CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Masters Thesis

Middleware for Large-scale Distributed Systems

César Ricardo da Silva Teixeira

CISTER-TR-151106

Middleware for Large-scale Distributed Systems

César Ricardo da Silva Teixeira

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Over the last few years the designing and implementation of applications have evolved to a new breed of applications that are used by a huge number of users at the same time and are capable of being executed in up to thousands of machines physically distributed, even geographically, such as the cloud computing systems, the new concept of “big data” and smart cities. The existence of several components of these systems, distributed in independent machines, brings inevitable issues in terms of designing and implementation of those systems in order to achieve flexible, scalable, robust, reliable and interoperable systems. It is extremely important to design and implement systems that can be capable of providing a communication and coordination among all the components of the system. The concept of implementing a Middleware seems to be a great option to solve most of these issues, allowing a system to communicate with other systems in a really fast, robust and secure way. The main goal of this thesis is to demonstrate that the usage of Middleware technologies to ensure the communication in distributed systems brings a huge number of advantages, such as interoperability between systems, robustness regarding the communication layer, scalability and high speed communications.

Instituto Superior de Engenharia do Porto

Middleware for Large-scale Distributed Systems

César Ricardo da Silva Teixeira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquitetura, Sistemas e Redes

Orientador: Professor Doutor Luís Lino Ferreira

Co-orientador: Doutor Michele Albano

Júri:

Presidente:

Professor Doutor Paulo Alexandre Fangueiro Oliveira Maio, Instituto Superior de Engenharia do Porto

Vogais:

Professor Doutor Pedro Alexandre Guimarães Lobo Ferreira Souto, Faculdade de Engenharia do Porto

Porto, Outubro 2015

Acknowledgements

I would like to take this opportunity to thank all the people that helped me throughout all these years of work to achieve a master's degree.

First of all, I would like to thank my teacher and advisor, Prof. Dr. Luís Lino Ferreira, who was extremely comprehensive and tolerant during the time I was writing and working on this thesis, providing me all the support and motivation I needed to accomplish this stage of my life.

Prof. Dr. Michele Albano, along with Prof. Dr. Luís Lino Ferreira was extremely important for me to get along with this thesis. He was always interested and supportive, trying to teach me everything I needed to work and to be able to complete this thesis.

There are no words to describe how much Prof. Dr. Luís Lino Ferreira and Prof. Dr. Michele Albano were important for me during the time I was writing this thesis, so I just would like to thank you both for helping me and most of all, for being true friends that I would never forget.

To my girlfriend, Rita Amaral, who I love the most and who was always there for me. For all the support and motivation that I needed to achieve this important goal, thank you my love.

To my parents and my little brother, for being the perfect family that everyone would like to have.

At last, but not least, I would like to thank all my friends for the support and great times we spent together, especially Bruno Saraiva for being the best friend.

Resumo

Nos últimos anos o aumento exponencial da utilização de dispositivos móveis e serviços disponibilizados na "*Cloud*" levou a que a forma como os sistemas são desenhados e implementados mudasse, numa perspectiva de tentar alcançar requisitos que até então não eram essenciais.

Analisando esta evolução, com o enorme aumento dos dispositivos móveis, como os "smartphones" e "tablets" fez com que o desenho e implementação de sistemas distribuidos fossem ainda mais importantes nesta área, na tentativa de promover sistemas e aplicações que fossem mais flexíveis, robutos, escaláveis e acima de tudo interoperáveis. A menor capacidade de processamento ou armazenamento destes dispositivos tornou essencial o aparecimento e crescimento de tecnologias que prometem solucionar muitos dos problemas identificados.

O aparecimento do conceito de Middleware visa solucionar estas lacunas nos sistemas distribuidos mais evoluídos, promovendo uma solução a nível de organização e desenho da arquitetura dos sistemas, ao memo tempo que fornece comunicações extremamente rápidas, seguras e de confiança. Uma arquitetura baseada em Middleware visa dotar os sistemas de um canal de comunicação que fornece uma forte interoperabilidade, escalabilidade, e segurança na troca de mensagens, entre outras vantagens.

Nesta tese vários tipos e exemplos de sistemas distribuídos e são descritos e analisados, assim como uma descrição em detalhe de três protocolos (XMPP, AMQP e DDS) de comunicação, sendo dois deles (XMPP e AMQP) utilzados em projecto reais que serão descritos ao longo desta tese.

O principal objetivo da escrita desta tese é demonstrar o estudo e o levantamento do estado da arte relativamente ao conceito de Middleware aplicado a sistemas distribuídos de larga escala, provando que a utilização de um Middleware pode facilitar e agilizar o desenho e desenvolvimento de um sistema distribuído e traz enormes vantagens num futuro próximo.

