FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Evaluating and Improving the Performance of the Arrowhead Framework

Rafael Teles da Rocha

Mestrado em Engenharia de Software

Supervisor: Pedro Souto

Second Supervisor: Luís Lino Ferreira

June 26, 2019

© Rafael Teles da Rocha, 2019

Evaluating and Improving the Performance of the Arrowhead Framework

Rafael Teles da Rocha

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

Chair: Prof. Nuno Flores
External Examiner: Prof. Ricardo Severino
Supervisor: Prof. Pedro Souto

June 26, 2019

Abstract

Industry 4.0 is currently understood in general terms as the "digitization" of a business' infrastructure. Most industries recognize the greatest challenges posed by Industry 4.0 for the future, but only a small portion of them feel prepared. However, due to Industry 4.0's promising benefits in terms of economic growth, world governments are actively involved in projects attempting to improve industry digitization. Consequently, this dissertation analyzes a work-in-progress software framework for Industry 4.0: the Arrowhead Framework. This framework was the main result of an European research project of the same name, and focuses on enabling interoperability between different industrial systems. However, the goal of this dissertation evolved throughout the course of the work.

At first, the objective was to evaluate and model the performance of the Arrowhead Framework, by testing its main use cases in certain stress scenarios. Indeed, by using stochastic Petri nets, the author was able to propose a performance model of the framework's intracloud and intercloud orchestration process. Through this model, it was possible to not only estimate the average response time for an orchestration request in both intra- and inter-cloud versions, but also to estimate the probability distribution of the Petri net being in a specific response state. For the Intracloud process, there was a 25% difference between the estimation and the actual result. As for the Intercloud process, there was a 37% difference.

Through this performance analysis, the author was also able to identify that the framework, and by extension its systems, had some potential performance setbacks, mostly related to how these were handling HTTP requests. In fact, because the performance results were so poor for one of the framework's systems (i.e., the Event Handler), it was decided to redesign it and change its implementation accordingly to improve its performance, by using appropriate software configurations and design patterns. The Event Handler (a message broker built over REST/HTTP), is a service whose performance is very important in most Arrowhead deployments.

Thus, by changing how the original Event Handler and its clients handled HTTP requests and thread creation, the enhanced version of the Event Handler is now able to achieve much higher levels of performance, evolving from an average latency of 666.3 ms to 8.95 ms. Actually, considering the average latency of both versions for the same test scenario, the Event Handler had an overall performance boost of over 98%. Similar modifications can be applied to other components of the Arrowhead Framework to improve their performance. As such, this reengineering process served as a case study in order to explore some possible performance improvements for the framework's other systems in the future.

Moreover, the author also proposed a Petri net model for the Event Handler in order to depict the performance impact of different thread pool configurations and CPU core availability. By employing a stochastic analysis on this Petri net, the goal was to then be able to predict the system's performance in order to guarantee the required quality of service. Regarding the model's

estimations, there was a 5.16% difference between the average latency and the estimated latency.

Keywords: Industry 4.0, Performance, Publish-Subscribe, HTTP, REST, Java, Petri Nets

Resumo

A Indústria 4.0 é atualmente compreendida em termos gerais como a “digitalização” da infraestrutura de um negócio. A maioria das indústrias reconhece os maiores desafios colocados pela Indústria 4.0 para o futuro, mas apenas uma pequena parte delas se sente preparada para o mesmo. No entanto, devido aos benefícios promissores da Indústria 4.0 em termos de crescimento económico, os governos mundiais encontram-se ativamente envolvidos em projetos que tentam melhorar a digitalização da indústria. Deste modo, esta dissertação analisa uma *software framework* (um trabalho em progresso) para a Indústria 4.0: o *Arrowhead Framework*. Esta *framework* foi o principal resultado de um projeto de investigação europeu com o mesmo nome e concentra-se em proporcionar a interoperabilidade entre diferentes sistemas industriais. No entanto, o objetivo desta dissertação evoluiu ao longo do curso do trabalho.

Inicialmente, o objetivo consistia em estabelecer um modelo de performance para a arquitetura do *Arrowhead Framework*, testando os seus principais casos de uso em certos cenários de grande carga de pedidos. De facto, usando redes de Petri estocásticas, o autor foi capaz de propor um modelo de performance dos processos de orquestração intracloud e intercloud da *framework*. Através deste modelo, foi possível estimar o tempo médio necessário para receber uma resposta de orquestração de uma *cloud Arrowhead* e também estimar a distribuição probabilística para a rede de Petri estar num estado específico. Para o processo *intracloud*, houve uma diferença de 25% entre a estimativa e o resultado real. Quanto ao processo *intercloud*, houve uma diferença de 37%.

Através desta análise de performance, o autor também foi capaz de identificar que a *framework*, e consequentemente os seus sistemas, apresentavam alguns potenciais problemas de performance, principalmente relacionados ao modo como estes lidavam com pedidos HTTP. Efetivamente, dado que os resultados de performance foram tão pobres para um dos sistemas da framework (i.e., o sistema Event Handler), foi decidido reformulá-lo e alterar a sua implementação de modo a melhorar o seu desempenho, usando configurações de *software* e *design patterns* apropriados. O Event Handler (um *message broker* construído sobre REST/HTTP) é um serviço cuja performance é muito importante na maioria das implementações do Arrowhead.

Assim, alterando a forma como o *Event Handler* original e os seus clientes manipulavam pedidos HTTP e criavam threads, a versão melhorada do *Event Handler* é agora capaz de atingir níveis muito mais altos de desempenho, evoluindo de uma latência média de 666,3 ms para 8,95 ms. De facto, considerando a latência média de ambas as versões para o mesmo cenário de teste, o *Event Handler* teve um aumento geral de performance acima de 98%. Modificações semelhantes podem ser aplicadas a outros componentes do Arrowhead Framework para melhorar o seu desempenho. Deste modo, este processo de reengenharia serviu como um caso de estudo para explorar algumas futuras melhorias de performance possíveis para os outros sistemas da framework.

Além disso, o autor também propôs um modelo de rede de Petri para o *Event Handler*, a fim de descrever o impacto de diferentes configurações de *threadpools* e disponibilidade de *CPU cores* na performance do sistema. Ao realizar uma análise estocástica nesta rede de Petri, o objetivo é

prever a performance do sistema para garantir a qualidade de serviço necessária. Em relação às estimativas do modelo, houve uma diferença de 5,16% entre a latência média e a latência estimada.

Keywords: Industry 4.0, Performance, Publish-Subscribe, HTTP, REST, Java, Petri Nets

Acknowledgements

First, I would like to thank my family for always supporting me and turning me into the person I am today. Without your help and love, I would not have reached this far into my journey.

I also want to thank my supervisor Prof. Pedro Souto, my co-supervisor Prof. Luís Lino Ferreira, and CISTER colleague Prof. Cláudio Maia for guiding and assisting me along this dissertation's development. It was a wonderful learning experience that helped me become a more professional and pragmatic engineer. Furthermore, I have to thank CISTER and everyone at FEUP for giving me the tools and opportunity to pursue my goals.

And last, but certainly not least, I want to thank all my friends and colleagues at CISTER for all the wonderful moments we shared together. Thank you for working with me, sharing your knowledge with me, making me laugh or helping me out when I most needed it. I shall always cherish those moments.

Rafael Rocha

To my parents, for all the love and support they have given me.

Rafael Rocha

Contents

1 Introduction 1
1.1 Motivation . . . 2
1.2 Research Goals . . . 2
1.3 Dissertation Structure . . . 2

2 Background 5
2.1 Demystifying Industry 4.0 . . . 5
2.1.1 Internet of Things . . . 6
2.1.2 Cyber-Physical Systems . . . 7
2.1.3 Big Data Analytics . . . 8
2.1.4 Cloud Computing . . . 8
2.2 Transitioning from Monoliths to Microservices, through REST . . . 9
2.3 The Arrowhead Framework . . . 10
2.3.1 Service Registry (Mandatory) . . . 12
2.3.2 Authorization System (Mandatory) . . . 12
2.3.3 Orchestration System (Mandatory) . . . 13
2.3.4 Event Handler . . . 13

3 Problem Statement 15

4 State of the Art 17
4.1 Message-Oriented protocols when REST is inadequate . . . 17
4.1.1 Advanced Message Queuing Protocol (AMQP) . . . 18
4.1.2 Message Queuing Telemetry Transport (MQTT) . . . 20
4.1.3 Simple Text Oriented Messaging Protocol (STOMP) . . . 21
4.1.4 Extensible Messaging Presence Protocol (XMPP) . . . 21
4.1.5 Data Distribution Service (DDS) . . . 22
4.1.6 Summarizing all mentioned messaging protocols . . . 22
4.2 Using Petri Nets for performance modeling . . . 23

5 Performance Evaluation of the Arrowhead Framework 27
5.1 Implementation of an Arrowhead system . . . 27
5.2 Intracloud Orchestration . . . 30
5.3 Intercloud Orchestration . . . 33

6 Modeling the performance of Arrowhead's Intracloud and Intercloud orchestration 37
6.1 Intracloud Orchestration . . . 41
6.1.1 Explaining the Petri net model for the Intracloud Orchestration . . . 41
6.1.2 Stochastic analysis of the Petri net model for the Intracloud orchestration 43

6.2 Intercloud Orchestration 45
6.2.1 Explaining the Petri net model for the Intercloud Orchestration 45
6.2.2 Stochastic analysis of the Petri net model for the Intercloud orchestration 47

7 Improving the performance of the Event Handler 51
7.1 The Event Handler 51
7.2 Original Implementation 52
7.3 Enhancements 52
7.3.1 Reuse open connections between the Publisher and the Event Handler . . 53
7.3.2 Establish a persistent connection between the Event Handler and each Subscriber 54
7.3.3 Reuse previously created threads in the Event Handler 58
7.4 Experimental setup 59
7.5 Performance evaluation of original version 59
7.6 Performance evaluation of enhanced version 60
7.6.1 Test Scenario A: 1 Publisher, 1 Subscriber, 2000 events 60
7.6.2 Test Scenario B: 1 Publisher, 1-6 Subscribers, 9000 events 61
7.6.3 Test Scenario C: 10 Publishers, 10 Subscribers, 10.000 events in total . . 62
7.6.4 Test Scenario D: 1 Publisher and 7 Subscribers (on same machine), different threadpool sizes in Event Handler 63
7.6.5 Test Scenario E: 1 Publisher and 7 Subscribers (each on a Raspberry Pi 1), different threadpool sizes in Event Handler 64

8 Modelling the Event Handler's performance 67
8.1 Explaining the Petri net model 69
8.2 The Petri net model 70
8.2.1 Comparing the model with the actual experiments 72
8.2.2 Interpreting the analysis results 73

9 Conclusions and Future Work 75
9.1 Results from the Dissertation 75
9.2 Additional Contributions 76
9.3 Further Work 76

A Workshop Demo at ISORC 2019 77

B Workshop Poster at ISORC 2019 83

C Accepted paper for IECON 2019 85

References 93

List of Figures

2.1 The four industrial revolutions [90] . . . 6
2.2 Monoliths vs. Microservices [62] . . . 10
2.3 Interconnected local collaborative clouds [80] . . . 11
2.4 The Arrowhead Core Systems [18] . . . 11
2.5 Overview of the Arrowhead Framework's mandatory systems [10] . . . 12
2.6 Overview of the Orchestration process [18] . . . 13
2.7 Overview of the Event Handler [80] . . . 14

4.1 Topic Exchange between Publishers and Subscribers in AMQP [35] . . . 19
4.2 Topic Subscription in MQTT [12] . . . 20
4.3 Examples of Petri Nets [17] . . . 24
4.4 Associating a probability density function of a delay which has an Erlang distribution. . . . 24
4.5 A transition carrying an enabling function. . . . 25
4.6 An inhibitor arc from "Place 2" to "transition". . . . 25

5.1 Sequence diagram of the Intracloud orchestration . . . 30
5.2 Testing environment for the intracloud orchestration . . . 31
5.3 Intracloud orchestration latency . . . 32
5.4 Frequency distribution of database query latency for Intracloud orchestration . . . 33
5.5 Sequence diagram of the Intercloud orchestration . . . 34
5.6 Testing environment for the intercloud orchestration . . . 35
5.7 Intercloud orchestration latency . . . 36

6.1 Example of the decision process for choosing an Erlang probability distribution, based on the data distribution, for a stochastic transition. . . . 39
6.2 Example of the decision process for choosing an Exponential probability distribution, based on the data distribution, for a stochastic transition. . . . 40
6.3 Stochastic Petri net model of Arrowhead's Intracloud Orchestration . . . 41
6.4 Sequence diagram of the Intracloud orchestration with annotations identifying the transitions used in the Petri net model . . . 42
6.5 Transient analysis of the Petri net model for the Intracloud orchestration . . . 44
6.6 Stochastic Petri net model of Arrowhead's Intercloud Orchestration . . . 45
6.7 Sequence diagram of the Intercloud orchestration with annotations identifying the transitions used in the Petri net model . . . 46
6.8 Transient analysis of the Petri net model for the Intercloud orchestration . . . 49

7.1 A simplified representation of the Event Handler system . . . 52
7.2 Testing environment for the official Event Handler . . . 59

7.3 End-to-end latency, for each of the two thousand messages sent, with the official Event Handler 60
7.4 End-to-end latency comparison of the two versions of the Event Handler 61
7.5 End-to-end latency distribution of 9000 events for one subscriber and six subscribers, with the enhanced Event Handler. 62
7.6 End-to-end latency distribution of 1000 events from 10 Publishers (each) to 10 Subscribers, with the enhanced Event Handler. 63
7.7 End-to-end latency of 3000 events from one publisher to seven subscribers (running on the same machine), with different thread pool sizes on the enhanced Event Handler (running on a Raspberry Pi 3 Model B). 64
7.8 End-to-end latency distribution from one publisher to 7 subscribers (each running on a Raspberry Pi 1), with different thread pool sizes on the enhanced Event Handler (running on a Raspberry Pi 3 Model B). 65

8.1 Stochastic Petri net model of the Event Handler running on a quad-core CPU . . 68
8.2 Transient analysis of the Petri net model 71
8.3 Average distribution of the estimated end-to-end latency, with four messages . . . 72

List of Tables

4.1 A comparison between all mentioned message protocols 23

7.1 Performance comparison between all tested threadpool sizes, for one publisher and 7 subscribers running on the same machine and the Event Handler running on a Raspberry Pi 3 Model B 64

7.2 Performance comparison between all tested threadpool sizes, for one publisher and 7 subscribers (each running on a Raspberry Pi 1) and the Event Handler running on a Raspberry Pi 3 Model B 65

Abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
DDS	Data Distribution Service
HTTP	Hypertext Transfer Protocol
IIoT	Industrial Internet of Things
IoT	Internet of Things
M2M	Machine-to-Machine
MQTT	Message Queuing Telemetry Transport
QoS	Quality of Service
REST	REpresentational State Transfer
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSE	Server-Sent Events
SOAP	Simple Object Access Protocol
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
VPN	Virtual Private Network
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
WP	Work Package

Chapter 1

Introduction

Every manufacturing company, be it large or small, is under constant pressure from its customers for its products to have better quality and lower cost. The fourth industrial revolution, Industry 4.0, aims to enable manufacturers to deliver these customer requirements by employing a digital, software-based framework to their factories. Effectively, this initiative is based on the concept of a fully-integrated industry, where the factory of the future is depicted as follows:

> “Products and services are flexibly connected via the internet or other network applications like the blockchain (consistent connectivity and computerization). The digital connectivity enables an automated and self-optimized production of goods and services including the delivering without human interventions ... The value networks are controlled decentralized while system elements ... are making autonomous decisions” [32]

Currently, Industry 4.0 is understood in general terms as the “digitization” of a business’ infrastructure, however there are still no reliable standards or definitions applied to what remains an arguably vague concept for businesses worldwide. As a result, most industries recognize the greatest challenges posed by Industry 4.0 for the future, but only a small portion of them feel prepared [15]. However, due to Industry 4.0’s promising benefits in terms of economic growth, world governments are actively involved in projects attempting to improve industry digitization [31].

Consequently, this dissertation analyzes a work-in-progress software framework for Industry 4.0, the Arrowhead Framework [10], which was the main result of an European research project of the same name, that focuses on enabling interoperability between different industrial systems. However, the focus of this thesis evolved throughout the course of the work. While at first the goal was to establish a performance model for the framework’s architecture (by testing it in certain stress situations), the opportunity arose to apply a reengineering process on the framework’s

message broker (by using appropriate software configurations and optimal design patterns) to significantly improve its performance. Ultimately, this system refactoring served as a case study in order to explore some possible performance improvements for the framework's other components.

1.1 Motivation

This dissertation's work is part of the European research project Productive4.0 [1], which is currently developing the Arrowhead Framework [10]. At present, the framework had not yet been subject of a performance evaluation and, as such, there was no perceived sense of how long the framework would take to perform certain operations, i.e., orchestration response times, average latency for message forwarding, which systems take longer to respond, and so forth. In fact, this necessity proved to be an interesting challenge to be tackled by this dissertation. Originally, the goal of the thesis was to evaluate and model the performance of the Arrowhead Framework (which was still completed), but because the performance results were so poor for one of the framework's systems, it was decided to redesign this system and change its implementation accordingly to improve its performance, by using appropriate software configurations and design patterns.

1.2 Research Goals

Given the already great amount of past and current smart manufacturing approaches and papers about them, the purpose of this dissertation is not to introduce yet another new approach to designing a software architecture for Industry 4.0. Instead, its goal is to break down a framework for Industry 4.0, describe its components along with their activities, connections, and interactions, while also analyzing how its architecture handles different stress situations, and to ultimately develop a performance model for it. Additionally, this analysis is also meant to provide critical feedback and enhancements, where design and implementation decisions that might lead to potential performance bottlenecks are detected and improved upon. In summary, this dissertation aims at answering the following research questions:

1. *How can the performance of a system or framework be modeled?*
2. *How can the correct use of design patterns and best practices affect a system's performance?*

1.3 Dissertation Structure

This dissertation is organized into nine main chapters. Chapter 2, *Background*, focuses on contextualizing the problem that this dissertation is trying to solve and its core technological pillars, detailing the technologies that will be analyzed in detail in this dissertation, among other important background information. Chapter 3, *Problem Statement*, goes into detail about the problem that this dissertation attempts to solve. Chapter 4, *State of the Art*, analyzes other projects, approaches,

technologies, or overall solutions that have tackled the same issue or were important/influential for this dissertation's development. Chapter 5, *Performance Evaluation of the Arrowhead Framework*, gives an overview of the implementation of an Arrowhead service, and evaluates the performance of the Arrowhead Framework in two of its main use cases: intracloud and intercloud service orchestration. Chapter 6, *Modeling the performance of Arrowhead's Intracloud and Intercloud orchestration*, explains the performance models of the Arrowhead framework for the two use cases whose performance was evaluated in Chapter 5. Furthermore, it validates this model by comparing its results with the results obtained in Chapter 5. Chapter 7, *Improving the performance of the Event Handler*, explains which system from the Arrowhead framework (the Event Handler) served as a case study to demonstrate possible performance improvements that could be applied to the other systems in the framework. Afterwards, it identifies the several performance bottlenecks in the Event Handler, explains the solutions for these problems and shows their implementation. Furthermore, this chapter also employs a performance evaluation on the original version of the Event Handler and also on its new implementation, for comparison. Chapter 8, *Modeling the Event Handler's performance*, similarly to Chapter 6, presents and explains the performance model developed for the Event Handler, and validates it against the experimental results presented in the previous chapter. Chapter 9, *Conclusions and Future Work*, summarizes and discusses the main results/contributions of this thesis.

Chapter 2

Background

In order to understand the problem statement, there first needs to be a clear understanding of the context. Therefore, this chapter explains the background context of the themes tackled in this dissertation by describing what Industry 4.0 is (Section 2.1) and its core technologies (Sections 2.1.1, 2.1.2, 2.1.3, and 2.1.4). Section 2.2 focuses on the implications of transitioning from a centralized system to a decentralized one, and what that means in terms of software architectures. Finally, Section 2.3 gives an overview of the Arrowhead Framework – a framework meant to assist the development of systems for Industry 4.0, and is also the main focus of this dissertation.

2.1 Demystifying Industry 4.0

The name "Industry 4.0" refers to the fourth industrial revolution – a new method to achieve results that were impossible a decade ago, thanks to the evolution of technology [68]. In fact, each revolution brought gargantuan changes to both the industry and society itself (see Figure 2.1): in the 19th Century, the first industrial revolution consisted in moving from farming to factory production; from the 1850s to World War I, the second revolution introduced steel, which lead to the early electrification of factories and the launch of mass production; from the 1950s to the 1970s, the third industrial revolution brought the migration from analogue technology to digital technology [68]. The fourth revolution, subsequently, is the move towards digitization.

Industry 4.0 can be often simply understood as the application of the generic concept of Cyber-Physical Systems (CPS) [88] [60] to industrial production systems, however, Industry 4.0 is in fact a more complex evolution, or rather, revolution. As [67] describes it, Industry 4.0 consists of an "information-intensive transformation" of manufacturing and other industries, connecting data, people, processes, services, systems and IoT-enabled industrial assets, and generating, leveraging and utilizing information in order to contribute to an ecosystem of industrial innovation and

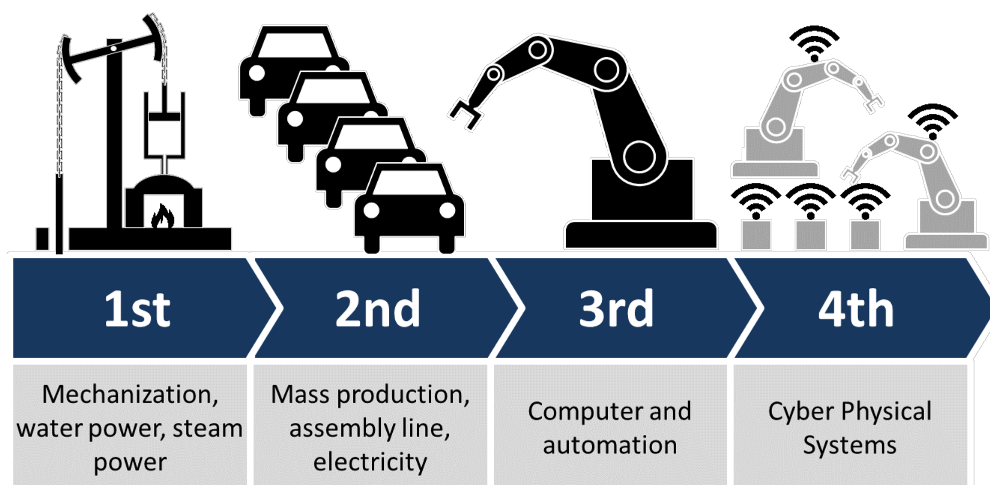


Figure 2.1: The four industrial revolutions [90]

collaboration. As such, promoters of this concept anticipate Industry 4.0 to bring vital advancements to industrial manufacturing processes, engineering, supply chain management, among other activities [43].

Essentially, this fourth industrial revolution distinguishes itself from the other ones by shifting from centrally controlled processes to decentralized production processes [57]. This consists in using large-scale Machine-to-Machine (M2M) and Internet of Things (IoT) deployments to help support increased automation, improved communication and monitoring, including self-diagnosis and more in-depth analysis, to provide a more productive industrial ecosystem [68]. As a result, factories will become increasingly automated and self-monitoring as the machines inside are enabled to analyze and communicate with each other and their human co-workers, granting companies much smoother processes that free up workers for other tasks [68].

One of the key promoters of Industry 4.0, the "Industrie 4.0 Working Group," developed the first recommendations for its implementation, which were published in April 2013 [43]. In this publication, the authors name some key technologies for Industry 4.0: IoT, CPS, Big Data, and Cloud Computing. These components are introduced subsequently.

2.1.1 Internet of Things

Internet of Things (IoT) is a computing concept that incites pervasive connection to the Internet, transforming everyday objects into connected devices [33]. The focus behind the IoT concept is to deploy multiple smart devices capable of sensing the surrounding environment, acquire environment data, send it (to the cloud) and analyze it (on the cloud), and finally construct useful information about said environment [33]. The connection of unusual objects to the Internet shall enhance the sustainability of both industry and society, allowing for efficient interactions between the physical world and its digital equivalent [33].

While IoT is usually portrayed as the groundbreaking technology for solving a vast majority of current society issues like smart cities, autonomous driving, pollution monitoring, among others, it also provides multiple solutions to problems in the industrial sector. As a subset of IoT, Industrial IoT (IIoT) covers the domains of Machine-to-Machine (M2M) and industrial communication technologies with automation applications [33].

IIoT leads to an improved comprehension of the manufacturing process, thus enabling efficient and sustainable production [33]. Communication in IIoT is machine-oriented and can run over a substantial wide range of sectors and activities. Some IIoT scenarios include legacy monitoring applications (such as machine monitoring in production plants) and innovative approaches for self-organizing systems (like autonomic industrial plants that require little to no human intervention) [59].

Given that IIoT derives from IoT, most general communication requirements of both domains are naturally very similar: low-cost support for the Internet ecosystem, usage of resource-constrained devices, and the need for network scalability and security [33]. However, there are also multiple communication requirements that are particular to each domain in regards to Quality of Service (which evaluates determinism, latency, throughput, among other requirements), availability and reliability, and security and privacy [33]. Effectively, IoT specializes on developing new communication standards that can connect novel devices to the Internet in a flexible and user-friendly way. By contrast, the current design of IIoT emphasizes on possible integration and interconnection of once isolated plants and working islands (or even machines), thus contributing to more efficient production and new services [59].

2.1.2 Cyber-Physical Systems

A cyber-physical system is defined as transformative technologies for managing interconnected systems between their computational capabilities and physical assets [53]. In other words, it is a system of collaborating IT elements, designed to control physical (mechanical, electronic) objects, where communication is done via the Internet (or other data infrastructures) in a closed environment [37]. Thus, a CPS generally consists of two main functional components [53]:

- Advanced connectivity that ensures real-time data acquisition from the physical world and information feedback from the cyber space;
- Intelligent data management, computational and analytics capability that constructs the cyber space.

Designing and deploying a cyber-physical production system can be done based on a five-level CPS structure proposed in [45]. It defines how engineers construct a CPS from the initial data acquisition, then analytics, and finally to the creation of real value for businesses. Among the five levels, the cognition and configuration levels are considered to be the most difficult to achieve [45].

In manufacturing, CPSs can improve quality and productivity through smart prognostics and diagnostics using data from different machines, networked sensors, and systems [45]. However, for more complex manufacturing systems, the integration of data from heterogeneous sources (different suppliers, different time stamps, and different data formats) can be a big challenge [45]. Therefore, the role of Big Data analytics for cyber-physical production systems will reach into design, manufacturing, maintenance, use, and reuse when engineers try to handle new types of data and problems [98].

2.1.3 Big Data Analytics

Big Data analytics consists on analyzing large data sets that contain a multitude of data types [84] to uncover hidden patterns, unknown correlations, data trends, among other useful business information [42]. By examining great amounts of data, an organization is capable of dealing with considerable information that can affect a business [91]. Thus, the main goal of big data analytics is to assist businesses in order to improve data understanding, and thus, be able to reach efficient and well-informed decisions.

As such, this process is rapidly emerging as a key IoT requirement to improve decision-making [64]. Big data analytics in IoT requires processing a large amount of data and storing it in various storage technologies. Given that a significant part of the unstructured data is acquired directly from web-enabled "things," Big Data implementations have to perform quick analytics with large queries to allow organizations to gain rapid insights, make fast decisions, and interact with people and other devices [64]. The interconnection of sensing and actuating devices gives the ability to share information across platforms through a unified architecture and develop a common operating picture for enabling innovative applications [64]. In the case of industrial manufacturing, Big Data analytics will enhance manufacturing efficiency by improving equipment service, reducing energy costs, and improving production quality [11].

2.1.4 Cloud Computing

Cloud computing, also known as simply "cloud," is the delivery of on-demand computing resources – from applications to data centers – over the internet [50]. An Oracle report from 2016 [75] revealed that of the 1200 technology decision-makers surveyed across Europe, Middle East, and Africa in midsize and large companies, 60% believe an integrated approach to cloud will unlock the potential of disruptive technologies, particularly in areas such as robotics and artificial intelligence. Cloud computing is a big shift from the traditional way businesses think about IT resources. As already mentioned in Section 2.1.1, one of the major goals of IIoT is to connect all machines and devices within the industry in order to produce valuable data that can be used for analysis. However, given the large amount of data that is generated by all these systems, data size often becomes too massive to handle and insight generation becomes complex in nature. Thus, cloud computing can help mitigate these problems by providing industry specific solutions, such as remote control and operation, predictive maintenance, and automation [3].

2.2 Transitioning from Monoliths to Microservices, through REST

When looking at the characteristics of enterprise applications for the fourth Industrial Revolution, special focus should first be given to the most fundamental necessary changes along the entire value chain, which in many cases are already taking place in the industrial world [93]:

- Self-control: CPS will function and interact independently;
- Self-organization: Different agents will cooperate with each other on the global IoT, leading to a decentralization of decisions;
- Complex algorithms for centralized supply chain planning must be swapped with less-complex decentralized algorithms;
- Customers, suppliers, and business partners must be strongly incorporated along the value chain;
- Responsiveness: Transparent decisions in decentralized control cycles enable fast reactions to change and disruptions.

Considering these changes that await, it would be correct to say that monolithic enterprise applications will become obsolete in Industry 4.0 [93]. A monolithic architecture is developed on a single programming stack, where each component is highly reliant on one another [8], and it can only scale in one dimension [89]. As depicted in Fig. 2.2, it can scale with an increasing transaction volume by running more copies of said application. However, this architecture cannot scale with a rising data volume – since each copy of the application will access all data, making caching less effective and increasing memory consumption and I/O traffic [89]. Moreover, different application components have different resource requirements, i.e., one might be CPU intensive while another might be memory intensive [89]. Hence, developers cannot scale each component independently in a monolithic architecture.

Thus, the most efficient way of achieving scalability and decentralization is with microservices and RESTful APIs [9]. The term *microservices* describes an architectural approach in which a developer builds an application as a set of frugal, comprehensible components that typically communicate via RESTful APIs. RESTful APIs are a fundamental building block for an IIoT cloud-based integration platform because of their lightweight, asynchronous, and stateless nature [9]. In addition, their uniform interface and the fact that they have become the standard for most cloud-based services makes integration efforts simpler and more sustainable than proprietary data exchange models [9].

When deployed in an IIoT scenario, this modern architectural approach allows organizations to react based on small changes in data or state. The use of APIs and microservices enables the system to orchestrate and combine discrete actions to achieve the intended result. The arrangement between RESTful APIs and the microservice architecture is a vital piece of the ongoing evolution of Industry 4.0 [9]. As organizations recognize the increasing strategic importance of

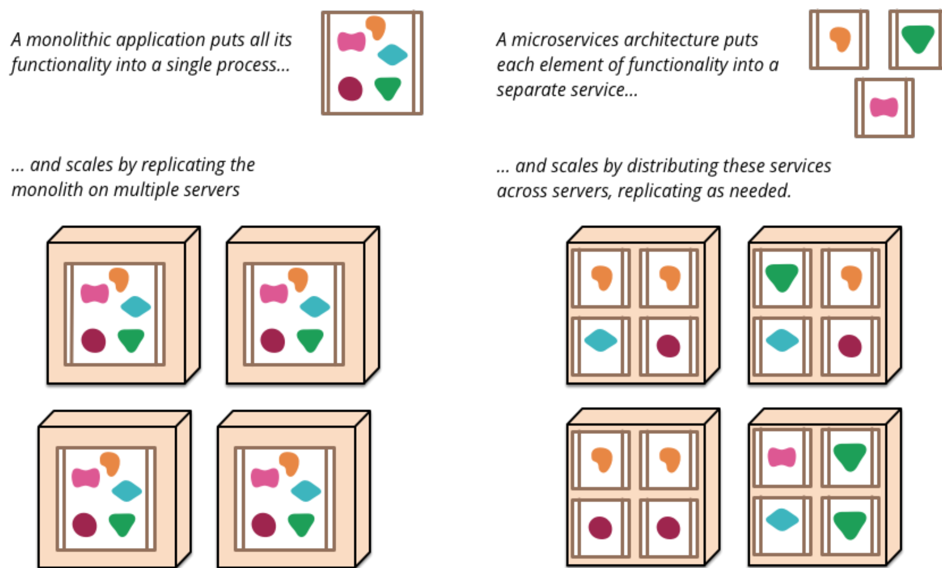


Figure 2.2: Monoliths vs. Microservices [62]

creating defensible, proprietary data and, as a result, continue to expand the data universe that they must integrate, they will demand the flexibility, scalability, and interoperability that the RESTful API/microservices combination provides [9].

