popularity in the IoT field because of their lower energy consumption [44]. These devices have limited resources. In order to understand the limitations of the Arrowhead Framework, especially on a device with limited resources, performance tests have been repeated on two devices with those architectures.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Implementation

## 5.1 Used Software and Hardware

For the implementation of our performance tests on the x86 based computer, we used a Virtual Machine (VM) with the following software:

Function	Used Software
Hypervisor for virtualization	Oracle VM Virtualbox 6.1.6
Virtual Machine Operating System	Ubuntu 18.04 64-Bit
Java Development Kit, JVM 1	OpenJDK 11.0.7 64-Bit, HotSpot JVM
Java Development Kit, JVM 2	OpenJDK 11.0.7 64-Bit, OpenJ9 0.20.0
Database	MySQL 5.7.29

The Virtual Machine had the following virtual hardware properties:

CPU:	6 Core CPU
RAM:	4096 MB
Hard Disk:	20 GB

The host machine, on which the Virtual Machine was running, had the following hardware and software properties:

CPU	Intel Core i7-10710U 6x 1.10GHz
RAM	16 GB
Hard Disk	512 GB SSD
Operating System	Ubuntu 20.04 64-Bit

For our performance tests on an ARM-based machine, we used a Raspberry Pi 4 with 4 GB RAM. Tests were directly performed on the host machine. On the Raspberry Pi the following software was running:

Function	Used Software
Operating System	Raspberry Pi OS 10
Java Development Kit, JVM	OpenJDK 11.0.6, HotSpot JVM
Database	MariaDB 10.3.22

On both platforms, the Arrowhead Framework 4.1.3 was running for our tests. On the official software repositories of Raspberry Pi OS, there is no MySQL. Instead, MariaDB is recommended as a replacement. MariaDB is based on MySQL and MariaDB 10.3 is almost fully compatible with MySQL 5.7 [24].

We installed the Arrowhead Framework natively from .deb files on both platforms. We chose Gatling 3.3.1 as our testing tool, and as the MQTT Broker, we used Apache ActiveMQ 5.15.12.

The resource consumption of the applications was monitored with JConsole.

## 5.2 Testing Environment Configurations

For better comparability of performance, response times are essential. For the comparison of the response times, we have raised the timeout limits of `connectTimeout`, `handshakeTimeout`, and `requestTimeout` of Gatling to 300,000 ms.

Furthermore, in all the Operating Systems the open-files limit was raised to 65,536.

The shared classes cache for OpenJ9 in VM was enabled with `groupAccess`, `persistent`, `cacheDir="/var/tmp/javasharedresources"` options, which enables all applications of a group in Ubuntu to access the same cache and that cache does not get deleted on system restart (See the specifications on [1]). All other options were left as default. See [1] for default values.

## 5.3 Generating Test Data

In order to execute our performance tests, as realistically as possible, we first generated test data, where every entry was unique. We fed our test data to Gatling through the CSV files. In the first step, we generated test data with service information. The following variables of a service were randomly generated: `serviceDefinition`, `systemName`, `port`, `authenticationInfo` (only in secure mode), `serviceUri`, `version`, `interfaces`. The detailed explanations of the respective fields can be found in the Service Registry section of this chapter. This data was created for 10,000 services.

This data was used for our Service Registry tests. The next step was to create data for authorization rules. To generate the intra-cloud and inter-cloud authorization rules, we inserted the 10,000 services into the database through the Arrowhead Framework. The Arrowhead Framework assigned IDs to each system and service. The services with their generated IDs were then exported to a CSV file and a backup file for MySQL.

For our Authorization tests, we need test data on the following parameters: `consumerId` (only for intra-cloud), `cloudId` (only for inter-cloud), `providerId`, `interfaceId`, `serviceDefinitionId`. These fields determine the authorization rules. The detailed explanations of the respective fields can be found in the Authorization section of this chapter. This data was also generated randomly, where we mapped producers with consumers, to generate 10,000 authorization rules. After inserting these rules into the database through the Arrowhead Framework, we once again exported this data to a CSV file and a backup file for MySQL.

For the Orchestrator tests, we need the following parameters in a single CSV file: `consumerName`, `port`, and `serviceDefinition` (of the requested service). The detailed explanations of the respective fields can be found in the Orchestrator section of this chapter. For the service discovery tests to succeed in the Orchestrator, sent requests should contain consumers with authorization rights for the requested service definition. In order to achieve this, we mapped the created authorization rules with the service information and exported this data also as a CSV file.

We generated 10,000 Arrowhead compliant X.509 identity certificates as authentication information to be used as part of the services in secure mode.

All test data was generated with our Java program, that was developed for test data generation for this project.

## 5.4 Service Registry

As mentioned in earlier chapters, the Service Registry does not have any dependency on other services and is responsible for the management of services. In Background, we have determined the service registration process as particularly resource-intensive. In order to test the service registration process, the following JSON file was sent simultaneously to the `/serviceregistry/register` interface of the Service Registry service:

```
{  "serviceDefinition": "${serviceDefinition}",
   "providerSystem": {
     "systemName": "${systemName}",
     "address": "localhost",
     "port": ${port},
     "authenticationInfo": "${AuthInfo}" // existent only in secure mode
   },
   "serviceUri": "${serviceUri}",
   "secure": // in tests with secure mode "TOKEN" ,
//in tests with insecure mode "NOT_SECURE"
   "metadata": {
     },
   "version": "${version}",
   "interfaces": [
```

```
"${interfaces}"  
]  
}
```

The variables starting with "\$" are taken from the CSV data. The variables have the following meaning:

- **serviceDefinition:** The name of the service definition. A random alphanumeric string was generated for this variable.
- **systemName:** The name of the provider system. A random alphanumeric string was generated for this variable.
- **port:** The port for the connection to this service. A random number between 0 and 65,536 was generated for this variable.
- **AuthInfo:** This authentication information is only provided if the service runs in secure mode. It is the public key of the system. For every service, there was a unique public key.
- **serviceUri:** This represents the URI for the connection to the service. A random alphanumeric string was generated for this variable.
- **version:** The version of the registered service. An integer between 0 and 30 was generated for this variable.
- **interfaces:** The interface that is provided by this service. Following options were used for our tests in insecure mode: "HTTP-INSECURE-JSON", "HTTP-INSECURE-XML", "HTTP-INSECURE-CSV", "HTTP-INSECURE-TEXT" and "HTTP-SECURE-JSON", "HTTP-SECURE-XML", "HTTP-SECURE-CSV", "HTTP-SECURE-TEXT" in secure mode.

First, we performed our tests on the Virtual Machine with HotSpot JVM. We have sent 100, 500, 1000, 5000, and 10,000 requests simultaneously in the secure mode. In the insecure mode, we have 10,000 requests only. For every test, the virtual machine was reset to its initial state, with the Arrowhead Framework freshly installed and empty tables in the database.

Our primary goal is, to determine the limit, where we can achieve an acceptable response time in different hardware platforms. Building on our results from the Virtual Machine with HotSpot JVM, we carried out the performance tests with the following configurations:

- **Virtual Machine, OpenJ9 default mode:** 1000, 5000, and 10,000 simultaneous requests on secure mode.



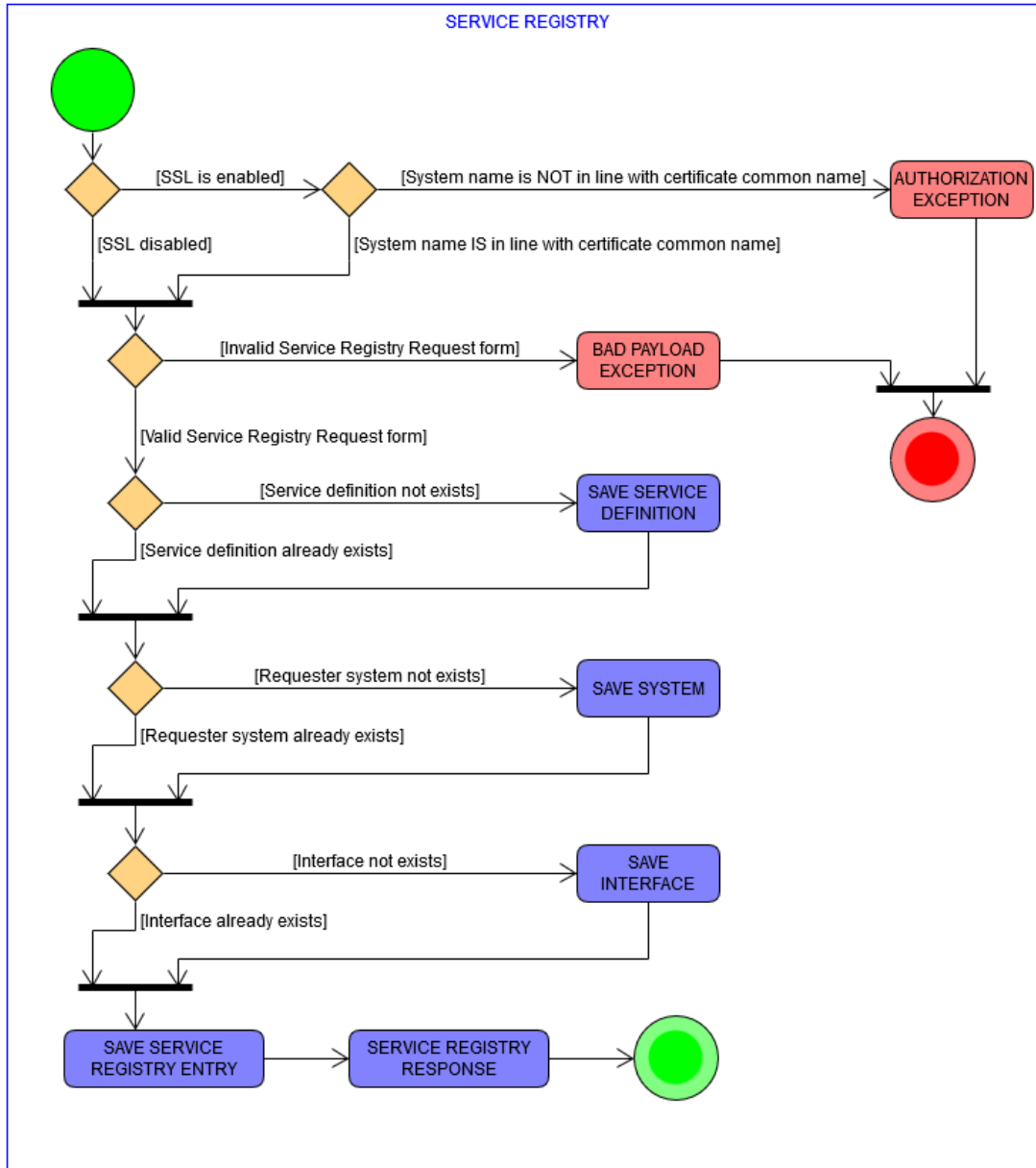


Figure 5.1: Service Registration Activity Diagram (copied from [35])

- **Virtual Machine, OpenJ9 with enabled shared classes cache:** 1000, 5000, and 10,000 simultaneous requests on secure mode.
- **Raspberry Pi:** 100 and 500 simultaneous requests on secure mode.

Other functionalities of Service Registry are tested as part of the integration tests in the Authorization and Orchestrator tests because they are dependent on the Service Registry.

In order to execute our tests, we deactivated the security mechanism of the Service Registry, which checks if the provider name in a registration request matches the common name of the security certificate. This was performed by commenting out some lines in `SRAccessControlFilter.java`.