Palavras-chave: Sistemas Distribuídos, Middleware, AMQP, XMPP, Publish-Subscribe, Arquitetura Orientada a Serviços

Abstract

Over the last few years the designing and implementation of applications have evolved to a new breed of applications that are used by a huge number of users at the same time and are capable of being executed in up to thousands of machines physically distributed, even geographically, such as the cloud computing systems, the new concept of “big data” and smart cities.

The existence of several components of these systems, distributed in independent machines, brings inevitable issues in terms of designing and implementation of those systems in order to achieve flexible, scalable, robust, reliable and interoperable systems. It is extremely important to design and implement systems that can be capable of providing a communication and coordination among all the components of the system.

The concept of implementing a Middleware seems to be a great option to solve most of these issues, allowing a system to communicate with other systems in a really fast, robust and secure way.

The main goal of this thesis is to demonstrate that the usage of Middleware technologies to ensure the communication in distributed systems brings a huge number of advantages, such as interoperability between systems, robustness regarding the communication layer, scalability and high speed communications.

Keywords: Distributed Systems, Middleware, AMQP, XMPP, Publish-Subscribe, Service-oriented Architecture

Index

Figures Index

Tables Index

Acronyms and Symbols

Acronyms List

AGG	Aggregator
AMQP	Advanced Message Queuing Protocol
DDS	Data Distribution Service
DER	Distributed Energy Resource
EBBI	Energy Brokerage & Business Intelligence
EDA	Event Driven Architecture
FIFO	First in First Out
FOA	FlexOffer Agent
JAXB	Java Architecture for XML Binding
MPG	Middleware Plugin
PubSub	Publish Subscribe
QoS	Quality of Service
RabbitMQ	Rabbit Message Queueing
REST	Representational State Transfer
SC	Supervisory Control
SOA	Service Oriented Architecture
VDM	Virtual Devices Module
VME	Virtual Market of Energy
XEP	XMPP Extension Protocol
XML	eXtensible Markup Language
XMPP	Extensible Message and Presence Protocol

1 Introduction

In recent years a new breed of applications are being developed capable of being executed in up to thousands of machines physically distributed, in many cases located all over the world, supporting millions of users. Global Internet services, cloud computing systems, "big data" analytics platforms in data centers and smart city environments with mobile devices are some examples of large-scale distributed systems.

These kind of systems relies on multiple components located on different computers communicating and coordinating actions with each other in order to achieve a common goal. Systems are increasing as never seen in the past, providing new features almost every day, that is why scalability is really important to take into account.

Users are more intolerant to failures, so robustness is also very important. Since there is a lot of relevant information being exchanged between users and/or system components, security is also of paramount importance when designing and implementing these systems. A critical part of a distributed system is the communication between all the components. Recently, the Middleware concept has been emerging among developers and system designers to provide fast, scalable, robust and secure interconnection between systems. The usage of a Middleware technology to ensure the communication among systems in a distributed system has the advantage of providing an abstraction in terms of what kind of systems are communicating, enhancing the interoperability in distributed systems.

This thesis pretends to demonstrate that the usage of Middleware technologies to ensure the communication in distributed systems brings a huge number of advantages, such as interoperability between systems, robustness regarding the communication layer, scalability and high speed communications.

At first, an Introduction to this thesis is given, explaining the context in which this thesis was written. Follows the State of the Art in distributed systems, followed by a detailed description

of several different Middleware architectures. A detailed description of three different communication protocols (AMQP, XMPP and DDS) is also provided, introducing those concepts and usability.

The Distributed Systems Middleware section can be described as continuation of the State of the Art chapter, providing a more detailed description of types of Middleware and possible, protocols and examples of how Middleware may be used for distributed systems.

Furthermore, since this thesis is in the context of two projects (ENCOURAGE Project and Arrowhead Project), a detailed description of both projects is given, describing and explaining the contributions given to the projects in two different sections, Design and Implementation of Components on the ENCOURAGE Architecture and Design and Implementation of Components on the Arrowhead Project respectively.

At last, in the Conclusions and Future Work chapter, the importance of the contributions and also the knowledge acquired during the time this thesis was written is provided, along with some more future work that may be done.

1.1 Context

In this subsection a brief description of both ENCOURAGE and Arrowhead projects is given, providing the context for this thesis. Each project is later described in detail in sections Design and Implementation of Components on the ENCOURAGE Architecture and Design and Implementation of Components on the Arrowhead Project, respectively.

The ENCOURAGE – acronym for Embedded iNtelligent COntrols for bUildings with Renewable generAtion and storage – Project aims to develop a platform which will contribute for the optimization of energy use in buildings and consequently participate in the smart grid environments in a near future.

With ENCOURAGE, it is supposed to save up to 20% of energy consumption through the proposed architecture, providing a scalable, performant and reliable communication between storage, consumption or generation devices in the same building or in different buildings. An important part of the project is the ability to have a systematic and a performant monitoring system which provides near real-time information regarding all devices, possibly showing them to users through different ways, such as social networks.