2.3 The Arrowhead Framework [1]

The Arrowhead project is a large European effort that aimed at normalizing the interaction between IoT applications through a Service Oriented Architecture (SOA). This effort targeted many application domains comprising industrial production, smart buildings, electro-mobility, and energy production. Services are exposed and consumed by (software) systems, which are executed on devices, which are physical or virtual platforms providing computational resources. The devices are grouped into local automation clouds (see Figure 2.3), which are self-contained, geographically co-located, independent from one another, and mostly protected from external access through security measures.

Arrowhead services are considered either application services (when implementing a use case), or core services (that provide support actions such as service discovery, security, service orchestration, and protocol translation). To facilitate application development, the core systems are included into the common Arrowhead Framework [51]. The Arrowhead Framework is intended to be either deployed at the industrial site, or accessed securely, for example through a Virtual Private Network

[1] The following explanation was mostly based on the papers "The Arrowhead Framework applied to energy management" [85] (whose author is the same as this dissertation's), "Quality of Service on the Arrowhead Framework" [63] and "Making system of systems interoperable – The core components of the arrowhead framework" [80]. Thus, this dissertation was given permission by the papers' respective authors to use the information and text available in these documents, provided that the original works and authors were properly referenced.

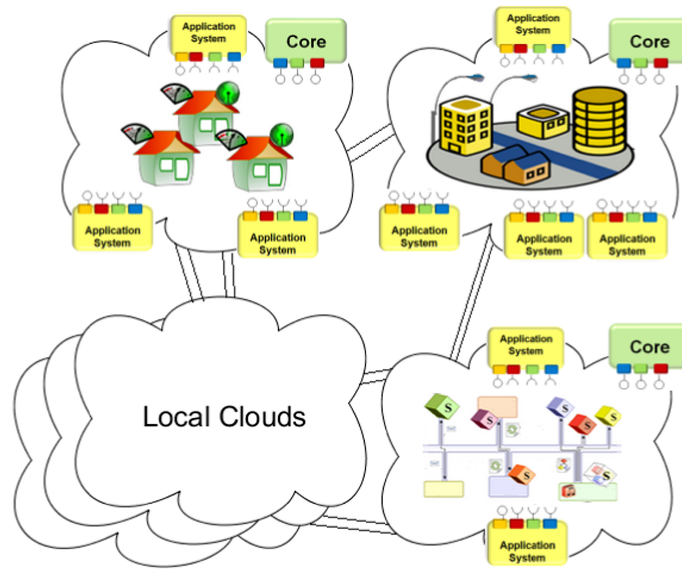


Figure 2.3: Interconnected local collaborative clouds [80]

(VPN). As per the Arrowhead approach, local clouds are governed through their own instances of the core systems (see Figure 2.4). There are two main groups of the core systems:

- The mandatory ones that need to be present in each local cloud (at level **I** in Figure 2.4);
- The automation supporting ones that further enhance the core capabilities of a local cloud (at levels **II** and **III** in Figure 2.4).

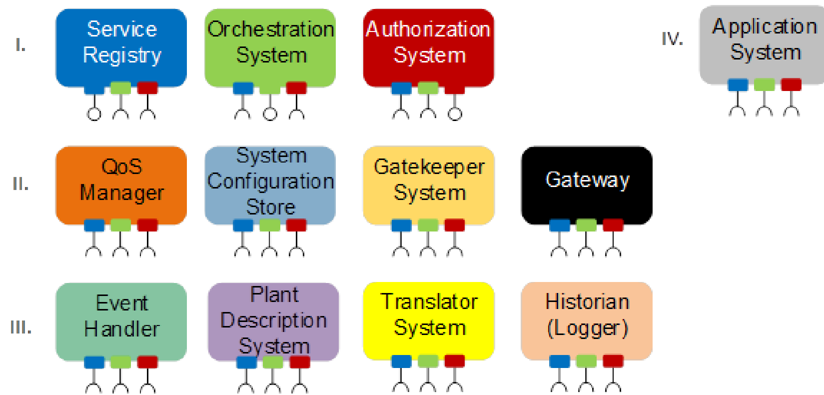


Figure 2.4: The Arrowhead Core Systems [18]

The core systems of Arrowhead focus on the maintenance of the local cloud itself and of non-functional requirements of use cases, and are included into, and shipped in the form of, the Arrowhead Framework [51]. Even in the most minimal local cloud, the core services take care of registration and discovery of services, systems and devices (ServiceDiscovery service, or SD), security (Authentication service, or AA), and orchestration of complex services (Orchestration service, or O), as portrayed by Figure 2.5. The application systems are also consumers of the core

services. The Orchestration service is used to assemble complex services, which may be comprised of several individual services. To this aim, services, systems and devices in an Arrowhead local cloud have to be registered, and through their registries (ServiceRegistry) the Orchestrator can access a global view of the local cloud.

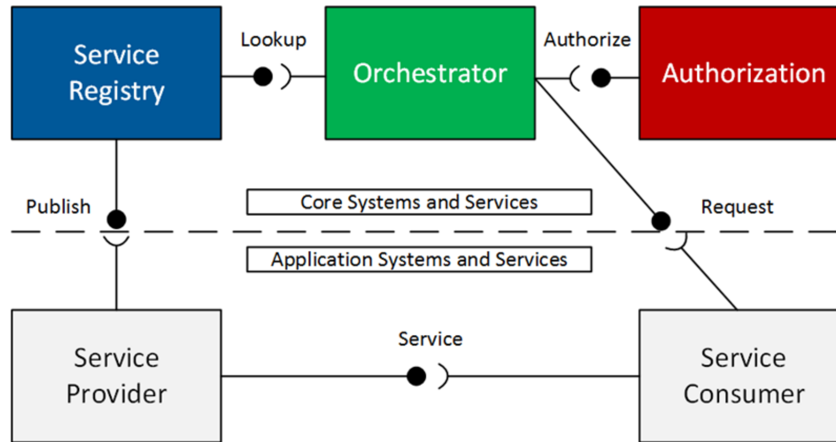


Figure 2.5: Overview of the Arrowhead Framework's mandatory systems [10]

For the sake of providing full context, the major Arrowhead systems that were used in this dissertation are subsequently presented.

2.3.1 Service Registry (Mandatory)

The Service Registry System keeps track of all active producing services within the network. It is used to ensure that all systems can find each other – even if endpoints are dynamically changed. To this ends, it supports a service registry functionality based on DNS and DNS-SD (using a DNS-SD BIND server and a DNS-SD Java library); since the Arrowhead Framework is a domain-based infrastructure. However, it also has a development-friendly version that relies on MySQL based REST module to facilitate the bootstrapping process of the framework (using the Hibernate ORM with a MySQL connector) [18].

All Systems within the network that have services producing information to the network shall publish its producing service within the Service Registry by using the Service Discovery service. Within a system of systems, the Service Registry further supports system interoperability through its capability of searching for specific service producer features, i.e. an application service producer with a specific type of output. In short, it enables systems to publish their own application services and lookup others'.

2.3.2 Authorization System (Mandatory)

The Authorization system controls that a service can only be accessed by an authorized consumer. It consists of two service producers and one service consumer and it maintains a list of access rules to system resources (i.e. services). The Authorization Management service provides the

possibility to manage the access rules for specific resources. The Authorization Control service provides the possibility of managing the access for an external service to a specific resource. The system uses the Service Discovery service to publish all its producing services within the Service Registry system.

2.3.3 Orchestration System (Mandatory)

The Orchestration system is a central component of the Arrowhead Framework and also in any SOA-based architecture [36]. In industrial applications the use of SOA for massive distributed system of systems requires Orchestration. It is utilized to dynamically allow the re-use of existing services and systems in order to create new services and functionalities [56]. It is the primary decision-maker that is aware of the current conditions in the SoS. Its primary task is to allocate Service Providers to the Service Requests sent in by Systems (see Figure 2.6). During this orchestration process the Orchestrator consults with the other Core Systems and makes a decision based on the responses.

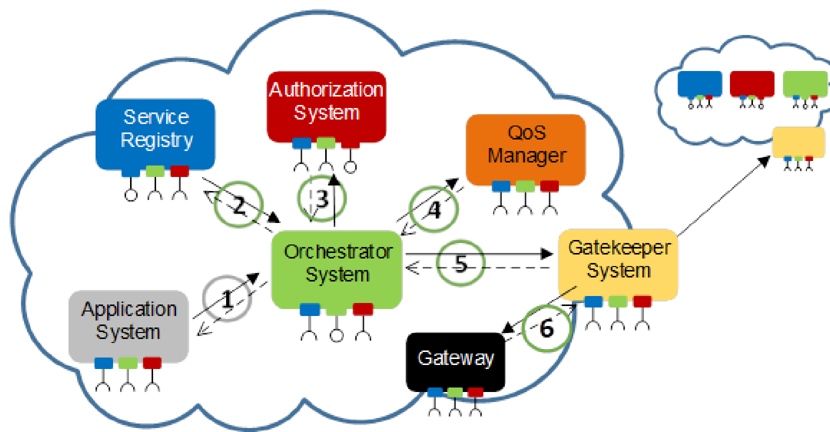


Figure 2.6: Overview of the Orchestration process [18]

2.3.4 Event Handler

The Event Handler System facilitates communication and data sharing between Application Systems in an Arrowhead network through "event propagation," following a Publish-Subscribe model (as depicted in Figure 2.7).

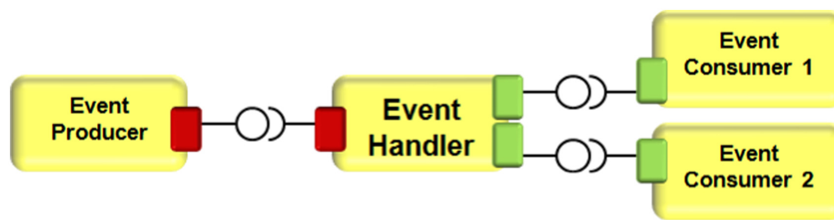


Figure 2.7: Overview of the Event Handler [80]

The Event Handler receives events from Event Producers and dispatches them to registered Event Consumers. The Event Handler is also responsible for logging events to persistent storage, registering event producers and consumers, and applying filtering rules (configured by Event Consumers) to incoming events. These rules can be, for example, based on message content or simply based on the severity level of the message, which varies from Debugging to Critical. The Event Handler can also be connected to an internal or external database system which is responsible for the permanent storage of all events. Effectively, the Event Handler works similarly to typical message broker, but implemented in REST-JSON, with the help of concurrency handling libraries.

Chapter 3

Problem Statement

This dissertation's work is part of the European research project Productive4.0 [1], which is currently developing the Arrowhead Framework. At present, the framework had not been analyzed from a performance standpoint and, as such, there was no perceived sense of how long the framework would take to perform certain operations, i.e., how long does it take to receive an orchestration response, which systems take longer to respond, what is the average latency for message forwarding, and so forth. In fact, this necessity proved to be an interesting challenge to be tackled by this dissertation.

Still, an exhaustive performance evaluation of the Arrowhead Framework would not be feasible for the scope of this thesis, since it is a rather complex framework that can even support an unbounded number of use-cases – after all, a framework can be built for implementing an infinite number of systems. Thus, in an attempt to limit the analysis' scope, the performance modeling was chosen to be use-case driven. As such, this dissertation's first objective was to show how one can develop a performance model, using two important use cases of the Arrowhead Framework. Regarding the performance modeling process, assembling performance models that correctly capture the different facets of system behavior is said to be an overly challenging task when applied to large and complex real-world systems [58]. This thesis to followed the undermentioned modeling methodology [58], which is based on existing work in software performance engineering by Menasce et al. [28, 27]:

1. **Establish performance modeling objectives –** Set concrete goals for the performance modeling effort.

2. **Characterize the system in its current state –** Develop a specification that includes detailed information on the system design and topology, the hardware and software components that it consists of, the communication and network infrastructure, among other characteristics.

3. **Characterize the workload –** The workload of the system should be described in a qualitative and quantitative manner.

4. **Develop a performance model –** Develop a performance model which depicts the different components of the system and its workload and captures the main factors affecting its performance.

5. **Validate, refine, and/or calibrate the model –** The model is considered valid if the performance metrics predicted by the model match the measurements on the real system within a certain acceptable margin of error [29]. If this does not happen, then the model must be refined or calibrated to more accurately reflect the modeled system and workload.

6. **Use the model to predict system performance –** The validated performance model is used to predict the performance of the system for the deployment configurations and workload scenarios targeted for analysis.

7. **Analyze results and address modeling objectives –** The results from the model predictions are analyzed and used to address the goals set in the beginning of the modeling study.

Furthermore, a major part in the development of a model consists in choosing the level of abstraction. Unfortunately, there are no specific rules for choosing this, so the decision is based primarily on the performance analyst's expertise and ingenuity. Nevertheless, the level of abstraction is the main element that distinguishes a model from a prototype or an emulator. Simulation is better suited to developing more thorough models, while analytical models are usually more abstract [65].

Ultimately, this performance evaluation serves as a proof-of-concept for more in-depth performance analysis that can be potentially done in the future by the Arrowhead community.

However, the focus of this dissertation evolved throughout the course of the work. Originally, the goal of the thesis was to evaluate and model the performance of the Arrowhead Framework, but because the performance results were so poor for one of the framework's systems (i.e., the Event Handler), it was decided to redesign it and change its implementation accordingly to improve its performance, by using appropriate software configurations and design patterns. Given that some of these bottlenecks were caused by design decisions that also permeate other Arrowhead systems, this reengineering process would consequently serve as a case study for potential performance improvements on the rest of the framework's components.

Chapter 4

State of the Art

Usually, a problem is never entirely unique. More often than not, similar challenges to any problem statement have already been tackled by other people with multiple solutions. Therefore, this chapter analyzes other projects, approaches, technologies, or overall solutions that have tackled the same issue or were important/influential for this dissertation's development.

Given the thesis's subject matter, Section 4.1 studies and compares different middleware approaches to messaging, which is often required for IoT systems and will be important for later chapters. Afterwards, Section 4.2 presents a mathematical modeling language used for modeling the performance of distributed systems, and explains why it is a good fit for this dissertation.

4.1 Message-Oriented protocols when REST is inadequate

Indeed, RESTful interactions have become essential to enterprise computing as it enables many APIs on the web today [92]. The reason behind this is because REST is considered to be much easier to learn and handle than other web service interface approaches such as SOAP (Simple Object Access Protocol), which is a mature protocol intended to expose individual operations as services that use XML to describe the content of the message [49]. Since REST is based on standard HTTP operations rather than XML, it uses verbs with exact connotations such as "GET" or "DELETE" which prevents ambiguity. REST resources are also assigned individual URIs, adding flexibility to a web service's design [73]. With REST, information that is produced and consumed is separated from the technologies that enable production and consumption. Furthermore, since REST is not constrained to XML like SOAP, it can return XML, JSON, YAML or any other format depending on what the client requests [70]. In fact, REST is the predominant architectural style used in this thesis' project, the Arrowhead Framework, where every Arrowhead system is a RESTful service, even the Event Handler system – a message broker that forwards messages between systems, using a publish-subscribe model.

However, a REST/HTTP architecture is mostly limited to the client-server model, where the client system requests data/service to a server, the server system then responds to the request by providing that data/service, closing the HTTP connection. In other words, the duration of the contract between client and server is a one-time invocation. On the other hand, an event-driven architecture (which is usually implemented using the publish-subscribe model), has the goal to establish a long-term contract between the client and the server, by keeping the connection open between the client and the message broker, using subscriptions. Thus, the client-server model, and consequently REST, is not always the most suitable choice for every situation. For some application domains, such as real-time systems that need to propagate real-time data to multiple other systems (e.g. a temperature sensor providing sensor readings to maintenance systems), an event-driven architecture is more appropriate.

Furthermore, while publish-subscribe can be implemented using HTTP, other messaging protocols that have a lower connection overhead, might be better suited for this purpose. Therefore, s survey of the most relevant messaging protocols – along with message-oriented middleware (MOM) that use them – is subsequently presented.

4.1.1 Advanced Message Queuing Protocol (AMQP)

AMQP (standardized by the OASIS consortium [72]) was developed as an open standard with the goal of replacing existing proprietary messaging middleware. OASIS states that three of the most important reasons to use AMQP are security, reliability and interoperability [6].

AMQP has two different specifications: AMQP 0.9.1 and AMQP 1.0, where each version has a completely different messaging paradigm. AMQP 0.9.1 implements the publish/subscribe paradigm, relying on an AMQP broker that handles the “exchanges” and the messages queues. The newer version of AMQP protocol, AMQP 1.0, is not tied to any particular messaging mechanism. This dissertation will only focus on the 0.9.1 implementation, herein mentioned as AMQP.

Regarding its messaging features, AMQP provides a wide range, including reliable queuing, topic-based publish/subscribe messaging, flexible routing, and transactions (see Figure 4.1). AMQP exchanges route messages directly, either in fanout form (i.e. broadcasts all messages to all queues), by topic, or based on headers [82]. Furthermore, it is possible to restrict access to queues and manage their depth. This protocol was designed for business messaging with the idea of offering a non-proprietary solution that can manage a large amount of message exchanges that could happen in a short time in a system.

Moreover, AMQP uses TCP for reliable transport and supports three different levels of QoS:

- **QoS 0:** Delivers on a best effort assumption, with no confirmation on message acknowledgment. This QoS level can be useful, for example, for a GPS tracker that sends data of a location every few minutes, over a long period of time. Therefore, it is adequate if the messages with GPS location are sometimes missing, because the general location is still known since most of the message updates have been received [52].

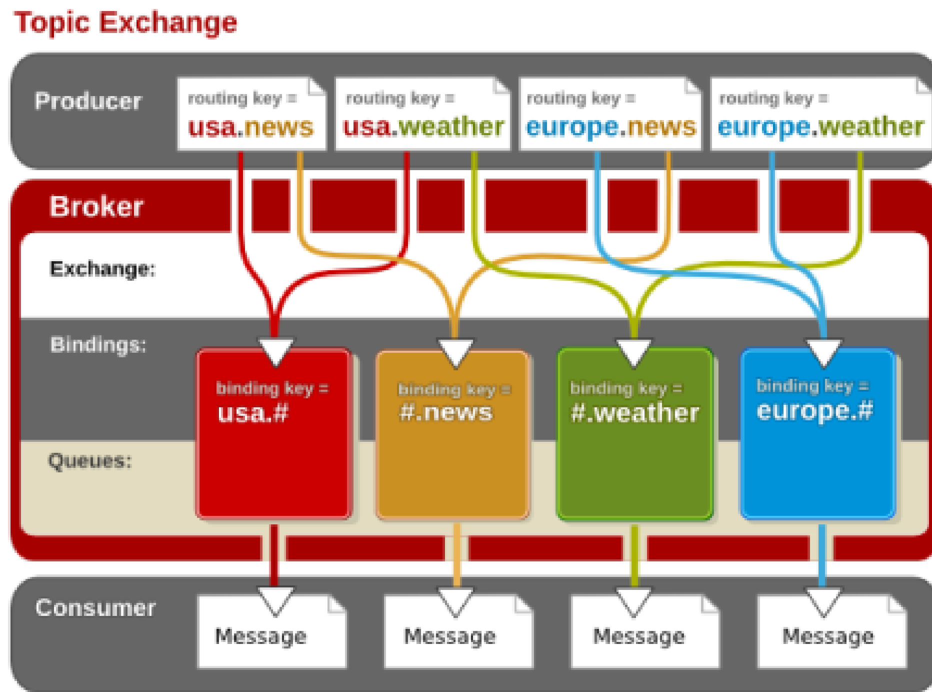


Figure 4.1: Topic Exchange between Publishers and Subscribers in AMQP [35]

- **QoS 1:** Guarantees that messages will reach their destination, so a message confirmation from subscribers is necessary. Thus, the subscriber must send an acknowledgement, and if it does not arrive in a defined period of time, the publisher will publish the message again [52].

- **QoS 2:** Assures that the message will be delivered exactly once, without message duplication [52].

Thus, for resource constrained systems where battery life is more important than reliable communication, QoS 0 is a valid option. For data exchange between more powerful systems, QoS 1 and QoS 2 are naturally better options. Finally, the AMQP protocol offers complementary security mechanisms for data protection through the TLS encryption protocol, and also for authentication by using SASL (Simple Authentication and Security Layer) [52].

Additionally, AMQP adoption has been remarkably strong: companies like VMWare use it in their virtualization products and cloud services, NASA uses it for the control plane of their Nebula Cloud Computing, and Google uses it for complex event processing to analyze user defined metrics [7]. In terms of available MOM for AMQP, the following brokers are some of the most popular ones: RabbitMQ [86], Apache ActiveMQ [38], and Apache Qpid [39].

4.1.2 Message Queuing Telemetry Transport (MQTT)

The design principles and aims of MQTT (also standardized by OASIS) are much simpler and focused than those of AMQP: it provides publish/subscribe messaging, does not use queues (despite its name), uses only topics instead of exchanges and bindings (see Figure 4.2), and was specifically designed for resource-constrained devices and non-ideal network connectivity conditions, such as low bandwidth and high latency (e.g., dial up lines and satellite links) [82, 52].

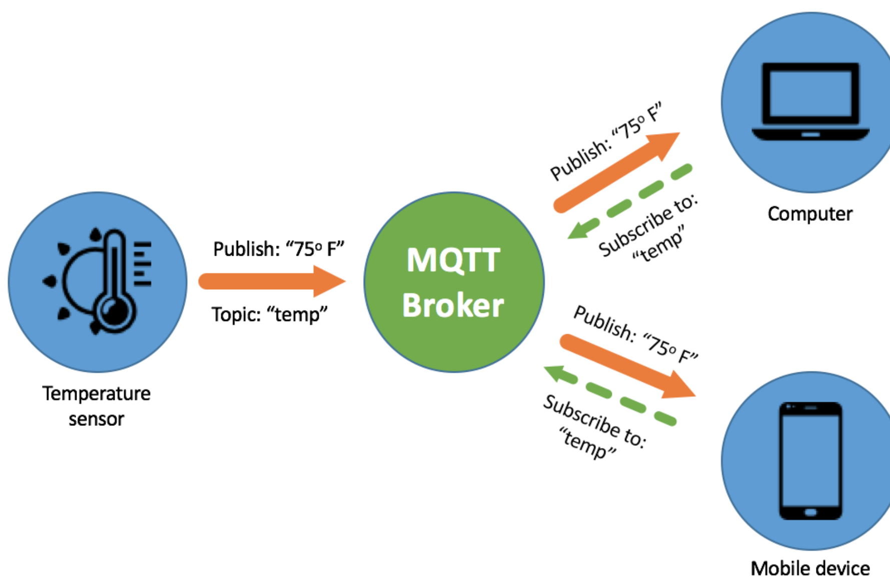


Figure 4.2: Topic Subscription in MQTT [12]

Thus, one of the advantages MQTT has over more full-featured "enterprise messaging" brokers is precisely its simplicity and its very small message header. Similarly to AMQP, MQTT runs on top of the TCP transport protocol, which ensures its reliability. Additionally, because MQTT has much lower power requirements than other reliable protocols like HTTP, it is considered one of the most prominent protocol solutions in constrained environments [82, 52]. As a matter of fact, Facebook uses it for their mobile applications [94].

For QoS, MQTT has the same three QoS levels as AMQP: QoS 0, 1, and 2 [71]. The higher the QoS level, the bigger amount of necessary resources it is to process a MQTT packet. Therefore, it is vital to adjust the QoS option to particular network conditions. Also, another important feature MQTT offers is the possibility to store some messages for new subscribers by setting a "retain" flag in published messages [52].

In conclusion, MQTT's strengths are simplicity and a compact binary packet payload (no message properties, compressed headers, much less verbose than a text-based protocol like HTTP), making it a preferable option for simple push messaging scenarios, e.g. temperature updates, machine operation logs, and mobile notifications [82]. Moreover, there are multiple MQTT-based

MOM that are open for use, which differ in their MQTT protocol implementation. Some of them are Mosquitto [69], RabbitMQ [86], Apache ActiveMQ [38], and HiveMQ [47].

4.1.3 Simple Text Oriented Messaging Protocol (STOMP)

Unlike the prior mentioned protocols, STOMP is text-based, making it more similar to HTTP, while also running over the TCP transport protocol. Like AMQP, STOMP provides a message header with properties, and a message body. The design principles of STOMP were to provide easy and widespread messaging interoperability among different programming languages, platforms and brokers [96].

It should be mentioned that STOMP does not use queues or topics: it uses a *SEND* semantic with a "destination" string. The broker itself must manually map onto something that it understands internally such as a topic, queue, or exchange. Consumers *SUBSCRIBE* to those destinations [82]. Since both the semantics and the detailed syntax of the destination tag are not defined in the official specification, different brokers can actually interpret the destination in different manners, which compromises the protocols interoperability [5].

However, STOMP is simple and lightweight (even though it is considered somewhat verbose), with a wide range of language bindings, and also provides some transactional semantics [97]. Regarding available MOM with STOMP implementations, Apache ActiveMQ [38], RabbitMQ [86], CoilMQ [61], and HornetQ [48] are some of the suggested brokers by the STOMP community [95].

4.1.4 Extensible Messaging Presence Protocol (XMPP)

XMPP is an open standard messaging protocol, originally designed for instant messaging and message exchange between applications [102]. Like STOMP, XMPP is also a text-based protocol, using Extensible Markup Language (XML), and implements both client/server and publish/subscribe approaches, running over TCP [103].

One of the most important characteristics of this protocol is security, making it one of the more secure messaging protocols mentioned in this State of the Art. Unlike other protocols such as MQTT, where TLS encryption is not built-in within the protocol specifications, XMPP specification already implements a TLS mechanism [103], which provides a reliable mechanism to ensure confidentiality and data integrity. Beside TLS, XMPP also implements SASL [103], which guarantees server validation through an XMPP-specific profile [26].

However, given its instant messaging origins, XMPP has some weaknesses that should be considered. Namely, the XML structure used on every message makes their sizes inconveniently large, especially when used in networks with bandwidth problems. Additionally, another downside is the lack of QoS, in fact, because XMPP runs on top of a persistent TCP connection and lacks an efficient binary encoding, it is not suited for unstable, low-power wireless networks [52]. However, there has been an effort to make XMPP better suited for IoT [16, 30], for instance, a lightweight publish/subscribe approach has been developed for resource constrained IoT devices, in order to

optimize the existing XMPP implementation [44]. In regards to available MOM for XMPP, the following brokers are some of the recommended ones by the XMPP community [104]: ejabberd [83], AstraChat [99], Openfire [24], and Tigase XMPP Server [100].

4.1.5 Data Distribution Service (DDS)

DDS is a real-time data-centric interoperability standard which uses a publish/subscribe approach. However, unlike the other previously mentioned publish/subscribe protocols, DDS is decentralized and based on peer-to-peer communication, and therefore does not rely on a message broker, ultimately removing a potential single point of failure for the whole system. Both communication sides are then decoupled from each other, and a publisher can publish data even if there are no interested subscribers. The data consumption is essentially anonymous, since the publishers do not inquire about who consumes their data [52].

Additionally, one of the most prominent features of the DDS protocol is its scalability, which comes from its support for dynamic discovery. The discovery process, achieved through a built-in discovery protocol, allows for subscribers to identify which publishers are present, and to specify information they are interested in (through topics) with the desired QoS (which are included in a very extensive set of policies), and for publishers to publish their data [23]. The various QoS policies manage a gargantuan amount of DDS features, such as data availability, data delivery, data timeliness, and resource utilization [101]. In order to communicate with each other, publishers and subscribers must then use the same topic (same name, type and a compatible QoS) [52].

Furthermore, DDS uses UDP/IP as its default transport protocol for interoperability purposes and multicast for anonymous discovery, nonetheless it can also support TCP/IP. Regarding security, DDS employs different solutions depending on the transport protocol that is being used. If TCP is the transport protocol of choice, then TLS can be used. In the case of UDP, the DTLS protocol is used [52]. Regarding available MOM with DDS implementations, the following middleware are free to use: OpenDDS [74] and Vortex DDS [2].

4.1.6 Summarizing all mentioned messaging protocols

Given the multiple protocols mentioned in this chapter, the following table (Table 4.1) summarizes each protocol's characteristics for easy comparison.

Table 4.1: A comparison between all mentioned message protocols

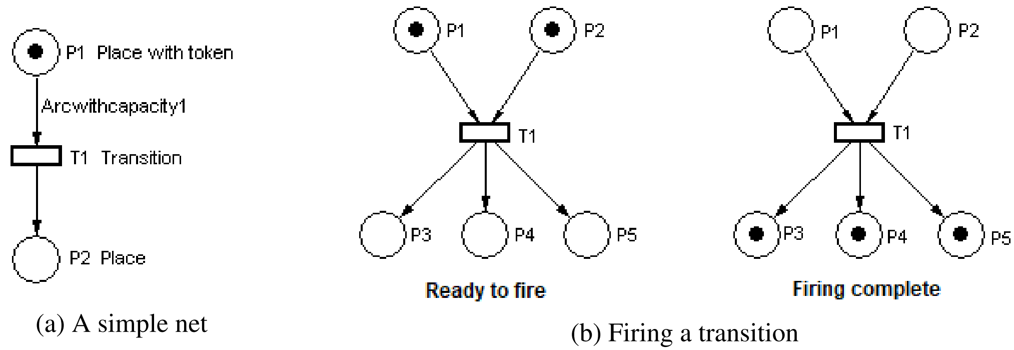
Messaging Protocol	Req/Resp	Pub/Sub	Service Discovery	Transport Protocol	Quality of Service	Security	Available implementations
AMQP	Yes (1.0)	Yes (0.9.1)	No	TCP/IP	3 optional levels	TLS (built-in)	RabbitMQ, Apache ActiveMQ, Apache Qpid
MQTT	No	Yes	No	TCP/IP	3 optional levels	TLS (optional)	Mosquitto, RabbitMQ, Apache ActiveMQ, HiveMQ
STOMP	Yes	Yes	No	TCP/IP	Application dependent	TLS (optional)	Apache ActiveMQ, RabbitMQ, CoilMQ, HornetQ
XMPP	Yes	Yes	Yes	TCP/IP	–	TLS (built-in)	ejabberd, AstraChat, Openfire, Tigase XMPP Server
DDS	No	Yes	Yes	UDP/IP TCP/IP	Multiple policies	DTLS (UDP) TLS (TCP) (optional)	OpenDDS, Vortex DDS

4.2 Using Petri Nets for performance modeling

One of this dissertation's goals was to develop a performance model, and while the modeling methodology to be followed had been chosen back in Section 3, a question still remained: what kind of analytical model is appropriate for this process?

Petri Network models (also known as Petri nets) are widely used for these situations [58, 105, 54]. Petri nets were proposed as an easy and convenient formality for the process of modeling systems that deal with concurrent activities [105], such as communication networks, multiprocessor systems, manufacturing systems and distributed databases. Their increasing popularity is due to their simple representation of concurrency and synchronization – which are not easily expressed in traditional formalisms – developed for analysis of systems with sequential behavior [105]. The risk, however, is that these models can become too detailed, and therefore intractable. Nevertheless, as explained in Section 3, the level of abstraction is dependent on each particular case, and its decision should be based primarily on the performance analyst's expertise and ingenuity.

Petri nets are bipartite directed graphs, in which the two types of vertices – named "places" and "transitions" – represent conditions and events (see Figure 4.3). Places may hold tokens (see Figure 4.3a). The state of a Petri net consists in its assignment of tokens to places [17]. An event can only happen when all conditions associated with it (represented by arcs aimed at the event) are satisfied. The occurrence of an event often leads to a new state (see Figure 4.3b), indicated by arcs directed from the event. Essentially, the occurrence of one event causes another event to occur, and so on [105].



(a) A simple net

(b) Firing a transition

Figure 4.3: Examples of Petri Nets [17]

Petri net models can then be extended with a temporal specification, by associating a firing delay to a transition – these models are called Timed Petri nets. This temporal specification can either be deterministic or probabilistic. When the specification of the firing delay is of probabilistic nature, the transition is associated with the probability density function or the probability distribution function of the delay (see Figure 4.4). These models are known as Stochastic Petri nets. Thus, when performing a stochastic analysis, the firing delays are randomly selected by the probability distributions associated with the transitions [46].

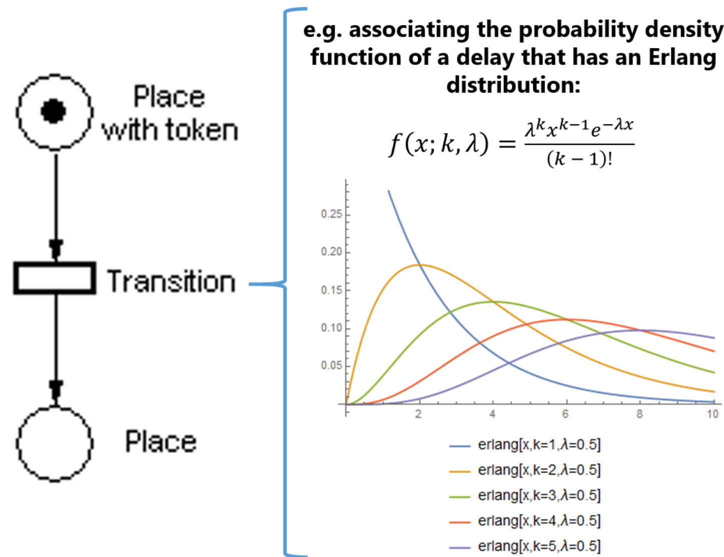


Figure 4.4: Associating a probability density function of a delay which has an Erlang distribution.