## 5.5 Authorization

The Authorization service is dependent on the Service Registry service. For our performance tests on the Authorization service, we inserted intra-cloud rules to the database through the Authorization service. For our intra-cloud Authorization rules, we have sent the following JSON files through `/authorization/mgmt/intracloud` interface:

```
{
  "consumerId": ${consumerId},
  "interfaceIds": [
    ${interfaceId}
  ],
  "providerIds": [
    ${providerId}
  ],
  "serviceDefinitionIds": [
    ${serviceDefinitionId}
  ]
}
```

For generating test data, the inter-cloud rules we sent the following JSON files through `/authorization/mgmt/intercloud` interface:

```
{
  "cloudId": 2,
  "interfaceIdList": [
    ${interfaceId}
  ],
  "providerIdList": [
    ${providerId}
  ]
}
```

```

],
"serviceDefinitionIdList": [
  ${serviceDefinitionId}
]
}

```

The variables starting with "\$" are taken from the CSV data. The variables have the following meaning:

- **consumerId:** The ID of the consumer service in the local cloud that has access to the service. This is existent only in intra-cloud authorization rules.
- **cloudId:** The ID of the cloud, that has access to this resource. This is existent only in inter-cloud authorization rules.
- **providerId:** The ID of the provider of this service. In our tests, we had only one provider ID in this field.
- **interfaceId:** The ID of the service interface.
- **serviceDefinitionId:** The ID of the service definition. We have only one ID in this field in our tests.

All our tests were done with valid IDs. It means that all tests should be successful. In these tests, Authorization service queries the Service Registry to check if the IDs of the service are valid and returns the information about the consumer and provider systems.

First, we performed our tests on the Virtual Machine with HotSpot JVM. We have sent 1000, 5000, and 10,000 requests simultaneously in the secure mode. In the insecure mode, we sent 10,000 requests only. For every test, the virtual machine was reset to its initial state, with the Arrowhead Framework freshly installed, only containing services in the database. Our primary goal is, to determine the limits, where we can achieve an acceptable response time in different hardware platforms. Building on our results from the Virtual Machine with HotSpot JVM, we conducted the performance tests with the following configurations:

- **Virtual Machine, OpenJ9 default mode:** 1000 simultaneous requests on secure mode.
- **Virtual Machine, OpenJ9 with enabled shared classes cache:** 1000 simultaneous requests on secure mode.
- **Raspberry Pi:** 100, and 1000 simultaneous requests on secure mode.

Other functionalities of the Authorization are tested as part of the integration tests in the Orchestrator tests since the Orchestrator is dependent on the Authorization.

## 5.6 Orchestrator

The Orchestrator is responsible for inter-cloud and intra-cloud service discovery. Their tests are essentially different.

### 5.6.1 Intra-cloud service discovery

Consumers may send requests to the Orchestrator to initiate intra-cloud service discovery. For our performance tests, we have sent the following JSON data to the `/orchestrator/orchestration` interface of the Orchestrator:

```
{
  "requesterSystem": {
    "systemName": "${consumerName}",
    "address": "localhost",
    "port": ${port},
    "authenticationInfo": ""
  },
  "requestedService": {
    "serviceDefinitionRequirement": "${serviceDefinition}",
    "securityRequirements": [
      "NOT_SECURE", "CERTIFICATE", "TOKEN"
    ],
    "metadataRequirements": {
    }
  },
  "orchestrationFlags": {
    "overrideStore": true
  }
}
```

The variables starting with "\$" are taken from the CSV data. The variables have the following meaning:

- **consumerName:** The name of the consumer system.
- **port:** The port of the consumer system
- **serviceDefinition:** The name of the requested service definition.

The orchestration flag `overrideStore` signifies that a dynamic service discovery should be performed instead of a store-based service discovery.

All our tests were done using valid data. That is to say, all tests should be successful. In these tests, when a request arrives, the Orchestrator contacts the Service Registry to find the requested service. If the requested service is found, it checks with the Authorization service to view if this consumer has access rights for this service. If the access rights are given, a token is generated, and a response is sent back to the consumer, in our case to Gatling (see Figure 5.2) .

First, we have performed our tests on the Virtual Machine with HotSpot JVM. We sent 100, 500, 1000, and 10,000 requests simultaneously in the secure mode. For every test, the virtual machine was reset to its initial state, with the Arrowhead Framework freshly installed and services and intra-cloud authorization rules present in the database.

Our primary goal is to determine the limits, where we can achieve an acceptable response time in different hardware platforms. Building on our results from the Virtual Machine with HotSpot JVM, we conducted the performance tests with the following configurations:

- **Virtual Machine, OpenJ9 default mode:** 100, 1000, and 10,000 simultaneous requests on secure mode.
- **Virtual Machine, OpenJ9 with enabled shared classes cache:** 100, 1000, and 10,000 simultaneous requests on secure mode.
- **Raspberry Pi:** 50, 70, and 100 simultaneous requests on secure mode.

### 5.6.2 Inter-cloud service discovery - Gatekeeper

Consumers may send requests to the Orchestrator to initiate the inter-cloud service discovery. As mentioned in Methodologies, in order to have a testing environment that is as realistic as possible, we deployed a second Arrowhead Framework instance (see Figures 4.1 and 4.2). The second instance of the Arrowhead Framework running in the local computer was modified to limit the resource consumption. We modified the doGSDPoll and doICN methods in the GatekeeperService.java file in the Gatekeeper. These methods are responsible for contacting the local Orchestrator for obtaining service information and authorization information. We commented out these functionalities and hardcoded a service and authorization information. This modified Gatekeeper sends the same response at all times, which massively reduces the computation that is required on this local cloud.

In order to test the inter-cloud service discovery, we considered two cases. In the first case, we sent our inter-cloud service discovery requests to the Orchestrator in the VM, and in the second case, we sent our requests to the Orchestrator in our localhost.

#### Orchestrator inter-cloud 1 - Requests sent to VM

In our first testing scenario, we sent the inter-cloud service discovery requests to the Orchestrator running on our VM. In this case, the Arrowhead Framework in the VM

## 5. IMPLEMENTATION

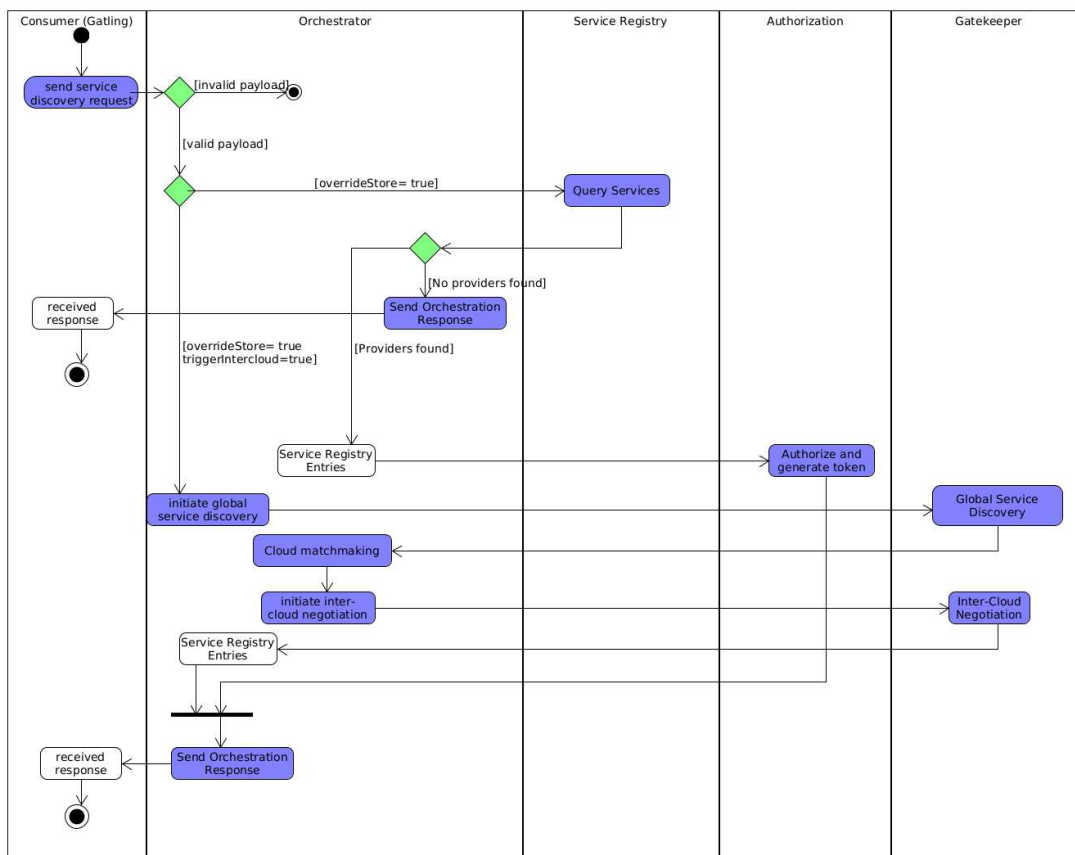


Figure 5.2: Orchestrator tests Activity Diagram [35]

searches for the services in our "dummy" Arrowhead Framework. The main computation and search process would take place in the "dummy" Arrowhead Framework. The following JSON request was sent to the Orchestrator:

```

{
  "requesterSystem": {
    "systemName": "${consumerName}",
    "address": "localhost",
    "port": ${port},
    "authenticationInfo": ""
  },
  "requestedService": {
    "serviceDefinitionRequirement": "${serviceDefinition}",
    "securityRequirements": [
      "NOT_SECURE", "CERTIFICATE", "TOKEN"
    ]
  }
}

```

```

    ],
    "metadataRequirements": {
    },
  },
  "orchestrationFlags": {
    "overrideStore": true,
    "triggerInterCloud" : true
  }
}

```

The variables starting with "\$" are taken from the CSV data. The variables have the following meaning:

- **consumerName:** The name of the consumer system.
- **port:** The port of the consumer system
- **serviceDefinition:** The name of the requested service definition.

The orchestration flags `overrideStore` and `triggerInterCloud`, meaning that a dynamic service discovery in the inter-cloud should be established instead of a store-based service discovery. In this case, service discovery in the intra-cloud is skipped.

First, we performed our tests on the Virtual Machine with HotSpot JVM. We sent 40, 100, and 1000 requests simultaneously in the secure mode. For every test, the virtual machine was reset to its initial state, with the Arrowhead Framework freshly installed and services with inter-cloud authorization rules present in the database.

Our primary goal is to determine the limits, where we can achieve an acceptable response time in different hardware platforms. Building on our results from the Virtual Machine with HotSpot JVM, we have conducted the performance tests with the following configurations:

- **Virtual Machine, OpenJ9 default mode:** 40 simultaneous requests on secure mode.
- **Virtual Machine, OpenJ9 with enabled shared classes cache:** 40 simultaneous requests on secure mode.
- **Raspberry Pi:** 20, 30, and 40 simultaneous requests on secure mode.

## Orchestrator inter-cloud 2 - Requests sent to "dummy" Arrowhead Framework

In our second test scenario for the Orchestrator, we sent the inter-cloud service discovery requests to the "dummy" Orchestrator running on the local computer. In this case, the "dummy" Arrowhead Framework searches for the services in our Arrowhead Framework on the VM. The main computation and search process occurs in the Arrowhead Framework on the VM. The JSON request is identical to our first inter-cloud test.