The project consists in three different ways that cooperate with each other to achieve the energy savings for which it is purposed. Initially, in order to control and coordinate actions with larger subsystems, such as HVAC (heating, ventilating, and air conditioning systems), lightning, renewable energy generation, thermal storage, energy saving, among others, a set of supervisory control strategies were developed.

The same supervisory control strategies are also capable of orchestrating, in terms of operations, the large number of devices in this kind of systems. This features will help to optimize the energy use correlating the occupants comfort, the energy costs and the environmental impacts with other important things like peoples' habits, weather conditions, characteristics of appliances, the local generations and storage of energy and market conditions.

Basically the supervisory control strategies will be capable of deciding what is best for each scenario, aiming the energy savings. Another important part of the project is the concept of an intelligent gateway, which holds embedded logic supporting inter-building energy exchange. This subsystem will be responsible for the communication between buildings in order to negotiate the usage of electricity that is produced locally.

Finally, a virtual representation of every device within the platform and the design and implementation of event-based middleware applications will enhance the advanced monitoring and diagnostics. Providing a reliable and systematic monitoring of the information exchanged between all components, the overall performance will be controlled and diagnosed before something happens, which will result in a sustained long-term energy savings.

While ENCOURAGE aims to provide an interoperable platform where all the components are linked through the usage of middleware-based architecture and applications, the Arrowhead Project focus mostly on cooperative automation, efficient and flexible of dynamic interactions between energy producers and consumers, between people and systems, between systems and systems, etc.

These types of complex interactions are examples of challenges that our society is facing, concerning energy and competitiveness. The Arrowhead project aims to provide a Service-Oriented architecture model, focused on the interoperability, adapted in terms of functions and performance. The interoperability for which the project proposes is intended to be demonstrated though real experiments in several domains, such as electro-mobility, smart buildings, infrastructures and smart cities, industrial production, energy production and energy virtual market.

The proposed framework consists in three main services, named Core Services, which will enable the discovery, authentication and authorization, and the implementation of the innovative process of orchestration between systems. A Service Discovery Service is capable of registering and unregistering service producers and consumers along with the discovery of service producers and consumers as well. An Authorization Core Service that will be responsible for managing and ensuring only secure communications and authorized systems within the framework.

Finally, an Orchestration Service is responsible for providing relevant information on which service producer or consumers a system should establish communication, taking into account several pre-defined rules, such as geographical location, CPU load, availability, etc.

1.2 Motivation

The design and implementation of distributed systems, even more in large-scale distributed systems, have some issues in terms of compatibility of systems, communications performance or even scalability. These kinds of issues are the main motivation for the writing of this thesis and research on the topic of Middleware for distributed systems.

The usage of a Middleware may bring several benefits to distributed systems. These advantages have to be studied, discussed and correctly chosen for different scenarios. The advantages in a specific use-case might be a disadvantage in another situation, which means that the integration of a Middleware in a distributed system is a complex task that should be considered, yet it needs to be studied and discussed for a long time.

In a distributed system, providing compatibility among all the systems involved is a hard task, sometimes it might not be possible. With the usage of a Middleware, in some cases, it may facilitate the integration of different systems. This integration might be easier due to the high level of transparency that a Middleware may bring to a system. Basically, all the systems would use common communication mechanisms, allowing different systems, in different programming languages or different platforms to be integrated.

The scalability is another issue of distributed systems. Most of the times, any distributed system may need to scale at some time. This may happen due to an increase on the number of users/systems in a distributed system, at the same time, the huge amount of information being exchanged in the system, replication of data, or for security reasons as well. The usage of Middleware technologies may be a good solution to solve this kind of issues, allowing systems to scale without systems or users notice that modification.

At last, the performance in distributed systems is extremely important, especially in cases in which the distributed system is in the context of industrial applications or smart grids, where a fast and reliable communication is needed for long periods of time. This is another motivation for the writing of this thesis, trying to prove that the usage of Middleware technologies may bring a safe, reliable and performant communication between the systems involved in a (large-scale) distributed system.

1.3 Objectives

The main goals of this thesis are the study and compare the advantages of using Middleware solutions to enable and facilitate the communication in large-scale distributed systems.

Using Middleware technologies in distributed systems might bring several advantages, however it is important but extremely hard to design and conceptualize the overall architecture. Otherwise, if the system is badly designed, it might become extremely hard to understand how

things work, to debug and maintain it. This thesis pretends to demonstrate and explain how a Middleware can be used in a distributed system, in order to take advantage of its features.

To be able to think, discuss and design a large-scale distributed system architecture using a Middleware as the communication layer, it is important to study different Middleware technologies and protocols. This research is important to choose the right protocol depending on the scenario a designer is facing, the requirements and goals of the distributed system. In this thesis, a detailed description of some protocols is given, allowing to acquire enough knowledge to choose the right solution, adapted to the scenario at hand.