As such, Stochastic Petri nets seem to be a good fit for this project, given that combining stochastic and deterministic timings into one model is particularly significant for representing network latency and, consequently, the performance analysis of networked systems. Moreover, the probabilistic approach may be advantageous as it can provide decent accuracy, while compromising on more general results.

Another thing to notice in Stochastic Petri nets is that time enabled transitions fire one at a time,

according to a sequence that is determined by the probabilistic structure of the model. In the case of immediate transitions, whenever multiple ones are enabled, the selection of the next transition to fire will be based on parameters other than the temporal ones. In fact, for immediate transitions, the firing delay is deterministically equal to zero so that it cannot be used to discriminate against the one that will actually fire [41].

Furthermore, aside from these time specifications, a transition is also able to carry an "enabling function" (i.e. a boolean expression) that only allows a token to go from one place to the next if a certain condition is satisfied – these conditions are related to the amount of tokens located in a particular place(s). These special transitions are sometimes identified with the letter "e" next to them (see Figure 4.5), depending on the Petri net editor used.

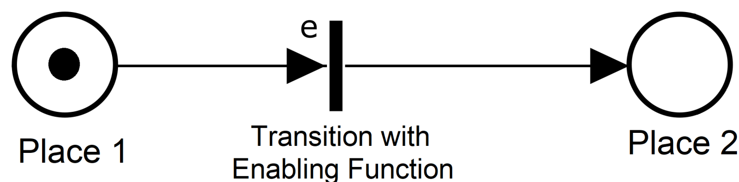


Figure 4.5: A transition carrying an enabling function.

Finally, another Petri net component should be mentioned – the inhibitor arc (see Figure 4.6). An inhibitor arc – i.e. an arc with a black circle on its end – is used to mandate that the transition must only fire when its place has no tokens. This could be used to, for example, ensure that a place must only carry one token at a time.

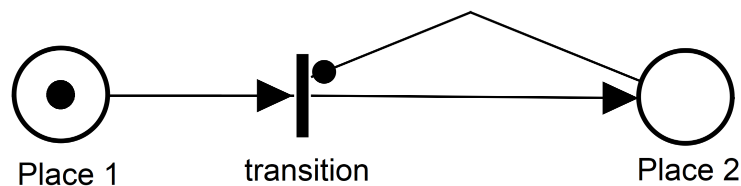


Figure 4.6: An inhibitor arc from "Place 2" to "transition".

Chapter 5

Performance Evaluation of the Arrowhead Framework

In order to evaluate the performance of the Arrowhead Framework, several tests were conducted using different testing environments for each use case scenario. As such, the following sections describe the purpose of each use case (among other details), their testing environment (i.e. what types of machines were deployed, which Arrowhead clouds were set up, and what are the network specifications to which these machines are connected to), and the messaging workload that was employed in order to stress different aspects of the software infrastructure.

Before going into detail about each use case, a slight code analysis of the common practices of each Arrowhead system's implementation is subsequently presented, since this information is important for later topics covered in this thesis.

5.1 Implementation of an Arrowhead system

Every Arrowhead system in the official Arrowhead repository [18] extends the *ArrowheadMain* abstract class, which is located in the *core-common* package. This class is responsible for the initial setup of an Arrowhead system, e.g., configuring and starting the system's web server, creating a database session (through the *DatabaseManager* class, using the Hibernate ORM), registering the core system services to the Service Registry, among other operations.

Regarding the web server setup, *ArrowheadMain* uses a REST-based architecture implemented on top of Grizzly [21] and Jersey [79]. Grizzly comprises: i) a core framework that facilitates the development of scalable event-driven applications using Java Non-blocking I/O API, and ii) both client-side and server-side HTTP services. Jersey is a framework that facilitates the development of RESTful Web Services and its clients, by providing an implementation of the JAX-RS API and some extensions.

JAX-RS is the "standard" specification for developing REST services in Java. Some implementations of JAX-RS include Jersey, RESTeasy, WebSphere, and Jello-Framework. Since Jersey's developers are the Oracle Corporation and the Eclipse Foundation, Jersey is considered to be the official implementation of JAX-RS. A simple example of the implementation of a JAX-RS resource using Jersey is shown in Listing 5.1.

```java
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
 * Root resource (exposed at "myresource" path)
 */
@Path("myresource")
public class MyResource {

    /**
     * Method handling HTTP GET requests. The returned object will be sent
     * to the client as "text/plain" media type.
     *
     * @return String that will be returned as a text/plain response.
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

Listing 5.1: Example of the implementation of a JAX-RS resource using Jersey [76].

The default use of Jersey (which uses servlets as its underlying mechanism [20]) will lead to the creation of a new thread for each request that is destroyed after its work is completed. Thus, RESTful services using standard Jersey will slow down when there are thousands of requests sent at the same time or at a very fast pace (the consequences of creating multiple threads are later explored in Section 7.3.3).

This thread policy with requests is usually enough for when a processing resource method takes a relatively short time to execute. However, in cases where resource method execution takes a long time to compute the result, JAX-RS's server-side asynchronous processing model should be used (using *AsyncResponse* as a parameter for the resource method). Through this, there is no association between a request processing thread and the client connection. The I/O container which handles incoming requests will never assume that a client connection can be safely

closed when a request processing thread returns. This methodology is able to increase throughput significantly, however it will still create a new thread when it does the actual work [77].

In order to solve this problem, the optimal solution is to use a threadpool, which reuses previously created threads to execute current tasks and offers a solution to the problem of thread creation overhead and resource consumption. However, while one could manually implement a threadpool using Java's *ExecutorService* for each resource method, the optimal solution is to use a web container that serves as an HTTP server for Jersey, and configure it to have a threadpool. While this accomplishes the same goal, it lowers the thread creation responsibility down a layer below Jersey and to the web container [4]. Grizzly is a popular implementation of these web containers and it is used by the *ArrowheadMain* class.

Regarding the Arrowhead systems themselves, all systems follow a similar approach to listing 5.1 for their JAX-RS resource (i.e. without using *AsyncResponse*), which means they do not reuse client connections and could thus suffer from connection overhead if multiple clients send requests at the same time. Furthermore, because *ArrowheadMain* uses Grizzly for its web container, these systems have the option to handle requests concurrently through a configured thread pool. However, the Grizzly HTTP server module in *ArrowheadMain* does not currently have a configured thread pool (as can be seen in Listing 5.2) and, by extension, none of the Arrowhead systems will then be able to efficiently handle multiple requests. These two factors will more than likely have an effect on the performance of each system and, consequently, on the performance of the orchestration process in its entirety.

```
1
2  public abstract class ArrowheadMain {
3    // ...
4
5    private void configureServer(HttpServer server) {
6      //Add swagger UI to the server
7      final HttpHandler httpHandler = new CLStaticHttpHandler(HttpServer.class.
           getClassLoader(), "/swagger/");
8      server.getServerConfiguration().addHttpHandler(httpHandler, "/api");
9      //Allow message payload for GET and DELETE requests - ONLY to provide custom
           error message for them
10     server.getServerConfiguration().setAllowPayloadForUndefinedHttpMethods(true);
11   }
12
13   // ...
14 }
```

Listing 5.2: Code snippet of *ArrowheadMain* class where the HTTP server (the web container) is configured.

5.2 Intracloud Orchestration

For the case of intracloud communication, its orchestration process is fairly simple, as shown in Figure 5.1. In fact, it only uses the three mandatory core Arrowhead systems: Orchestrator, Service Registry, and Authorization. Essentially, this type of communication is used when a consumer requests service X to the Orchestrator from an Arrowhead cloud that has a service provider for that request. The Orchestrator returns the provider's address to the consumer, as long as the consumer is properly authorized. Thus, all operations are held inside the same Arrowhead cloud (hence the name "*intra*"cloud).

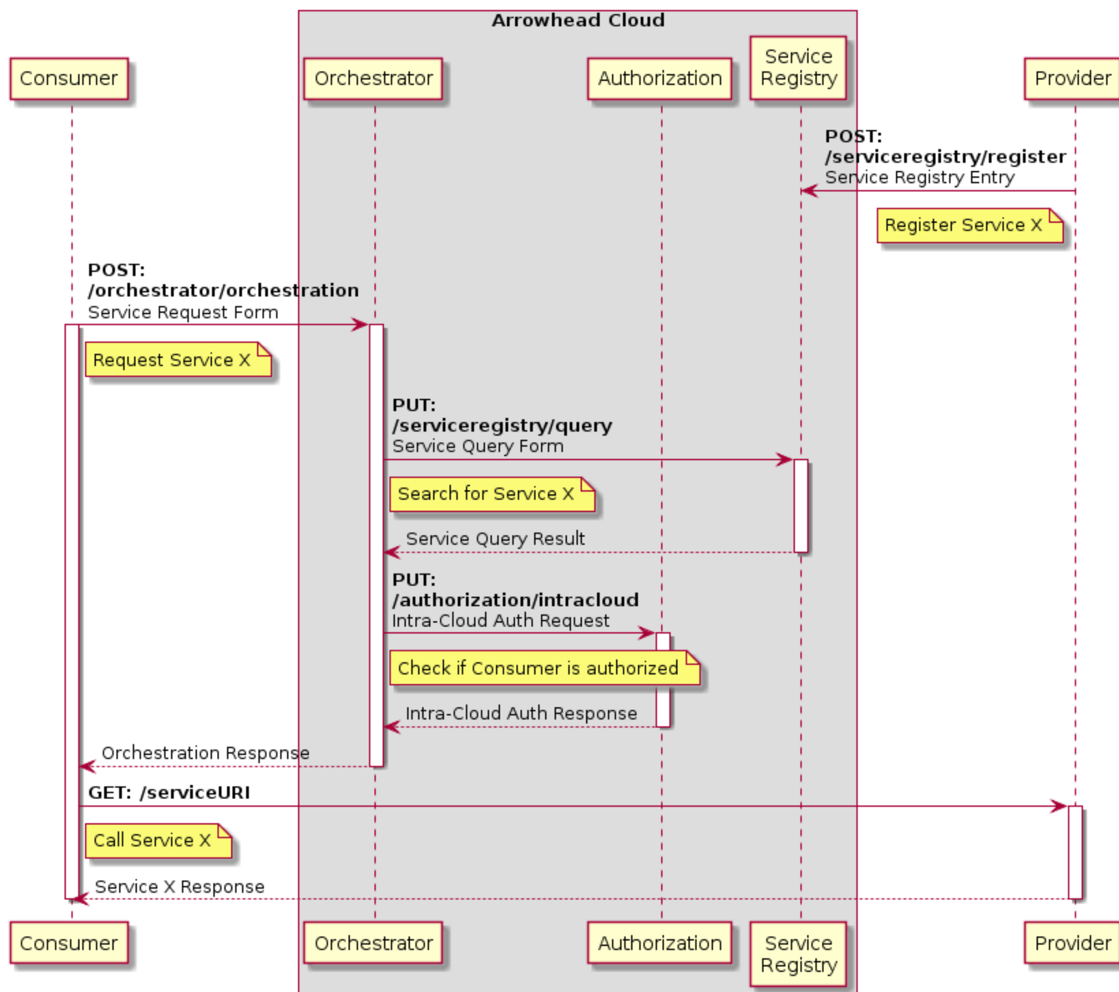


Figure 5.1: Sequence diagram of the Intracloud orchestration

Regarding the testing environment, all parties – the Consumer, the Service Provider, and the Arrowhead Cloud (which includes all core systems) – were deployed on different machines (see Figure 5.2). In terms of the messaging workload, 100 requests (whose contents can be seen in Listing 5.3) were sent to the cloud's Orchestrator, with a one second delay between each message sent. Each request is around 700 bytes long, sent on a 100 Mb/s Switched Ethernet LAN. To

measure the latency between each system, each time one of these systems sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. Moreover, all system clocks were synchronized using a local NTP server, which provides accuracies in the range of 0.1 ms [66].

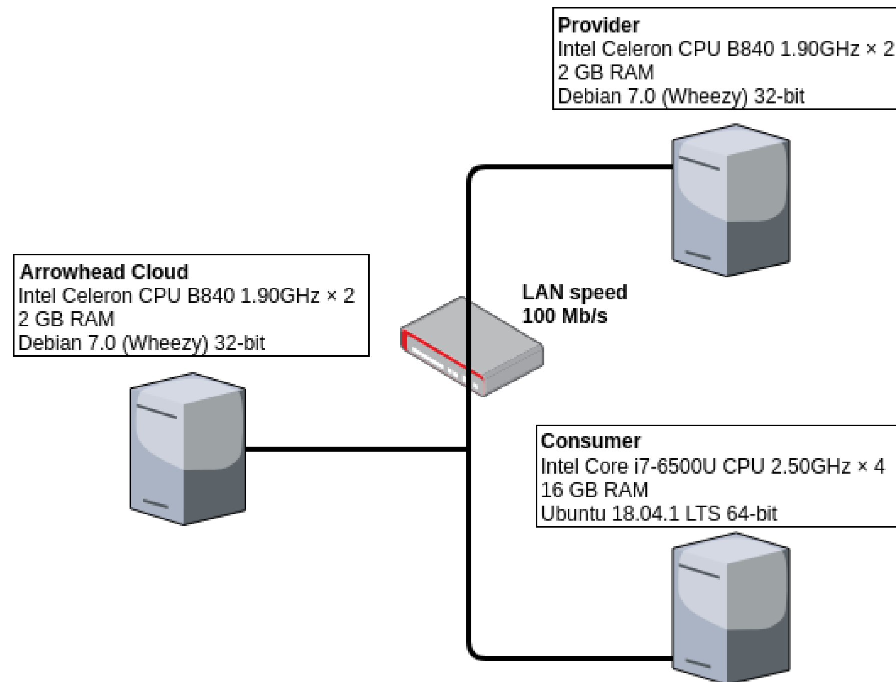


Figure 5.2: Testing environment for the intracloud orchestration

```
1  {
2    "requesterSystem" : {
3      "systemName" : "client1",
4      "address" : "192.168.60.108",
5      "port" : 8080,
6      "authenticationInfo" : "null"
7    },
8    "requestedService" : {
9      "serviceDefinition" : "IndoorTemperature",
10     "interfaces" : [ "json" ],
11     "serviceMetadata" : {
12       "unit" : "celsius"
13     }
14   },
15   "orchestrationFlags" : {
16     "onlyPreferred" : false,
17     "overrideStore" : true,
18     "externalServiceRequest" : false,
19     "enableInterCloud" : true,
```

```
20     "enableQoS" : false,
21     "matchmaking" : false,
22     "metadataSearch" : true,
23     "triggerInterCloud" : false,
24     "pingProviders" : false
25   },
26   "preferredProviders" : [ ],
27   "requestedQoS" : { },
28   "commands" : { }
29 }
```

Listing 5.3: Service request sent from the Consumer to the Orchestrator in JSON

After sending 100 requests to the Orchestrator, the average elapsed time until the Consumer receives an orchestration response to its request was approximately 598.12 ms, with standard deviation of 139.46 ms and with a maximum latency of 1530 ms. The corresponding latency for each orchestration response can be seen in Figure 5.3.

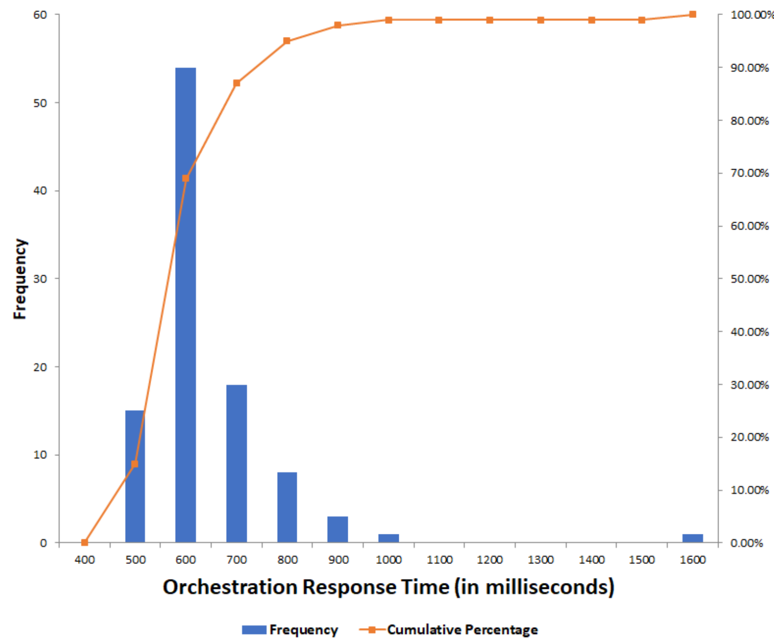
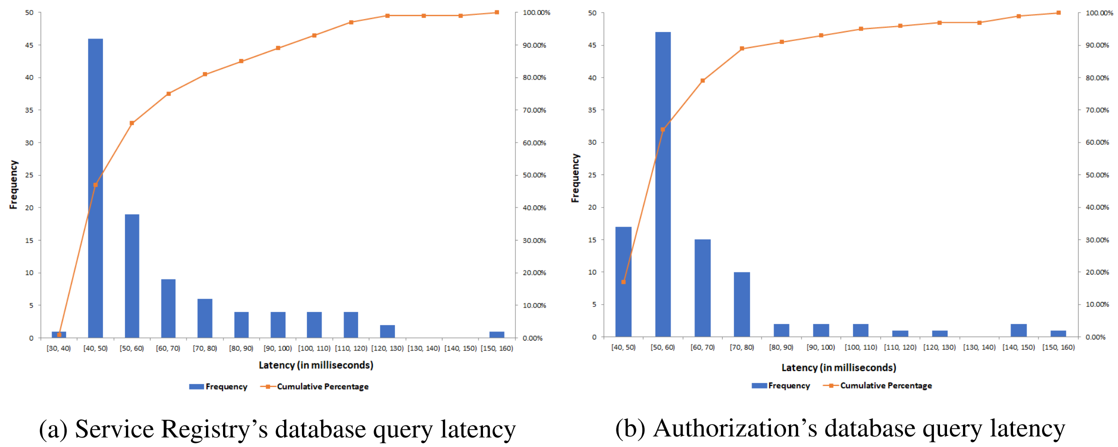


Figure 5.3: Intracloud orchestration latency

Furthermore, Service Registry's database queries took an average of 62.16 ms (with a maximum of 159 ms), while Authorization's queries took an average of 63.11 ms (with a maximum of 152 ms). Figure 5.4 shows the frequency distribution of these query latencies.



(a) Service Registry's database query latency (b) Authorization's database query latency

Figure 5.4: Frequency distribution of database query latency for Intracloud orchestration

An average orchestration latency of 598 ms does appear to be somewhat lengthy, given that this process consists of a simple service query and authorization validation on a private network with 0.597 ms of round-trip delay time (RTT). Furthermore, considering that all core Arrowhead systems are located inside the same machine, all requests between these systems are sent to localhost, so it would be expected for this to be a less prolonged process. As such, this amount of latency could be acceptable for some time-critical systems, however, the fact that some orchestration responses have a latency of more than one second could compromise the system's quality of service.

5.3 Intercloud Orchestration

With respect to the intercloud communication, the orchestration process starts the same way as in the intracloud process, however in this scenario the Arrowhead cloud (hereinafter identified, along with its systems, with the letter "A") does not have the requested service in Service Registry A. Thus, Orchestrator A will turn to the cloud's Gatekeeper system (Gatekeeper A) and start a Global Service Discovery (GSD).

In a GSD, Gatekeeper A will contact other registered neighbor clouds' Gatekeepers and request for service X. If any of these clouds send a positive confirmation, then Gatekeeper A saves the cloud's address in a list. After a GSD, Gatekeeper A will thus send to Orchestrator A a list of Arrowhead clouds that provide the requested service. In case the consumer specified a preferred provider, the Orchestrator returns its address if its in the list. Otherwise, if the preferred provider is not on the list or if the consumer did not specify a preferred provider, the Orchestrator returns the first one in the list. Either way, the consumer is then able to connect to a service provider from a different cloud (in other words, Cloud B's services).

The full orchestration process is displayed on Figure 5.5. In addition, a demonstration of this scenario was created and hosted on a Git repository for future reference [1].

[1] URL of the Git Repository: `https://github.com/Rafa-Rocha/arrowhead-intercloud-rest-demo`

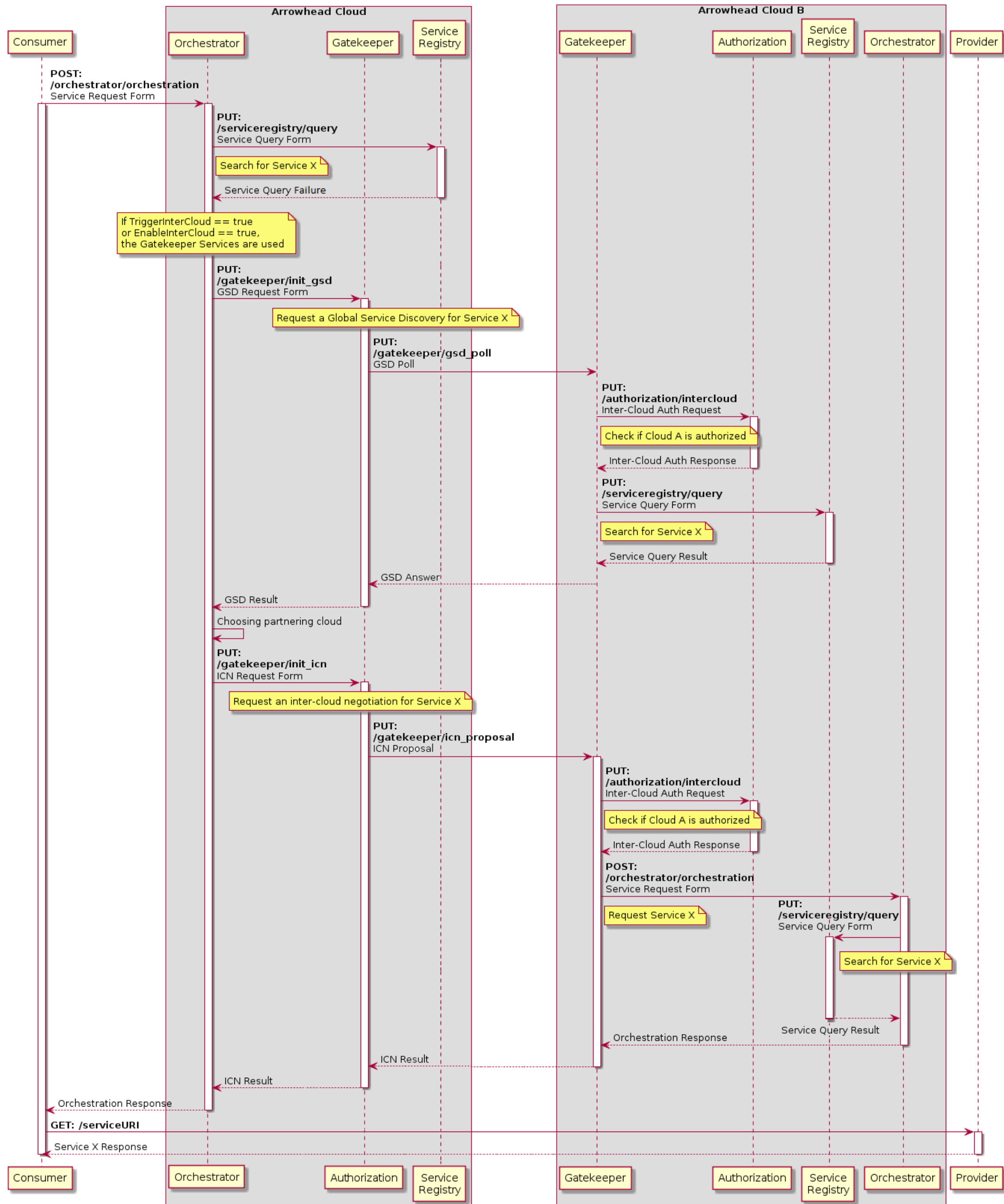


Figure 5.5: Sequence diagram of the Intercloud orchestration

In relation to the testing environment, similarly to intracloud's testing environment, all parties – including the neighbor clouds – were deployed on different machines (see Figure 5.6). In terms of the messaging workload, 100 requests (with the same contents as Listing 5.3) were sent to the cloud's Orchestrator, with a one second delay between each message sent. Since the requests have the same content as before, each is around 700 bytes long, sent on a 100 Mb/s Switched Ethernet LAN. All system clocks were also synchronized using a local NTP server.

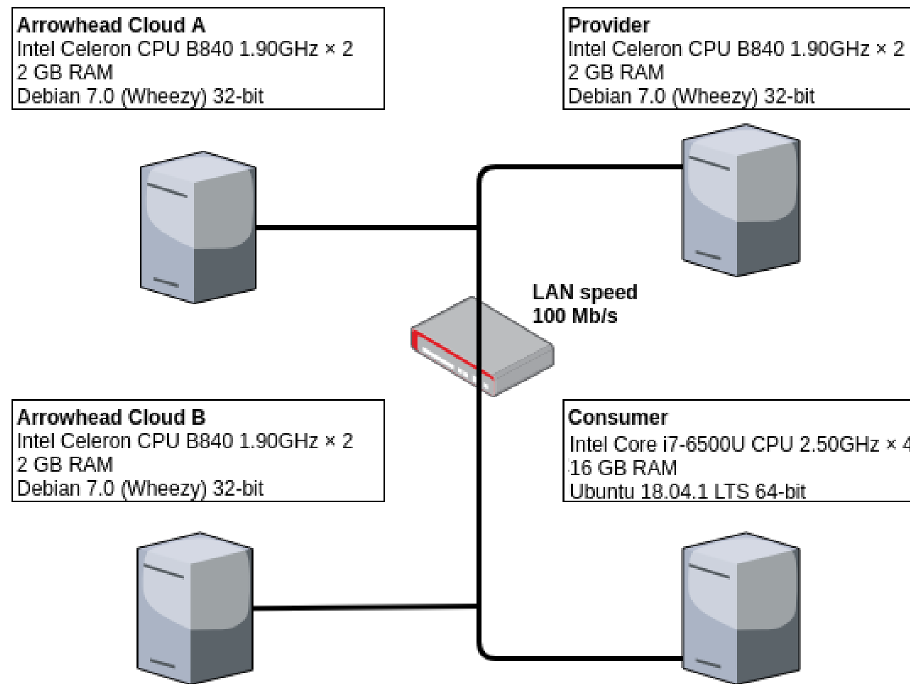


Figure 5.6: Testing environment for the intercloud orchestration

After sending 100 requests to Cloud A's Orchestrator, the average elapsed time until the Consumer receives an orchestration response to its request was approximately 4566.6 ms, with standard deviation of 933.6 ms and with a maximum latency of 10103 ms. The corresponding latency for each service request can be seen in Figure 5.7. Moreover, Service Registry A's database queries took an average of 225.3 ms (with a maximum of 4165 ms), while Service Registry B and Authorization B's queries took an average of 55.2 ms (with a maximum of 114 ms) and 59.42 ms (with a maximum of 187 ms), respectively.

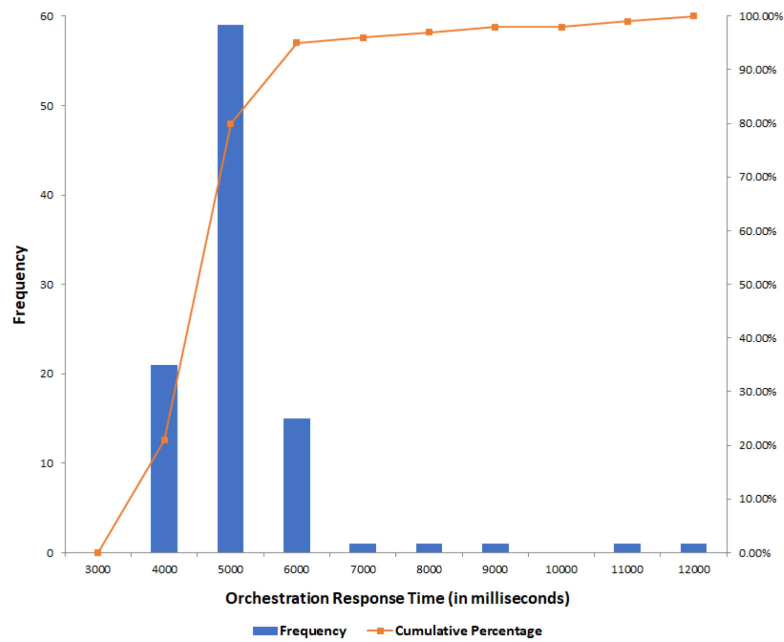


Figure 5.7: Intercloud orchestration latency

Given that these tests were employed on a private network and with databases that only have around 60 entries each, for the full orchestration process to take more than 1500 ms and averaging the 4000 ms is telling that there is a bottleneck in the request chain, and it is more than likely related to how these Arrowhead systems deal with incoming requests and their execution (as was pointed out in Section 5.1).

Chapter 6

Modeling the performance of Arrowhead's Intracloud and Intercloud orchestration

An important goal of this work consisted in modeling the performance of Arrowhead's two types of cloud orchestration: intracloud and intercloud. This, in return, would be highly valuable for the Arrowhead community since it would then be possible to visualize potential bottlenecks in any of the framework's core systems and to officially have a representation of the framework's performance. In regard to what kind of models should be chosen for this process, as Section 4.2 has shown, Petri net models are indicated for representing stochastic network scenarios. Thus, it was decided that Arrowhead's performance should be modeled through Petri nets.

Petri nets are usually utilized to model small and specific situations, however for this case the goal is to represent the performance of the whole orchestration process of a system of systems, thus the same type of granularity applied to typical Petri net examples cannot be replicated to this project's scenario. As such, the places used in the following Petri nets represent a full Arrowhead system, while the transitions between each system represent request/response latency or database query execution times.

For the stochastic analysis of the model, it was decided to use Oris Tool [81] since it was one of the only software tools that was found which provided analysis tools for stochastic Petri nets. GreatSPN [34] was also another possible tool that was experimented with, however, Oris's tools were preferred to GreatSPN's, because of its user-friendly interface and easier to understand analysis results, and also because of its more varied stochastic transitions (i.e. Erlang distribution transitions, instead of only exponential and fixed time transitions).

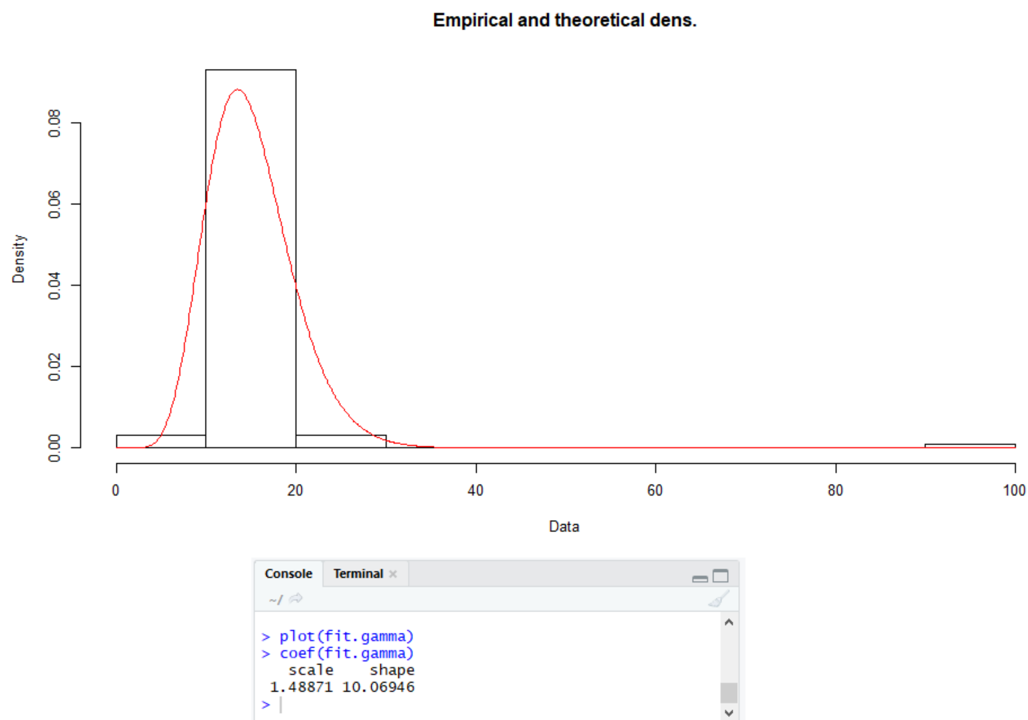
While on the subject of transitions, the following Petri nets use two different types of stochastic transitions, namely transitions that follow an Erlang probability distribution (named "Erlang transitions" or "Erl transitions") and transitions that follow an exponential probability distribution

(named "Exponential transitions" or "Exp transitions"). However, there are some cases where latency or execution times follow a Gamma distribution instead. Unfortunately, for these types of distributions, Oris only provides Erlang transitions – whose distribution is already based on the Gamma distribution, with the main difference being that Erlang's shape (k) parameter has to be an integer value, instead of a real number. Thus, considering this limitation for cases where a Gamma distribution is present, there was no other way than to use the Erlang transition and round the shape parameter to an integer.

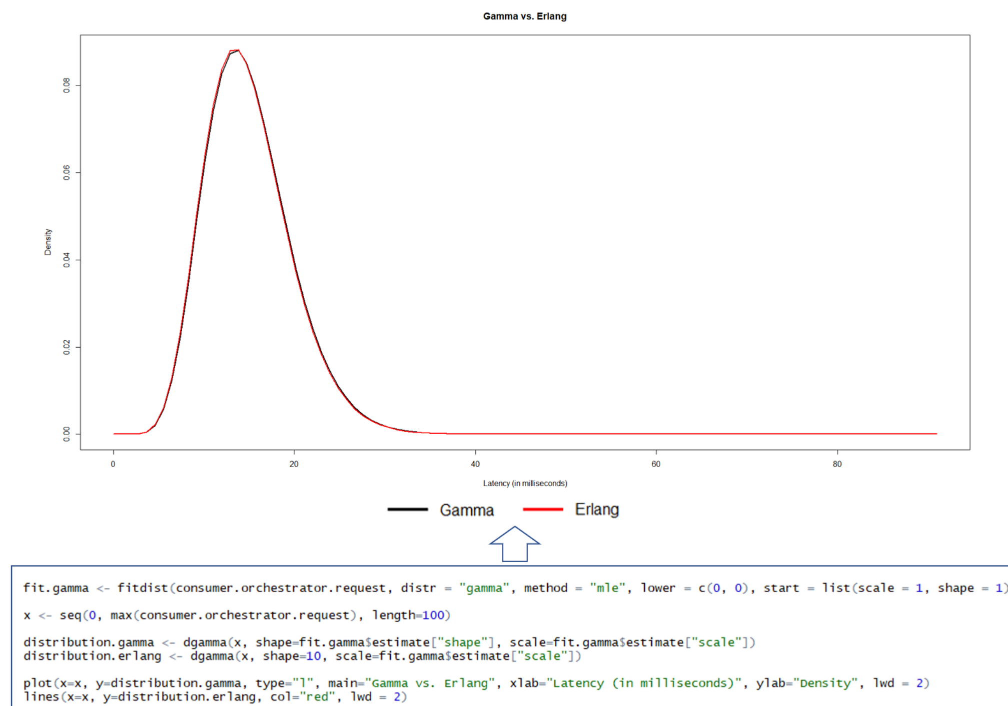
Moreover, in order to identify the type of probability distribution (and its parameter values) that fit the latency distribution of each request/response/database query, an R package called "*fitdistrplus*" was used. This package contains a function named "*fitdist*", which attempts to fit a probability distribution to a series of data (in other words, probability distribution fitting). Unfortunately, *fitdist* does not support Erlang or Exponential distributions, however it does support the Gamma distribution. Thus, given that both Erlang and Exponential distributions are derived from the Gamma distribution, *fitdist* was used to calculate the optimal parameters that fit a Gamma distribution to the latency data (see Listing 6.1). After checking the calculated distribution, the transition is then chosen based on the distribution's shape (see Figures 6.1a and 6.2a). We found that this approximation with either Exponential or Erlang distributions was very close to the original Gamma distribution, as seen in Figures 6.1b and 6.2b.