First, we performed our tests on the Virtual Machine with HotSpot JVM. We sent 50, 100, and 1000 requests simultaneously in the secure mode. For every test, the virtual machine was reset to its initial state, with Arrowhead Framework freshly installed and services and inter-cloud authorization rules present in the database.

Our primary goal is to determine the limits, where we can achieve an acceptable response time in different hardware platforms. Building on our results from the Virtual Machine with HotSpot JVM, we have conducted the performance tests with the following configurations:

- **Virtual Machine, OpenJ9 default mode:** 50 simultaneous requests on secure mode.
- **Virtual Machine, OpenJ9 with enabled shared classes cache:** 50 simultaneous requests on secure mode.
- **Raspberry Pi:** 10, 20, and 30 simultaneous requests on secure mode.

## 5.7 Gatling Configuration

As we have noted earlier, we used Gatling as our testing tool. We have raised the timeout limits `connectTimeout`, `handshakeTimeout`, `requestTimeout` of Gatling to 300,000 ms. In tests in secure mode, we configured the key store, as well. All other configuration values were left as default (See specification [10] for default values). Following code snippet is an example of a test in Gatling:

```
val feeder = csv("Name_of_the_csv_file.csv").queue

val stringBody =
  """JSON data to be sent"""

val httpProtocol = http
  .baseUrl("Base_URL_of_testing_object")
  .acceptHeader("text/html,application/xhtml+xml,
application/xml;q=0.9,*/*;q=0.8")
  .doNotTrackHeader("1")
```



```

        .acceptLanguageHeader("en-US, en;q=0.5")
        .acceptEncodingHeader("gzip, deflate")
        .userAgentHeader("Mozilla/5.0 (Windows NT 5.1; rv:31.0)
        Gecko/20100101 Firefox/31.0")

    val scn = scenario("SimulationName").feed(feeder).exec(http("request_1").
        post("URL_of_object").body(StringBody(stringBody)).asJson)

    setUp(
        scn.inject(atOnceUsers(Count_of_simultaneous_requests))
    ).protocols(httpProtocol)

```

In the code snippet, first we feed the test data through a CSV file in our test and define the JSON data that need to be filled with the test data. Then the HTTP protocol with the base URL and headers are defined. Gatling sends the requests simultaneously to our defined URL for our test scenario. The method `atOnceUsers` defines, that the number of requests should be sent simultaneously.

## 5.8 Limitations

The CPUs that we utilized use, as most CPUs today, the technique of dynamic frequency scaling. That is to say, the clock speed of the CPU is adjusted automatically depending on the demand, the power mode and environment temperature, and other variables. The users are offered very limited options to alter this behavior. We cannot guarantee that all tests were conducted at the same clock speed in our performance tests. However, it is our expectation that these differences were not major since the laptop's environment temperature and power mode remained almost identical. This behavior is displayed on our Intel CPU and ARM CPU of Raspberry Pi [14] [27].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Results and Discussion

In this chapter, we present the results of our tests. The results of the Arrowhead Framework components are based on our tests on secure mode. We also have a separate section, where we compared the performance in secure and insecure mode. For memory consumption, we present the peak heap memory usage. If the error count is not shown in the charts, there were no errors found.

## 6.1 Service Registry

In this section, we present our results from our performance tests for Service Registry. In Figure 6.1, we present the results of our experiment on Service Registry with the HotSpot JVM on the VM, where we sent 10,000, 5000, 1000, 500, and 100 requests simultaneously. We observe that even with 10,000 simultaneous requests, the Service Registry component can handle the requests with a mean response time of 41.14 seconds with only six errors. In tests with less simultaneous requests, the mean response time decreases rapidly. On 5000 simultaneous requests, we have a mean response time of 22.8 seconds and seven errors, on 1000 simultaneous requests a mean response time of 6.93 seconds and three errors, on 500 simultaneous requests a mean response time of 5.39 seconds and five errors. On 100 simultaneous requests, we have a mean response time of 1.28 seconds and four errors. Even with simultaneous requests as low as 100 requests, the reason for errors lies in a bug with concurrent database entries. See Section 6.6 for details on this bug. Our results reveal that on the VM, we can guarantee an acceptable response time under 10 seconds with <1000 simultaneous requests (see Figure 6.1). However, this number is highly dependent on the hardware configuration. In Figure 6.2, we present the results of our experiment on Raspberry Pi 4, where we sent 500 and 100 simultaneous requests. On the Raspberry Pi 4, on 500 simultaneous requests, we have a mean response time of 45.93 seconds and five errors, and on 100 simultaneous requests, a mean response time of 7.34 seconds and seven errors. On the Raspberry Pi 4, we achieve an acceptable response

## 6. RESULTS AND DISCUSSION

time only with <100 simultaneous requests. The ARM-based processors like the one found on Raspberry Pi cannot reach the performance of x86 based processors, like on our VM.

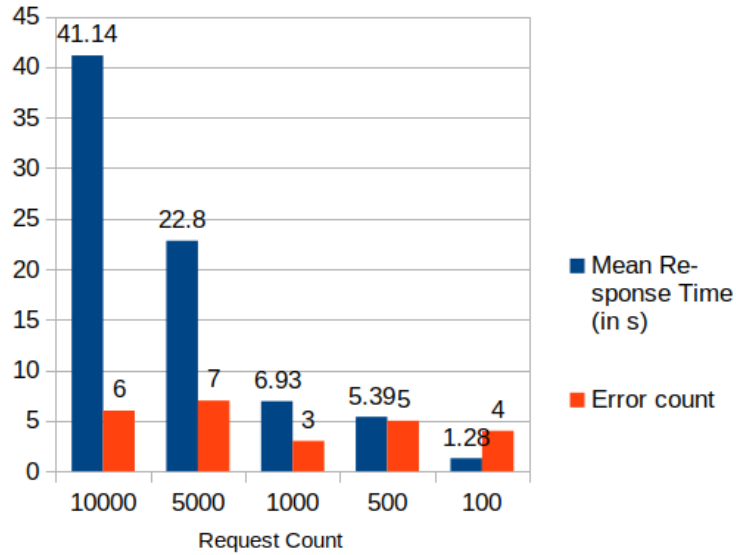


Figure 6.1: Service Registry Mean Response Times with error count on the x86 HotSpot JVM

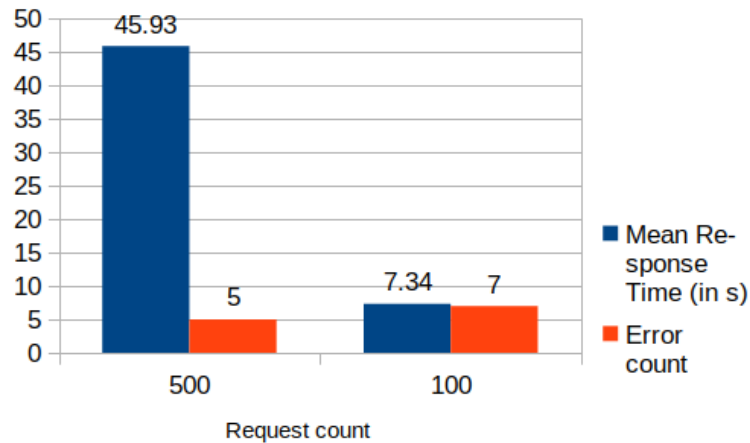


Figure 6.2: Service Registry tests - Response Time on Raspberry Pi with the HotSpot JVM

We also compared the performance and memory consumption of different JVM con-

figurations. In Figure 6.3 and 6.4, we present the memory consumption and mean response times of the Service Registry on the HotSpot JVM, on OpenJ9 with default configurations, and on OpenJ9 with shared classes cache with 10,000, 5000, and 1000 simultaneous requests. On 10,000 requests, the HotSpot JVM had a mean response time of 41.14 seconds with 980 MB memory consumption, OpenJ9 with default configurations 52.14 seconds with 980 MB memory consumption, and OpenJ9 with enabled shared classes cache a mean response time of 50.36 with 980 MB memory consumption. On 5000 simultaneous requests, the HotSpot JVM had a mean response time of 22.80 seconds with 680 MB memory consumption, OpenJ9 with default configurations a mean response time of 26.74 seconds 772 MB memory consumption, and OpenJ9 with enabled shared classes cache a mean response time of 26.41 seconds with 651 MB memory consumption. On 1000 simultaneous requests, the HotSpot JVM had a mean response time of 6.93 seconds with 266 MB memory consumption, OpenJ9 with default configurations a mean response time of 10.86 seconds with 286 MB memory consumption, and OpenJ9 with shared classes cache a mean response time of 8.22 seconds with 250 MB memory consumption. In all of our tests, the HotSpot JVM with default configurations yielded the best performance with the lowest mean response time. Even though OpenJ9 with default configurations incurred the highest memory consumption in all the tests, it also displayed the worst performance with the highest response times. OpenJ9 with shared classes cache (SCC) delivered the lowest memory consumption, but it also showed higher response times than the HotSpot JVM (see Figures 6.3 6.4). In this test, we discuss that the HotSpot JVM is suitable for use cases where we expect high traffic and high performance on Service Registry. OpenJ9 SCC is suitable when we are dealing with devices with limited resources that do not expect a high Service Registry traffic. OpenJ9 with default configurations increased memory consumption and reduced the performance, rendering it unsuitable for any use case.

## 6.2 Authorization

In this section, we present our results from our performance tests for Authorization. In Figure 6.5, we present the results of our experiment on Authorization with the HotSpot JVM on the VM, where we sent 10,000, 5000, 1000 requests simultaneously. On the same Figure, we also present the results of the tests performed with OpenJ9 using default configurations and OpenJ9 with shared classes cache (SCC), where we sent 1000 requests. In our tests with the HotSpot JVM on the VM with 10,000 simultaneous requests, the Authorization component has a mean response time of 44.3 seconds, on 5000 simultaneous requests a mean response time of 24.4 seconds, and on 1000 requests a mean response time of 7.89 seconds. Our results reveal that on the VM, we can guarantee an acceptable response time under 10 seconds with <1000 simultaneous requests (see Figure 6.5). However, this number is highly dependent on the hardware configuration. In Figure 6.6, we present the results of our experiment on Raspberry Pi 4, where we have sent 1000 and 100 requests simultaneously. On the Raspberry Pi 4, on 1000 simultaneous requests, we have a mean response time of 76.17 seconds, and on 100 simultaneous requests a mean

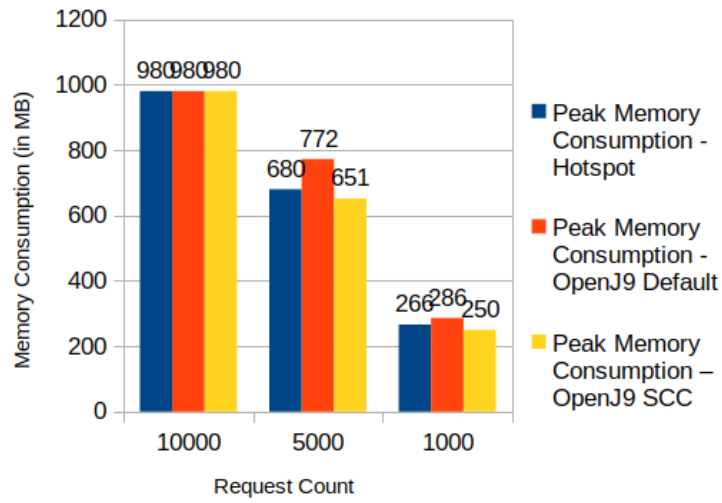


Figure 6.3: Service Registry tests - Memory Consumption on different JVM configurations

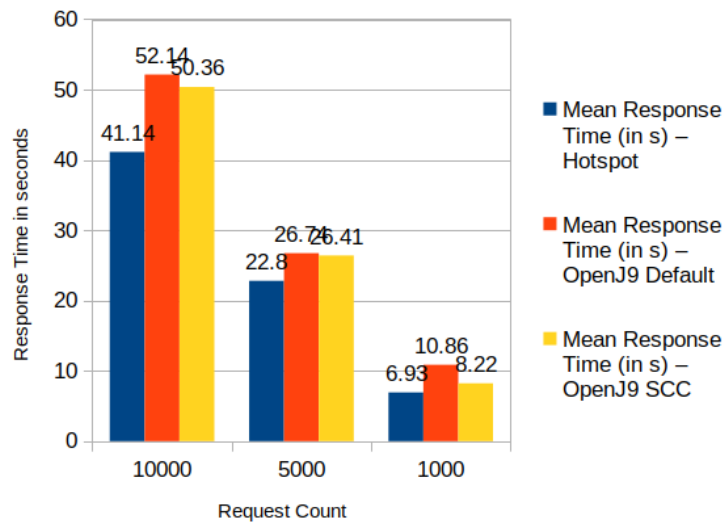


Figure 6.4: Service Registry tests - Response Time on different JVM configurations

response time of 9.21 seconds. On Raspberry Pi 4, we have a mean acceptable response time only with <100 simultaneous requests.