This thesis has been supported by the Arrowhead and ENCOURAGE European projects, which provided the use-cases used in this thesis. Thesis also takes into account the overall performance of the system by providing several performance experiments in some near real scenarios.

1.4 Thesis Overview

This thesis is divided in six chapters:

The Introduction chapter presents the scope of this thesis by describing its context and the motivations for the study and development of all the work that resulted in the writing of this thesis. A brief explanation of the benefits of using Middleware for distributed systems is also given, along with the goals that were defined in the beginning of this work.

The State of the Art chapter introduces the current state of technologies that are used in large-scale distributed systems. A full description and characterization of distributed systems is given in this section, providing a detailed description of possible distributed systems architectures.

The Distributed Systems Middleware chapter gives a detailed explanation of different Middleware architectures and how they are characterized. A more detailed explanation of protocols like AMQP and XMPP is provided, since those are the two used protocols for both projects discussed in this thesis.

The Design and Implementation of Components on the ENCOURAGE Architecture and Design and Implementation of Components on the Arrowhead Project chapters respectively give an in-depth view of ENCOURAGE and Arrowhead Projects, providing a detailed description of both architectures, components of the architectures and contributions to both projects.

Finally, the Conclusions and Future Work chapter gives the conclusions about this thesis and provides some ideas for future works.

2 State of the Art

The evolution of the Internet and the increase on the number of computer networks over the years lead to new challenges on the design and implementation of distributed systems. Former distributed systems solution were designed for small and closed networks of computers, such as home or companies intranets, are no longer enough to support the appearance the necessities of large distributed systems, composed from tiny devices, like temperature sensors, to powerful computing clouds.

The way distributed systems architectures were thought had to change to embrace large-scale distributed systems, such the web search, massively multiplayer online games, financial trading, cloud computing, P2P networks, smart devices, wearable devices, embedded systems, smart grids, smart cities, eHealth, etc.

Middleware-based architectures are been proven as a good solution to solve this problem, providing a highly decoupled way of designing and supporting the exchange of messages between systems. Middleware systems are capable of supporting complex function like, the routing the messages, security and QoS.

Three main subsections compose this chapter. The Distributed Systems subsection describes what actually distributed systems are, including some definitions, how they can be characterized and what are the pros and cons of this kind of systems. The Distributed Systems Architectures subsection presents a set of architectures that can be used when deploying distributed systems, in different scenarios.

At last, the Distributed Systems Applications subsection is intended to provide a more detailed description of several examples of distributed systems, where some of them are related to the projects where I was involved, such as Smart Grids and Smart Cities.

Middleware brings several advantages to distributed systems architectures, but first it is important to explain what actually distributed systems are, how they can be characterized, why they are important and where we can find examples of distributed systems nowadays. This information is provided in the next subsection of Distributed Systems.

2.1 Distributed Systems

Applications and systems are evolving faster than ever and all kind of devices are in constant communication, exchanging huge amounts of information. This evolution brings lots of challenges regarding interoperability, scalability, data storage, computational power (especially on mobile devices like smartphones, tablets or low power devices) [1], etc.

The term "distributed systems" was originally used to identify computer networks where the components of the network, usually computers, were distributed geographically in some area [2]. Distributed systems are systems where several components in a network communicate and coordinate tasks passing only messages among them to achieve a common goal [1].

According to Tanenbaum, Van Steen, a distributed system can be defined as "*a collection of independent computers that appears to its users as a single coherent system*". This means that for an end-user, an application or system is located only at one place or is only a single piece of software, whilst in fact the application or system may be distributed across multiple computers, physically near or geographically distributed [3].

The main motivation to design and implement distributed systems is the ability to share resources between computers, systems or even users. A resource can be anything, but what really matters is that resources are useful things that can be shared between systems in a network, like printers, shared disks or webpages in case of Internet. The Internet of Things (IoT) is another concept that has gained lot of importance in the past decade [4].

Internet of Things is one of the recent and most important motivations for the deployment of distributed systems, mainly due to the capabilities it can provide [4] [5]. The main goal of IoT is to have all the objects around us connected to the Internet and communicating with each other without, or with a minimal, human intervention, trying to provide a better world for humanity. This objects, also known as smart-objects [6] due to its intelligence, should be capable of acquiring information of what we like or want and act according to that to provide us something in return [7].

Figure 1 is intended to highlight the most common challenges of distributed systems.