```
1  library(fitdistrplus)
2  library(logspline)
3
4  consumer.orchestrator.request <- c(15,17,20,17,17,18,17,16,20,16,15,
5                                     17,16,15,18,17,17,16,15,91,16,14,
6                                     14,16,15,14,15,12,15,14,12,15,14,
7                                     13,14,15,15,14,13,10,15,15,13,13,
8                                     14,15,13,13,13,14,14,14,14,14,14,
9                                     16,21,14,13,13,13,16,23,13,13,12,
10                                    15,13,12,13,13,12,12,13,13,12,13,
11                                    12,13,13,10,11,12,14,13,13,11,11,
12                                    12,12,11,13,11,12,12,13,11,9,29)
13
14 # using the option "lower" to impose limits on the parameter search space
15 # see link for further details: https://stats.stackexchange.com/questions/158163/
       why-does-this-data-throw-an-error-in-r-fitdistr
16
17 fit.gamma <- fitdist(consumer.orchestrator.request, distr = "gamma",
18              method = "mle", lower = c(0, 0), start = list(scale = 1, shape = 1))
19
20 summary(fit.gamma)
21 plot(fit.gamma)
22 coef(fit.gamma)
```

Listing 6.1: Code snippet in R demonstrating the use of fitdist for distribution fitting on the Consumer-Orchestrator request latency data.

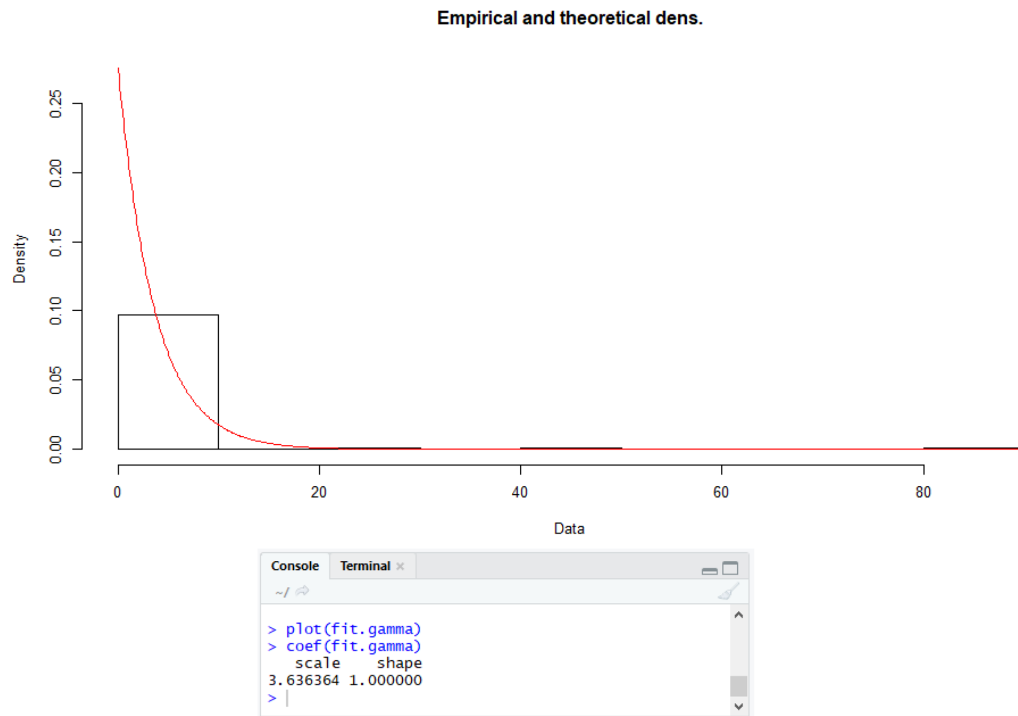


(a) Consumer-Orchestrator request latency distribution with an Erlang-like shape. Therefore, for the Petri net model, this type of data is represented by using an Erlang transition with *fitdist*'s estimated parameter, while rounding the shape parameter to an integer (in this case, 10).

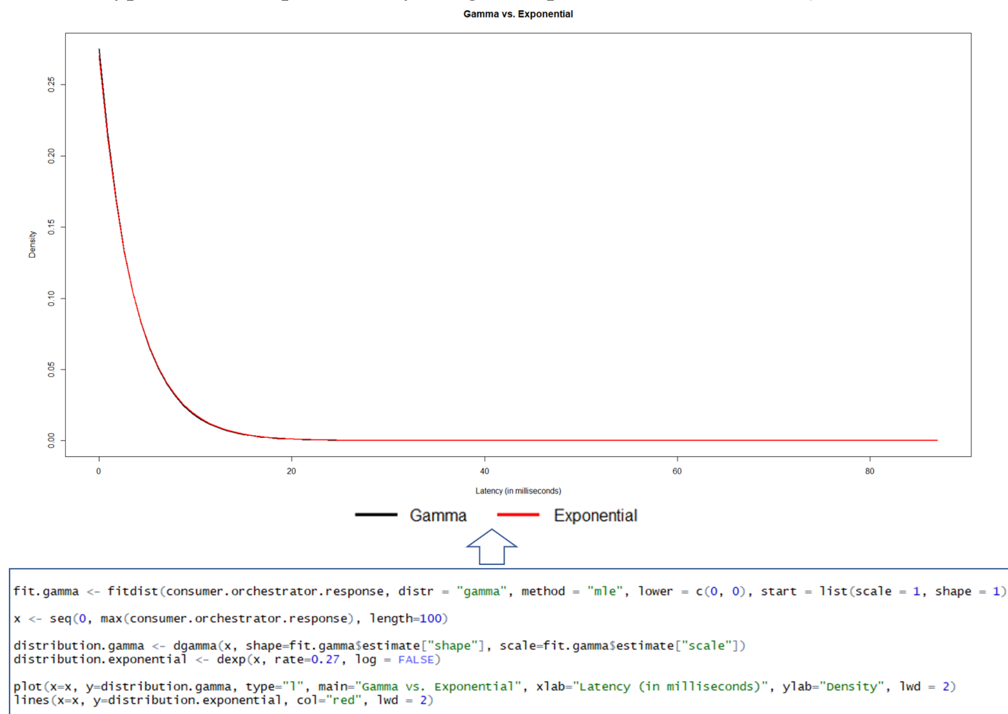


(b) A comparison between the Gamma and Erlang distributions. Gamma uses *fitdist*'s estimated parameters, while Erlang rounds *fitdist*'s shape parameter to an integer (in this case, 10 instead of 10.0646).

Figure 6.1: Example of the decision process for choosing an Erlang probability distribution, based on the data distribution, for a stochastic transition.



(a) Consumer-Orchestrator response latency distribution with an exponential-like shape. Thus, for the Petri net model, this type of data is represented by using an Exponential transition with *fitdist*'s estimated values.



(b) A comparison between the Gamma and Exponential distributions. Gamma uses *fitdist*'s estimated parameters, while Exponential only uses *fitdist*'s rate parameter (in this case, 0.27, since the rate is the reciprocal of the scale parameter).

Figure 6.2: Example of the decision process for choosing an Exponential probability distribution, based on the data distribution, for a stochastic transition.

Admittedly, this is not the most correct way of representing a distribution, given that there might exist some discrepancies between the observed values and the values expected under the model in question (which can be observed using the "goodness of fit" test [25]). Thus, the predicted latency simulated through the Petri net's stochastic analysis will never be completely representative of the actual latency. However, given Oris's limitations and the fact that no other Petri net tool supports other distribution transitions and are not able to perform a transient stochastic analysis (which will be further explained in Section 6.1.2) like Oris, this decision was deemed acceptable for the purposes of this thesis.

6.1 Intracloud Orchestration

For the intracloud orchestration process, the developed Petri net is displayed in Figure 6.3, while a detailed explanation of its logic is further explained in Section 6.1.1, and finally its stochastic analysis is presented in Section 6.1.2.

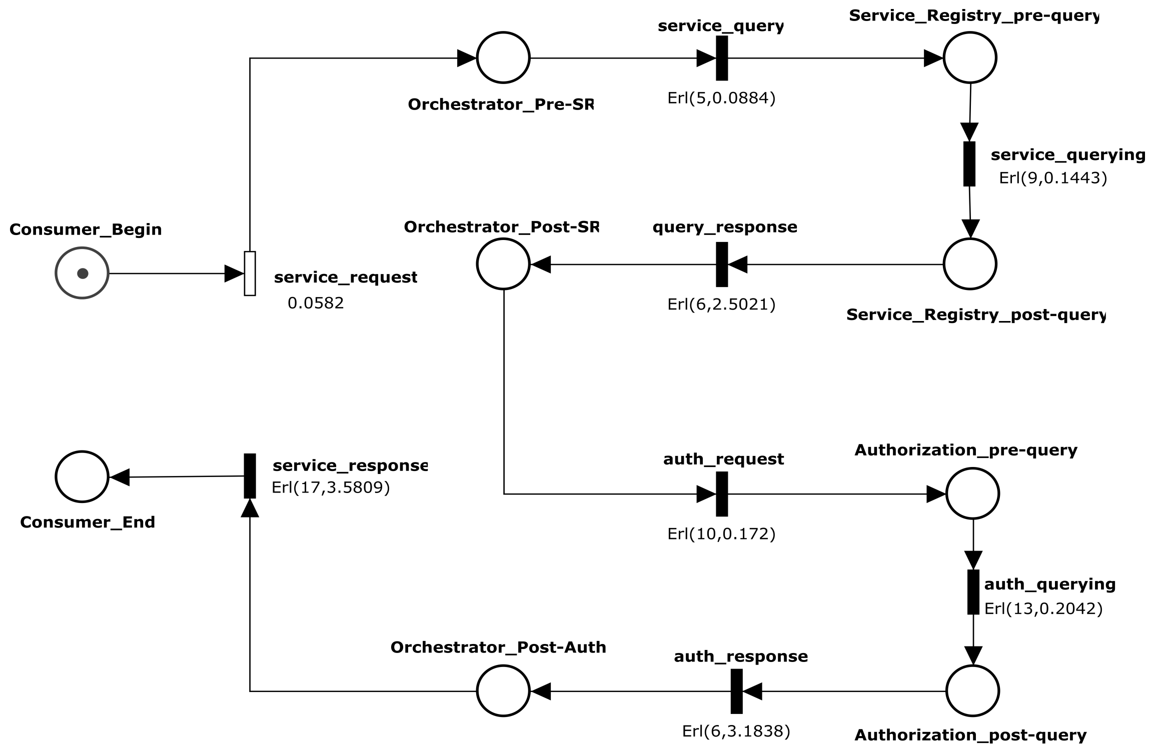


Figure 6.3: Stochastic Petri net model of Arrowhead's Intracloud Orchestration

6.1.1 Explaining the Petri net model for the Intracloud Orchestration

Starting with the Consumer system, this system is represented in the Petri net by two different places: *Consumer_Begin* and *Consumer_End*. This decision was made in order to easily identify the two Consumer states for the Petri net's stochastic analysis. In fact, this same state logic was

used for the other Arrowhead systems as well. As such, the Orchestrator was divided into three places: Pre-SR (i.e. before sending a request to Service Registry), Post-SR (i.e. after sending a request to Service Registry and before sending a request to Authorization), and Post-Auth (i.e. after sending a request to Authorization and before sending a response to Consumer). Similarly, both the Service Registry and Authorization have a pre- and post-query place (i.e. before and after executing a database query). Figure 6.4 shows the same sequence diagram as in Figure 5.1, but with some annotations identifying the transitions used in the Petri net model.

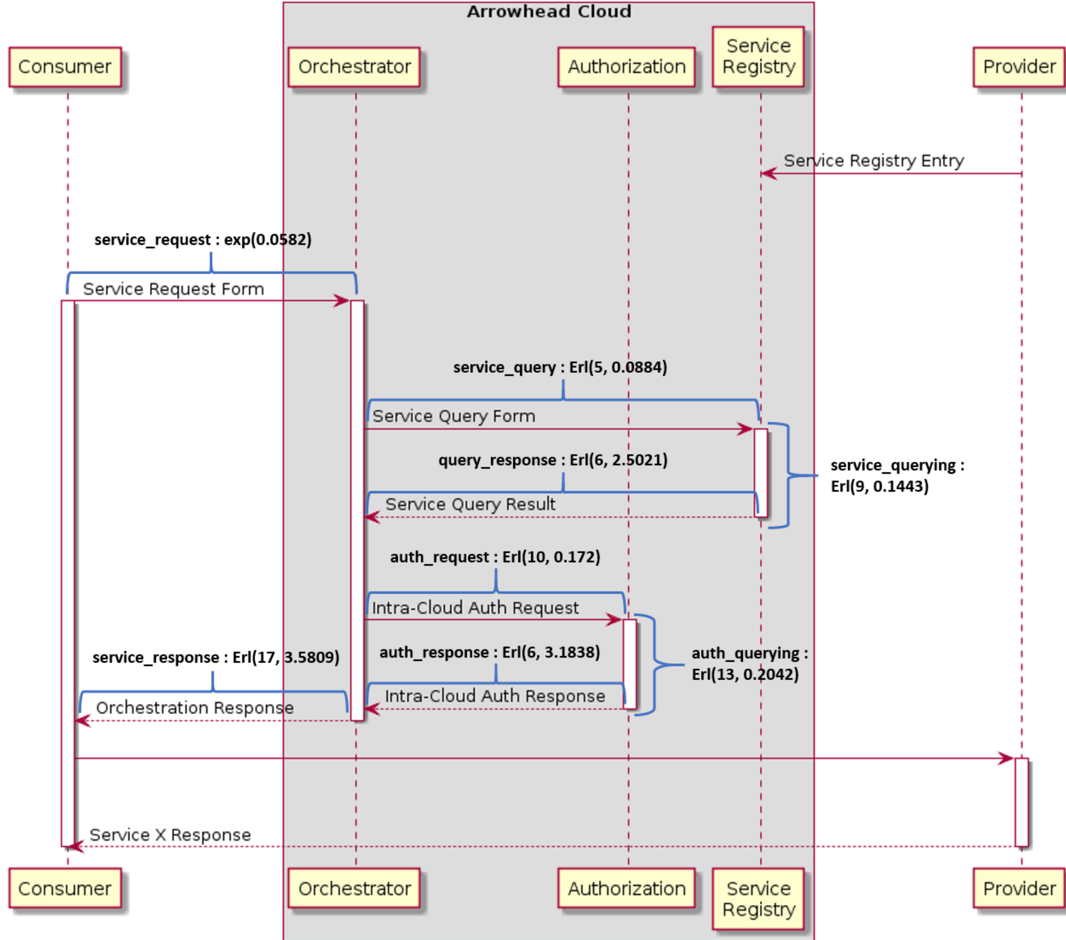


Figure 6.4: Sequence diagram of the Intracloud orchestration with annotations identifying the transitions used in the Petri net model

Regarding the transitions used in the model, the ones in black are Erlang transitions, while the ones in white are Exponential transitions. Thus, when the Consumer sends a service request to the Orchestrator, the time it takes for a request to reach the Orchestrator has an exponential distribution with a rate (λ) equal to 0.0582. Next, the Orchestrator sends a request to the Service Registry, where the request's latency has an Erlang distribution with shape 5 and rate 0.0884. The Service Registry then executes a database query to look for the requested service. This query's execution time has an Erlang distribution with shape 9 and rate 0.1443. The Service Registry

then sends its response to the Orchestrator, which has an Erlang distribution with shape 6 and rate 2.5021. The Orchestrator then sends a request to Authorization to validate if Consumer can have access to the service. This request's latency has an Erlang distribution with shape 10 and rate 0.172. The Authorization executes a database query, whose execution time has an Erlang distribution with shape 13 and rate 0.2042. Next, the Authorization responds to the Orchestrator. This authorization response's latency has an Erlang distribution with shape 6 and rate 3.1838. Finally, the Orchestrator then sends a service response to the Consumer, where the network latency has an Erlang distribution with shape 17 and rate 3.5809.

6.1.2 Stochastic analysis of the Petri net model for the Intracloud orchestration

Regarding the stochastic analysis, Oris provides a tool for transient analysis which consists in analyzing the probabilities of a process transitioning from one place to the other at a specific instant in time. Thus, the analysis creates a matrix – in which the "time" variable is the row and the "possible arrival state" variable is the column, where each row represents a probability distribution for that instant in time, which means that the sum of all values in each row must equal 1. A chart was then created via the matrix's analysis data – with "time" as the X-axis and "place probability" as the Y-axis – and is displayed in Figure 6.5.

In relation to the analysis results, first, the *Consumer_Begin* place starts with a probability of 1, since it is the only place that contains a token at that initial instant. However, as time goes on, its probability starts to decrease non-linearly until it reaches 0. While *Consumer_Begin*'s probability decreases, *Orchestrator_Pre-SR*'s probability begins to rise, because it is the next place in the sequence. This pattern then repeats itself for the other places in the Petri net. Furthermore, another pattern can also be observed: right after a place probability peaks at a certain instant (e.g. *Orchestrator_Pre-SR* peaks at the 35.10 ms instant) the probability curves of its previous place and its next place cross each other, respectively decreasing and increasing non-linearly (e.g. around 3.9 ms after *Orchestrator_Pre-SR* peaks, *Consumer_Begin* starts having fewer probabilities of still holding a token than *Service_Registy_Pre-Query*). Moreover, it should also be noted that *Service_Registy_Post-Query* and *Orchestrator_Post-Auth* have very low overall probabilities because these two are surrounded by faster transitions than the rest of the other places. Thus, the token will spend less time inside these places, which consequently lowers the probability of these places to hold a token for a long time. Finally, as the token transitions from one place to the next, the *Consumer_End*'s probability becomes the highest of all (i.e. 0.443) at the 258 ms instant, and it increases until it ultimately reaches approximately 1 (at the 448.60 ms instant), i.e. once there is almost 0% probability of any other system to still be holding the token.

If comparing these results to the actual data, there is a 25% difference between the estimated time to 100% receive an orchestration response (i.e. approximately 448.60 ms) and the actual average time (i.e. 598.12 ms). In other words, the estimation misses the actual data by 149.52 ms.

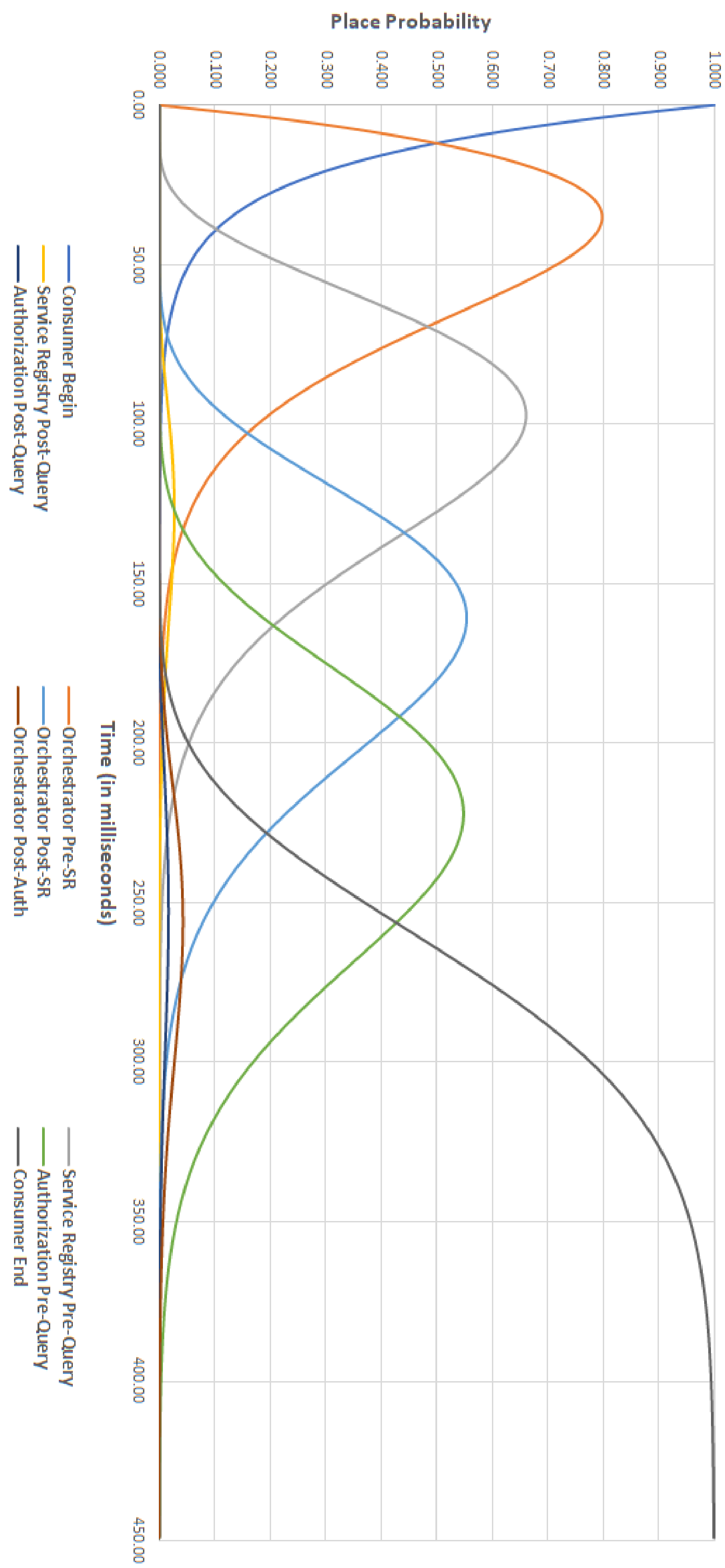


Figure 6.5: Transient analysis of the Petri net model for the Intracloud orchestration

6.2 Intercloud Orchestration

Regarding the intercloud orchestration process, the developed Petri net is displayed in Figure 6.6. In Section 6.2.1 we explain this model and the results of its stochastic analysis are presented in Section 6.2.2.

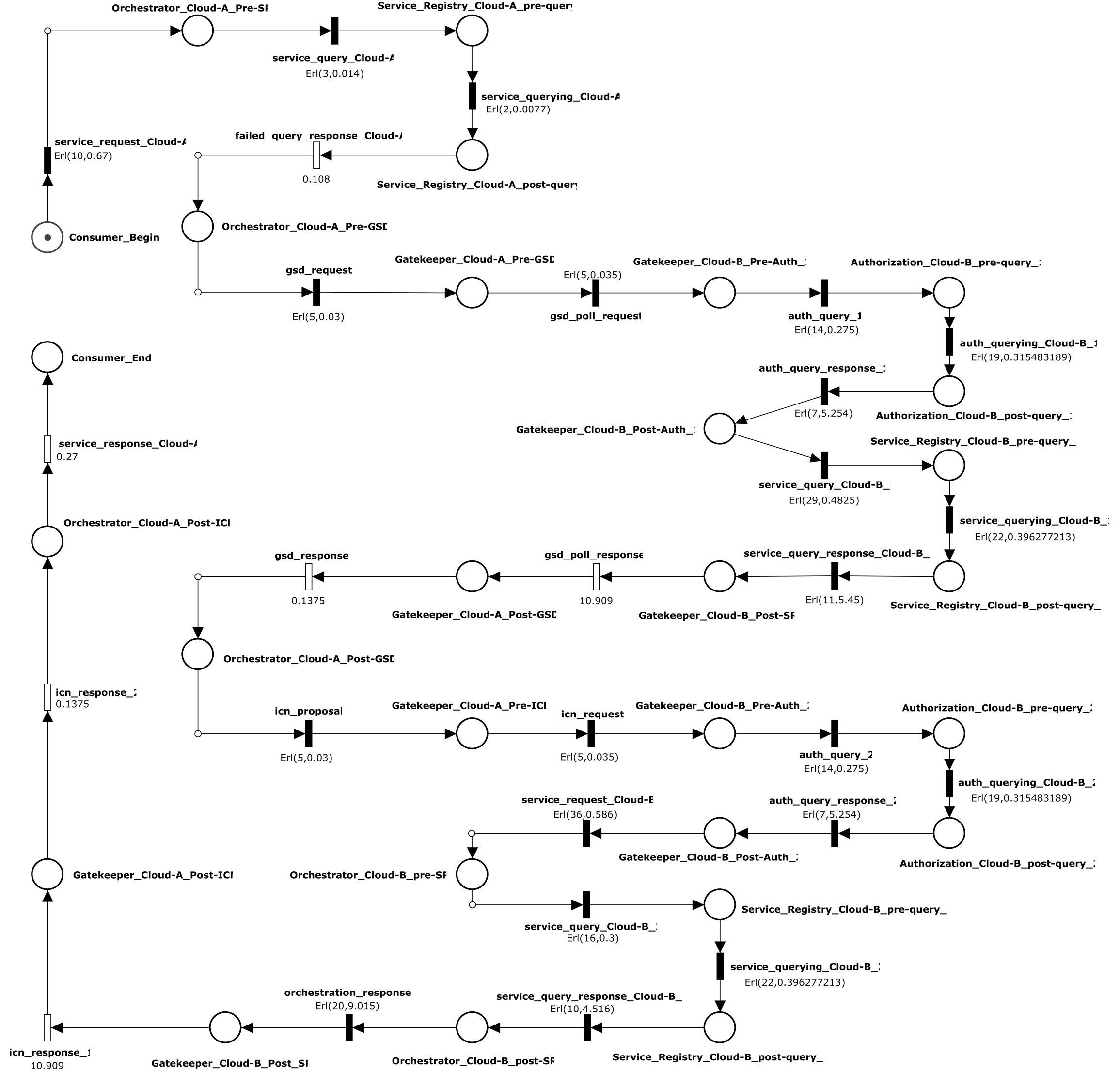


Figure 6.6: Stochastic Petri net model of Arrowhead's Intercloud Orchestration

6.2.1 Explaining the Petri net model for the Intercloud Orchestration

Given the nature of the orchestration process, the Petri net for the Intercloud orchestration shares many similarities to the Intracloud's Petri net. As such, the same logic of dividing Arrowhead systems by their states was used. The only real difference is that the number of required system interactions for intercloud orchestration is more than triple the ones for intracloud orchestration,

thus more places are necessary. Similar to Section 6.1.1, Figure 6.7 shows the same sequence diagram as in Figure 5.5, however with some annotations identifying the transitions used in the Petri net model.

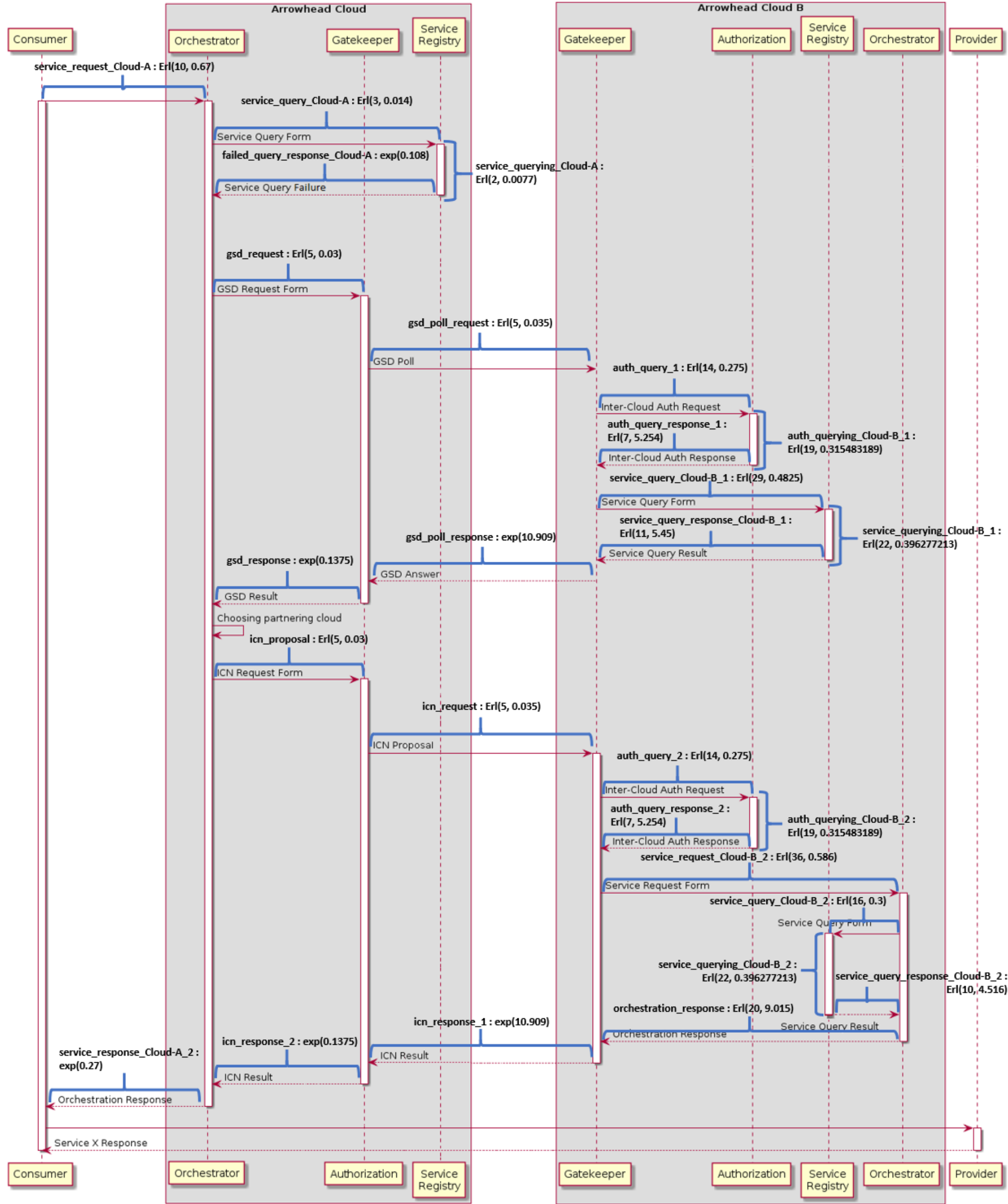


Figure 6.7: Sequence diagram of the Intercloud orchestration with annotations identifying the transitions used in the Petri net model

Initially, the process is similar to the intracloud orchestration: Consumer sends a service request to the Orchestrator A (Erlang distribution with shape 10 and rate 0.67) and Orchestrator A sends a request to the Service Registry A (Erlang distribution with shape 3 and rate 0.014). However, Service Registry A then executes a database query which fails to find the required service (Erlang distribution with shape 2 and rate 0.0077) and sends its response to Orchestrator A

(exponential distribution with rate 0.108). Orchestrator A sends a GSD request to Gatekeeper A (Erlang distribution with shape 5 and rate 0.03), and thus Gatekeeper A sends a GSD request to Gatekeeper B (Erlang distribution with shape 5 and rate 0.035).

Gatekeeper B then executes the usual authorization and service registry queries. First, Gatekeeper B sends a authorization query to Authorization B (Erlang distribution with shape 14 and rate 0.275) to check if Cloud A is authorized, Authorization B executes its database query (Erlang distribution with shape 19 and rate 0.315) and then returns the response to Gatekeeper B (Erlang distribution with shape 7 and rate 5.5254). Gatekeeper B then sends a service query to Service Registry B (Erlang distribution with shape 29 and rate 0.4825), Service Registry B executes its query (Erlang distribution with shape 22 and rate 0.396) and sends its response to Gatekeeper B (Erlang distribution with shape 11 and rate 5.45).

Gatekeeper B then sends its GSD response to Gatekeeper A (exponential distribution with rate 10.909). Gatekeeper A sends the GSD response to Orchestrator A (exponential distribution rate 0.1375). Orchestrator sends it ICN proposal to Gatekeeper A (Erlang distribution with shape 5 and rate 0.03), and Gatekeeper A sends it to Gatekeeper B (Erlang distribution with shape 5 and rate 0.035). Next, Gatekeeper B and Authorization B execute the same authorization process, with the same probability distributions. Once Gatekeeper B receives the authorization response, it sends a orchestration request to Orchestrator B (Erlang distribution with shape 36 and rate 0.586). Orchestrator B then sends a service query to Service Registry B (Erlang distribution with shape 16 and rate 0.3). Service Registry executes the same database query as before (with the same probability distribution), and sends its response to Orchestrator B (Erlang distribution with shape 10 and rate 4.516). Orchestrator B then sends the orchestration response to Gatekeeper B (Erlang distribution with shape 20 and rate 9.015).

Afterwards, Gatekeeper B sends the ICN response to Gatekeeper A (exponential distribution with rate 10.909), who then sends it to Orchestrator A (exponential distribution with rate 0.1375). Finally, Orchestrator A sends a service response to Consumer (exponential distribution with rate 0.27).

6.2.2 Stochastic analysis of the Petri net model for the Intercloud orchestration

Regarding the stochastic analysis results (which are displayed in Figure 6.8), a few initial patterns are similar to Intracloud's. However, because the latency is much higher for some of the requests/responses, the probability curves for some of the Petri net's places are also much wider. Since there is a whole slew of different curves (which some are even hard to discern in the chart), not all states will be heavily described here, seeing that a detailed analysis on each probability curve would not bring much value to these observations. Nonetheless, this overview will still describe the major points of the analysis results.

First, just like before, the *Consumer_Begin* place starts with a probability of 1, and as time goes on, its probability starts to decrease non-linearly until it reaches 0. While *Consumer_Begin*'s probability decreases, *Orchestrator_Cloud-A_Pre-SR*'s probability begins to rise, because it is the next place in the sequence. *Orchestrator_Cloud-A_Pre-SR* keeps hold of the token for around

202 ms, until its probability curve starts to get lower than the one from the next place in the sequence, *Service_Registy_Cloud-A_Pre-Query*. The place afterwards, *Service_Registy_Cloud-A_Post-Query*, has a very low probability curve because it is surrounded by faster transitions than the rest of the other places. Subsequently, the place probabilities for *Orchestrator_Cloud-A_Pre-GSD*, *Gatekeeper_Cloud-A_Pre-GSD*, *Orchestrator_Cloud-A_Post-GSD*, and *Gatekeeper_Cloud-A_Pre-ICN* stand out from the rest, since these have more significant latency distributions. Finally, after the 1422 ms instant, *Consumer_End*'s probability becomes the highest one in the group and continues to rise non-linearly until it reaches 1, at around the 2867 ms instant.

If comparing these results to the actual data, there is a 37% difference between the estimated time to 100% receive an orchestration response (i.e. approximately 2867 ms) and the actual average time (i.e. 4566.6 ms). In other words, the estimation misses the actual data by 1699.6 ms.

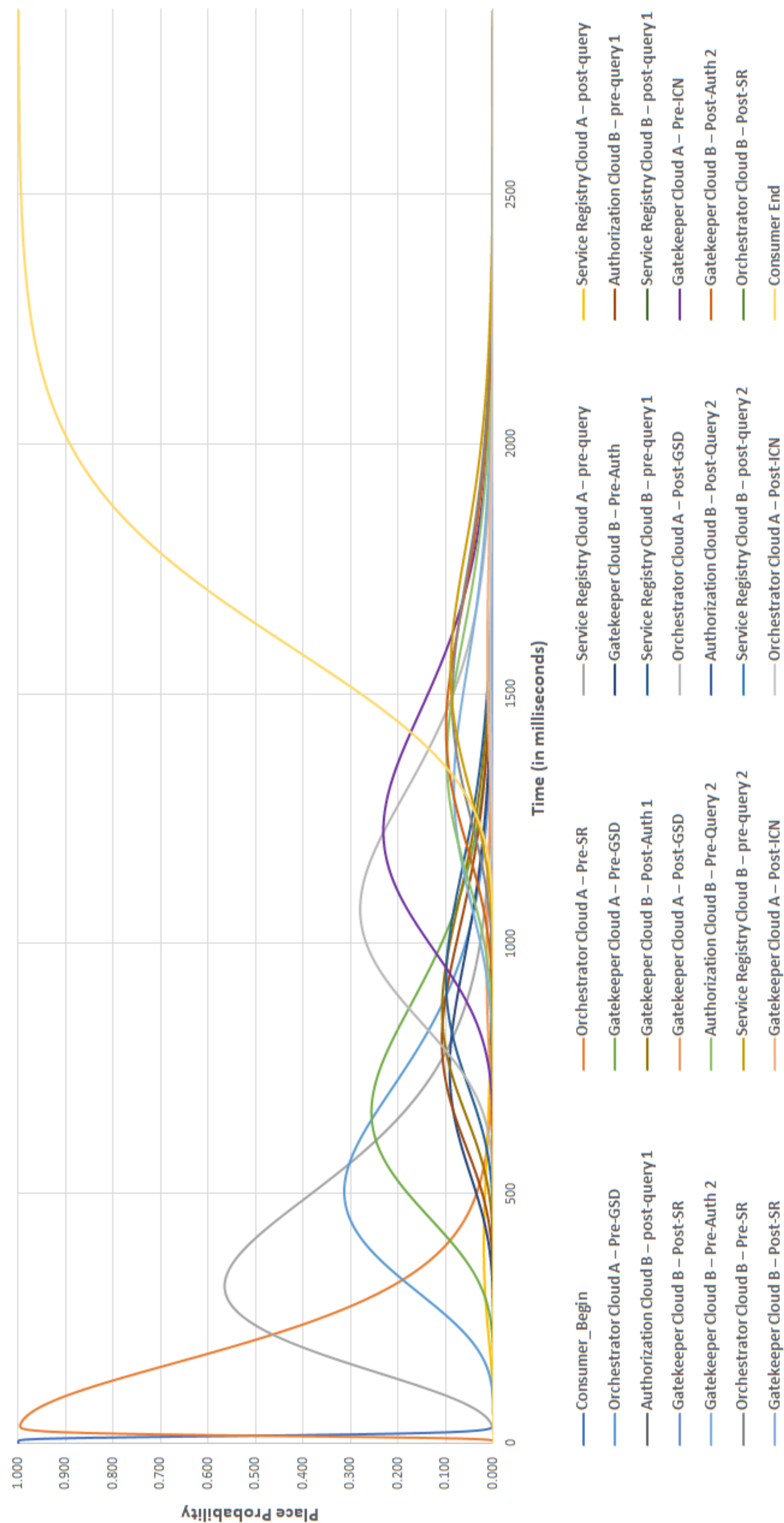


Figure 6.8: Transient analysis of the Petri net model for the Intercloud orchestration

Chapter 7

Improving the performance of the Event Handler

For systems that require low response times, the type of latency that was experienced in the orchestration tests may not be acceptable in an industrial context. Nonetheless, this can be improved, however it would be necessary to change each Arrowhead system's specific implementation since each system was developed for a particular use case. As such, since the Arrowhead Framework is a very large project, it would not be realistically feasible to change the entire framework, given that the rest of the Productive4.0 teams are also working on their own Arrowhead systems.

However, another Arrowhead system (i.e. the Event Handler) was also the target of this thesis' performance evaluation. However, because its performance results were so poor (as will be shown in this chapter), we decided to redesign it and change its implementation accordingly to improve its performance, by using appropriate software configurations and design patterns. Given that some of these bottlenecks were caused by design decisions that also permeate other Arrowhead systems, this reengineering process served as a case study for this dissertation to explore potential performance improvements for the rest of the framework's components.

7.1 The Event Handler

The Arrowhead core systems are accompanied by automation supporting services that further improve the core capabilities of a local cloud, from measuring quality of service to enabling message propagation between multiple systems. The Event Handler is one of these optional supporting systems. The Event Handler, is used to propagate updates from a producer service to one or more consumer applications. In this sense, the Event Handler serves as a REST/HTTP(S) implementation of a publish-subscribe message broker, handling the distribution of messages (or events) from publishers to multiple subscribers (as shown in Figure 7.1).

For an Arrowhead publisher service to continuously notify its subscribers within its performance requirements, the Event Handler's performance is obviously of extreme importance. There are two important performance parameters to take into account in a publish-subscribe setting: i) the end-to-end delay for a message to go from a producer to a consumer; and ii) the message throughput. i.e., the number of messages which can be sent per time unit and processed by the Event Handler. These two performance parameters are evaluated in this work.

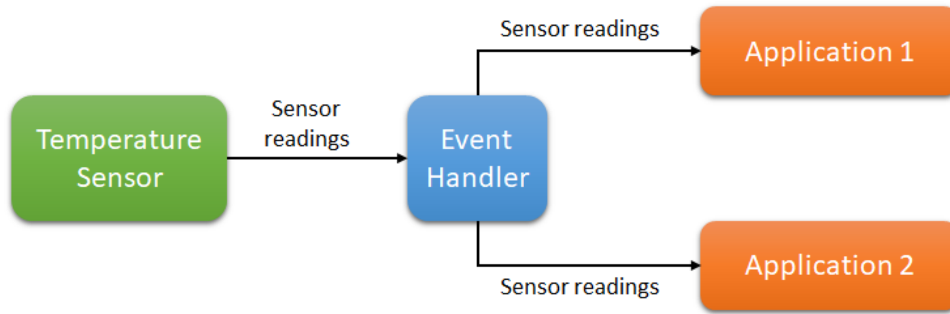


Figure 7.1: A simplified representation of the Event Handler system

7.2 Original Implementation

The Event Handler's implementation in the official Arrowhead repository [18] uses the same Jersey/Grizzly setup as the other Arrowhead systems. Once again, the Grizzly HTTP server module in the Event Handler does not currently have a configured thread pool. Thus, as explained in Section 5.1, the system will most likely not be able to efficiently handle multiple requests. Moreover, for the client applications that are meant to use the Event Handler, i.e. the publishers and subscribers, the Arrowhead Consortia provides client skeletons to be extended with the developers' own application code [19]. These client skeletons also use the same Jersey/Grizzly setup and server configuration as the Arrowhead systems, since these are also web applications.

7.3 Enhancements

This Section first describes the reengineering process that was performed on the Event Handler, explaining each problem, the design change required to solve the issue, and finally its implementation. The final result is available in a Git repository[1].

Motivated by the poor performance of the original Event Handler, we analyzed the code of the Publisher, the Event Handler and the Subscriber, and detected two major problems. The first problem was that none of the three components reused HTTP connections. This has a major performance impact on communications, since establishing a connection from one system to another consists of multiple packet exchanges between two endpoints (i.e. connection handshaking),

[1]URL for the Git repository: https://github.com/Rafa-Rocha/arrowhead-improved-event-handler

which can cause major overhead, especially if each connection is used to exchange a single small HTTP message [40]. In fact, a much higher data throughput is achievable if open connections are reused to execute multiple requests. This problem required a different solution for the three systems:

- The Publisher had to use a connection pool so that it could reuse its connections to the Event Handler (see Section 7.3.1);
- The Event Handler had to use Jersey's own Server-Sent Events mechanism to establish a persistent connection to each of its Subscribers (see Section 7.3.2).

The second problem was the thread policy used by the Event Handler, which created a new thread for every incoming request, which would then greatly impact the machine's available RAM and response times. Thus, the Event Handler required a thread pool to manage incoming requests in a less wasteful manner, as threads can be reused among different requests (see Section 7.3.3).

7.3.1 Reuse open connections between the Publisher and the Event Handler

In order to reuse open connections between the Publisher and the Event Handler, the best choice was to implement a connection pool on the Publisher, via the Apache HTTP Client on Jersey's transport layer (see Listing 7.1). On an Apache HTTP Client [40], the client can maintain a maximum number of connections on a per endpoint basis (which can be configured), so a request for an endpoint for which the client already has a persistent connection available in the pool will be handled by reusing a connection from the pool rather than creating a brand-new connection.

In the new version of the Publisher, we configured the connection pool to a single connection per destination. Only one connection per route was set in order to maintain message order, since using multiple parallel connections might lead to the processing of messages out of sequential order. Additionally, in a typical Arrowhead deployment, published events are very small messages, therefore using more connections per destination would most likely not improve performance.