We have repeated the tests with 1000 simultaneous requests on OpenJ9 and recorded the response times and memory consumption of the Authorization and Service Registry components. In Figure 6.5 the response times and in Figure 6.7 the memory consumptions

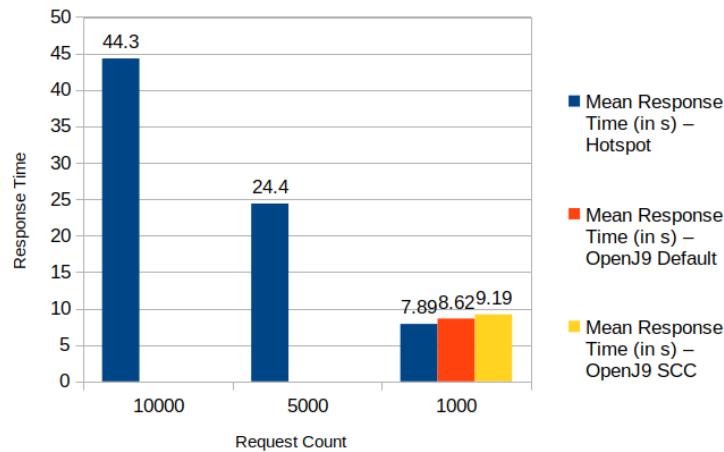


Figure 6.5: Authorization tests - Mean Response Times - OpenJ9 tests were only done with 1000 requests

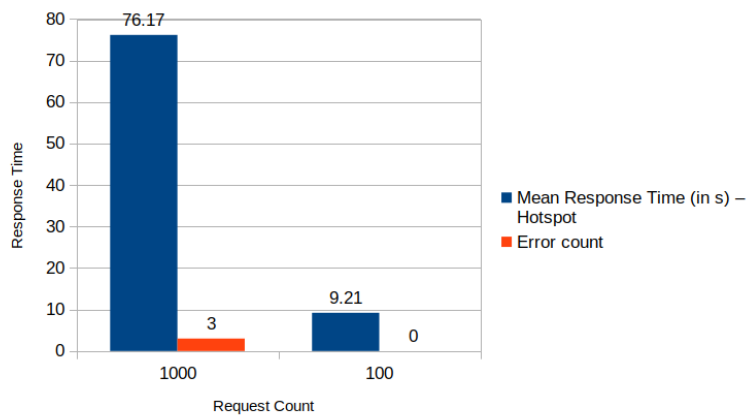


Figure 6.6: Authorization tests - Mean Response Times with error count on Raspberry Pi

are presented. On OpenJ9 with default configuration, we have a mean response time of 8.62 seconds, and on OpenJ9 with shared classes cache a mean response time of 9.19 seconds. The memory consumption of the Service Registry component on the HotSpot was 190 MB, on OpenJ9 with default configurations 113 MB, and on OpenJ9 with shared classes cache 97 MB. The memory consumption of the Authorization component was 260 MB on the HotSpot JVM, 306 MB on OpenJ9 with default configurations, and 245 MB on OpenJ9 with shared classes cache.

In our test, the HotSpot JVM with default configurations, achieved the best performance with the lowest mean response time. Considering the sum of the memory consumption

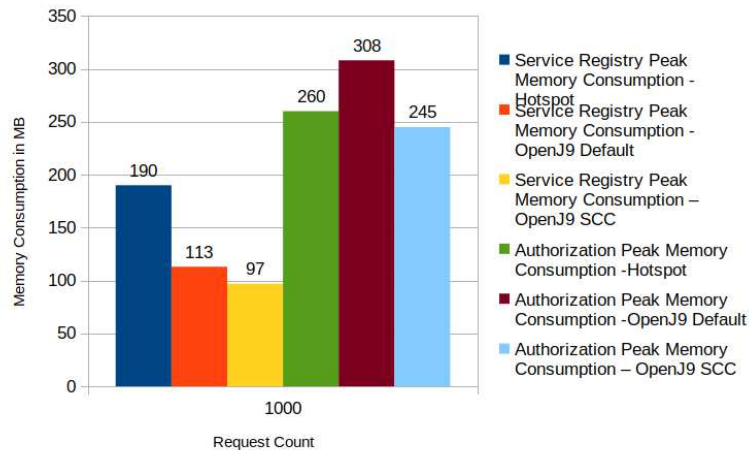


Figure 6.7: Authorization tests - Memory Consumption on different JVM configurations

of the Service Registry and Authorization services, we observe that the HotSpot also incurred the highest memory consumption with 450 MB. OpenJ9 with SCC delivered the lowest memory consumption with 343 MB, but it also displayed the worst performance with the highest mean response time (see Figure 6.5 and 6.7). As in Service Registry tests, we discuss that the HotSpot JVM is suitable for use cases where we expect a high traffic on the Authorization service and a high performance. The difference between the sum of memory consumption of the HotSpot JVM and OpenJ9 SCC is 107 MB, which may play a significant role in devices that are equipped with limited resources. OpenJ9 with SCC is particularly suitable for devices with limited resources and expected low traffic. OpenJ9 with default configurations delivered a total memory consumption of 421 MB, which is less than that of HotSpot, and higher response times than the HotSpot JVM. We cannot recommend OpenJ9 running with default configurations for most use cases, since the advantage of a smaller memory consumption is negligible.

## 6.3 Orchestrator

In this section, we present our results from our performance tests for the Orchestrator.

### 6.3.1 Orchestrator - Intra-cloud service discovery

In Figure 6.8, we present the results of our experiment on the Orchestrator intra-cloud search with the HotSpot JVM on the VM, where we sent 10,000, 1000, 500, and 100 requests simultaneously. On the same Figure, we also present the results of the tests with OpenJ9 with default configurations, and with OpenJ9 with shared classes cache (SCC), where we sent 10,000, 1000 and 100 requests. In our tests with the HotSpot JVM on the VM sending 10,000 simultaneous requests, we have a mean response time of 89.66



seconds, on 1000 simultaneous requests a mean response time of 16.16 seconds, on 500 simultaneous requests a mean response time of 5.6 seconds, and on 100 simultaneous requests a mean response time of 3.79 seconds. Our results reveal that on the VM, we can guarantee an acceptable response time that is under 10 seconds with <500 simultaneous requests. However, this number is highly dependent on the hardware configuration. In Figure 6.9, we present the results of our experiment on Raspberry Pi 4, where we have sent 100, 70, and 50 simultaneous requests. On the Raspberry Pi 4, on 100 simultaneous requests, we have a mean response time of 14.57 seconds. On 70 simultaneous requests, a mean response time of 13.28 seconds, and on 50 simultaneous requests, we have a mean response time of 10.47 seconds. On Raspberry Pi 4, we have an acceptable response time with only <50 simultaneous requests.

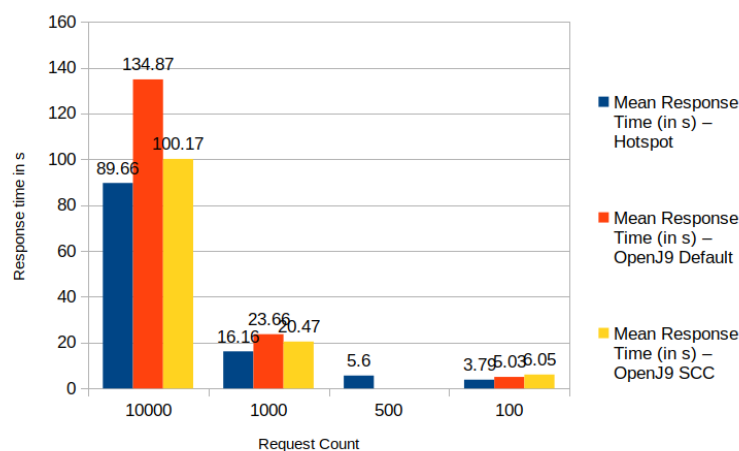


Figure 6.8: Orchestrator intra-cloud - Mean Response Time on different JVMs

We also compared the performance and memory consumption of different JVM configurations. In Figure 6.8 the response times, in Figure 6.10 the memory consumptions and in 6.11 the error counts are presented, where we sent 10,000, 1000, and 100 simultaneous requests. On 10,000 simultaneous requests, on the HotSpot JVM, we have a mean response time of 89.66 with 213 errors, on OpenJ9 with default configurations a mean response time of 134.87 seconds with 945 errors, and on OpenJ9 with enabled shared classes cache 100.17 seconds with 749 errors. On 1000 simultaneous requests, on the HotSpot JVM, we have a mean response time of 16.16 with 93 errors, on OpenJ9 with default configurations a mean response time of 23.66 seconds with 237 errors, and on OpenJ9 with enabled shared classes cache 20.47 seconds with 161 errors. On 100 simultaneous requests, on the HotSpot JVM, we have a mean response time of 3.79 seconds with no errors, on OpenJ9 with default configurations a mean response time of 5.03 seconds with no errors and on OpenJ9 with enabled shared classes cache 6.06 seconds with no errors. In our tests with the HotSpot JVM with 10,000 simultaneous requests, Service Registry had a memory consumption of 250 MB, Authorization component 230 MB, and Orchestrator 1000 MB.