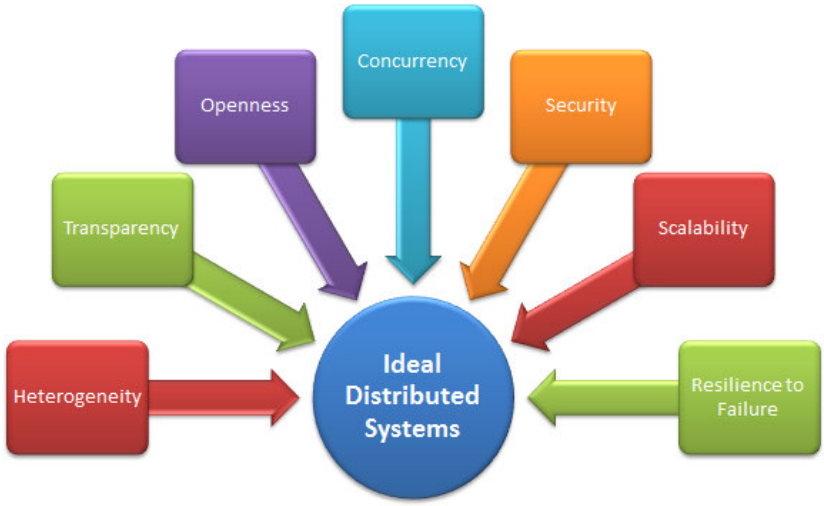

Figure 1 - Distributed Systems Challenges

In Figure 1, one of the challenges of distributed systems is the heterogeneity among computers and networks, which can be applied to multiple scenarios, such as networks, computers hardware, operating systems, programming languages or even different implementations by different developers. This is an issue for the design of distributed systems, which aims to offer an infrastructure where all the systems may communicate with each other. The term middleware tries to mask this heterogeneity in distributed systems.

Middleware can be a piece of software that gives developers a high abstraction regarding the differences between all systems that may interact, from hardware to operating systems, networks or programming languages. Another approach for this heterogeneity challenge is the concept of mobile code, where pieces of code are transferred from one computer to another, for example with virtual machines, to be executed.

Transparency is another challenge when designing distributed systems that aims to give the end user the perception of a unique system, even if it is composed by several independent components. The ANSA Reference Manual identifies eight forms of transparency, such as Access Transparency, Location Transparency, Concurrency Transparency, Replication Transparency, Failure Transparency, Mobility Transparency, Performance Transparency and Scaling Transparency [1].

The most important forms of transparency are the Access and Location, where the local or remote resources can be accessed identically and resources can be accessed without knowing the exact location of them [1].

The Concurrency refers to the fact that multiple processes may interact at the same time with shared resources and Replication refers to the existence of multiple instances of resources, without knowledge of users or developers, to improve the reliability and performance of a system. Failure Transparency provides the ability to users or other programs to finish

completely their tasks even if some components of the whole system is down or broken, hardware or software [1] [3].

The Mobility Transparency allows the exchange of resources among users of other systems without interfere with the system or users operations. The Performance Transparency allows the reconfiguration of the system along its execution if needed. At last, the Scaling Transparency refers to the ability to expand the system without interfere with the main structure of the system [1].

The Openness is another challenge of distributed systems and it is determined by the possibility of extending the main system by adding more services of resources. This implies the distribution of documentation to developers in order to create new resources and make possible the integration of new compliant components [1] [3].

The Concurrency in a distributed system means that each resource of that system might be accessed by several clients at the same time. The distributed system must handle those concurrent resource accesses, synchronizing them to maintain data consistent [1] [3].

Another challenge in distributed systems is Security. The information that is exchanged and stored in a distributed system needs to be protected and secured, since it might be extremely important to users. The security in resources follows three main characteristics: confidentiality, protecting resources from unauthorized individuals; integrity, protecting against corruption or its modification; and availability, keeping a resources always available by any means [1] [3].

Scalability is another challenge of distributed systems and also one of the most important ones. Distributed systems should be capable of expanding in cases it is needed, for example as load vary. The difficulty of this challenge is the fact that when a system expands, it should not interfere with the main system. This means, for example, that data should be replicated to prevent data loss [1].

If more computational processing or more data storage capacity is needed, distributed systems are perfect to handle this kind of issues. In any system, reliability is also very important and once again, distributed systems are helpful since redundant components can be added at any time, for example using virtualization techniques.

The most recent evolution of the virtualization techniques, which allows the creation of multiple Virtual Machines instead of having multiple physical machines, brought innovative and easy ways to greatly facilitate the deployment of distributed systems [8]. This kind of techniques reduces the time spent of systems administration, increases the uptime and allows to create multiple Virtual Machines [9] [10] locally.

Finally, the Resilience to Failures is another challenge identified in distributed systems. In distributed systems, failures of hardware or software components should be detected and handled whenever possible. In terms of data corruption, checksums are an example of how this

can be detected. In any way, failures should be hidden to the end users, providing some features to avoid systems to fail as a whole [1].

Distributed systems provide huge savings in terms of cost, since there is no need to have mainframes to make all the processing. Compared to mainframes, distributed are extremely powerful, being possible to have an enormous number of computers processing and storing information at the same time. In this way, we can assume that distributed systems brought an incredible increase of performance.