```
1  if (CONNECTION_POOL_SIZE > 0) {
2    // Connection Pooling
3    PoolingHttpClientConnectionManager connectionManager =
4                     new PoolingHttpClientConnectionManager();
5
6    connectionManager.setDefaultMaxPerRoute(CONNECTION_POOL_SIZE);
7
8    configuration.property(
9        ApacheClientProperties.CONNECTION_MANAGER,connectionManager);
10
11   configuration.connectorProvider(new ApacheConnectorProvider());
12 }
```

Listing 7.1: Configuring the connection pool on the Publisher's HTTP client.

7.3.2 Establish a persistent connection between the Event Handler and each Subscriber

The Event Handler also did not reuse previously created HTTP connections to its subscribers, consequently adding a large overhead on each message's end-to-end delay, due to the establishment of one connection per forwarded message. Thus, to avoid creating a connection to each subscriber on every request, the solution was to use Jersey's Server-Sent Events (SSE) [78] mechanism in the Event Handler.

The SSE mechanism can be used to handle a one-way publish-subscribe model. When the Subscriber sends a request to the Event Handler, the Event Handler holds a connection between itself and the Subscriber until a new event is published. When an event is published, the Event Handler sends the event to the Subscriber, while keeping the connection open so that it can be reused for the next events. The Subscriber processes the events sent from the Event Handler individually and asynchronously without closing the connection (see Listing 7.2). Therefore, the Event Handler can reuse one connection per Subscriber.

```
1  EventListener listener = new EventListener() {
2    public void onEvent(InboundEvent inboundEvent) {
3      System.out.println(
4        inboundEvent.readData(String.class) + " at "
5          + ZonedDateTime.now().toInstant().toEpochMilli());
6    }
7  };
8
9  for (String eventType : EVENT_TYPES) {
10   EventSource eventSource =
11       EventSource.target(new CustomWebTarget(target, eventType)).build();
12
13   eventSource.register(listener, eventType);
14
15   eventSource.open();
16
17   subscribedEvents.put(eventType, eventSource);
18 }
```

Listing 7.2: Subscriber's event listener for incoming events.

Subscribers that wish to listen to SSE events have to send a GET request to Event Handler's URI "/subscription", which is handled by the *subscribe(HttpHeaders)* resource method (see Listing 7.3). The method creates a new *EventOutput* representing the connection to the requesting subscriber.

```
1  @GET
2  @Path("subscription")
```

```java
@Produces(SseFeature.SERVER_SENT_EVENTS)
public EventOutput subscribe(@Context HttpHeaders httpheaders) {
  String eventType = httpheaders.getHeaderString("eventtype");

  final EventOutput eventOutput = new EventOutput();

  EventHandlerService.addSubscription(eventType, eventOutput);

  return eventOutput;
}
```

Listing 7.3: Event Handler's endpoint for subscribers to subscribe themselves on a type of event.

Afterwards, *addSubscription(eventType, eventOutput)* registers this *eventOutput* instance with a broadcaster instance, using its add(eventOutput) method. Naturally, an *SseBroadcaster* is not bound to any message topic, because it sends all messages to all subscribers, which in this case is not ideal because the Event Handler has to deal with different message topics (in this case, event types) that some systems might be subscribed to, but others might not. Thus, it was necessary to create a *Map* object to store all *SseBroadcasters* by their corresponding topic (see Listing 7.4). Since the time complexity of *Map*'s get() and put() operations is O(1), this decision will (theoretically) have a minimal overhead in the Event Handler's performance. Furthermore, the *Map* implementation used for this case is a *ConcurrentHashMap*, which provides thread-safety and memory-consistent atomic operations.

```java
private static final Map<String, SseBroadcaster> SSE_BROADCASTERS =
                             new ConcurrentHashMap<>();

// (...)

public static void addSubscription(String eventType, EventOutput eventOutput) {
  // register event type if new
  if (!SSE_BROADCASTERS.containsKey(eventType)) {
    SSE_BROADCASTERS.put(eventType, new SseBroadcaster());
  }

  // add subscription
  SSE_BROADCASTERS.get(eventType).add(eventOutput);
}

private static void publishEvent(PublishEvent eventPublished) {
  OutboundEvent event = buildEvent(eventPublished);

  if (SSE_BROADCASTERS.containsKey(eventPublished.getEvent().getType())) {
    System.out.println(
      "Going to send message " + eventPublished.getEvent().getPayload()
        + " at " + ZonedDateTime.now().toInstant().toEpochMilli());
```

```
23
24     SSE_BROADCASTERS.get(eventPublished.getEvent().getType()).broadcast(event);
25   }
26 }
```

Listing 7.4: Event Handler's service class which handles subscriptions and published events.

Finally, the *subscribe()* resource method method then returns the *eventOutput*, which causes Jersey to bind the *eventOutput* instance with the requesting subscriber and send the response HTTP headers to the subscriber. The subscriber's client connection remains open and the subscriber is now waiting, ready to receive new SSE events. All events are written to the *eventOutput* by the corresponding broadcaster later on.

When a publisher wants to broadcast a new event to subscribers listening on a specific event type, it sends a POST request to Event Handler's URI "/publish" with the message content (see Listing 7.5). A new SSE outbound event is built in the standard way and passed to the corresponding broadcaster in the *publishEvent(eventPublished)* method (see Listing 7.4). The broadcaster internally invokes write(OutboundEvent) on all registered EventOutputs. After that the resource method returns a standard text response to the publisher to inform that the message was successfully broadcast.

```
1  @POST
2  @Path("publish")
3  public Response publishEvent(@Valid PublishEvent eventPublished,
4                  @Context ContainerRequestContext requestContext) {
5
6    System.out.println(
7      "Received message " + eventPublished.getEvent().getPayload() + " at " +
         ZonedDateTime.now().toInstant().toEpochMilli()
8    );
9
10   if (eventPublished.getEvent().getTimestamp() == null) {
11     eventPublished.getEvent().setTimestamp(ZonedDateTime.now());
12   }
13
14   // (...)
15
16   EventHandlerService.publishEvent(eventPublished);
17
18   return Response.status(Status.OK).build();
19 }
```

Listing 7.5: Event Handler's service class which handles subscriptions and published events.

The advantage of using *SseBroadcaster* is that it internally identifies and also handles client disconnections. When a subscriber closes the connection, the broadcaster detects this and removes the stale connection from the internal collection of the registered *EventOutputs*, as well as freeing

all the server-side resources associated with the stale connection. Additionally, the *SseBroadcaster* is thread-safe, so that clients can connect and disconnect at any time and *SseBroadcaster* will always broadcast messages to the most recent collection of registered and active set of subscribers.

For a subscriber to subscribe itself to an SSE event, it needs to create an *EventSource* instance, using a *WebTarget* object. However, *EventSource* by itself does not provide a way to pass query parameters, consequently prohibiting the subscriber of sending the event type it is interested in. Thus, it was necessary to create a *CustomWebTarget* class, which implements the *WebTarget* class, and pass the event type in the request's headers (see Listing 7.6).

```
1  public class CustomWebTarget implements WebTarget {
2
3    private WebTarget base;
4
5    private String eventType;
6
7    public CustomWebTarget(WebTarget base, String eventType) {
8      this.base = base;
9      this.eventType = eventType;
10   }
11
12   // Injecting the header whenever someone requests a Builder (like EventSource
         does):
13   @Override
14   public Builder request() {
15     return base.request().header("eventtype", eventType);
16   }
17
18   @Override
19   public Builder request(String... paramArrayOfString) {
20     return base.request(paramArrayOfString).header("eventtype", eventType);
21   }
22
23   @Override
24   public Builder request(MediaType... paramArrayOfMediaType) {
25     return base.request(paramArrayOfMediaType).header("eventtype", eventType);
26   }
27
28   @Override
29   public Configuration getConfiguration() {
30     return base.getConfiguration();
31   }
32
33   // All other methods from WebTarget are delegated as-is
34   // (...)
35 }
```

Listing 7.6: Subscriber's custom web target for sending event types in a request's header.

7.3.3 Reuse previously created threads in the Event Handler

As explained in Section 5.1, if the Grizzly HTTP server's threadpool is not configured, Grizzly follows Jersey's model of generating a new thread for each request, by default. In other words, with every wave of X number of requests sent to the Event Handler, Jersey will allocate the same X number of server threads almost simultaneously and closes them soon afterwards [13]. Naturally, this leads to overhead (thread creation and teardown and context switching between thousands of threads) and a large consumption of system memory (host OS must dedicate a memory block for each thread stack; with default settings, just four threads consume 1 Mb of memory [14]), which becomes largely inefficient.

The solution for this is to configure a thread pool on the Grizzly HTTP server module, which will reuse threads instead of destroying them. The key question is, what should be the optimal thread pool size for this scenario? While there is no clear-cut answer for this, it is usually suggested that if the HTTP request are CPU bound (as in this case), the amount of threads should be (at maximum) equal to the number of CPU cores in the host machine [22]. Otherwise, if the requests are I/O bound then more threads can successfully run in parallel.

Thus, we decided to determine the pool size by an empirical process, which consisted in starting with the same number of threads as the number of CPU cores and increasing them until there was no discernible improvement in throughput. Through this process, 10 ms average latency was achieved with a thread pool of 64 threads. Section 7.6 shows the effect of the thread pool size on the Event Handler's performance.