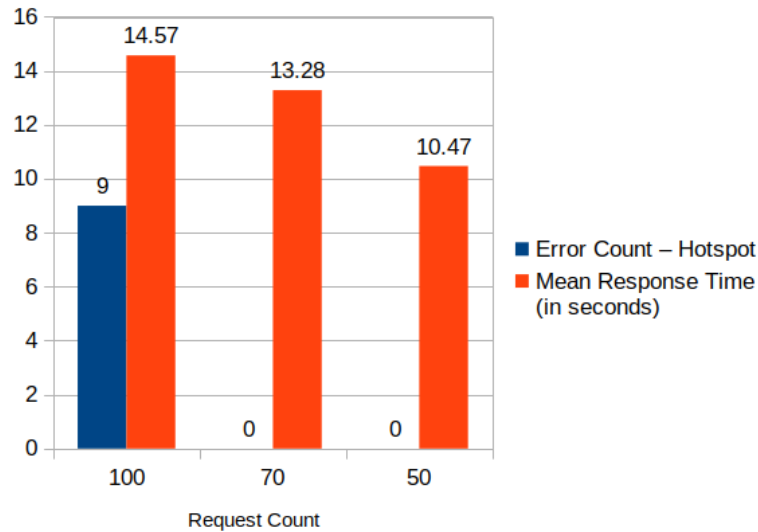


Figure 6.9: Orchestrator intra-cloud test results on Raspberry Pi

In tests with the HotSpot JVM with 1000 simultaneous requests, Service Registry had 270 MB memory consumption, Authorization 180 MB, and Orchestrator 260 MB. When we sent 100 simultaneous requests on the HotSpot JVM, Service Registry had 126 MB memory consumption, Authorization 219 MB, and Orchestrator 130 MB. In our tests with OpenJ9 JVM default with 10,000 simultaneous requests, Service Registry had a memory consumption of 160 MB, Authorization component 142 MB, and Orchestrator 1000 MB. In tests with OpenJ9 JVM default with 1000 simultaneous requests, Service Registry had 130 MB memory consumption, Authorization 150 MB, and Orchestrator 280 MB. When we sent 100 simultaneous requests on OpenJ9 JVM default, Service Registry had 140 MB memory consumption, Authorization 135 MB, and Orchestrator 137 MB. In our tests with OpenJ9 with enabled SCC with 10,000 simultaneous requests, Service Registry had a memory consumption of 160 MB, Authorization component 142 MB, and Orchestrator 1000 MB. In tests with OpenJ9 with enabled SCC with 1000 simultaneous requests, Service Registry had 130 MB memory consumption, Authorization 150 MB, and Orchestrator 280 MB. When we sent 100 simultaneous requests on OpenJ9 with enabled SCC, Service Registry had 140 MB memory consumption, Authorization 135 MB, and Orchestrator 137 MB.

In our test, the HotSpot JVM with default configurations yielded the best performance with the lowest mean response time and lowest error count. However, we observe that the HotSpot also incurred the highest memory consumption in all of our tests (see Figure 6.10). Even though OpenJ9 with default configurations had a higher memory consumption than OpenJ9 with SCC, it delivered a higher response time in most tests and a higher error count overall. OpenJ9 with SCC had the lowest memory consumption of all, and

its performance was better or comparable with OpenJ9 with default configuration (see Figures 6.10 and 6.8). As in previous tests, we discuss that the HotSpot JVM is suitable for use cases where we expect a high traffic on the Orchestrator intra-cloud service discovery and a high performance. The difference of the sum of memory consumption between the HotSpot JVM and OpenJ9 SCC goes up to 250 MB, which may play a significant role on devices with limited resources. OpenJ9 with SCC is particularly suitable for devices that are equipped with limited resources and expected low traffic. OpenJ9 with default configurations displayed the worst performance in most of the tests and had a higher memory consumption than OpenJ9 with SCC. We cannot recommend it for any use case.

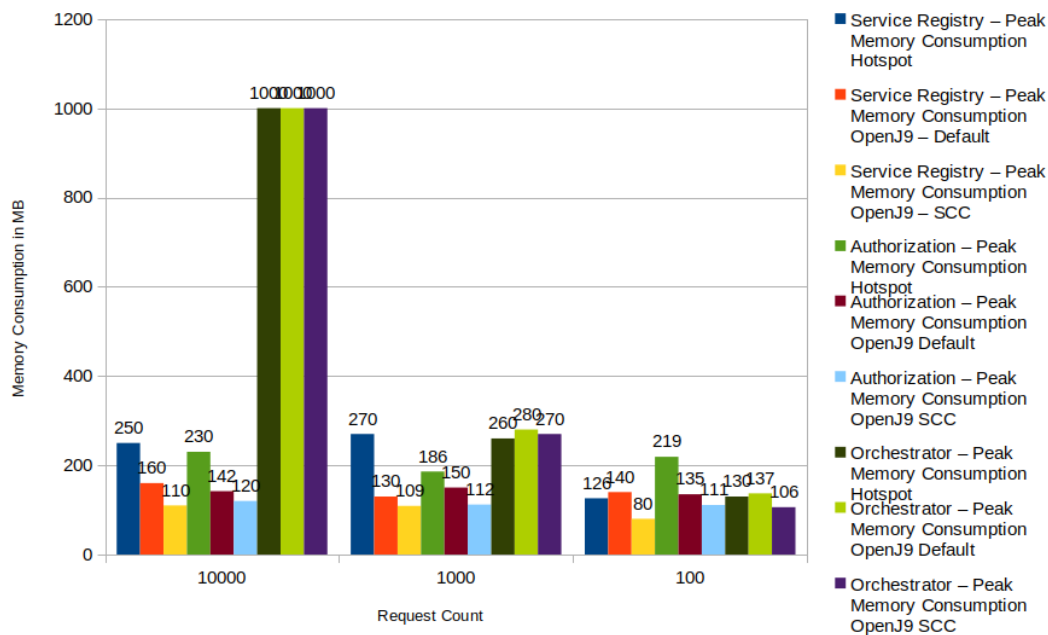


Figure 6.10: Orchestrator intra-cloud - Memory Consumption of different components on different JVMs

### 6.3.2 Orchestrator - Inter-cloud service discovery

As explained in Methodologies and Implementation, we have two types of inter-cloud tests.

#### Orchestrator - Inter-cloud 1 - Requests sent to VM

In Figure 6.12, we present the results of our experiment on the Orchestrator inter-cloud search with the HotSpot JVM on the VM, where we sent 1000, 100, and 40 requests simultaneously. In our tests with the HotSpot JVM on the VM with 1000

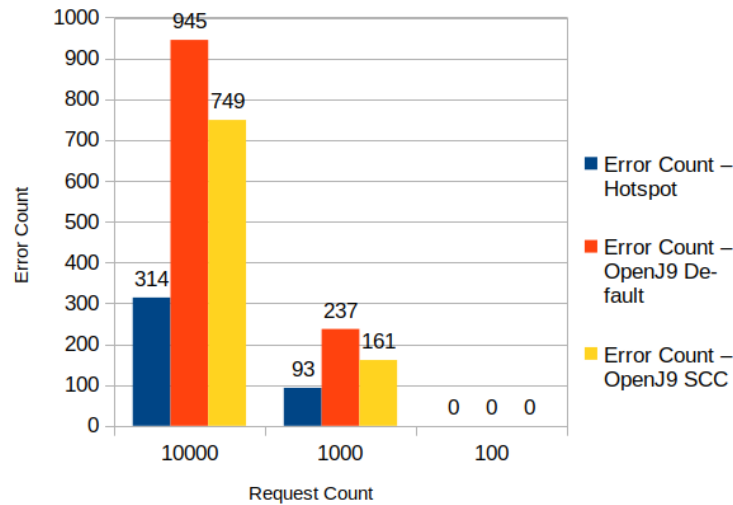


Figure 6.11: Orchestrator intra-cloud - Error count in tests

simultaneous requests we have a mean response time of 26.9 seconds with 591 errors, on 100 simultaneous requests a mean response time of 12.26 seconds with 50 errors, and on 40 simultaneous requests a mean response time of 7.08 seconds with no errors. Our results reveal that on the VM, we can guarantee an acceptable response time under 10 seconds with <40 simultaneous requests. However, this number is highly dependent on the hardware configuration. In Figure 6.13, we present the results of our experiment on Raspberry Pi 4, where we sent 40, 30, and 20 requests simultaneously. On the Raspberry Pi 4, on 40 simultaneous requests, we have a mean response time of 11.19 seconds with three errors, on 30 simultaneous requests a mean response time of 12.74 seconds with no errors. On 20 simultaneous requests, we have a mean response time of 9.05 seconds with no errors. On the Raspberry Pi 4, we have an acceptable response time with only <20 simultaneous requests (see Figure 6.13).

We have also compared the performance and memory consumption of different JVM configurations. In Figure 6.14 the response times and in Figure 6.15 the memory consumptions are presented, where we have sent 40 simultaneous requests. On 40 simultaneous requests, on the HotSpot JVM, we have a mean response time of 7.08, on OpenJ9 with default configurations a mean response time of 6.94 seconds, and on OpenJ9 with enabled shared classes cache 8.37 seconds. In our tests with the HotSpot JVM with 40 simultaneous requests, Service Registry had a memory consumption of 170 MB, Authorization component 138 MB, Orchestrator 190 MB, and Gatekeeper 160 MB. In our tests with OpenJ9 JVM default with 40 simultaneous requests, Service Registry had a memory consumption of 102 MB, Authorization component 72 MB, Orchestrator 121 MB, and Gatekeeper 122 MB. In our tests with OpenJ9 with enabled SCC with 40 simultaneous requests, Service Registry had a memory consumption of

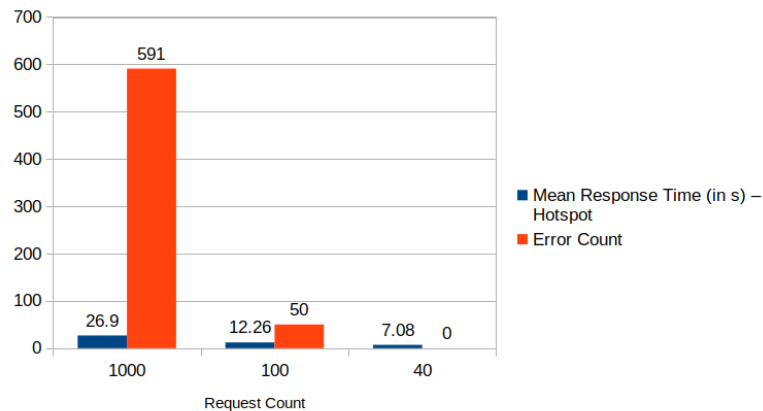


Figure 6.12: Orchestrator inter-cloud 1 - mean response times and error counts on x86 HotSpot

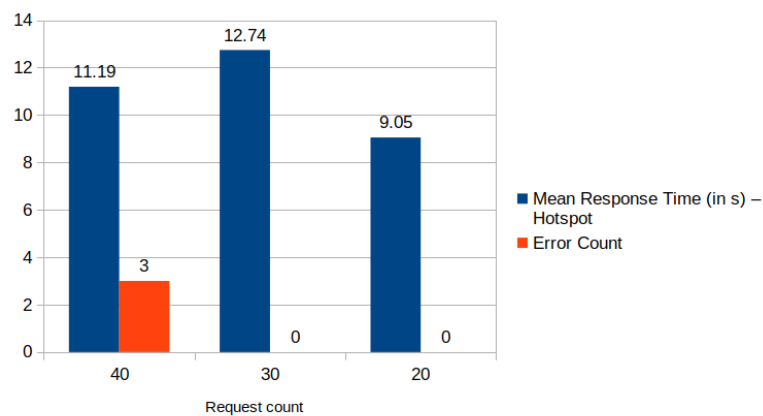


Figure 6.13: Orchestrator inter-cloud 1 - mean response times and error counts on Raspberry Pi

112 MB, Authorization component 99 MB, Orchestrator 100 MB, and Gatekeeper 96 MB. In our test, the HotSpot JVM with default configurations and OpenJ9 with default configurations yielded the best performance with the lowest mean response time and lowest error count (see Figure 6.14). But we observe that HotSpot also incurred the highest memory consumption on all our tests (see Figure 6.15). When we compare the sum of all the components' memory consumption, we observe that OpenJ9 default consumed about 417 MB memory, and OpenJ9 with SCC consumed about 407 MB memory. The difference is minimal, but once again we observe that OpenJ9 with SCC delivered the lowest memory consumption of all, and its performance was also the worst

with the highest mean response times. As in previous tests, we discuss that the HotSpot JVM is suitable for use cases where we expect a high traffic on the Orchestrator inter-cloud service discovery and a high performance. The difference of the sum of memory consumption between the HotSpot JVM and OpenJ9 SCC goes up to 251 MB, which may play a significant role on devices with limited resources. OpenJ9 with SCC is particularly suitable for devices with limited resources and expected low traffic. OpenJ9 with default configurations yielded a better performance than that of OpenJ9 SCC and a similar memory consumption with OpenJ9 SCC, making it also an option for devices with limited resources in this use case.