There are some applications that are used every day and are naturally distributed, such as the Web, the email, instant messaging (IM) applications or even social network, like Facebook, Google+, Twitter, etc. In terms of cons, distributed systems rely on networks. Since network capacity is limited, this can be a bottleneck in this kind of systems.

A big issue that concerns every system designer or administrator is the security [11]. When networks were composed by a small number of computers, it was easier to manage and maintain security policies, however in distributed systems, it isn't so easy to achieve.

Finally, the software complexity tends to increase as long as the distributed system gets bigger and there is a need to make every component of the architecture interactive. There are alternatives, ways of reducing that complexity, for example with the usage of a middleware-based architecture, exposing the same interface to every application.

There are several examples of distributed systems, such as Enterprise Information Systems, the Massive Multiplayer Online Games (MMOGs), Code Offloading, Smart Grids and Smart Cities, Cyber Physical Systems and more recently the Cloud-based systems. Each one of this examples is described in more detail in a separate section of this chapter.

2.2 Distributed Systems Architectures

Distributed systems can be designed in several ways, originating multiple types of architectures, depending on the context in which the system will rely on. Distributed systems architectures might be centralized or decentralized.

In centralized systems, every component of the architecture has knowledge of the state of the system whilst in a decentralized system, each component works by itself, not knowing (it can also know, but that is not the strictly needed) the state of other components.

Centralized architectures provide a total aware of the overall system with a unique point of failure, however in some cases, the non-existence of a single point of failure is an advantage, for example if in a distributed system on of the components of the architecture fails, it can exist another similar or equal node that is able to perform the same tasks, resulting in a total awareness of the failures by the end users of the system [12].

Within centralized architectures, relies a common and widely used type of architecture, the client-server architecture. Client-server architectures consist on having components providing services, acting as servers, and other components acting as clients that use services provided by servers [13].

In this kind of architectures, clients and servers may be located in the same machine but they can be distributed in different machines as well. Usually clients interact with servers through a request-response model, where a client contacts the server requesting for some information, and then it waits for an answer from the server. In the end the client receives the answer and handle it. Figure 2 presents an example of how a client-server architecture looks like.

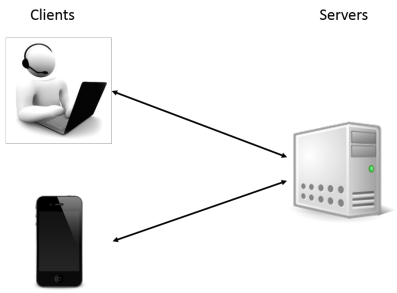

Figure 2 - Example of a Client-Server Architecture

Applications, which implements client-server architectures, can be structured in different layers, each one with a specific function. The three-tier architecture, also known as multi-tier architecture [14] [15], provides three different layers, the user-interface layer, the processing layer and finally the data layer. Each layer interacts with another layer, in a hierarchical manner.

The user-interface layer provides only the graphical user interface (GUI) to the user, without having any processing or logic capabilities. The middle-layer, or the processing layer, is responsible for processing the entire logic of the application that is then sent to the bottom layer, the data layer, responsible for the data management of the application (for example through databases, files, etc.).

Figure 3 shows an example of the three-tier architecture.

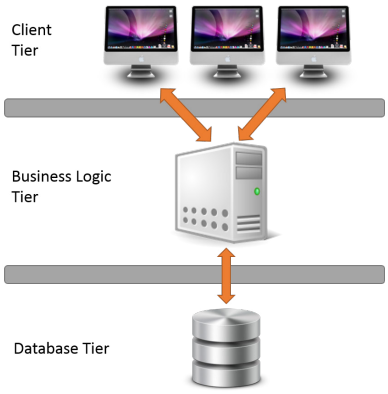

Figure 3 - Example of Three-tier Architecture

In the cyber physical systems context, this kind of client-server architecture is commonly used with a two-layered view, consisting in a layer responsible only for the user-interface, since it is generally smaller devices without processing capabilities, and a procession and data layer, which resides in the server. This scenario usually results in more loaded network traffic due to the need of the clients to send the entire raw data to the server for the processing.

Another example of a centralized architecture is the Distributed objects architecture, which consists in several objects interacting with each other, but opposing to the client-server architecture, in this case objects are viewed as equal among them. There is no difference between clients and servers, since every object in the architecture might act as server or client at the same time.

The opposite of centralized are the decentralized architectures which means that in this kind of architectures, the components of the architecture are not aware of the state of the other components. Peer-to-peer [16] or SOA [17] are examples of decentralized architectures.

Peer-to-peer architectures consist in making clients interact directly with other clients without the need to use a server as a bridge [18]. The peer-to-peer technology is widely used in several systems like instant messaging, gaming of distributed data management and bit torrent systems [19].

This kind of architectures are a very good approach for multimedia streaming, giving the fact that each client may share its bandwidth with other clients, without the need to overload the streaming servers. Figure 4 presents an example of how a Peer-to-Peer architecture may be represented.