```
1  private static Client createClient(SSLContext context) {
2    ClientConfig configuration = new ClientConfig();
3    configuration.property(ClientProperties.CONNECT_TIMEOUT, 30000);
4    configuration.property(ClientProperties.READ_TIMEOUT, 30000);
5
6    // Configuring the thread pool
7    configuration.property(ClientProperties.ASYNC_THREADPOOL_SIZE, THREAD_POOL_SIZE);
8
9    Client client;
10   if (context != null) {
11     client = ClientBuilder.newBuilder().sslContext(context)
12           .withConfig(configuration).hostnameVerifier(allHostsValid).build();
13   } else {
14     client = ClientBuilder.newClient(configuration);
15   }
16
17   client.register(JacksonJsonProviderAtRest.class);
18
19   return client;
20 }
```

Listing 7.7: Configuring the Event Handler's thread pool

7.4 Experimental setup

In order to evaluate the Event Handler's performance, we conducted a test on the system, with one Publisher sending 2000 events (sequentially, with no delay) to the Event Handler, which connects to just one Subscriber. Each request is 71 bytes long, on a 100 Mb/s Switched Ethernet LAN. To measure the latency between Publisher, Event Handler, and Subscriber, each time one of these components sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. The Event Handler and the Subscriber were deployed on Raspberry Pis. There are two main reasons for choosing this platform:

1. When testing software in a resource-constrained platform, bottlenecks become more obvious and easier to identify;

2. Raspberry Pi hardware is well documented and its usage is widespread for industrial and IoT applications.

The testing environment is displayed in Figure 7.2, basically constituted by a publisher, a subscriber and the Event Handler, with all clocks synchronized using a local NTP server.

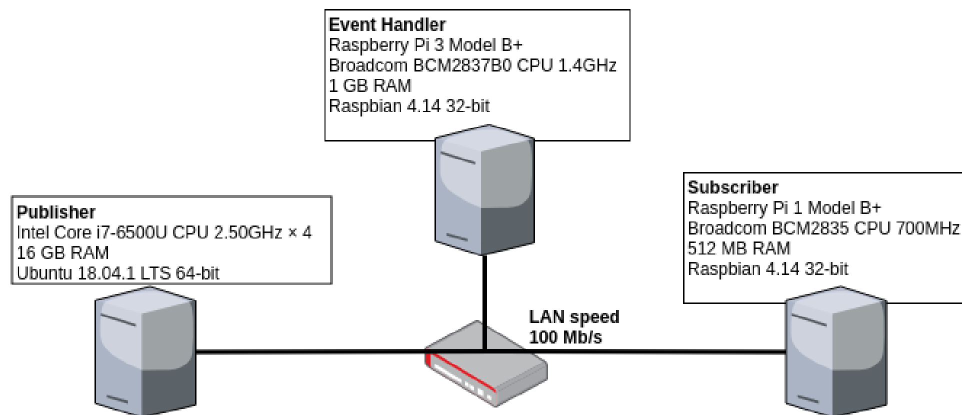


Figure 7.2: Testing environment for the official Event Handler

7.5 Performance evaluation of original version

After sending 2000 events to the original Event Handler, 41.9% of these events had an end-to-end latency greater than 100 ms, and 20.3% of these had a latency greater than 1s, with an average of approximately 666.3 ms. Moreover, the maximum latency reaches the 4.9 s, as can be seen in Figure 7.3. This type of performance is a symptom of a bottleneck in the system.

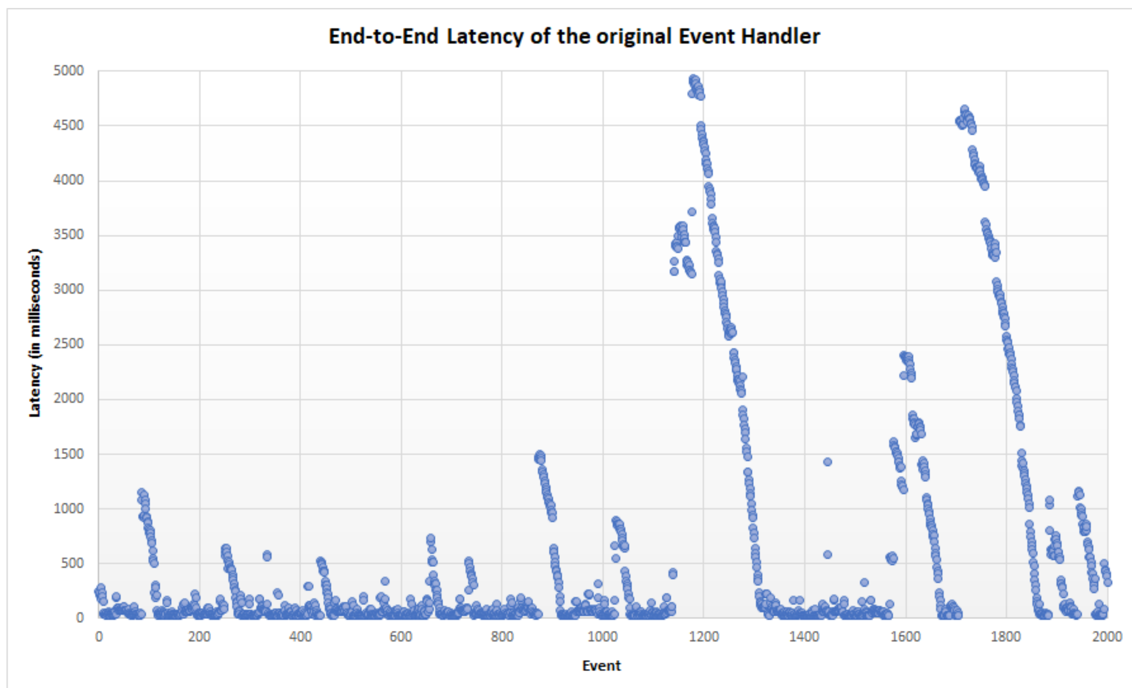


Figure 7.3: End-to-end latency, for each of the two thousand messages sent, with the official Event Handler

7.6 Performance evaluation of enhanced version

After the major refactoring on the original Event Handler, the "enhanced" version was put to the test on multiple different workload scenarios.

7.6.1 Test Scenario A: 1 Publisher, 1 Subscriber, 2000 events

The first test was with the same environment and workload as the original version. After repeating the same testing process, the test results were exceedingly better than the previous version's (see Figure 7.4), with an average end-to-end latency of approximately 8.95 ms and a maximum latency of 32.00 ms.

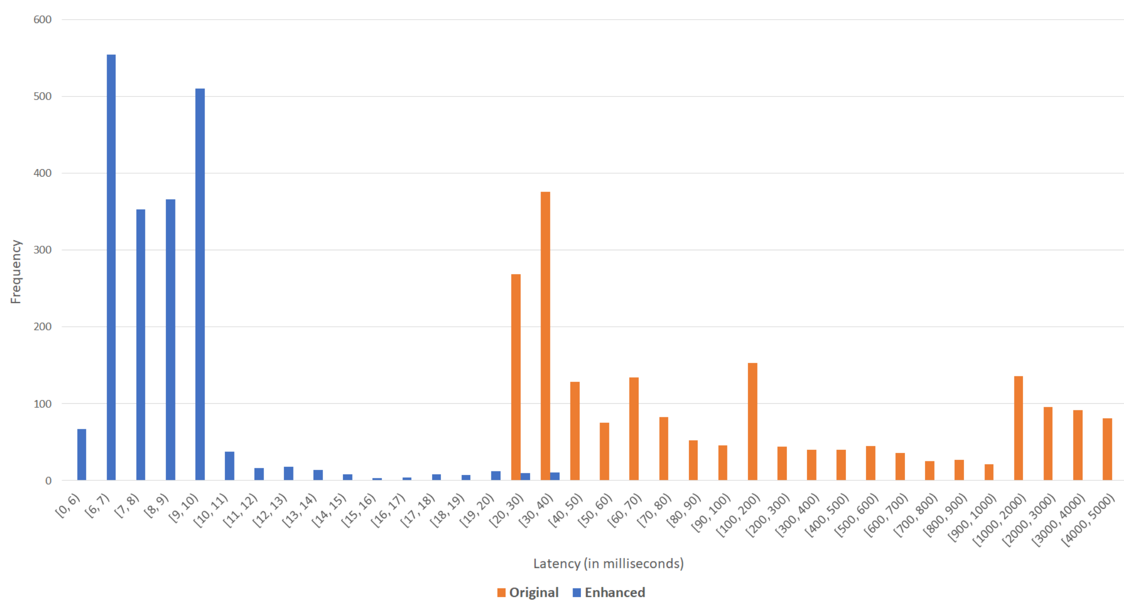


Figure 7.4: End-to-end latency comparison of the two versions of the Event Handler

7.6.2 Test Scenario B: 1 Publisher, 1-6 Subscribers, 9000 events

Afterwards, two other scenarios were tested:

1. Instead of 2000 events, the Publisher shall send 9000 events, in order to detect potential bottlenecks;

2. The same scenario as scenario 1, however, instead of using a single Subscriber, six different Subscribers were used.

The test results of these experiments showed a similar performance increase. For scenario 1, the average end-to-end latency was 8.98 ms, with a maximum latency of 52.00 ms. As for scenario 2, the average end-to-end latency was 10.68 ms, with a maximum latency of 45.67 ms, measured between all six subscribers. A histogram with the end-to-end latency distribution for these two scenarios is displayed in Figure 7.5.

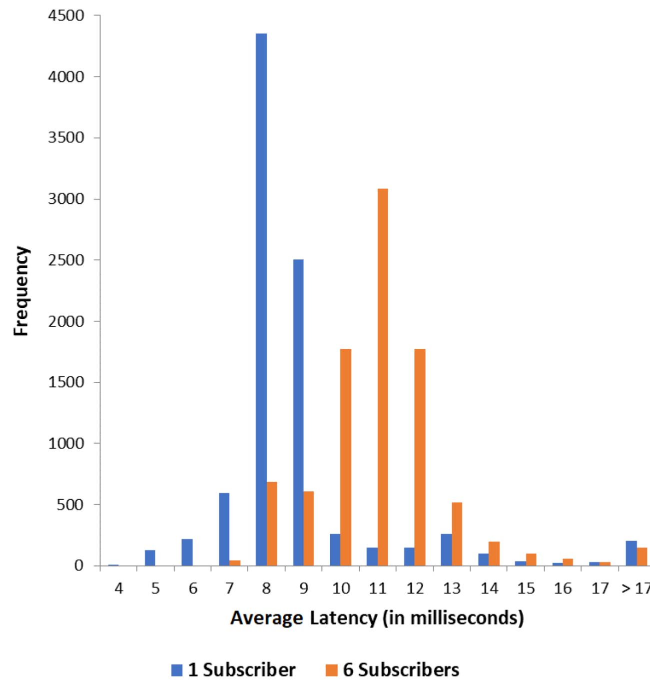


Figure 7.5: End-to-end latency distribution of 9000 events for one subscriber and six subscribers, with the enhanced Event Handler.

7.6.3 Test Scenario C: 10 Publishers, 10 Subscribers, 10.000 events in total

To further analyze how scalable the new Event Handler is, another test was carried out using 10 publishers and 10 subscribers, with each publisher sending 1000 events to the Event Handler (i.e. 10,000 messages in total) with no delay between each message sent, and every subscriber system subscribing to the same "event type" (i.e. message topic). The test results are displayed in Figure 7.6.

While more than half of the latency values were below 30 ms (with an average latency of 24.95 ms, and the most frequent interval being between 9 to 13 ms), there was a higher number of cases where the end-to-end latency got above the 90 ms, with the maximum latency being 319 ms. This can indicate that the Event Handler might still not be completely capable of easily scaling with a large and frequent number of messages between multiple publishers and subscribers. Nevertheless, it is still a gargantuan improvement over the original version's performance.

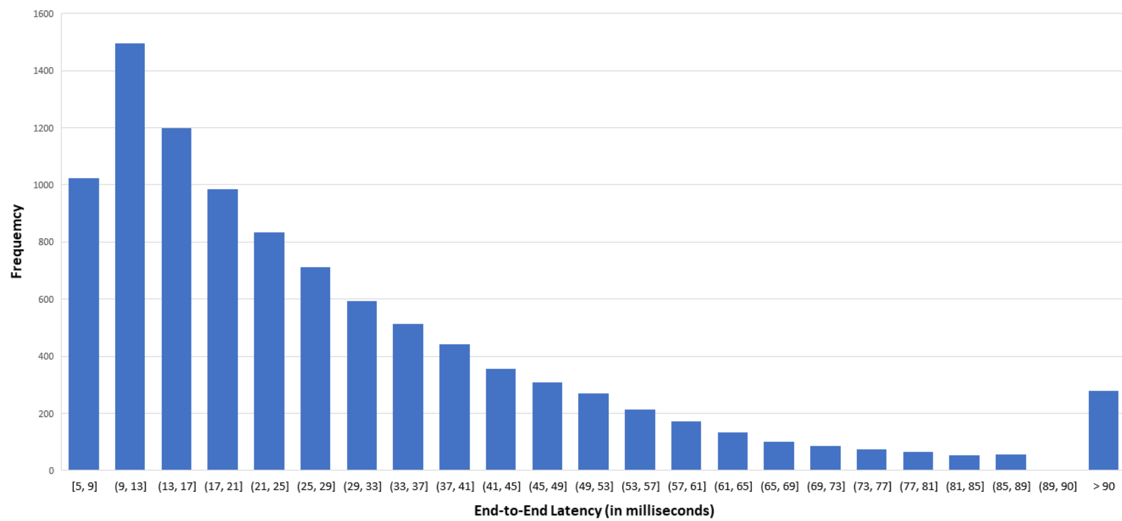


Figure 7.6: End-to-end latency distribution of 1000 events from 10 Publishers (each) to 10 Subscribers, with the enhanced Event Handler.

7.6.4 Test Scenario D: 1 Publisher and 7 Subscribers (on same machine), different threadpool sizes in Event Handler

Additional tests were carried out on the Event Handler to evaluate the effect of the threadpool size on its performance, while running on a Raspberry Pi 3. As previously mentioned in Section 7.3.3, 64 threads were enough to achieve a satisfactory performance for multiple use scenarios. The decision process for the threadpool size consisted in using all powers of 2 between the number of Raspberry Pi 3's CPU cores (i.e. 4 cores) and 64. In other words: 4, 8, 16, 32 and 64.

These tests were initially done with one publisher and 7 subscribers running on the same machine, while the Event Handler ran on a Raspberry Pi 3 Model B. In this scenario, the performance differences between threadpool sizes were as in Figure 7.7 and Table 7.1. The threadpool size of 64 was the one with lower latency. However, in this test case, 64 might not always be the best choice: we get an improvement of the average latency of less than 0.25 ms with respect to 4 threads, but the maximum latency increases from 52 to 82.

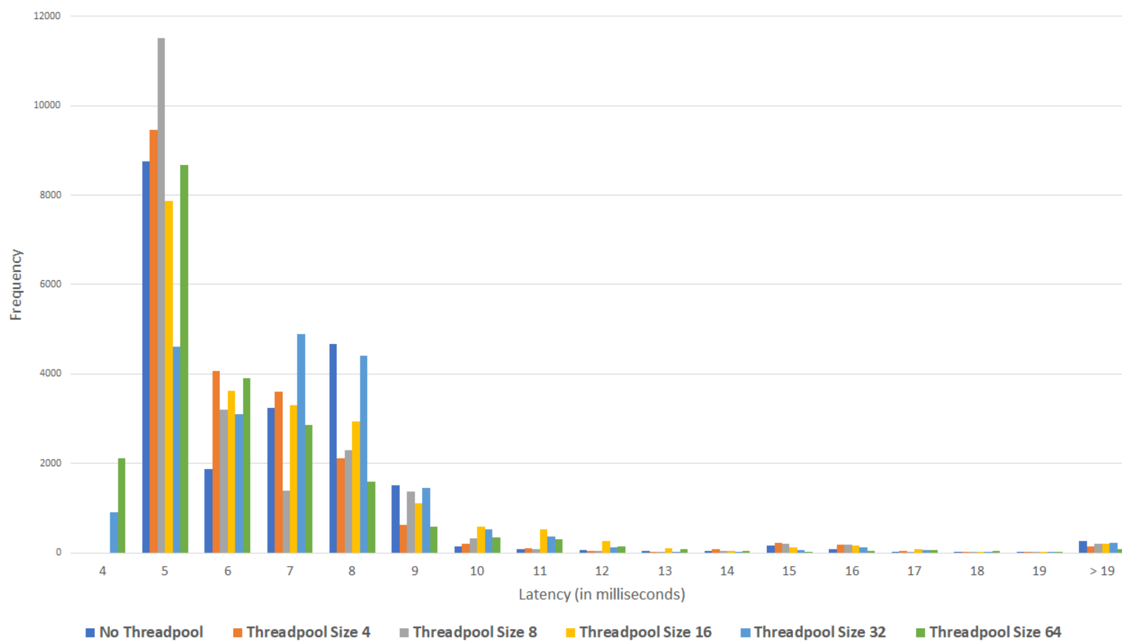


Figure 7.7: End-to-end latency of 3000 events from one publisher to seven subscribers (running on the same machine), with different thread pool sizes on the enhanced Event Handler (running on a Raspberry Pi 3 Model B).

Table 7.1: Performance comparison between all tested threadpool sizes, for one publisher and 7 subscribers running on the same machine and the Event Handler running on a Raspberry Pi 3 Model B

Threadpool Size	Average Latency (ms)	Standard Deviation (ms)	Maximum Latency (ms)
No threadpool	6.803	3.152	69
4	6.433	2.794	52
8	6.370	2.978	52
16	6.930	3.076	63
32	7.179	2.899	57
64	6.199	2.617	82

7.6.5 Test Scenario E: 1 Publisher and 7 Subscribers (each on a Raspberry Pi 1), different threadpool sizes in Event Handler

Another similar test scenario was carried out, but this time each one of the 7 subscribers ran on a Raspberry Pi 1. Unfortunately, due to time constraints, it was not possible to test all threadpool sizes (8 and 16 were left out). However, based on the ones that did get tested, the performance difference between each threadpool size was substantially more significant than in the previous scenario, with a significant difference between the average latency, as seen in Figure 7.8 and Table 7.2. Similar to the prior scenario, the threadpool size of 64 was the one with the lowest average latency between all tested sizes. Although, it appears that there also a significant difference

in the maximum latency, with the pool with 32 threads having a maximum latency that is almost one order of magnitude smaller than that for 64 threads.

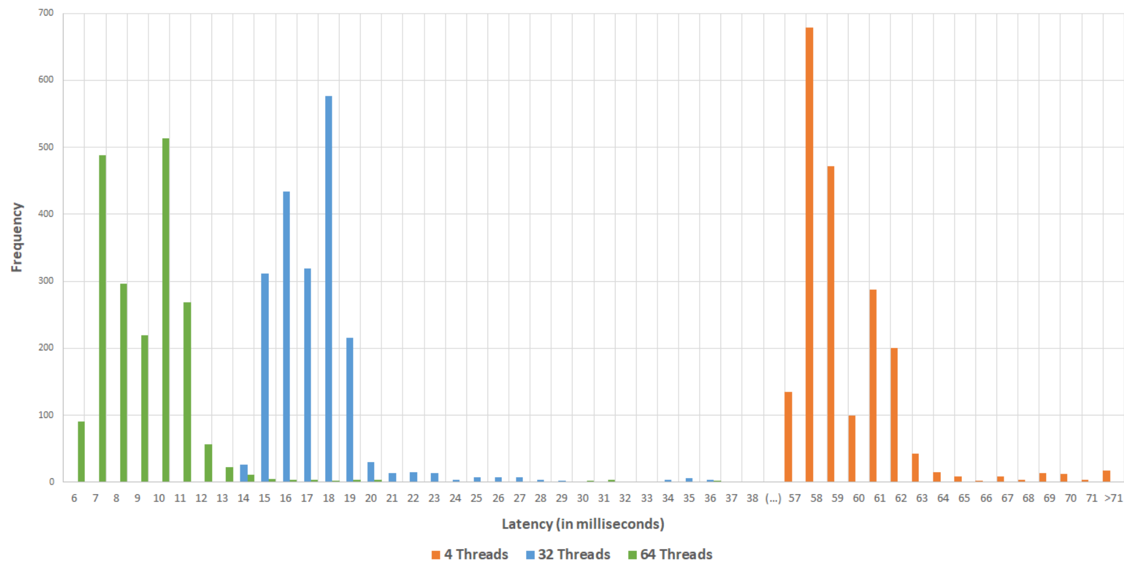


Figure 7.8: End-to-end latency distribution from one publisher to 7 subscribers (each running on a Raspberry Pi 1), with different thread pool sizes on the enhanced Event Handler (running on a Raspberry Pi 3 Model B).

Table 7.2: Performance comparison between all tested threadpool sizes, for one publisher and 7 subscribers (each running on a Raspberry Pi 1) and the Event Handler running on a Raspberry Pi 3 Model B

Threadpool Size	**Average Latency (ms)**	**Standard Deviation (ms)**	**Maximum Latency (ms)**
4	59.996	3.461	96
32	17.832	3.467	49
64	9.584	5.483	450

Chapter 8

Modelling the Event Handler's performance

In order to estimate the performance of different host machines running the improved Event Handler system, it is necessary to take into account specific thread pool configurations, number of CPU cores, and network latency. As such, it was decided that a performance model should be developed in order to depict these variables. Thus, similar to the Intracloud and Intercloud orchestrations' performance modeling, the Event Handler's performance was also modeled through Petri nets.

To develop such a model, Lu & Gokhale's methodology [55] seemed to be a clear fit for this work's purpose, since it was used to model the performance of a Web server with a thread pool architecture. Thus, the resulting Petri net – which is displayed in Figure 8.1 – followed a few of the paper's guidelines, however with some clear distinctions which will be further explained in this chapter.

In relation to the stochastic transitions chosen for the model, the same process from Chapter 6 was used to determine the probability distribution that best fit the latency distribution of each request/response. At the time of developing the Petri net model, the distribution fitting process was applied on the latency data acquired from the tests with 1 and 6 Subscribers (Figure 7.5).

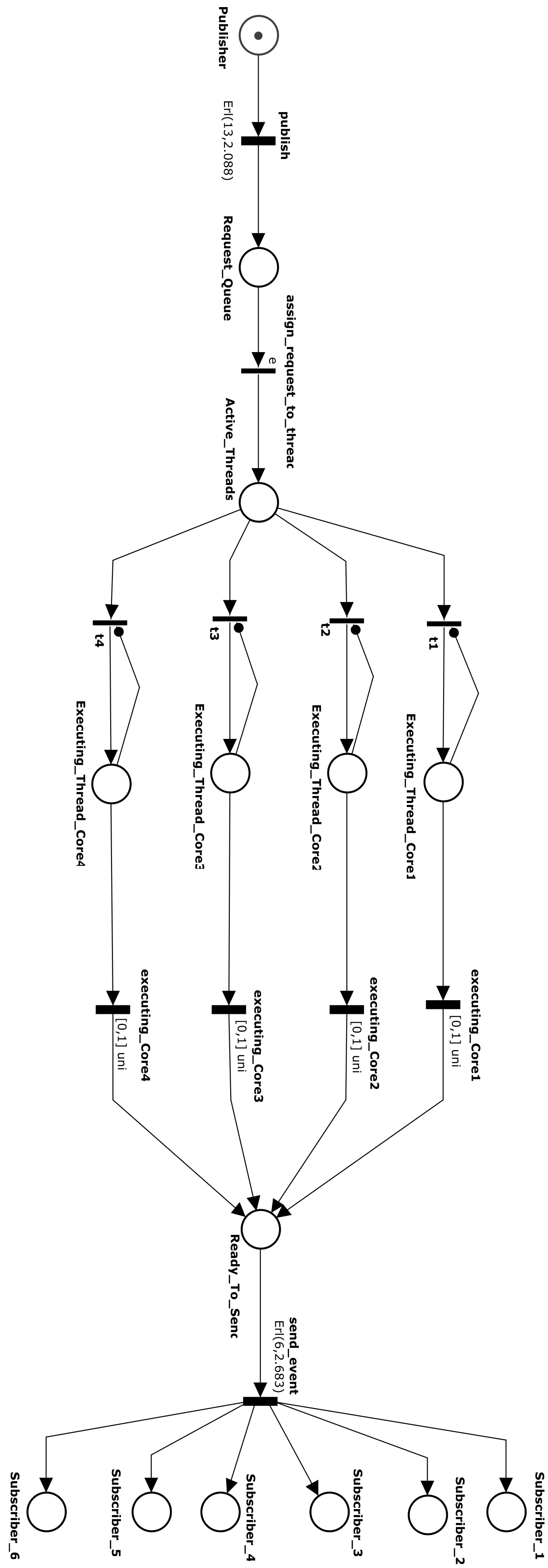


Figure 8.1: Stochastic Petri net model of the Event Handler running on a quad-core CPU

8.1 Explaining the Petri net model

The underlying logic in the developed Petri net is similar to [55]'s proposed model, however while Lu & Gokhale's model is specific to single-core CPUs, our model characterizes the Event Handler's execution in a quad-core CPU (given that it is running in a Raspberry Pi 3 Model B). Regarding the model itself, the Publisher place (the circle on the left) represents the Publisher, and the publish transition (the black vertical wide bar) represents the time it takes for a published event to be transmitted and reach the Event Handler. The *Request_Queue* place holds unprocessed requests, while the *assign_request_to_thread* transition represents the Event Handler's thread pool limit – only assigning requests to a thread if the total number of active threads (represented by the token sum in the *Active_Threads* place and *Executing_Thread_CoreX* places) has not exceeded the specified limit. In the Petri net, this condition is executed through an enabling function in the *assign_request_to_thread* transition. Once a request is assigned to a thread, the thread is executed by one of the host machine's CPU cores. The *Executing_Thread_CoreX* place (*X* should be replaced by the corresponding core) signifies the thread's execution, while the *executing_CoreX* transition represents the amount of time it takes to execute. An inhibitor arc, i.e. the arc with a black circle on its end, is used from *Executing_Thread_CoreX* to the respective *tX* transition to avoid the firing of transition *tX* when *Executing_Thread_CoreX* already has a token, therefore guaranteeing that only one request is being executed on that specific CPU core.

Once a thread finishes a CPU run, Lu & Gokhale's model contemplated the probability of the request either getting a successive CPU run, needing to access I/O, or getting fulfilled and exiting the system. This is because their model is for a Web Service, where many times one needs to access the disk, and going back to the CPU queue is a way to model preemption. Since these situations do not make much sense for the Event Handler, it was decided to tone down the model's complexity and assume that each request gets fulfilled the first time it finishes a CPU run. As such, once the *executing_CoreX* transition finishes, it sends a token to *Ready_to_Send*, where the event is ready to be sent to its subscribers.

Several real experiments have been performed in order to fine tune the model with real data extracted from several test runs from where the values for each request type were derived (i.e., considering requests sent from Publisher to Event Handler, requests sent from Event Handler to each Subscriber), and the CPU execution time for each request, and determine their most appropriate probability distribution function to be applied in the Petri net model. In fact, as mentioned back in the beginning of this section, the probability distributions were based on the latency data acquired from the tests with 1 and 6 Subscribers, which was the data available at the time. We determined that the requests sent from the Publisher to the Event Handler had a Gamma distribution with shape = 13.235 and rate = 2.088. However, given Oris's limitations (explained back in Section 6), the shape parameter was then rounded to an integer value (i.e. 13). Similarly, the requests sent from the Event Handler to its Subscribers also had a Gamma distribution with shape = 6.235 and rate = 2.683, where shape was then rounded to 6, to likewise satisfy the Erlang distribution requirements. Finally, the CPU execution times in the Event Handler (i.e. executing_CoreX) were

decided to be represented as transitions with a uniform distribution, where the early finish time is 0 ms and the late finish time is 1 ms (considering that in the test results 75% of the execution times took 0 ms, while the other 25% took 1 ms).

8.2 The Petri net model

Regarding the stochastic analysis, similarly to the intracloud and intercloud orchestrations, a chart was created via the Oris Tools' estimations – with "time" as the X-axis and "place probability" as the Y-axis – and is displayed in Figure 8.2.

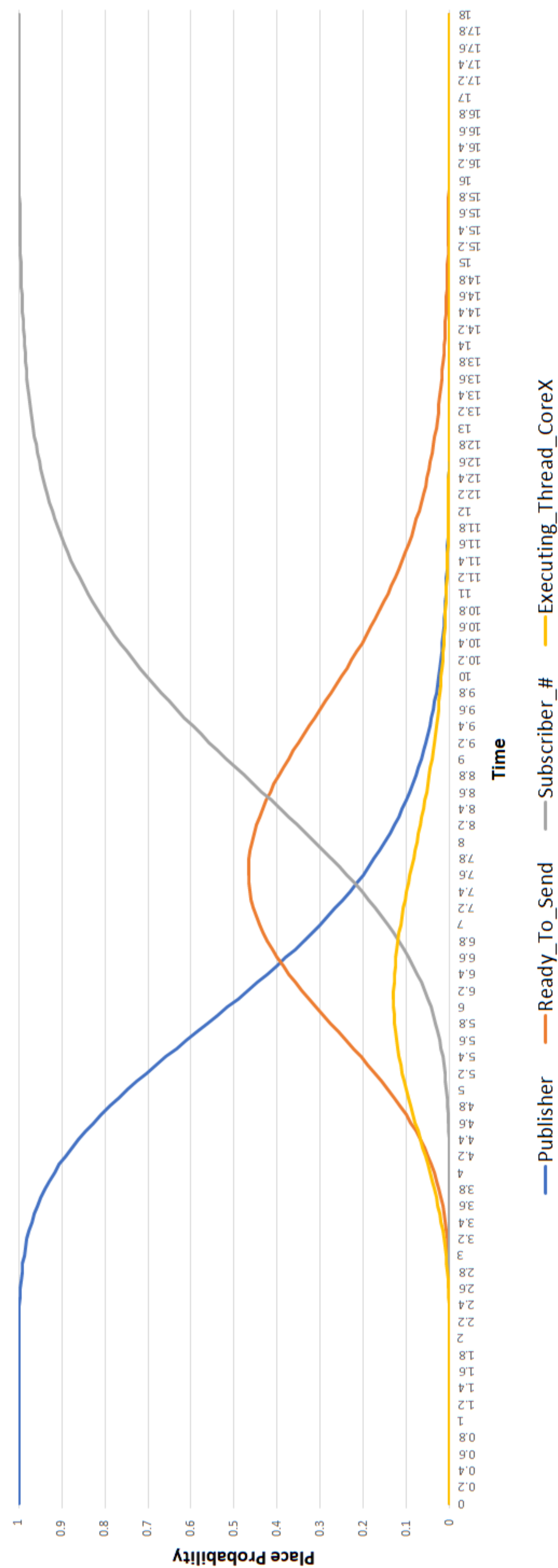


Figure 8.2: Transient analysis of the Petri net model

8.2.1 Comparing the model with the actual experiments

Overall, the values collected from the model match the results obtained in the experiments of the enhanced Event Handler. In the Petri Net model, the probability distribution for *Subscriber_#* to receive the published message is only higher than Publisher, *Executing_Thread_CoreX*, and *Ready_to_Send* after the 8.5 ms instant. One can see that this value is matched with the experiment results for one Subscriber reported in Figure 7.5, where it is possible to see that around 60% of the events were delivered with an 8 ms latency. Whereas for the six Subscribers tests, where the average end-to-end latency is approximately 10.68 ms, the corresponding probability distribution is 80.4%.

In addition to the initial stochastic analysis with one token, another stochastic analysis was performed with four tokens (i.e. four messages) to examine how the model scales with the processing of multiple messages. The same transient analysis matrix was calculated, and the distribution of the estimated end-to-end latencies for four messages is depicted in Figure 8.3, juxtaposed with the real test results from Figure 7.5. Unfortunately, due to some processing limitations of the Oris tool, the author was unable to assess the performance for more than four tokens.

Nevertheless, this stochastic analysis with four tokens is able to capture the latency interval for most messages, i.e. from 8 to 16 ms, which mostly goes in hand with the event latency distributions of the test results. However, the author feels that these latency estimations must still be further improved in order to fine tune the probability for each latency and also to capture a wider range of latencies, since the more extreme latencies (i.e. below 8 ms and above 17 ms) are not represented. In terms of improving these estimations, this could be done by: i) changing the probability distributions and the parameters chosen for each transition; or ii) changing the Petri net model itself.

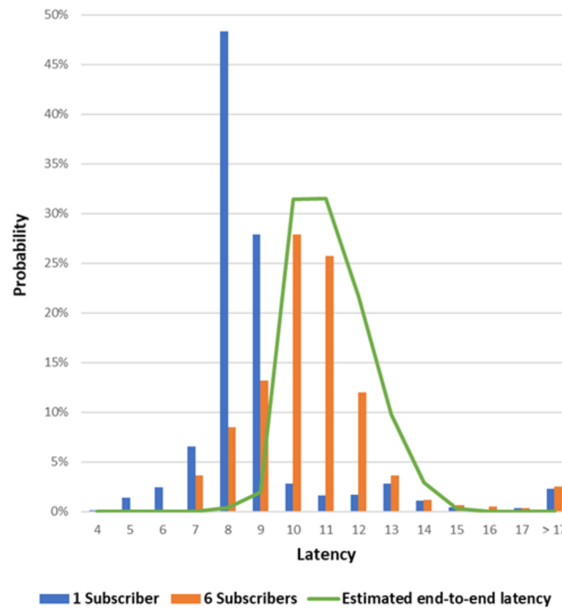


Figure 8.3: Average distribution of the estimated end-to-end latency, with four messages

8.2.2 Interpreting the analysis results

First, the only places that are present in the transient analysis chart (Figure 8.2) are *Publisher*, *Ready_to_Send*, *Subscriber_#* and *Executing_Thread_CoreX*. The reason for this is because the other places (aside from *Request_Queue*) only depend on immediate transitions, thus the token will not spend any time in these places, meaning that these do not have an impact in the overall processing time. Although *Request_Queue* is linked to an immediate transition (i.e. *assign_request_to_thread*), this transition is restricted to the Event Handler's thread pool size, which (as explained previously) is represented by the token sum in the *Active_Threads* place, the *Executing_Thread_CoreX* places, and *Ready_to_Send* place. Since only one token is sent in this particular analysis, *Request_Queue* will not be storing any tokens, thus it will not be present in this chart.

Until time 2.1, the probability of a token being in *Publisher* is approximately 1, whereas the other places are still 0, because the Publisher takes at least 2 ms to send an event to the Event Handler. Between time 2 and 13.6 ms, the probability of the Publisher sending a message decreases nonlinearly to 0, while the exact opposite happens to the Subscriber, i.e. the probability that *Subscriber_#* has received the token rises nonlinearly to 1. In fact, at time 7.6 ms, the two curves cross each other, which means that, beyond this point, there is a higher probability of an event having reached the respective Subscriber, than it still being published by the Publisher. Furthermore, from 2.1 to 13.5 ms, the probability of the token being in *Executing_Thread_CoreX* has an almost Gaussian distribution, which means that once the message is sent from the Publisher, it is processed by the Event Handler for a maximum of 1 second. After this process, the message is then ready to be sent. Indeed, from 2.5 to 17 ms, similar to *Executing_Thread_CoreX*, the probability of the token being in *Ready_to_Send* also has a Gaussian distribution, meaning that once the Event Handler is ready to send the published event, the Publisher has already sent the message, and the Subscriber is about to receive it – hence the probability decrease in Publisher and the increase in *Subscriber_#* right after the probability peak in *Ready_to_Send*.

According to the analysis's time estimations, the "maximum" time it takes to send an event (i.e. with a 99% chance) from the Publisher to the Event Handler (i.e., when the probability for the Publisher place reaches approximately 0) is around 13.6 ms, while the estimated "latest" time for a Subscriber to receive an event (i.e., when the probability for the *Subscriber_#* places reaches approximately 1) is around 17.1 ms. Nevertheless, there is a 99% chance that Subscribers will receive the published event around 14.3 ms. Furthermore, the probability for the *Ready_To_Send* place to hold a token peaks (47%) at the 7.6 ms, which means that the Event Handler is ready to send the published event to its subscribers at this instant, 47% of the times.

Chapter 9

Conclusions and Future Work

This chapter recaps the most relevant points of this dissertation and describes the results obtained from the project, mentions additional work and contributions done throughout the thesis's development, and finally explains the project's limitations and future improvements.

9.1 Results from the Dissertation

By using stochastic Petri nets, we were able to propose a performance model of the Intracloud and Intercloud orchestration process of the Arrowhead framework. Through this model, it was possible to not only estimate the average response time for an orchestration request in both intra- and inter-cloud versions, but also to estimate the probability distribution of the Petri net being in a specific response state. For the Intracloud process, there was a 25% difference between the estimation and the actual result. As for the Intercloud process, there was a 37% difference.

Through these performance evaluations, it was also possible to identify that the framework, and by extension its systems, had some potential performance setbacks, mostly it handles HTTP requests. Thus, we have decided to improve the performance of the Event Handler (a message broker built over REST/HTTP), a service whose performance is very important in most Arrowhead deployments, by using appropriate software configurations and design patterns.

By changing how the original Event Handler and its clients handled HTTP requests and thread creation, the enhanced version of the Event Handler is now able to achieve the initial goal of an average end-to-end latency of 10 ms. In fact, considering the average latency of both versions for the same test scenario (one publisher sending 2000 events to one subscriber), the Event Handler had an overall performance boost of over 98%. Moreover, we also proposed a Petri net model for the Event Handler in order to estimate the overall end-to-end latency probability of each component (Publisher, Event Handler, and Subscribers). The model estimates that the subscriber has a high probability of getting the published event around the 8.5 ms. These estimations stand mostly

true to the actual values, where the percentage difference between the average latency (8.95 ms) and the estimated latency (8.5 ms) is around 5.16%.

Similar modifications can be applied to other components of the Arrowhead Framework to improve their performance.

9.2 Additional Contributions

Throughout the development of this dissertation, additional work was done to disseminate this project's results.

- **Improving the performance of a Publish-Subscribe message broker**
  - **Authors:** Rafael Rocha, Cláudio Maia, Luis Lino Ferreira, Pedro Souto, Pal Varga
  - **Conference:** Demo in 22nd IEEE International Symposium on Real-Time Computing (ISORC 2019)
  - **Dissemination Items:** Appendix A and Appendix B [87]
- **Improving and modelling the performance of a Publish-Subscribe message broker**
  - **Authors:** Rafael Rocha, Cláudio Maia, Luis Lino Ferreira, Pal Varga
  - **Conference:** Accepted for 45th Annual Conference of the IEEE Industrial Electronics Society (IECON 2019)
  - **Dissemination Items:** Appendix C

9.3 Further Work

Regarding the Event Handler, the system's performance might still be able to improve even further than its current state by optimizing the Event Handler's thread pool size and the Publisher's connection pool. However, the gains to be had are most likely marginal.

In relation to the Petri net models developed in this dissertation, they could certainly be further improved, either by changing the probability distributions and their parameters chosen for each transition or by changing the Petri net model itself. For example, as mentioned in Section 8.2.1, for the Event Handler's Petri net, the probability for each latency should be better fine tuned and should be able to capture a wider range of latencies, since the more extreme latencies (i.e. below 8 ms and above 17 ms) are not represented. As for the Intracloud and Intercloud Petri nets, their estimations were also not completely applicable to the actual values, with a 25% and 37% difference, respectively. These issues are expected to be the focus for future research work.

Appendix A

Workshop Demo at ISORC 2019

CISTER

Research Centre in
Real-Time & Embedded
Computing Systems

Demo

Improving the performance of a Publish-Subscribe message broker

Rafael Rocha

Cláudio Maia

Luis Lino Ferreira

Pedro Souto

Pal Varga

CISTER-TR-190403

Improving the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia, Luis Lino Ferreira, Pedro Souto, Pal Varga

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

The Arrowhead Framework, a SOA-basedframework for IoT applications, provides the Event Handlersystem: a publish/subscribe broker implemented withREST/HTTP(S). However, the existing implementation of theEvent Handler suffers from message latency problems that arenot acceptable for industrial applications. Thus, this paperdescribes the refactoring process of this system that enabled itto reach acceptable levels of latency.

© 2019 CISTER Research Center
www.cister-labs.pt

Improving the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia, Luis Lino Ferreira
CISTER Research Center, ISEP
Polytechnic Institute of Porto
Porto, Portugal
{rtdrh, crr, llf}@isep.ipp.pt

Pedro Souto
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
pfs@fe.up.pt

Pal Varga
Dept. of Telecomm. and Media Informatics, Budapest University of Technology and Economics
Budapest, Hungary
pvarga@tmit.bme.hu

***Abstract*—The Arrowhead Framework, a SOA-based framework for IoT applications, provides the Event Handler system: a publish/subscribe broker implemented with REST/HTTP(S). However, the existing implementation of the Event Handler suffers from message latency problems that are not acceptable for industrial applications. Thus, this paper describes the refactoring process of this system that enabled it to reach acceptable levels of latency.**

Keywords—Performance, Publish-Subscribe, HTTP, REST, SOA, Java

I. Introduction

The Arrowhead Framework [1] aims at using a service-oriented approach for IoT applications. It includes a set of Core Services [1] (e.g. service discovery, orchestration and authentication) that support the interaction between Application Services, such as services capable of providing sensor readings. One of the available Arrowhead systems, the Event Handler, is used for sending periodic updates from a producer service to several other consumer applications. In this sense, the Event Handler serves as a REST/HTTP(S) implementation of a publish-subscribe message broker, thus the Event Handler does not process events, it only handles their distribution from publishers to multiple subscribers. For an Arrowhead service to continuously notify its subscribers on time, the Event Handler's performance is of extreme importance. However, the existing implementation of the Event Handler suffers from several end-to-end message latency problems (leading up to a maximum of almost 5 seconds to deliver some messages), mostly due to the wasteful creation of threads and HTTP connections, which also lead to unnecessarily high CPU and memory usage, which particularly affects resource-constrained host machines. Therefore, a refactoring is necessary to address these performance woes, in order to achieve an average end-to-end latency of 50ms.

II. Analysing the original version of Event Handler

Event Handler's implementation in the official Arrowhead repository [2] uses a combination of Grizzly (a framework designed to take advantage of the Java Non-blocking I/O API) for its HTTP server, and Jersey (a framework designed to support JAX-RS APIs) for its RESTful API. Furthermore, no thread pool configuration was found for the Grizzly HTTP server module. Moreover, for the client applications that are meant to use the Event Handler, i.e. the publishers and subscribers, the Arrowhead Consortia provides client skeletons to be extended with the developers' own application code [2]. These client skeletons use the same Jersey/Grizzly setup and server configuration as the Arrowhead systems.

A. The testing environment

In order to evaluate the Event Handler's performance, we conducted a stress test on the system, with one Publisher sending two thousand requests per second to the Event Handler, which connects to just one Subscriber. Each request weighs 71 bytes (measured with Wireshark), on a network with 100 Mb/s LAN speed. To calculate the latency between Publisher, Event Handler, and Subscriber, each time a system sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. We deployed the Event Handler on a Raspberry Pi 3 Model B+ and the Subscriber on a Raspberry Pi 1 Model B+. Furthermore, in order to ensure that end-to-end latency was correctly calculated, the clock on all machines was synchronized using a local NTP server, which provides accuracies generally in the range of 0.1ms.

B. Performance

After sending two thousand events to the original Event Handler, 41.9% of these events had an end-to-end latency greater than 100ms, and 20.3% of these had a latency greater than 1s, with an average of approximately 666.3ms. But the maximum latency reaches 4.9s. Naturally, this type of performance is not acceptable for industrial applications, and thus, the official implementation of the Event Handler was revised.

III. Improving the Event handler