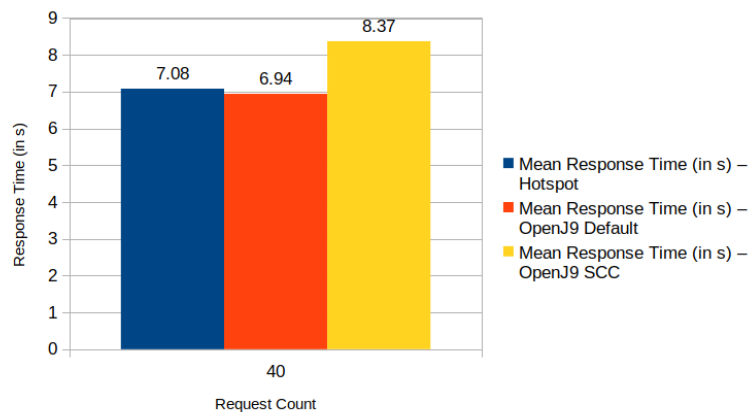


Figure 6.14: Orchestrator inter-cloud 1 - mean response time on different JVMs

### Orchestrator - Inter-cloud 2 - Requests sent to "dummy" Arrowhead Framework

In Figure 6.16, we present the results of our experiment on the Orchestrator inter-cloud search with the HotSpot JVM on the VM, where we sent 1000, 100, and 50 requests simultaneously. In our tests with the HotSpot JVM on the VM with 1000 simultaneous requests we have a mean response time of 37.26 seconds with 638 errors, on 100 simultaneous requests a mean response time of 11.17 seconds with 29 errors, and on 50 simultaneous requests a mean response time of 7.85 seconds with no errors. Our results reveal that on the VM, we can guarantee an acceptable response time that is under 10 seconds with <50 simultaneous requests (see Figure 6.16). However, this number is highly dependent on the hardware configuration. In Figure 6.17, we present the results of our experiment on the Raspberry Pi 4, where we sent 30, 20, and 10 requests simultaneously. On the Raspberry Pi 4, on 30 simultaneous requests we have a mean response time of 16.82 seconds with 3 errors, on 20 simultaneous requests a mean response time of 12.5 seconds with no error, and on 10 simultaneous requests we have a mean response time of 7.98 seconds with no error. On the Raspberry Pi 4, we have an acceptable response time with only <10 simultaneous requests.

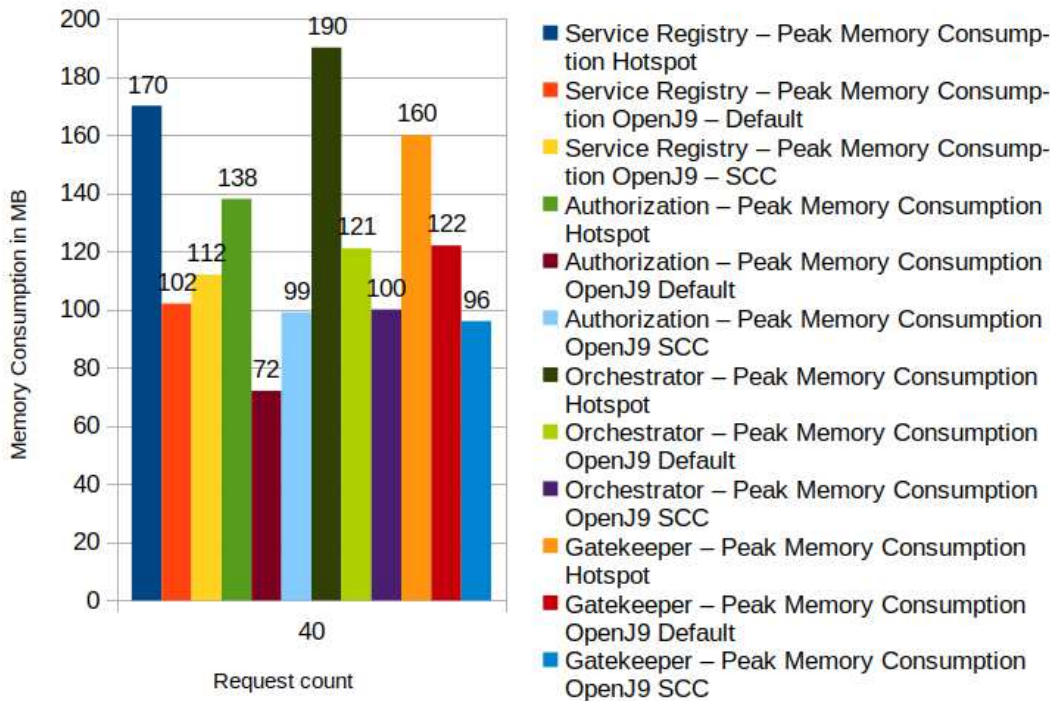


Figure 6.15: Orchestrator inter-cloud 1 - memory consumptions of different components on different JVMs

We also compared the performance and memory consumption of different JVM configurations. In Figure 6.18 the response times and in Figure 6.19 the memory consumptions are presented, where we sent 50 simultaneous requests. On 50 simultaneous requests, on the HotSpot JVM we have a mean response time of 7.85, on OpenJ9 with default configurations a mean response time of 8.72 seconds and on OpenJ9 with enabled shared classes cache 8.7 seconds. In our tests with the HotSpot JVM with 50 simultaneous requests, the Service Registry had a memory consumption of 203 MB, Authorization component 151 MB, Orchestrator 179 MB, and Gatekeeper 155 MB.. In our tests with OpenJ9 JVM default with 50 simultaneous requests, Service Registry had a memory consumption of 145 MB, Authorization component 130 MB, Orchestrator 130 MB, and Gatekeeper 105 MB. In our tests with OpenJ9 with enabled SCC with 50 simultaneous requests, Service Registry had a memory consumption of 93 MB, Authorization component 105 MB, Orchestrator 74 MB and Gatekeeper 93 MB. the HotSpot JVM with default configurations yielded the best performance with the lowest mean response time (see Figure 6.18). But we also observe that the HotSpot also incurred the highest memory consumption on all of our tests (see Figure 6.19). We see that OpenJ9 with SCC had the lowest memory consumption but also the worst performance. In our tests, it also produced three errors on 40 simultaneous requests where other JVM configurations

## 6. RESULTS AND DISCUSSION

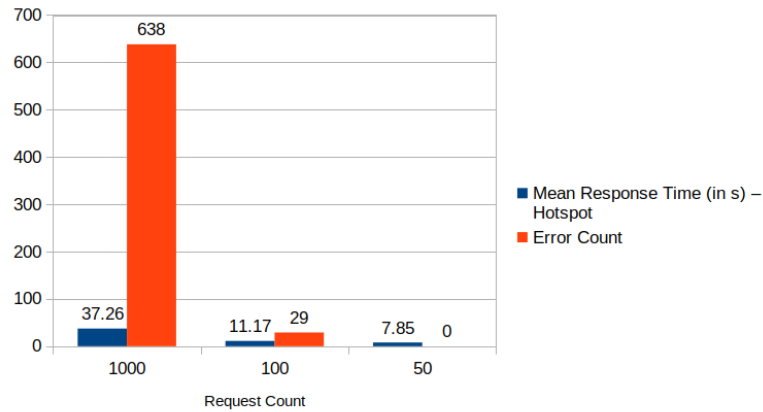


Figure 6.16: Orchestrator inter-cloud 2 - mean response times and error counts on x86 HotSpot

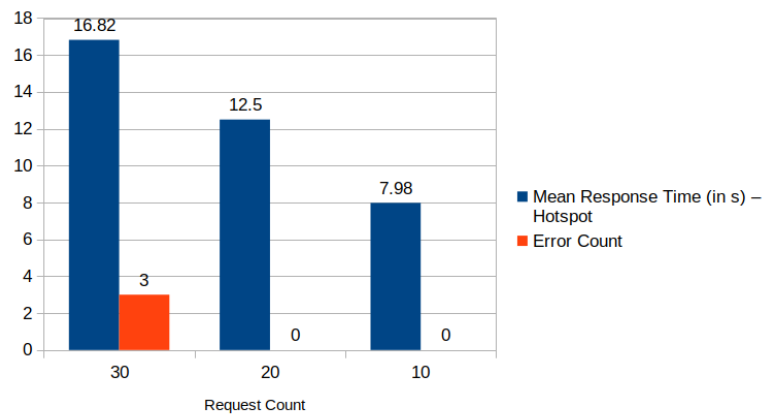


Figure 6.17: Orchestrator inter-cloud 2 - mean response times and error counts on Raspberry Pi

were able to handle the requests without problems. As in previous tests, we discuss that the HotSpot JVM is suitable for use cases where we expect a high traffic on the Orchestrator inter-cloud service discovery and a high performance. The difference of the sum of memory consumption between the HotSpot JVM and OpenJ9 SCC goes up to 323 MB, which may play a big role on devices that are equipped with limited resources. OpenJ9 with SCC is particularly suitable for devices with limited resources and expected low traffic. OpenJ9 with default configurations displayed a better performance than OpenJ9 SCC and a higher memory consumption than OpenJ9 SCC, which makes it an alternative for devices where the HotSpot JVM cannot be used due to limited resources



but also high performance is needed and OpenJ9 with SCC is not an option.