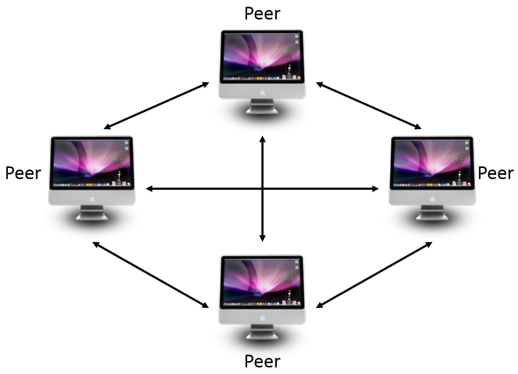

Figure 4 - Peer-to-Peer Architecture

At last, Service-oriented architecture is the point of view of how a distributed system might be designed. According to Thomas Erl a service-oriented architecture is defined as “architectural model that aims to enhance the efficiency, agility and productivity of an enterprise by positioning services as the primary means through which solution logic is represented in support of the realization of strategic goals associated with service-oriented computing” [17].

This means that the integration with old systems, but with new systems as well, of this type of architecture will bring many advantages in terms of performance and scalability.

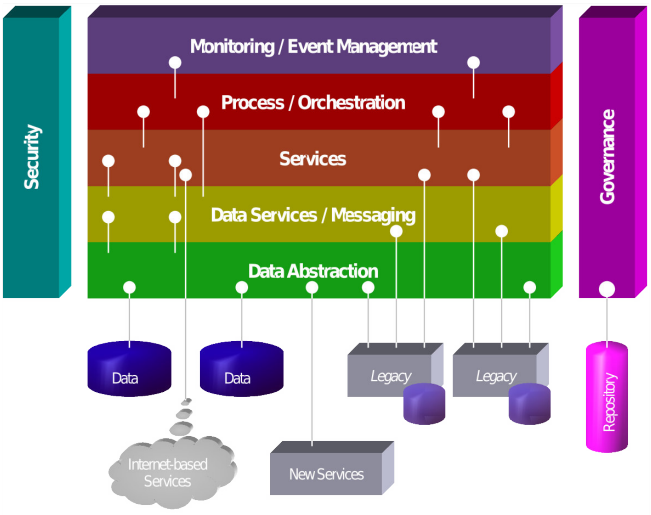

Figure 5 - Service-oriented Architecture meta-model (The Linthicum Group, 2007)

The fundamental unit of SOA is the actual service. A service is defined as independent software program that has its own capabilities depending on the context it is inserted in, which are passive of externally invocation by consumers of that service. A service, however, might be composed by more services, forming a composed service [17]. Figure 6 depicts two different kinds of services, the normal ones and the composed ones.

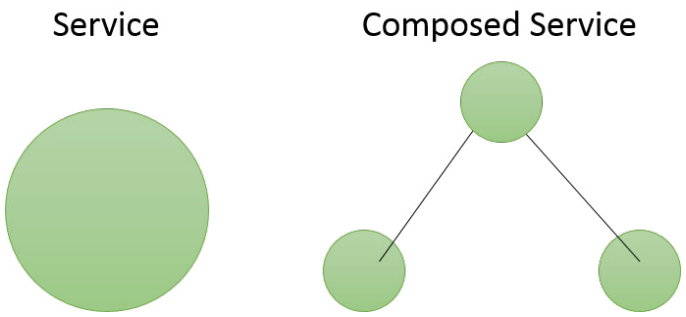

Figure 6 - Service representation in SOA

The implementation of service-oriented architectures brings many advantages, such as the increased federation, the higher interoperability and the openness since the vendor diversity increases.

2.3 Distributed Systems Applications

In this subsection the examples of distributed systems referred in the Distributed Systems Architectures subsection are described in more detail. Each one of the examples depicts different ways where distributed systems can be applied.

2.3.1 Enterprise Information Systems

The evolution of technology during the last decade brought some challenges to how companies provide their services and manage internal and external information. Information technologies started to be integrated into each service or process inside a company trying to provide more agile and unified systems, becoming easier to use or maintain.

In the past, companies had two main concerns in order to compete with other companies, the quality of the products and the price they provide. Nowadays, companies must be more focused not only on the quality and price of their products, but also on the flexibility and a quick response to customer demands [20]. This new approach enhances the competitiveness with other companies [21].

Companies are now, more than ever interested in introducing information technologies to their processes, integrating their own internal systems and evolve to a more centralized system where everything is possible to monitor and manage from a set of integrated dashboards accessing multiple company processes [21].

Figure 7 depicts an Enterprise Resource Planning system architecture. This kind of systems can have impact in many sectors of companies. An ERP system would integrate all the independent department systems of a company (as a distributed system) in order to be manageable and monitored through a unique and flexible set of dashboards [21].