To improve the Event Handler's performance, each endpoint – the Publisher, the Event Handler, and the Subscriber – had to be addressed. Thus, after a code analysis, two major problems were detected. The first problem was that none of the three components reused connections. This has a major performance impact on system communications, since establishing a connection from one system to another is rather complex and consists of multiple packet exchanges between two endpoints (connection handshaking), which can cause major overhead, especially for small HTTP messages [3]. In fact, a much higher data throughput is achievable if open connections are re-used to execute multiple requests. This problem required a different solution between the three systems: 1) the Publisher had to use a connection pool so that it could reuse its connections to the Event Handler; 2) the Event Handler had to use Jersey's own Server-Sent Events mechanism to establish a persistent connection to each of its Subscribers. The second problem consisted in the Event Handler creating a new thread for every incoming request, which would then greatly impact the machine's available RAM and response times. Thus, the Event Handler required a thread pool to manage incoming requests in a less wasteful manner.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

A. *Connection Pool in the Publisher*

In order to re-use open connections between the Publisher and the Event Handler, the best choice was to implement a connection pool, via the Apache HTTP Client, on Jersey's transport layer. According to the Apache Software Foundation [3], the client maintains a maximum limit of connections on a per route basis (which can be configured), so a request for a route for which the client already has a persistent connection available in the pool will be handled by renting a connection from the pool rather than creating a brand-new connection. For the final test, only one connection per route was set.

B. *Server-Sent Events in the Event Handler and Subscriber*

The Event Handler also did not re-use previously created connections to its subscribers, consequently adding a large overhead to the end-to-end latency of each published event. Contrary to the previous problem's solution though, in this case, Jersey itself already offered a mechanism to handle a one-way publish-subscribe model: Server-Sent Events (SSE). According to the Jersey documentation [4], by using SSE, when the Subscriber sends a request to the Event Handler, the Event Handler holds a connection between itself and the Subscriber until a new event is published. When an event is published, the Event Handler sends the event to the Subscriber, while keeping the connection open so that it can be used for the next events. The Subscriber processes the events sent from the Event Handler individually and asynchronously without closing the connection. Therefore, the Event Handler can reuse one connection per Subscriber.

C. *Thread Pool in the Event Handler*

By default, if the thread pool configuration of the Grizzly HTTP server module is left untouched, Jersey generates a new thread for each request. In other words, with every wave of two thousand requests sent to the Event Handler, Jersey will allocate around that same amount of server threads simultaneously, only for them to be de-allocated soon afterwards [5]. Naturally, this leads to a great amount of overhead (thread creation and teardown, context switching between thousands of threads) and a large consumption of system memory (host OS must dedicate a memory block for each thread stack; with default settings, just four threads consume 1 Mb of memory [6]), which becomes largely inefficient. Nonetheless, the solution for this is relatively simple: configure a thread pool on the Grizzly HTTP server module, which will reuse threads instead of destroying them. The process to identify the optimal pool size was to start with the same number of threads as the available number of CPU cores and increase them until there is no discernible improvement in throughput. Through this, the 50ms latency goal was achieved on a thread pool of 64 threads.

IV. PERFORMANCE EVALUATION OF THE ENHANCED VERSION

After the major refactoring on the original Event Handler, the "enhanced" version was put to the test on a similar testing environment and workload as the original. However, instead of just one Subscriber, it was decided to test the Event Handler with seven different Subscribers, so as to ensure that all changes would have a major effect on performance. After repeating the same testing process, the test results were exceedingly better than the previous version's (see Fig. 1), with an average of approximately 46.2ms of end-to-end latency per request, and a maximum latency of approximately 114ms.

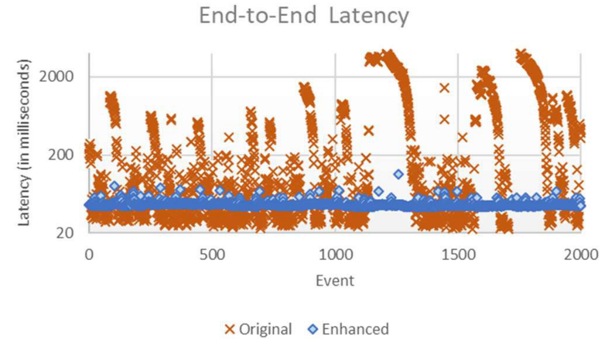


Fig. 1. End-to-end latency of the two versions of the Event Handler.

V. CONCLUSIONS AND FUTURE WORK

By changing how the original Event Handler and its clients handled HTTP requests and thread creation, the enhanced version of the Event Handler is now able to achieve the initial goal of reaching an average end-to-end latency of 50ms. In fact, by considering the average latencies of both versions, it is safe to say that the Event Handler had an overall performance boost of over 93%. Nevertheless, the authors theorize that the system's performance might still be able to improve even further than its current state by optimizing the Event Handler's thread pool size and the Publisher's connection pool.

ACKNOWLEDGMENTS

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234); also, by EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 737459 (Productive4.0 project) and by the Portuguese National Innovation Agency (ANI) under the European Regional Development Fund (FEDER), through the "Portugal 2020" (PT2020) partnership, within the framework of the System of Incentives to Research and Technological Development (SII&DT) and the Operational Program for Competitiveness and Internationalization (POCI), within project FLEXIGY, nº 34067 (AAC nº 03/SI/2017).

REFERENCES

[1] P. Varga, et al, "Making system of systems interoperable – The core components of the arrowhead framework", Journal of Network and Computer Applications, Volume 81, 2017, Pages 85-95, ISSN 1084-8045.

[2] "Arrowhead Consortia", GitHub, 2019. [Online]. Available: https://github.com/arrowhead-f. [Accessed: 22- Mar- 2019].

[3] The Apache Software Foundation, "Chapter 2. Connection management", Hc.apache.org, 2019. [Online]. Available: https://bit.ly/2TQpBjK. [Accessed: 21- Mar- 2019].

[4] Project Jersey, "Chapter 14. Server-Sent Events (SSE) Support", Docs.huihoo.com, 2019. [Online]. Available: https://bit.ly/2TpoQsK. [Accessed: 21- Mar- 2019].

[5] N. Babcock, "Know Thy Threadpool: A Worked Example with Dropwizard", Nbsoftsolutions.com, 2016. [Online]. Available: https://bit.ly/2Js4Shm. [Accessed: 21- Mar- 2019].

[6] "Why using many threads in Java is bad", Iwillgetthatjobatgoogle.tumblr.com, 2012. [Online]. Available: https://bit.ly/2HCpHVE. [Accessed: 21- Mar- 2019].

Appendix B

Workshop Poster at ISORC 2019

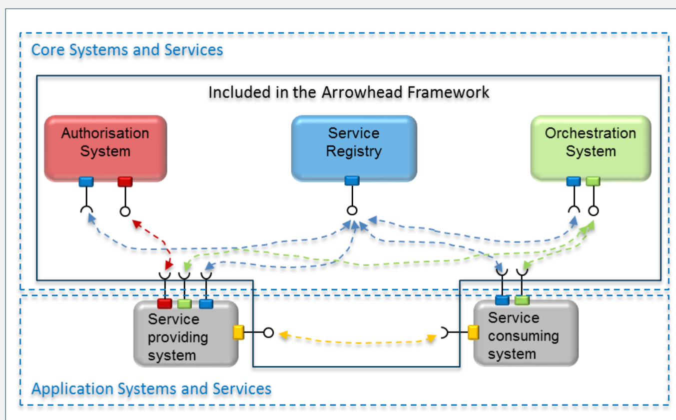
Improving the performance of a Publish-Subscribe message broker

CISTER – Research Centre in Real-Time & Embedded Computing Systems

Rafael Rocha, Cláudio Maia, Luis Lino Ferreira, Pedro Souto, Pal Varga
{rtdrh, crr, llf}@isep.ipp.pt, pfs@fe.up.pt, pvarga@tmit.bme.hu

Arrowhead Framework

- Consists of a SOA framework for IoT applications
- Based on three core services:
  - Service Discovery
  - Orchestration
  - Authentication
- Several other services:
  - Event Handler, QoS Manager, Gatekeeper, etc.

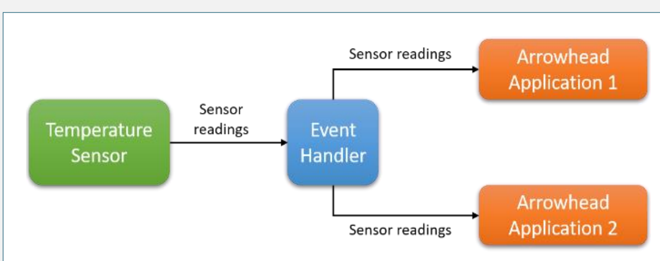


Event Handler

Used for sending periodic updates from a producer service to several consumer services. It is also based on a SOA approach.

- REST/HTTP(S) implementation of a publish-subscribe message broker
- Industrial applications require:
  - High throughput
  - Low end-to-end delay

However, the existing implementation of the Event Handler suffered from several end-to-end message latency problems (leading up to a maximum of almost 5 seconds) and also lead to unnecessarily high CPU and memory usage, which particularly affects resource-constrained machines.



Improving the Event Handler

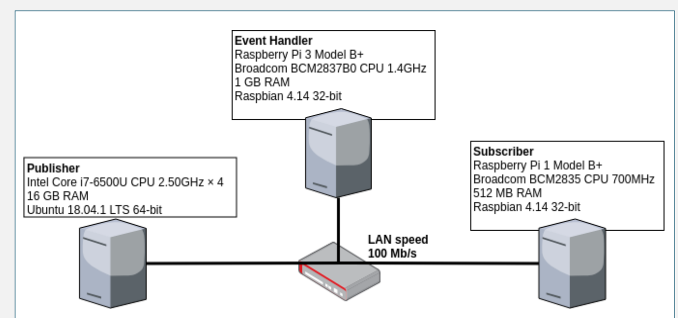
Problems:

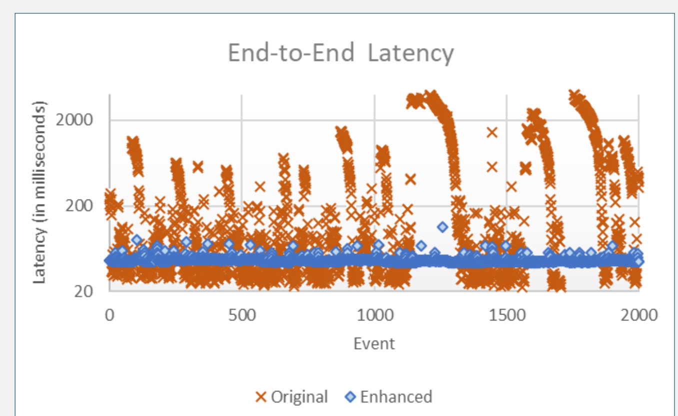
1. None of the three components (Producer, Consumer and the Event Handler) reused connections
2. Event Handler creates a new thread for every incoming request

Solutions:

1. A connection pool on the Publisher side
   - A request for a route for which the client already has a persistent connection available in the pool will be handled by renting a connection from the pool rather than creating a brand-new connection
2. Server-Sent Events between the Event Handler and Subscriber
   - Event Handler asynchronously sends data to Subscribers once the client-server connection is established by the Subscribers
3. Thread Pool in the Event Handler

Experimental results





This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234); also, by EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 737459 (Productive4.0 project) and by the Portuguese National Innovation Agency (ANI) under the European Regional Development Fund (FEDER), through the "Portugal 2020" (PT2020) partnership, within the framework of the System of Incentives to Research and Technological Development (SII&DT) and the Operational Program for Competitiveness and Internationalization (POCI), within project FLEXIGY, nº 34067 (AAC nº 03/SI/2017).

CISTER Research Centre
ISEP, Polytechnic Institute of Porto
Rua Dr. Antº Bernardino de Almeida, 431
4249-015 Porto, Portugal

+351 228 340 502
www.cister.isep.ipp.pt
cister-info@isep.ipp.pt
facebook.com/cisterrealtime

isep Instituto Superior de Engenharia do Porto
P.PORTO

Appendix C

Accepted paper for IECON 2019

Improving and modelling the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia,
Luis Lino Ferreira
CISTER Research Center, ISEP
Polytechnic Institute of Porto
Porto, Portugal
{rtdrh, crr, llf}@isep.ipp.pt

Pal Varga
Dept. of Telecomm. and Media Informatics, Budapest University of Technology and Economics
Budapest, Hungary
pvarga@tmit.bme.hu

***Abstract*—The Event Handler – a publish-subscribe broker implemented over REST/HTTP(S) – is an auxiliary system of the Arrowhead framework for Industrial IoT applications. However, during the course of this work we found that the existing implementation of the Event Handler suffers from serious performance issues. This paper describes the reengineering process that ultimately enabled it to reach much more acceptable levels of performance, by using appropriate software configurations and design patterns. Additionally, we also illustrate how this enhanced version of the Event Handler can be modeled using Petri nets, to depict the performance impact of different thread pool configurations and CPU core availability. Where the main objective of this model is to enable the prediction of the system performance to guarantee the required quality of service.**

Keywords—Performance, Publish-Subscribe, HTTP, REST, SOA, Java, Petri Nets, Real-Time

I. Introduction

The Arrowhead Framework [1] aims at using a service-oriented approach (SOA) for IoT applications, by providing a set of services [1] that support the interaction between applications, such as services capable of providing sensor readings. One of the available Arrowhead systems, the Event Handler (EH), is used to propagate updates from a producer service to one or more consumer applications. In this sense, the EH serves as a REST/HTTP(S) implementation of a publish-subscribe message broker, handling the distribution of messages (or events) from publishers to multiple subscribers (as is portrayed on Fig. 1).

For an Arrowhead publisher service to continuously notify its subscribers within its performance requirements, the EH's performance is of extreme importance. There are two important performance parameters to take into account in a publish-subscribe setting: i) the end-to-end delay for a message to go from a producer to a consumer; and ii) the message throughput. i.e., the number of messages which can be sent per time unit and processed by the EH. These two performance parameters are evaluated in this work and then modelled using Petri nets in order to enable offline analysis for each scenario.

However, the existing implementation of the EH suffers from several end-to-end message latency problems (leading up to a maximum of almost 5 seconds to deliver some messages), mostly due to the wasteful creation of threads and HTTP connections, which also lead to unnecessarily high CPU and memory usage, which particularly affects resource-constrained host machines. Therefore, a code refactoring was necessary in order to achieve an average end-to-end latency of 10 ms.

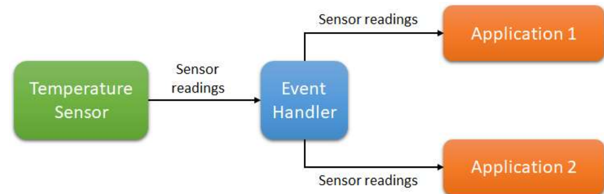


Fig. 1. A simplified representation of the Event Handler system where a sensor monitoring application publishes data that is consumed by two Arrowhead applications via Event Handler.

This paper starts with a brief description of the Arrowhead Framework and its EH system, which highlights its performance issues and how it can be solved. Then it follows by showing how to model the EH using Petri Nets. Finalizes with some conclusions about the work.

II. The Event Handler

A. The Arrowhead Framework

The Arrowhead Framework is the result of a set of European projects in which SOA principles have been applied to IoT and industrial applications. As the main result of the Arrowhead project, the framework continued its development independently and is now being used in multiple industrial installations and further developed in other projects. It aims at enabling all systems to work in a common and unified approach – leading towards high levels of interoperability. The software framework includes a set of Core Services [1] (service discovery, orchestration and authentication) that support the interaction between Application Services.

The Arrowhead Framework builds upon the local cloud concept, where local automation tasks should be encapsulated and protected from outside interference. Application services are not able to communicate with services outside the local cloud (intra-cloud orchestration), except with other Arrowhead compliant local clouds (inter-cloud orchestration). Each local cloud must contain, at least, the three mandatory core systems: Service Registry, Authorization and Orchestration. Thus, enabling the communication between Arrowhead application services. These core systems are then accompanied by automation supporting services that further improve the core capabilities of a local cloud, from measuring quality of service to enabling message propagation between multiple systems. The Event Handler (EH) is one of these supporting systems.

B. The Event Handler (original version)

The (Arrowhead's) EH uses a REST-based architecture implemented on top of Grizzly [2] and Jersey [3]. Grizzly comprises: i) a core framework that facilitates the

development of scalable event-driven applications using Java Non-blocking I/O API, and ii) both client-side and server-side HTTP services. Jersey is a framework that facilitates the development of RESTful Web Services and its clients, by providing an implementation of the standard JAX-RS API (which is the "standard" specification for developing REST services in Java) and some extensions.

The standard use of Jersey (which uses servlets as its underlying mechanism) will lead to the creation of a new thread for each request and then destroy the thread after its work is completed. Thus, RESTful services using standard Jersey will slow down when there are thousands of requests sent at the same time or at a very fast pace (later explored in section II-C3). In order to solve this problem, several implementations of servlet containers (also known as web containers) can provide a thread pool, which reuses previously created threads to execute current tasks and offers a solution to the problem of thread creation overhead and resource consumption. This in turn lowers the thread creation responsibility down a layer below Jersey and to the web container [4]. Grizzly is a popular implementation of these web containers.

However, the Grizzly HTTP server module in the EH does not currently have a configured thread pool. Thus, it will most likely not be able to efficiently handle multiple requests. Moreover, for the client applications that are meant to use the EH, i.e. the publishers and subscribers, the Arrowhead Consortia provides client skeletons to be extended with the developers' own application code [5]. These client skeletons use the same Jersey/Grizzly setup and server configuration as the Arrowhead systems.

1) The testing environment

In order to evaluate the EH's performance, we conducted a test on the system, with one Publisher sending 2000 events (sequentially, with no delay) to the EH, which connects to just one Subscriber. Each request is 71 bytes long, on a 100 Mb/s Switched Ethernet LAN. To measure the latency between Publisher, EH, and Subscriber, each time one of these components sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. We deployed the EH and the Subscriber on Raspberry Pis.

There are two main reasons to use this platform: i) when testing software in a resource-constrained platform, bottlenecks become more obvious and easier to identify; ii) Raspberry Pi hardware is heavily documented and its usage is widespread for industrial and IoT applications. The testing environment is displayed in Fig. 2, basically constituted by a publisher, a subscriber and the EH, with all clocks synchronized using a local NTP server, which provides accuracies in the range of 0.1 ms [6].

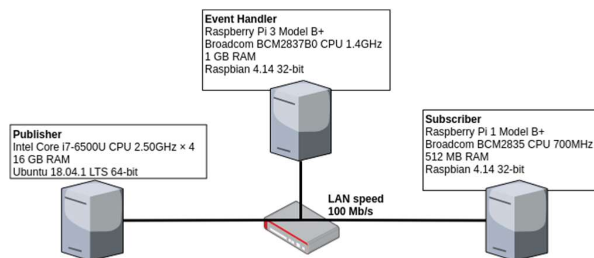


Fig. 2. Testing environment for the official Event Handler.

2) Performance evaluation

After sending 2000 events to the original EH, 41.9% of these events had an end-to-end latency greater than 100 ms, and 20.3% of these had a latency greater than 1 s, with an average of approximately 666.3 ms. Moreover, the maximum latency reaches the 4.9 s. This type of performance is a symptom of a bottleneck in the system. Consequently, the official implementation of the EH was revised.

C. Improving the Event Handler

A code analysis was performed to the Publisher, the EH, and the Subscriber. Two major problems were detected. The first problem was that none of the three components reused connections. This has a major performance impact on communications, since establishing a connection from one system to another is rather complex and consists of multiple packet exchanges between two endpoints (connection handshaking), which can cause major overhead, especially for small HTTP messages [7]. In fact, a much higher data throughput is achievable if open connections are re-used to execute multiple requests. This problem required a different solution for the three systems: a) the Publisher had to use a connection pool so that it could reuse its connections to the EH (see section II-C1); b) the EH had to use Jersey's own Server-Sent Events mechanism to establish a persistent connection to each of its Subscribers (see section II-C2). The second problem consisted in the EH creating a new thread for every incoming request, which would then greatly impact the machine's available RAM and response times. Thus, the EH required a thread pool to manage incoming requests in a less wasteful manner, as threads can be reused among different requests (see section II-C3).

1) Reuse open connections between the Publisher and the Event Handler

In order to reuse open connections between the Publisher and the EH, the best choice was to implement a connection pool on the Publisher, via the Apache HTTP Client on Jersey's transport layer. On an Apache HTTP Client [7], the client maintains a maximum number of connections on a per endpoint basis (which can be configured), so a request for an endpoint for which the client already has a persistent connection available in the pool will be handled by reusing a connection from the pool rather than creating a brand-new connection.

On our setup, only one connection per route was set in order to maintain message order, since using multiple parallel connections might lead to the processing of messages out of order.

2) Establish a persistent connection between the Event Handler and each Subscriber

The EH also did not reuse previously created connections to its subscribers, consequently adding a large overhead on each message end-to-end delay, due to the establishment of a connection. Thus, to avoid creating a connection to each subscriber on every request, we used Jersey's Server-Sent Events (SSE) [8] mechanism in the new implementation of the EH.

The SSE mechanism can be used to handle a one-way publish-subscribe model. When the Subscriber sends a

request to the EH, the EH holds a connection between itself and the Subscriber until a new event is published. When an event is published, the EH sends the event to the Subscriber, while keeping the connection open so that it can be reused for the next events. The Subscriber processes the events sent from the EH individually and asynchronously without closing the connection. Therefore, the EH can reuse one connection per Subscriber.

3) Reuse previously created threads in the Event Handler

As explained in section II-B, if the Grizzly HTTP server's threadpool is not configured, Grizzly follows Jersey's model of generating a new thread for each request, by default. In other words, with every wave of two thousand requests sent to the EH, Jersey will allocate 2000 server threads almost simultaneously and closes them soon afterwards [9]. Naturally, this leads to a great amount of overhead (thread creation and teardown and context switching between thousands of threads) and a large consumption of system memory (host OS must dedicate a memory block for each thread stack; with default settings, just four threads consume 1 Mb of memory [10]), which becomes largely inefficient.

The solution for this is to configure a thread pool on the Grizzly HTTP server module, which will reuse threads instead of destroying them. The key question is, what should be the optimal thread pool size for this scenario? While there is no clear-cut answer for this, it is usually suggested that if the HTTP request is CPU bound (as in this case), the amount of threads should be (at maximum) equal to the number of CPU cores in the host machine [11]. Otherwise, if the request is more I/O bound then more threads can successfully run in parallel. Therefore, the empirical process of identifying the optimal pool size consisted in starting with the same number of threads as the number of CPU cores and increasing them until there was no discernible improvement in throughput. Through this process, an interesting 10 ms average latency was achieved with a thread pool of 64 threads.

III. Performance Evaluation

After the major refactoring on the original EH, the "enhanced" version was put to the test on a similar environment and workload as the original. After repeating the same testing process, the test results were exceedingly better than the previous version's (see Fig. 3), with an average end-to-end latency of approximately 8.95 ms and a maximum latency of 32.00 ms.

Additionally, two other scenarios were tested: 1) instead of 2000 events, the Publisher shall send 9000 events, in order to detect potential bottlenecks; 2) the same scenario as scenario 1, however, instead of using a single Subscriber, six different Subscribers were used. Test results showed a similar performance increase. For scenario 1, the average end-to-end latency was 8.98 ms, with a maximum latency of 52.00 ms. As for scenario 2, the average end-to-end latency was 10.68 ms, with a maximum latency of 45.67 ms, measured between all six subscribers. A histogram with the end-to-end latency distribution for these two scenarios is displayed in Fig. 4.

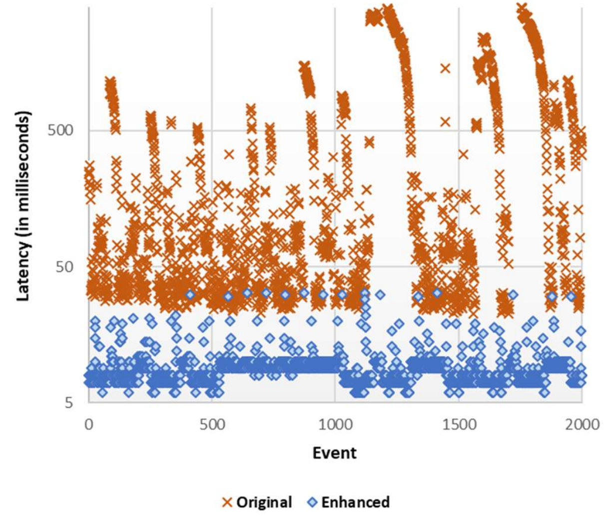


Fig. 3. End-to-end latency of the two versions of the Event Handler.

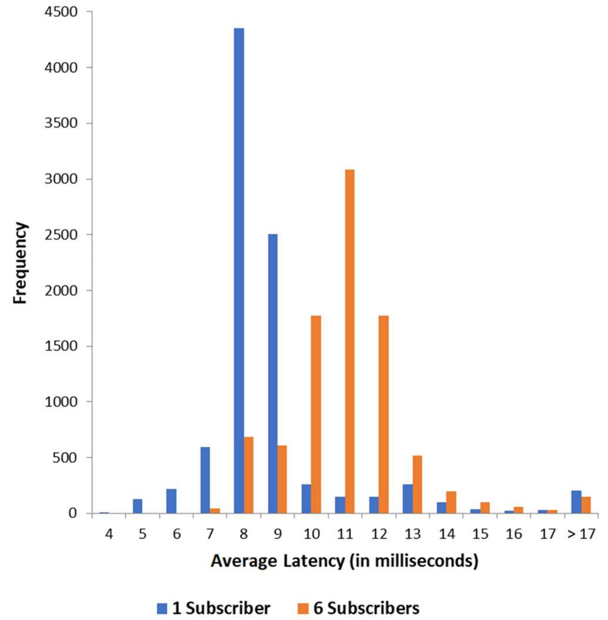


Fig. 4. End-to-end latency distribution of 9000 events for one subscriber and six subscribers, with the enhanced Event Handler.

IV. Modelling the Event Handler's Performance

In order to be able to predict the performance of different applications supported by the EH system, it is necessary to take into account specific thread pool configurations, number of CPU cores and communication latencies and model it. Such a model was developed using Petri nets, which easily allows modeling systems that deal with concurrent activities [12, 13], such as communication networks, multiprocessor systems, and manufacturing systems.

To develop this Petri net model, we adapted Lu & Gokhale's methodology [14] which has been previously used to model the performance of a Web server with a thread pool architecture. The resulting Petri net is displayed in Fig. 5. For the stochastic analysis of the model, we decided to use Oris

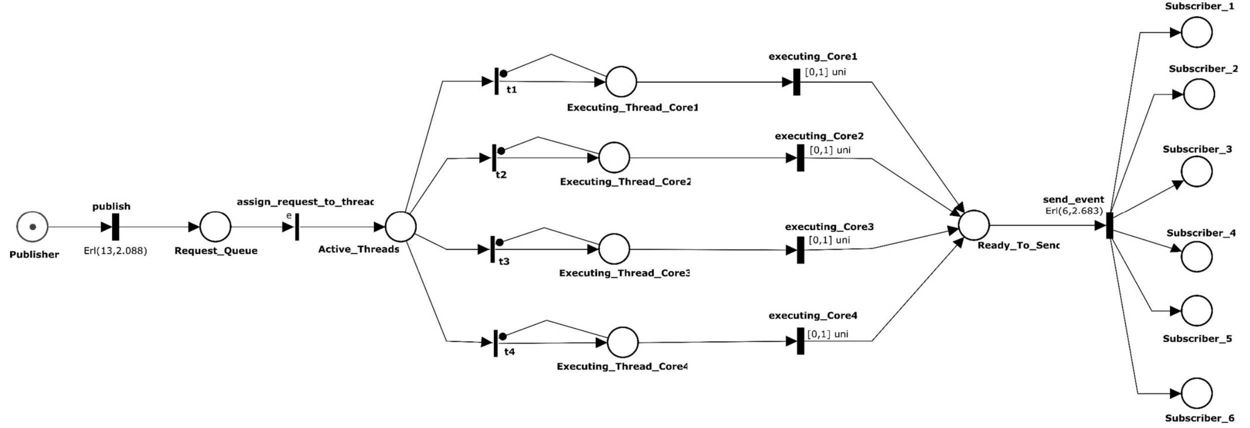


Fig. 5. Stochastic Petri net model of the Event Handler running on a quad-core CPU.

Tool [15] since it was one of the only open source tools with stochastic Petri nets analysis capabilities.

A. Explaining the Petri net model

Our model characterizes the EH's execution in a quad-core CPU (since real results have been obtained using a Raspberry Pi 3 Model B). Regarding the model itself, the *Publisher* place (the circle on the left) represents the Publisher, and the *publish* transition (the black vertical wide bar) represents the time it takes for a published event to be transmitted and reach the EH. The *Request_Queue* place holds unprocessed requests, while the *assign_request_to_thread* transition represents the EH thread pool limit – only assigning requests to a thread if the total number of active threads (represented by the token sum in *Active_Threads*, *Executing_Thread_CoreX,* and *Ready_to_Send* places) has not exceeded the specified limit. In the Petri net, this condition is executed through an enabling function (i.e. a boolean expression) in the transition, hence the letter "*e*" next to the *assign_request_to_thread* transition. Once a request is assigned to a thread, the thread is executed by one of the CPU cores. The *Executing_Thread_CoreX* place (*X* should be replaced by the corresponding core) represents the thread's execution, while the *executing_CoreX* transition represents the amount of time it takes to execute. An inhibitor arc (which is used to mandate that the transition must only fire when the place has no tokens) is used from *Executing_Thread_CoreX* to the respective *tX* transition to avoid the firing of transition *tX* when *Executing_Thread_CoreX* already has a token, therefore guaranteeing that only one request is being executed on a specific CPU core. Once the *executing_CoreX* transition finishes, it sends a token to *Ready_to_Send*, where the event is ready to be sent to its subscribers.

Several real experiments have been performed in order to fine tune the module with real data extracted from several test runs from where we derive the values for each request type (i.e., considering requests sent from Publisher to EH, requests sent from EH to each Subscriber), and the CPU execution time for each request, and determine their most appropriate probability distribution function to be applied in the Petri net model. We determined that the requests sent from the Publisher to the EH had a Gamma distribution with *shape* (k) = *13.235* and *rate* (λ) = *2.088*. However, Oris only provides transitions with an Erlang distribution which is a particular case of the Gamma distribution, where k should be an integer value. Similarly, the requests sent from the EH to its Subscribers also had a Gamma distribution with $k = 6.235$ and $\lambda = 2.683$, where k was then rounded to 6, to likewise satisfy the Erlang distribution requirements. Finally, the CPU execution times in the EH (i.e. *executing_CoreX*) were decided to be represented as transitions with a uniform distribution, where the early finish time is 0 ms and the late finish time is 1 ms.

B. Stochastic analysis of the Petri net model

Oris provides a tool for transient analysis which consists in analyzing the probability of a process transitioning from one place to the other at a specific instant in time. Thus, the analysis creates a chart – in which the "time" variable is used as the X-axis and the "possible arrival state" variable (in other words, the place probability) is used as the Y-axis, where each time instant represents a probability distribution, which means that the sum of all values in each time instant must equal 1. This chart is displayed in Fig. 6.

1) Interpreting the analysis results

First, the only places that are present in the chart are *Publisher*, *Ready_to_Send*, *Executing_Thread_CoreX* and *Subscriber_#*. The reason for this is because the other places (aside from *Request_Queue*) only depend on immediate transitions, thus the token will not spend any time in these places, meaning that these do not have an impact in the overall processing time. Although *Request_Queue* is linked to an immediate transition (i.e. *assign_request_to_thread*), this transition is restricted to the EH's thread pool size, which (as explained previously) is represented by the token sum in the *Active_Threads* place, the *Executing_Thread_CoreX* places, and the *Ready_to_Send* place. Since only one token is sent in this particular analysis, *Request_Queue* will not be storing any tokens, thus it will not be present in this chart.