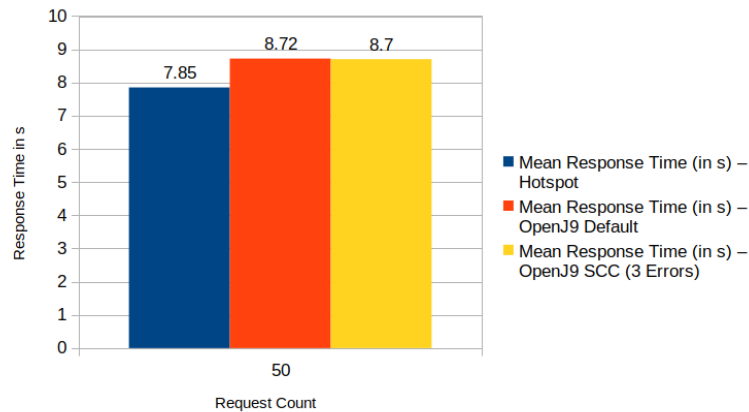


Figure 6.18: Orchestrator inter-cloud 2 - mean response time on different JVMs

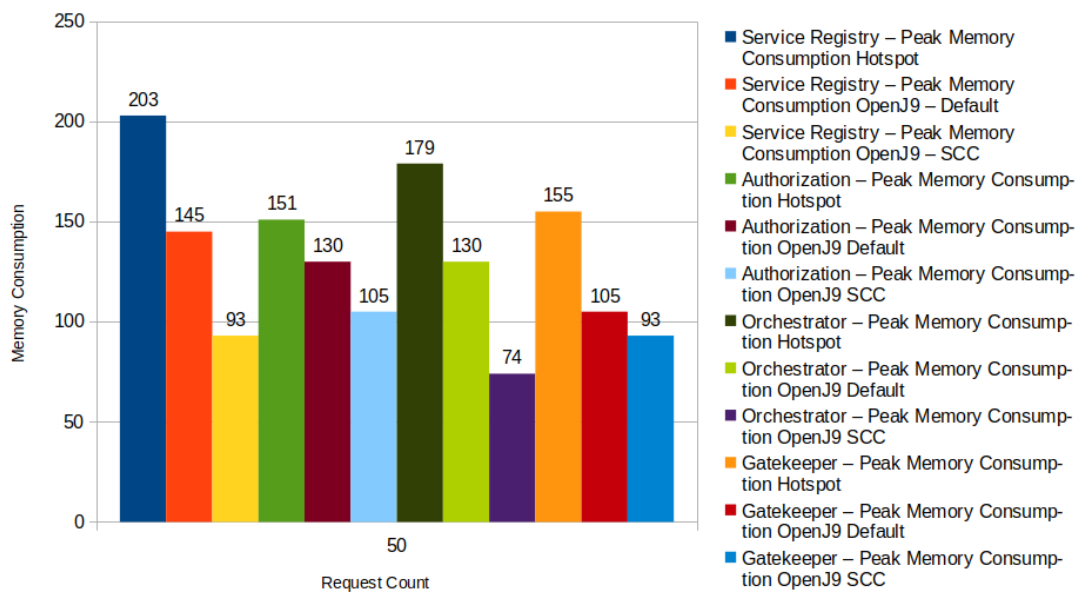


Figure 6.19: Orchestrator inter-cloud 2 - memory consumptions of different components on different JVMs

## 6.4 Secure mode vs. Insecure mode

We also compared the performance of some components in insecure mode. In Figure 6.20 the response times of Service Registry and Authorization are presented in secure and insecure mode, where we have sent 10,000 simultaneous requests. In our tests with 10,000 simultaneous requests, Service Registry had a mean response time of 41.14 seconds in secure mode and 19.57 seconds in insecure mode. The Authorization service had a mean response time of 44.3 seconds in secure mode and 23.11 seconds in insecure mode. As expected, the security mechanisms of the Arrowhead Framework have reduced the performance of the components massively (see Figure 6.20). The difference was partially more than 100% comparing the response times. Depending on the use case, users of the framework may consider running the framework in insecure mode to improve the performance.

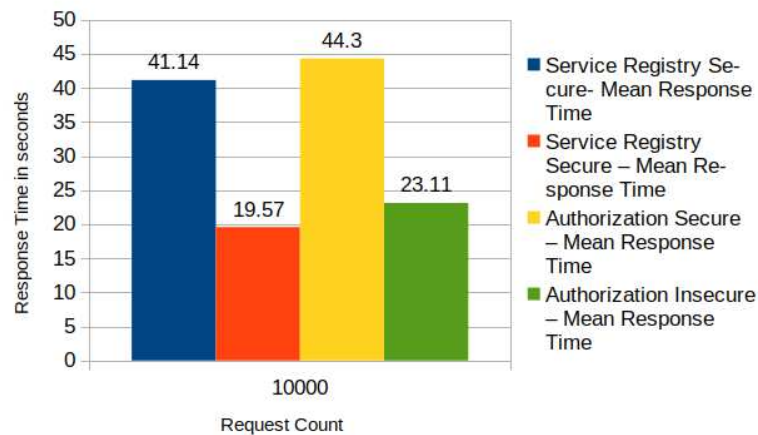


Figure 6.20: Service Registry and Authorization mean response times in secure and insecure mode

## 6.5 Startup Times

We compared the startup times of different components on different JVM configurations, as well. In Figure 6.21 the startup times of Service Registry, Authorization, Orchestrator and Gatekeeper are presented in different JVM configurations. Service Registry's startup time was 23.21 seconds on HotSpot, 24.8 seconds on OpenJ9 with default configurations, and 9.89 seconds on OpenJ9 with SCC. Authorization's startup time was 25.65 seconds on HotSpot, 25.36 seconds on OpenJ9 with default configuration and 10.63 seconds on OpenJ9 with SCC. Orchestrator's startup time was 24.76 seconds on HotSpot, 24.88 seconds on OpenJ9 with default configuration and 10.39 seconds on OpenJ9 with SCC. Gatekeeper's startup time was 25.11 seconds on HotSpot, 25.95 seconds on OpenJ9 with default configuration and 10.51 seconds on OpenJ9 with SCC. OpenJ9 with default

configurations and the HotSpot yielded very similar results. OpenJ9 with enabled SCC brought a significant decrease in startup times as expected (see Figure 6.21). Lower startup times are particularly beneficial in use cases where new instances of applications need to be started quickly when the demand increases. OpenJ9 with SCC increases the scalability of the framework by reducing the startup times.

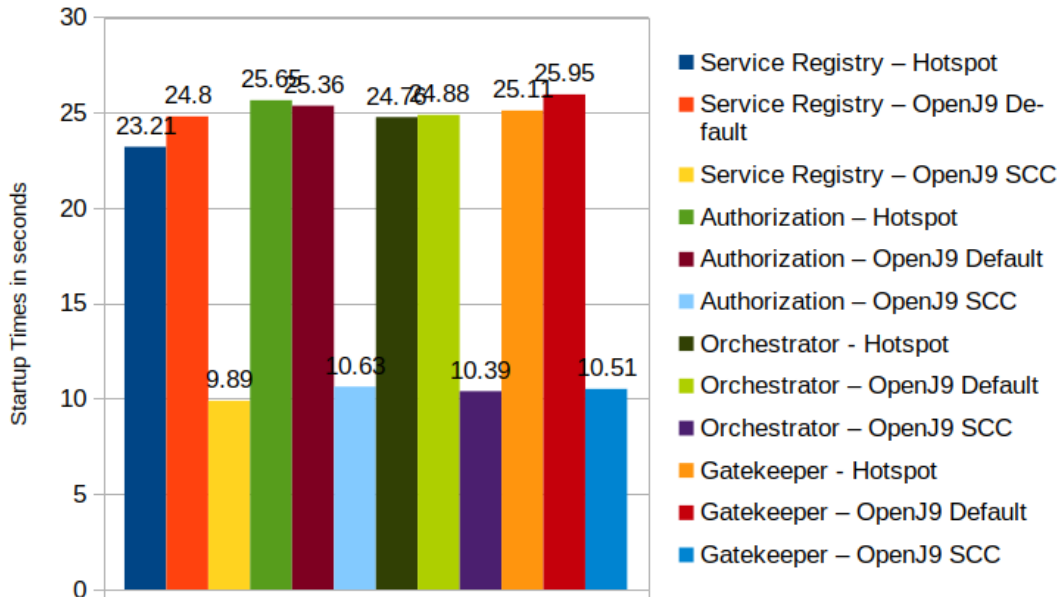


Figure 6.21: Startup Times of different components on different JVM configurations

## 6.6 Found Errors

We encountered some errors in our tests. In Service Registry, we found a concurrency issue that we spotted when we tried to simultaneously register two services with the same service interface. We reported the issue in the Github Repository of the Arrowhead Framework. The issue was fixed by the developers and integrated into the next release of Arrowhead Framework 4.2.0 [9].

We encountered some further problems when testing the startup times of the components with OpenJ9 with shared classes cache. The startup times were extremely longer every time we restarted the operating system. After discussing this issue with the developers on Github Repository of OpenJ9, we found out that OpenJ9 with SCC and enabled groupAccess, the cache is saved in the tmp folder of Ubuntu. Since Ubuntu deletes the contents of the /tmp folder on each restart, OpenJ9 would need to recreate the cache on every restart. Developers of OpenJ9 decided to update the documentation after our remark to address this possible issue when using groupAccess on Ubuntu [32].

We encountered other errors the majority of which were timeout issues, where some components did not respond on time.

### 6.7 Discussion

In our tests, we have seen that the Arrowhead Framework could cope with a high number of requests without crashing. The exception handling in the event of extreme high request count functioned properly. However, we have also observed that the number of simultaneous requests that a component can successfully handle, depends largely on the use case and the hardware configuration. The lower limit of requests may be as high as 1000 requests or as low as ten requests. As we expected, the intra-cloud and inter-cloud service discovery process is very resource-intensive and has the lowest lower limits in all our tested use cases. Developers using the Arrowhead Framework should consider the expected traffic when deciding on the hardware configuration. For expected high traffic with a high performance requirement, a modern x86 based processor with at least 8 GB of RAM is recommended. When only low traffic on the Framework is expected, ARM-based computers like the Raspberry Pi 4 may be eligible. Even in this case, a 4 GB RAM is recommended.

As a result of our comparison of JVMs, we can conclude that the HotSpot JVM yielded the best performance with the lowest mean response times in our tests. However, this high performance also resulted in the highest memory consumption among the test subjects.. OpenJ9 with default configurations mostly delivered y the worst performance with the highest mean response times, showing similar or less memory consumption than HotSpot. OpenJ9 SCC had the lowest memory consumption but also displayed higher mean response times than HotSpot. Developers using the Arrowhead Framework should deliberate the different performances and memory consumptions of different JVM configurations. For devices with expected low traffic without specific performance requirements, OpenJ9 SCC and hardware with limited resources may be used, which results in lower energy and hardware costs. For devices with high traffic, high performance requirements and better availability, HotSpot is the optimal solution.

We have seen that Open J9 SCC has a significant advantage regarding the start-up times of the applications. The lower start-up times with OpenJ9 SCC allow us to better scale up applications e.g., in the event of higher demand. This increases the scalability and availability of the applications. By better scalability, energy and hardware costs can also be better managed. However, this solution may only be eligible for use cases where the best performance is not required since OpenJ9 reduces the performance of the applications.

# CHAPTER 7

## Future Work

The Arrowhead Framework is still evolving. With their latest release 4.2.0, multiple components have been added to the framework. In future work, the performance of the new components should be evaluated. What is more, we did not test all the interfaces of the components that we have tested. In future work new testing scenarios should be defined in order to discover other possible bottlenecks and errors.

OpenJ9 has started developing a version of their JVM for 64-bit Linux on ARM and released an early access version with OpenJ9 0.20.0 [12]. In our tests, we have observed that OpenJ9 with shared classes cache is particularly interesting for devices that are equipped with limited resources thanks to the lower memory consumption. In future work it should be tested whether OpenJ9 can maintain its advantage on memory consumption also on architectures with AArch64, as well. OpenJ9 SCC also has many configuration options, such as the size of the cache. In future work, it should be tested whether other possible configurations such as a bigger cache size, have any impact on memory consumption and/or performance. There are other JVMs available on the market, such as GraalVM [17]. In future work, this product should also be tested and compared with HotSpot and OpenJ9.

HotSpot has also introduced a shared classes cache for application classes with JDK 10. We wish to test in future work whether the implementation of Oracle has any impact on the performance and/or memory consumption as with OpenJ9.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Conclusion

In this thesis, we have evaluated the performance of Service Registry, Authorization, Orchestrator, and Gatekeeper components of the Arrowhead Framework. We have studied and considered the mean response times, error counts, and peak memory consumptions of the components for the evaluation. We have also tested to find out if using an alternative JVM such as OpenJ9 has any impact on performance and memory consumption.

Our work is the first academic research that evaluates the Arrowhead Framework from a performance perspective. Our tests enabled us to determine the limits of the components by sending multiple simultaneous requests to them. We also detected a bug in Service Registry, which was consequently corrected and included in the following release.

With our results, we can offer recommendations for the users of the Arrowhead Framework. We have observed that in a device that is equipped with limited resources such as the Raspberry Pi 4, the limits for an acceptable performance can go as low as ten requests/sec.. Users of the framework should take this matter into account when using the Arrowhead Framework, and if high traffic is expected, they should consider installing hardware with a stronger performance. Another remarkable result was that we were able to reduce the memory consumption and startup times of the components without changing anything in the codebase by just using OpenJ9 with shared classes cache. However, this option is best for use cases, where we do not expect a high load on the device, since OpenJ9 with shared classes cache delivers a lower performance outcome than HotSpot.

We created a testing environment that can be made use of in future tests, which we have mentioned in Future Work chapter. Our results help users of the Arrowhead Framework decide on and choose the best hardware and software solutions in different use cases. This may save money and effort for stakeholders and improve the performance of the Framework. It also helps the developers of the Arrowhead Framework to further enhance

the performance of the framework in future releases.

## List of Figures

1.1	Some of the technologies associated with IoT. (copied from [56]) . . . . .	2
3.1	Mandatory core services with interactions enabling the service exchange between two application systems. (copied from [41]) . . . . .	13
3.2	Arrowhead Framework Core services deployed . . . . .	15
3.3	Inter-cloud communication over Gatekeeper (copied from [35]) . . . . .	15
3.4	Compilation process of a Java class file . . . . .	17
4.1	Testing Environment . . . . .	20
4.2	Testing Environment for the second version of Orchestrator tests . . . . .	23
5.1	Service Registration Activity Diagram (copied from [35]) . . . . .	29
5.2	Orchestrator tests Activity Diagram [35] . . . . .	34
6.1	Service Registry Mean Response Times with error count on the x86 HotSpot JVM . . . . .	40
6.2	Service Registry tests - Response Time on Raspberry Pi with the HotSpot JVM . . . . .	40
6.3	Service Registry tests - Memory Consumption on different JVM configurations . . . . .	42
6.4	Service Registry tests - Response Time on different JVM configurations . . . . .	42
6.5	Authorization tests - Mean Response Times - OpenJ9 tests were only done with 1000 requests . . . . .	43
6.6	Authorization tests - Mean Response Times with error count on Raspberry Pi . . . . .	43
6.7	Authorization tests - Memory Consumption on different JVM configurations . . . . .	44
6.8	Orchestrator intra-cloud - Mean Response Time on different JVMs . . . . .	45
6.9	Orchestrator intra-cloud test results on Raspberry Pi . . . . .	46
6.10	Orchestrator intra-cloud - Memory Consumption of different components on different JVMs . . . . .	47
6.11	Orchestrator intra-cloud - Error count in tests . . . . .	48
6.12	Orchestrator inter-cloud 1 - mean response times and error counts on x86 HotSpot . . . . .	49
6.13	Orchestrator inter-cloud 1 - mean response times and error counts on Raspberry Pi . . . . .	49
6.14	Orchestrator inter-cloud 1 - mean response time on different JVMs . . . . .	50
	60	



---

6.15	Orchestrator inter-cloud 1 - memory consumptions of different components on different JVMs . . . . .	51
6.16	Orchestrator inter-cloud 2 - mean response times and error counts on x86 HotSpot . . . . .	52
6.17	Orchestrator inter-cloud 2 - mean response times and error counts on Raspberry Pi . . . . .	52
6.18	Orchestrator inter-cloud 2 - mean response time on different JVMs . . . . .	53
6.19	Orchestrator inter-cloud 2 - memory consumptions of different components on different JVMs . . . . .	53
6.20	Service Registry and Authorization mean response times in secure and insecure mode . . . . .	54
6.21	Startup Times of different components on different JVM configurations . . . . .	55



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography

- [1] -Xshareclasses - Eclipse OpenJ9. <https://www.eclipse.org/openj9/docs/xshareclasses/>.
- [2] ActiveMQ. <https://activemq.apache.org/>.
- [3] AOT Compiler - OpenJ9. <https://www.eclipse.org/openj9/docs/aot/>.
- [4] Apache JMeter - Apache JMeter™. <https://jmeter.apache.org/>.
- [5] Apache Tomcat. <https://tomcat.apache.org/>.
- [6] The Arrowhead Framework vision and objective. <https://arrowhead.eu/arrowheadframework/why-how>.
- [7] AWS IoT - Amazon Web Services. <https://aws.amazon.com/iot/>.
- [8] Cl4cds. <https://simonis.github.io/cl4cds/>.
- [9] Concurrency issue when registering services · Issue #221 · arrowhead-f/core-java-spring. <https://github.com/arrowhead-f/core-java-spring/issues/221>.
- [10] Configuration – Gatling Open-Source Load Testing Documentation. <https://gatling.io/docs/current/general/configuration>.
- [11] Eclipse OpenJ9 - Class data sharing. <https://www.eclipse.org/openj9/docs/shrc/>.
- [12] Eclipse OpenJ9 0.20.0 Release Notes. <https://www.eclipse.org/openj9/docs/version0.20/#limited-support-for-64-bit-linux-on-arm>.
- [13] FIWARE. [www.fiware.org](http://www.fiware.org).
- [14] Frequently Asked Questions about Enhanced Intel SpeedStep® Technology... <https://www.intel.com/content/www/us/en/support/articles/000007073/processors.html>.
- [15] Gatling Open-Source Load Testing – For DevOps and CI/CD. <https://gatling.io/>.
- [16] Gatling vs. JMeter - DZone Performance. In *Dzone.Com*.
- [17] GraalVM. <https://www.graalvm.org/>.

- [18] Hotspot - Class Data Sharing. <https://docs.oracle.com/en/java/javase/11/vm/class-data-sharing.html#GUID-7EAA3411-8CF0-4D19-BD05-DF5E1780AA91>.
- [19] Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025. <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>.
- [20] IoT Hub | Microsoft Azure. <https://azure.microsoft.com/en-us/services/iot-hub/>.
- [21] Java HotSpot Virtual Machine Performance Enhancements. <https://docs.oracle.com/en/java/javase/14/vm/java-hotspot-virtual-machine-performance-enhancements.html#GUID-85BA7DE7-4AF9-47D9-BFCF-379230C66412>.
- [22] Java SE Core Technologies. <https://www.oracle.com/java/technologies/javase/javase-core-technologies-apic.html>.
- [23] JIT Compiler - OpenJ9. <https://www.eclipse.org/openj9/docs/jit/>.
- [24] MariaDB versus MySQL - Compatibility. <https://mariadb.com/kb/en/mariadb-vs-mysql-compatibility/>.
- [25] OpenJ9. <https://www.eclipse.org/openj9/>.
- [26] OpenJ9 - Performance. [https://www.eclipse.org/openj9/oj9\\_performance.html](https://www.eclipse.org/openj9/oj9_performance.html).
- [27] Raspberry Pi Documentation. <https://www.raspberrypi.org/documentation/hardware/raspberrypi/management.md>.
- [28] Siemens | MindSphere. <https://siemens.mindsphere.io/en>.
- [29] SiteWhere. <https://github.com/sitewhere/sitewhere>.
- [30] SmartThings. <https://www.smartthings.com/>.
- [31] Spring Boot. <https://spring.io/projects/spring-boot>.
- [32] Startup Times are significantly worse on OS startup with SCC · Issue #10087 · eclipse/openj9. <https://github.com/eclipse/openj9/issues/10087>.
- [33] Class sharing in Eclipse OpenJ9, June 2018.
- [34] Comparing Jenkins startup in Docker. <https://blog.openj9.org/2019/09/17/comparing-jenkins-startup-in-docker/>, September 2019.
- [35] Arrowhead Framework 4.1.3 Github. Arrowhead Consortia, July 2020.
- [36] Quarkus and Eclipse OpenJ9: Exceptional Performance across Platforms. <https://blog.openj9.org/2020/01/16/quarkus-and-eclipse-openj9-exceptional-performance-across-platforms/>, January 2020.

- [37] Watson IoT Platform. <https://www.ibm.com/internet-of-things/solutions/iot-platform/watson-iot-platform>, June 2020.
- [38] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464, December 2015.
- [39] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376, 24.
- [40] Mauro A.A. da Cruz, Joel J.P.C. Rodrigues, Arun Kumar Sangaiah, Jalal Al-Muhtadi, and Valery Korotaev. Performance evaluation of IoT middleware. *Journal of Network and Computer Applications*, 109:53–65, May 2018.
- [41] Jerker Delsing. *IoT Automation: Arrowhead Framework*. CRC Press, February 2017.
- [42] John Esquiagola, Laisa Costa, Pablo Calcina, Geovane Fedrecheski, and Marcelo Zuffo. Performance Testing of an Internet of Things Platform:. In *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, pages 309–314, Porto, Portugal, 2017. SCITEPRESS - Science and Technology Publications.
- [43] Jasmin Alexandra Guth. *Architectural Design of an Abstraction Layer for the Integration of Heterogeneous Cyber-Physical Systems*. PhD thesis, University of Stuttgart.
- [44] Eclipse Foundation Inc. Eclipse IoT Developer Survey Results. <https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf>.
- [45] Felix Larrinaga Barrenechea, Iñigo Aldalur Ceberio, Miren Illarramendi Reza-bal, Mikel Iturbe Urretxa, Txema Perez Lazare, Gorka Unamuno Eguren, Jon Salvidea Campuzano, and Inaxio Lazkanoiturburu. ANALYSIS OF TECHNOLOGICAL ARCHITECTURES FOR THE NEW PARADIGM OF THE INDUSTRY 4.0. *DYNA INGENIERIA E INDUSTRIA*, 94(1):267–271, 2019.
- [46] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, May 2014.
- [47] Daniel Macedo, Luiz Affonso Guedes, and Ivanovitch Silva. A dependability evaluation for Internet of Things incorporating redundancy aspects. In *Proceedings of the 11th IEEE International Conference on Networking, Sensing and Control*, pages 417–422, Miami, FL, USA, April 2014. IEEE.

- [48] Roberto Morabito, Ivan Farris, Antonio Iera, and Tarik Taleb. Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge. *IEEE Internet of Things Journal*, 4(4):1019–1030, August 2017.
- [49] Subbiah Muthiah, Ramakrishnan Venkatasubramanian, and Cognizant Technology Solutions. The Internet of Things: QA Unleashed. page 6.
- [50] Jakob Nielsen. *Usability Engineering*. Interactive Technologies. AP Professional, Cambridge, Mass., 1993.
- [51] Victor Estuardo Araujo Soto. *PERFORMANCE EVALUATION OF SCALABLE AND DISTRIBUTED IOT PLATFORMS FOR SMART REGIONS*. PhD thesis, Luleå University of Technology.
- [52] Frédéric Tounquet and Clément Alaton. Benchmarking smart metering deployment in the EU-28. Technical report.
- [53] Rob van Kranenburg. *The Internet of Things: A Critique of Ambient Technology and the All-Seeing Network of RFID*. Institute of Network Cultures, Amsterdam, 2008.
- [54] Konstantinos Vandikas and Vlasios Tsiatsis. Performance Evaluation of an IoT Platform. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 141–146, September 2014.
- [55] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. AOT vs. JIT: Impact of profile data on code quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems - LCTES 2017*, pages 1–10, Barcelona, Spain, 2017. ACM Press.
- [56] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, November 2014.