Until time 2.1 ms, the probability of a token being in *Publisher* is approximately 1, whereas the other places are

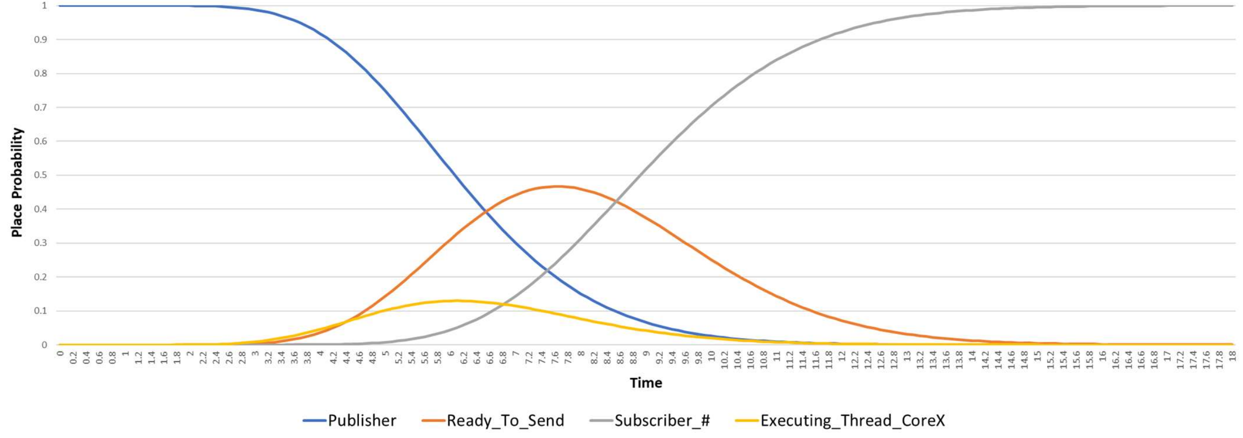


Fig. 6. Transient analysis of the Petri net model with one token.

still 0, because the Publisher takes at least 2 ms to send an event to the EH. Between time 2 and 13.6 ms, the probability of the Publisher sending a message decreases nonlinearly to 0, while the exact opposite happens to the Subscriber, i.e., the probability that *Subscriber_#* has received the token rises nonlinearly to 1. In fact, at time 7.6 ms, the two curves cross each other, which means that, beyond this point, there is a higher probability of an event having reached the respective Subscriber, than it still being published by the Publisher. Furthermore, from 2.1 to 13.5 ms, the probability of the token being in *Executing_Thread_CoreX* has an almost Gaussian distribution, which means that once the message is sent from the Publisher, it is processed by the EH for a maximum of 1 second. After this process, the message is then ready to be sent. Indeed, from 2.5 to 17 ms, similar to *Executing_Thread_CoreX,* the probability of the token being in *Ready_to_Send* also has a Gaussian distribution, meaning that once the EH is ready to send the published event, the Publisher has already sent the message, and the Subscriber is about to receive it – hence the probability decrease in *Publisher* and the increase in *Subscriber_#* right after the probability peak in *Ready_to_Send*.

According to the analysis's time estimations, the "maximum" time it takes to send an event (i.e. with a 99% chance) from the Publisher to the EH (i.e., when the probability for the *Publisher* place reaches approximately 0) is around 13.6 ms, while the estimated "latest" time for a Subscriber to receive an event (i.e., when the probability for the *Subscriber_#* places reaches approximately 1) is around 17.1 ms. Nevertheless, there is a 99% chance that Subscribers will receive the published event around 14.3 ms. Furthermore, the probability for the *Ready_To_Send* place to hold a token peaks (47%) at the 7.6 ms, which means that the EH is ready to send the published event to its subscribers at this instant, 47% of the times.

2) Comparing the model with the actual experiments

Overall, the values collected from the model match the results obtained in the experiments of the enhanced EH. In the Petri Net model, the probability distribution for *Subscriber_#* to receive the published message is only higher than *Publisher*, *Executing_Thread_CoreX*, and *Ready_to_Send* after the 8.5 ms instant. One can see that this value is matched with the experiment results for one Subscriber reported in Fig. 4, where it is possible to see that around 60% of the events were delivered with an 8 ms latency. Whereas for the six Subscribers tests, where the average end-to-end latency is approximately 10.68 ms, the corresponding probability distribution is 80.4%.

C. Validating the Petri net model

In addition to the initial stochastic analysis with one token, another stochastic analysis was performed with four tokens to examine how the model scales with the processing of multiple messages. The same transient analysis matrix was calculated, and the distribution of the estimated end-to-end latencies for four messages is depicted in Fig. 7, juxtaposed with the real test results from Fig. 4. Unfortunately, due to some processing limitations of the Oris tool, we were unable to assess the performance for more than four tokens. Nevertheless, this stochastic analysis with four tokens is able to capture the latency interval for most messages, i.e. from 8 to 16 ms, which mostly goes in hand with the event latency distributions of the test results. However, the authors feel that these latency estimations must still be further improved in order to fine tune the probability for each latency and also to capture a wider range of latencies, since the more extreme latencies (i.e. below 8 ms and above 17 ms) are not represented. In terms of improving these estimations, this could be done by: i) changing the probability distributions and the parameters chosen for each transition; or ii) changing the Petri net model itself.

V. Conclusions and future work

By changing how the original EH and its clients handled HTTP requests and thread creation, the enhanced version of the EH is now able to achieve the initial goal of reaching an average end-to-end latency of 10 ms. In fact, by considering the average latencies of both versions, it is safe to say that the EH had an overall performance boost of over 93%. Nonetheless, the authors agree that the system's performance might still be able to improve even further than its current state by optimizing the EH's thread pool size and the Publisher's connection pool. Moreover, we propose a Petri net model for the EH in order to estimate the overall end-to-

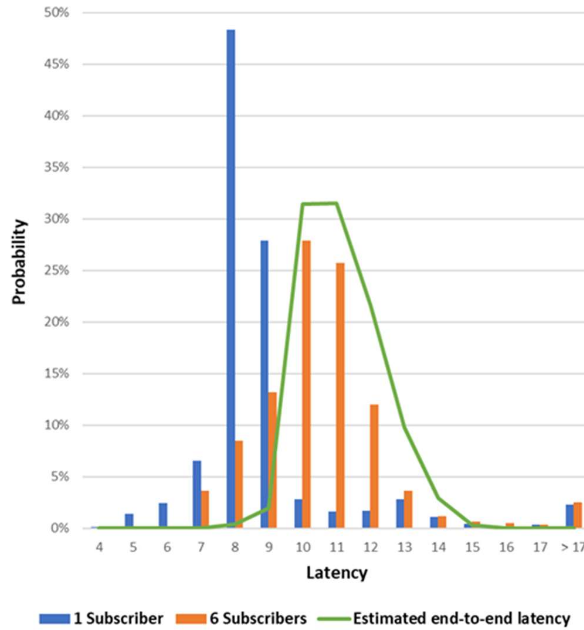


Fig. 7. Estimated end-to-end latency probability for four messages.

end latency probability of each component (Publisher, EH, and Subscribers). Results show that the model provides a good estimation of results. However, it could still be further improved, either by changing the probability distributions and their parameters chosen for each transition or by editing the Petri net model itself. Nevertheless, these questions are expected to be the focus for future research work.

Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234); also, by EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 737459 (Productive4.0 project) and by the Portuguese National Innovation Agency (ANI) under the European Regional Development Fund (FEDER), through the "Portugal 2020" (PT2020) partnership, within the framework of the System of Incentives to Research and Technological Development (SII&DT) and the Operational Program for Competitiveness and Internationalization (POCI), within project FLEXIGY, nº 34067 (AAC nº 03/SI/2017).

References

[1] P. Varga, et al, Making system of systems interoperable – The core components of the arrowhead framework, Journal of Network and Computer Applications, Volume 81, 2017, Pages 85-95, ISSN 1084-8045.

[2] Project Grizzly, Javaee.github.io, 2019. [Online]. Available: https://javaee.github.io/grizzly/. [Accessed 22 Mar 2019].

[3] Jersey, Jersey.github.io, 2019. [Online]. Available: https://jersey.github.io/. [Accessed 22 Mar 2019].

[4] Jersey @ManagedAsync and copying data between HTTP thread and Worker thread, Stack Overflow, 2019. [Online]. Available: https://stackoverflow.com/questions/31137134/jersey-managedasync-and-copying-data-between-http-thread-and-worker-thread. [Accessed 21 Mar 2019].

[5] Arrowhead Consortia, GitHub, 2019. [Online]. Available: https://github.com/arrowhead-f. [Accessed 22 Mar 2019].

[6] D. Mills, Network Time Synchronization Research Project, Eecis.udel.edu, 2019. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp.html. [Accessed 21 Mar 2019].

[7] The Apache Software Foundation, Chapter 2. Connection management, Hc.apache.org, 2019. [Online]. Available: http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html. [Accessed 21 Mar 2019].

[8] Project Jersey, Chapter 14. Server-Sent Events (SSE) Support, Docs.huihoo.com, 2019. [Online]. Available: . [Accessed 21 Mar 2019].

[9] N. Babcock, Know Thy Threadpool: A Worked Example with Dropwizard, Nbsoftsolutions.com, 2016. [Online]. Available: https://nbsoftsolutions.com/blog/know-thy-threadpool-a-worked-example-with-dropwizard. [Accessed 21 Mar 2019].

[10] Why using many threads in Java is bad, Iwillgetthatjobatgoogle.tumblr.com, 2012. [Online]. Available: http://iwillgetthatjobatgoogle.tumblr.com/post/38381478148/why-using-many-threads-in-java-is-bad. [Accessed 21 Mar 2019].

[11] Project Grizzly, "Project Grizzly - Best Practices", Javaee.github.io, 2018. [Online]. Available: https://javaee.github.io/grizzly/bestpractices.html. [Accessed 21 Mar 2019].

[12] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Transactions on Software Engineering, Volume 32, Issue7, pages 486–502, 2006.

[13] W. M. Zuberek. Timed petri nets in modeling and analysis of manufacturing systems. Emerging Technologies, Robotics and Control Systems, Volume 1, 2007.

[14] J. Lu, S. Gokhale, Performance Analysis of a Web Server with Dynamic Thread Pool Architecture, Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering, 2010, Pages 99-105.

[15] M. Paolieri, et al, Oris Tool - Analysis of Timed and Stochastic Petri Nets, Oris-tool.org, 2019. [Online]. Available: https://www.oris-tool.org/. [Accessed 15 May 2019].

References

[1] Productive 4.0. Productive 4.0 - a european co-funded innovation and lighthouse project on digital industry. Productive 4.0. Available at https://productive40.eu/, Accessed last time in 31 October 2018, 2017. Cited on pages 2 and 15.

[2] Adlink. Vortex dds. Adlink. Available at http://www.prismtech.com/vortex, Accessed last time in 20 December 2018. Cited on page 22.

[3] K. Ahmad. Why cloud computing is mandatory for industrial iot? Available at https://www.linkedin.com/pulse/why-cloud-computing-mandatory-industrial-iot-kabir-ahmad/, Accessed last time in 11 July 2018, 2017. Cited on page 8.

[4] Alden. Jersey @managedasync and copying data between http thread and worker thread. Available at https://stackoverflow.com/questions/31137134/jersey-managedasync-and-copying-data-between-http-thread-and-worker-thread, Accessed last time in March 2019, 2019. Cited on page 29.

[5] P. G. Alves. *A Distributed Security Event Correlation Platform for SCADA*. PhD thesis, Faculdade de Ciências e Tecnologia da Universidade de Coimbra, Rua Sílvio Lima, Pólo II da Universidade de Coimbra, 3030-790 Coimbra, 2014. Cited on page 21.

[6] AMQP. Amqp is the internet protocol for business messaging. AMQP. Available at http://www.amqp.org/about/what, Accessed last time in 14 December 2018, 2018. Cited on page 18.

[7] AMQP. Products and success stories. AMQP. Available at http://www.amqp.org/about/examples, Accessed last time in 14 December 2018, 2018. Cited on page 19.

[8] O. Ansari. Micro services pipeline for an industrial internet of things (iiot) framework... towards data engineering. Medium. Available at https://bit.ly/2CBbVPz, [Accessed 31 October 2018], July 2018. Cited on page 9.

[9] C. Araujo. A software-defined industrial world: Using apis and microservices to enable industry 4.0 (white paper). Intellyx. Available at https://bit.ly/2BRcdQL, [Accessed 31 October 2018], March 2017. Cited on pages 9 and 10.

[10] Arrowhead. Arrowhead | ahead of the future. Arrowhead. Available at https://www.arrowhead.eu/arrowheadframework, Accessed last time in 05 December 2018, 2013. Cited on pages xi, 1, 2, and 12.

[11] Inc. Audio-Tech Business Book Summaries. Industry 4.0 and the u.s. manufacturing renaissance. *Trends E-Magazine, Issue 146*, pages 4–10, jun 2015. Cited on page 8.

[12] Axway. Api builder and mqtt for iot – part 1. Axway Amplify Blog. Available at https://s3.amazonaws.com/www.appcelerator.com.images/MQTT_1.png, Accessed last time in 17 December 2018, 2018. Cited on pages xi and 20.

[13] Nick Babcock. Know thy threadpool: A worked example with dropwizard. Available at https://nbsoftsolutions.com/blog/know-thy-threadpool-a-worked-example-with-dropwizard, Accessed last time in March 2019, 2019. Cited on page 58.

[14] I Will Get That Job At Google blog. Why using many threads in java is bad. Available at http://iwillgetthatjobatgoogle.tumblr.com/post/38381478148/why-using-many-threads-in-java-is-bad, Accessed last time in March 2019, 2019. Cited on page 58.

[15] D. Buhr. Social innovation policy for industry 4.0. Friedrich-Ebert-Stiftung. Available at https://library.fes.de/pdf-files/wiso/11479.pdf, [Accessed 31 October 2018], 2017. Cited on page 1.

[16] X. Che and S. Maag. A passive testing approach for protocols in internet of things. *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 678–684, 2013. Cited on page 21.

[17] Ming Chen. Petri nets. Universitat Bielefeld. Available at https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html, Accessed last time in 26 December 2018. Cited on pages xi, 23, and 24.

[18] Arrowhead Consortia. Arrowhead framework 4.1. GitHub, Inc. Available at https://github.com/arrowhead-f/core-java, Accessed last time in 05 December 2018, 2013. Cited on pages xi, 11, 12, 13, 27, and 52.

[19] Arrowhead Consortia. Arrowhead framework client skeletons in java. GitHub, Inc. Available at https://github.com/arrowhead-f/client-java, Accessed last time in May 2019, 2019. Cited on page 52.

[20] Oracle Corporation. Interface servlet. Available at https://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html, Accessed last time in March 2019, 2019. Cited on page 28.

[21] Oracle Corporation. Project grizzly. Available at https://javaee.github.io/grizzly/, Accessed last time in March 2019, 2019. Cited on page 27.

[22] Oracle Corporation. Project grizzly - best practices. Available at https://javaee.github.io/grizzly/bestpractices.html, Accessed last time in March 2019, 2019. Cited on page 58.

[23] A. Corsaro. The data distribution service tutorial. PrismTech. Available at https://www.researchgate.net/publication/273136749_The_Data_Distribution_Service_Tutorial, Accessed last time in 20 December 2018, 2014. Cited on page 22.

[24] Dave Cridland. Openfire. ignite realtime. Available at https://www.igniterealtime.org/projects/openfire/, Accessed last time in 17 December 2018. Cited on page 22.

[25] Curious. How do i check if my data fits an exponential distribution? Available at `https://stats.stackexchange.com/questions/76994/how-do-i-check-if-my-data-fits-an-exponential-distribution`, Accessed last time in March 2019, 2019. Cited on page 41.

[26] P. Brizzi A. Lotito R. Tomasi D. Conzon, T. Bolognesi and M. A. Spirito. The virtus middleware: An xmpp based architecture for secure iot communications. *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, 2012. Cited on page 21.

[27] R. Fonseca D. Menasce, V. Almeida and M. Mendesr. A methodology for workload characterization of e-commerce sites. *Proc. First ACM Conf. Electronic Commerce*, pages 119–128, 1999. Cited on page 15.

[28] V. Almeida D. Menasce and L. Dowdy. *Capacity Planning and Performance Modeling—From Mainframes to Client-Server Systems*. Prentice Hall, 1994. Cited on page 15.

[29] V. Almeida D. Menasce and L. Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall, First edition, 2004. Cited on page 16.

[30] D. Renzel I. Koren R. Klauck D. Schuster, P. Grubitzsch and M. Kirsche. Global-scale federated access to smart objects using xmpp. *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*, pages 185–192, 2014. Cited on page 21.

[31] R. Davies. Industry 4.0 - digitalisation for productivity and growth. European Parliamentary Research Service. Available at `https://bit.ly/1Ms6C1N`, Accessed last time in 16 November 2018, September 2015. Cited on page 1.

[32] M. Rüsch E. Hofmann. Industry 4.0 and the current status as well as future prospects on logistics. *Computers in Industry*, pages 23–34, August 2017. Cited on page 1.

[33] S. Han U. Jennehag E. Sisinni, A. Saifullah and M. Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, November 2018. Cited on pages 6 and 7.

[34] Susanna Donatelli Elvio G. Amparore. The home of the new greatspn graphical editor. Available at `http://www.di.unito.it/~amparore/mc4cslta/editor.html`, Accessed last time in May 2019, 2019. Cited on page 37.

[35] Red Hat Enterprise. Topic exchange. Siguniang's Blog. Available at `https://siguniang.files.wordpress.com/2012/05/topic-exchange.png`, Accessed last time in 17 December 2018. Cited on pages xi and 19.

[36] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR Service-Oriented Computing Series. Upper Saddle River, NJ 07458, USA: Prentice Hall, Pearson Education, Inc.,, First edition, 2007. Cited on page 13.

[37] D. Augenstein F. Schoenthaler and T. Karle. Design and governance of collaborative business processes in industry 4.0. *Proceedings of the Workshop on Cross-organizational and Cross-company BPM (XOC-BPM) co-located with the 17th IEEE Conference on Business Informatics (CBI 2015)*, August 2015. Cited on page 7.

[38] The Apache Software Foundation. Apache activemq. The Apache Software Foundation. Available at http://activemq.apache.org/, Accessed last time in 17 December 2018. Cited on pages 19 and 21.

[39] The Apache Software Foundation. Apache qpid. The Apache Software Foundation. Available at https://qpid.apache.org/, Accessed last time in 17 December 2018. Cited on page 19.

[40] The Apache Software Foundation. Chapter 2. connection management. Available at http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html, Accessed last time in March 2019, 2019. Cited on page 53.

[41] Gianfranco Balbo Gianni Conte Giovanni Chiola, Marco Ajmone Marsan. Generalized stochastic petri nets: A definition at the net level and its implications. In *IEEE Transactions on Software Engineering*, February 1993. Cited on page 25.

[42] N. Golchha. Big data-the information revolution. *International Journal of Applied Research, vol. 1, no. 12*, pages 791–794, sep 2015. Cited on page 8.

[43] W. Wahlster H. Kagermann and J. Helbig. Recommendations for implementing the strategic initiative industrie 4.0: Final report of the industrie 4.0 working group. acatech. Available at https://bit.ly/2PuOZ8h, Accessed last time in 03 July 2018, April 2013. Cited on page 6.

[44] P. Wang H. Wang, D. Xiong and Y. Liu. A lightweight xmpp publish/subscribe scheme for resourceconstrained iot devices. *IEEE Access 5 (2017)*, pages 16393—16405, 2017. Cited on page 22.

[45] D. Siegel H.A. Kao, W. Jin and J. Lee. A cyber physical interface for automation systems-methodology and examples. *Machines*, pages 96–106, May 2015. Cited on pages 7 and 8.

[46] Serge Haddad. Stochastic petri net. ENS Paris-Saclay & CNRS & Inria. Available at http://www.lsv.fr/~haddad/PNcourse-part2.pdf, Accessed last time in 15 June 2018. Cited on page 24.

[47] HiveMQ. Hivemq. HiveMQ. Available at https://www.hivemq.com/, Accessed last time in 17 December 2018. Cited on page 21.

[48] HornetQ. Hornetq. HornetQ. Available at http://hornetq.jboss.org/, Accessed last time in 17 December 2018. Cited on page 21.

[49] C. Hunsaker. Rest vs soap: When is rest better? Stormpath. Available at https://stormpath.com/blog/rest-vs-soap, Accessed last time in 11 December 2018, 2018. Cited on page 17.

[50] IBM. What is cloud computing? Available at https://www.ibm.com/cloud/learn/what-is-cloud-computing, Accessed last time in 25 Oct 2018. Cited on page 8.

[51] L. L. Ferreira M. Albano P. P. Pereira O. Carlsson H. Derhamy J. Delsing, P. Varga. *The Arrowhead Framework architecture, Chapter 3 of IoT Automation: Arrowhead Framework*. CRC Press, 2017. Cited on pages 10 and 11.

[52] A. Jukan X. Masip-Bruin J. Dizdarević, F. Carpio. A survey of communication protocols for internet-of-things and related challenges of fog and cloud computing integration. *ACM Computing Surveys, Vol. 1*, apr 2018. Cited on pages 18, 19, 20, 21, and 22.

[53] B. Bagheri J. Lee and H.-A. Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Proceedings of the Third Conference on Object-Oriented Technologies and Systems*, pages 18–23, January 2015. Cited on page 7.

[54] J. Ding J. Lyu and H. Luh. Petri nets for performance modelling study of client-server systems. *International Journal Of Systems Science*, pages 565–571, 1998. Cited on page 23.

[55] Swapna S. Gokhale Jijun Lu. Performance analysis of a web server with dynamic thread pool architecture. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, pages 99–105, January 2010. Cited on pages 67 and 69.

[56] A. Colombo G. Kreutz K. Nagorny, R. Harrison. A formal engineering approach for control and monitoring systems in a service-oriented environment. *IEEE International Conference on Industrial Informatics (INDIN)*, page 480–487, 2013. Cited on page 13.

[57] H. Kagermann. Change through digitization – value creation in the age of industry 4.0. In A. Pinkwart H. Albach, H. Meffert and R. Reichwald, editors, *Management of Permanent Change*. Springer Fachmedien Wiesbaden, 2015. Cited on page 6.

[58] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering, Volume 32, Issue 7*, pages 486–502, 2006. Cited on pages 15 and 23.

[59] W. He L. D. Xu and S. Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, November 2014. Cited on page 7.

[60] X. Liu L. Sha, S. Gopalakrishnan and Q. Wang. Cyber-physical systems: A new frontier. *Machine Learning in Cyber Trust*, 2009. Cited on page 5.

[61] H. Lellelid. Coilmq. GitHub. Available at `https://github.com/hozn/coilmq`, Accessed last time in 17 December 2018. Cited on page 21.

[62] J. Lewis and M. Fowler. Microservices. Martin Fowler. Available at `https://martinfowler.com/articles/microservices.html`, [Accessed 31 October 2018], March 2014. Cited on pages xi and 10.

[63] J. Silva R. Duarte L. L. Ferreira M. Albano, P. M. Barbosa and J. Delsing. Quality of service on the arrowhead framework. *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, pages 1–8, jun 2017. Cited on page 10.

[64] A. Gani A. Karim I. Abaker Targio Hashem A. Siddiqa I. Yaqoob M. Marjani, F. Nasaruddin. Big iot data analytics: Architecture, opportunities, and open research challenges. *IEEE Access 5*, mar 2017. Cited on page 8.

[65] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1989*. Springer Berlin Heidelberg, 1990. Cited on page 16.

[66] David Mills. Network time synchronization research project. Available at https://www.eecis.udel.edu/~mills/ntp.html, Accessed last time in March 2019, 2019. Cited on page 31.

[67] Mike Moore. Industry 4.0: the fourth industrial revolution – guide to industrie 4.0. i-SCOOP. Available at https://www.i-scoop.eu/industry-4-0/, Accessed last time in 03 July 2018. Cited on page 5.

[68] Mike Moore. What is industry 4.0? everything you need to know. TechRadar. Available at https://www.techradar.com/news/what-is-industry-40-everything-you-need-to-know, Accessed last time in 03 July 2018, April 2018. Cited on pages 5 and 6.

[69] Mosquitto. Mosquitto. Mosquitto. Available at https://mosquitto.org/, Accessed last time in 17 December 2018. Cited on page 21.

[70] MuleSoft. What is a restful api? MuleSoft. Available at https://www.mulesoft.com/resources/api/restful-api, Accessed last time in 11 December 2018, 2018. Cited on page 17.

[71] OASIS. Mqtt version 3.1.1 - oasis standard. OASIS. Available at http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, Accessed last time in 17 December 2018, 2014. Cited on page 20.

[72] OASIS. Oasis. OASIS. Available at https://www.oasis-open.org/, Accessed last time in 14 December 2018, 2018. Cited on page 18.

[73] Service Objects. Why rest is so popular. Service Objects. Available at https://www.serviceobjects.com/resources/articles-whitepapers/why-rest-popular, Accessed last time in 11 December 2018, 2018. Cited on page 17.

[74] OpenDDS. Opendds. OpenDDS. Available at http://opendds.org/, Accessed last time in 20 December 2018. Cited on page 22.

[75] Oracle. Compute services: How an integrated approach to cloud will pave the way for innovation. Available at https://go.oracle.com/LP=38751?elqCampaignId=45117/?, Accessed last time in 11 July 2018, 2016. Cited on page 8.

[76] Eclipse Foundation Oracle Corporation. Chapter 1. getting started. Available at https://jersey.github.io/documentation/latest/getting-started.html, Accessed last time in March 2019, 2019. Cited on page 28.

[77] Eclipse Foundation Oracle Corporation. Chapter 11. asynchronous services and clients. Available at https://jersey.github.io/documentation/latest/async.html, Accessed last time in March 2019, 2019. Cited on page 29.

[78] Eclipse Foundation Oracle Corporation. Chapter 15. server-sent events (sse) support. Available at https://jersey.github.io/documentation/latest/sse.html, Accessed last time in March 2019, 2019. Cited on page 54.

[79] Eclipse Foundation Oracle Corporation. Jersey. Available at https://jersey.github.io/, Accessed last time in March 2019, 2019. Cited on page 27.

[80] L. L. Ferreira J. Eliasson M. Johansson J. Delsing I. Martinez de Soria P. Varga, F. Blomstedt. Making system of systems interoperable - the core components of the arrowhead framework. *Journal of Network and Computer Applications*, pages 85–95, mar 2017. Cited on pages xi, 10, 11, and 14.

[81] Marco Paolieri. Oris tool - analysis of timed and stochastic petri nets. Available at https://www.oris-tool.org/, Accessed last time in May 2019, 2019. Cited on page 37.

[82] A. Piper. Choosing your messaging protocol: Amqp, mqtt, or stomp. VMware Blogs. Available at https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html, Accessed last time in 14 December 2018, 2018. Cited on pages 18, 20, and 21.

[83] ProcessOne. ejabberd. ProcessOne. Available at https://www.process-one.net/en/ejabberd, Accessed last time in 17 December 2018. Cited on page 22.

[84] J. Coughlin R. Mital and M. Canaday. Using big data technologies and analytics to predict sensor anomalies. *Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference*, page 84, sep 2014. Cited on page 8.

[85] L. L. Ferreira F. Relvas R. Rocha, M. Albano and L. Matos. The arrowhead framework applied to energy management. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–10, June 2018. Cited on page 10.

[86] RabbitMQ. Rabbitmq. RabbitMQ. Available at https://www.rabbitmq.com/, Accessed last time in 17 December 2018. Cited on pages 19 and 21.

[87] Luis Lino Ferreira Pedro Souto Pal Varga Rafael Rocha, Cláudio Maia. Improving the performance of a publish-subscribe message broker. Available at www.cister.isep.pt/docs/1500, Accessed last time in May 2019, 2019. Cited on page 76.

[88] Lui Sha John Stankovic Ragunathan Rajkumar, Insup Lee. Cyber-physical systems: The next computing revolution. *Proceedings of the 47th Design Automation Conference, DAC '10*, pages 731–736, June 2010. Cited on page 5.

[89] C. Richardson. Pattern: Monolithic architecture. Microservice Architecture. Available at https://microservices.io/patterns/monolithic.html, [Accessed 31 October 2018], 2017. Cited on page 9.

[90] Christoph Roser. Industrial revolutions and future view. AllAboutLean.com. Available at AllAboutLean.com, Accessed last time in 03 July 2018, 2015. Cited on pages xi and 6.

[91] Philip Russom. *Big Data Analytics*. TDWI Research, 2011. Cited on page 8.

[92] J. Schabowsky. Rest vs messaging for microservices – which one is best? Solace. Available at https://solace.com/blog/products-tech/experience-awesomeness-event-driven-microservices, Accessed last time in 11 December 2018, 2017. Cited on page 17.

[93] PROMATIS software GmbH. Smart factory, smart supply chain ... are oracle applications smart enough? PROMATIS software GmbH. Available at https://www.promatis.ch/wp-content/uploads/sites/13/2015/04/WP_Smart-Factory_e.pdf, Accessed last time in 31 October 2018, 2015. Cited on page 9.

[94] J. Speed. Rest is for sleeping. mqtt is for mobile. Mobilebit. Available at https://mobilebit.wordpress.com/2013/05/03/rest-is-for-sleeping-mqtt-is-for-mobile/, Accessed last time in 17 December 2018, 2013. Cited on page 20.

[95] STOMP. Implementations. STOMP. Available at https://stomp.github.io/implementations.html, Accessed last time in 17 December 2018. Cited on page 21.

[96] STOMP. Stomp. STOMP. Available at https://stomp.github.io, Accessed last time in 17 December 2018. Cited on page 21.

[97] STOMP. Stomp protocol specification, version 1.2. STOMP. Available at https://stomp.github.io/stomp-specification-1.2.html, Accessed last time in 17 December 2018. Cited on page 21.

[98] A. Stork. Visual computing challenges of advanced manufacturing and industrie 4.0. *IEEE Computer Graphics and Applications, vol. 35, no. 2*, pages 21–25, 2015. Cited on page 8.

[99] Rockliffe Systems. Astrachat. Rockliffe Systems. Available at http://www.astrachat.com/HostedForBusiness.aspx, Accessed last time in 17 December 2018. Cited on page 22.

[100] Tigase. Tigase xmpp server. Tigase. Available at https://tigase.net/content/tigase-xmpp-server, Accessed last time in 17 December 2018. Cited on page 22.

[101] Vortex. Quality of service. PrismTech. Available at http://download.prismtech.com/docs/Vortex/html/ospl/DDSTutorial/qos.html, Accessed last time in 20 December 2018. Cited on page 22.

[102] XMPP. History of xmpp. XMPP. Available at https://xmpp.org/about/technology-overview.html, Accessed last time in 17 December 2018. Cited on page 21.

[103] XMPP. An overview of xmpp. XMPP. Available at https://xmpp.org/about/technology-overview.html, Accessed last time in 17 December 2018. Cited on page 21.

[104] XMPP. Xmpp servers. XMPP. Available at https://xmpp.org/software/servers.html, Accessed last time in 17 December 2018. Cited on page 22.

[105] W. M. Zuberek. Timed petri nets in modeling and analysis of manufacturing systems. *Emerging Technologies, Robotics and Control Systems, Volume 1*, 2007. Cited on page 23.