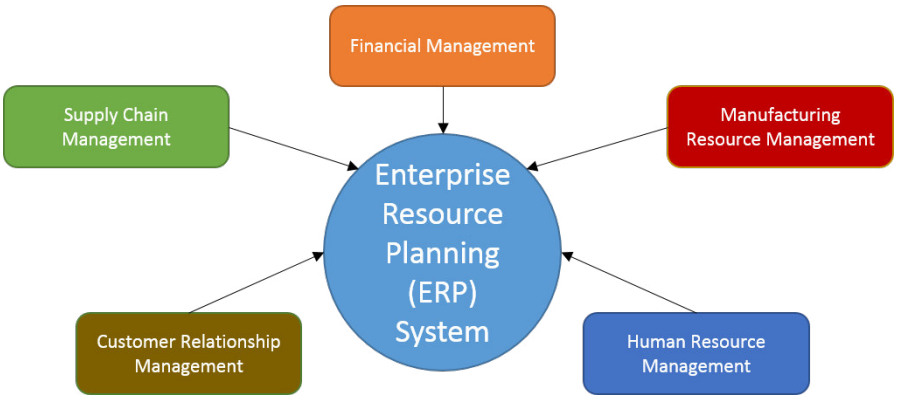

Figure 7 - Enterprise Resource Planning (ERP) System Architecture

An example of an ERP systems is the SAP Enterprise Resource Planning (ERP) [22], developed by The German company SAP [23], that provides several modules, each module operating in a specific sector or the company, such as financial management, manufacturing or human resources.

The SAP ERP claims to be a system that increases the competitiveness with integrated, fast and flexible business solutions; accelerates time to market innovative, individualized products and services; simplifies corporate structure, market channel and business scenario management; improves corporate resource and asset utilization and obviously a greater customer satisfaction; and is a consolidated foundation for the latest mobile, cloud-based, and in-memory technologies [22].

NetSuite is a cloud-based ERP system and it is the most deployed ERP system all over the world. They claim that the NetSuite ERP "delivers the proven, comprehensive financial management capabilities required to grow a changing, complex business". NetSuite ERP takes business beyond traditional accounting software by streamlining operations across your entire organization and providing you with the real-time visibility you need to make better, faster decisions [24].

Microsoft Dynamics Great Plains (GP) is another example of an ERP system, which claims to "help businesses gain greater control over their financials, better manage their inventory and operations, and make informed decisions that help drive business success. It's quick to implement and easy to use, with the power to support your growth ambition" [25].

2.3.2 Massive Multiplayer Online Games

Massive Multiplayer Online Games are another example of how distributed systems are currently used. This kind of distributed systems are designed in a way that can support thousands of users at the same time, experiencing an almost real interaction with a virtual world. In most cases, this virtual worlds are shared among all users, and the fast propagation of

actions and events through all users is a challenge to the designers of this kind of distributed systems.

Some important examples of MMOGs are the Sony's EverQuest II [26] and the EVE Online [27] from the Finnish company CCP Games .

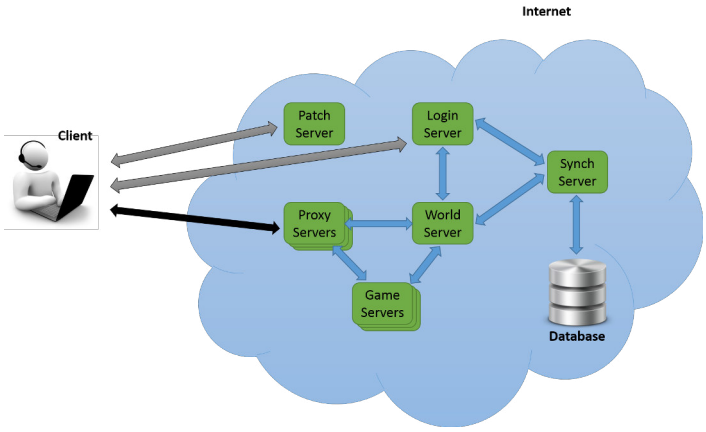

Figure 8 - Example of MMOG Architecture

Figure 8 depicts an example of a MMOG architecture with all its components and interactions. The architectures is composed by a set of distributed servers, each one of them responsible for a specific task. First a client connects to a Patch Server, which is responsible for the verification of clients' game, checking if the game is updated. Then, clients connect to the Login Server, which is responsible to query the database in order to check if that client is already registered.

After the authentications, a client connect to a Proxy Server that is responsible to forward data from the client to the Game Servers and backwards, or compressing and decompressing data. Game Servers are the servers where clients will be playing and interacting, which are in constant communication with the World Server.

The World Server is the master server in the cluster, which means that the main features of the game are handled in this server. The existence of several Game Servers provides a high scalable and performant system, by adding new Game Servers, for example as load increases.

Finally, when something needs to be persisted, a database is used to keep that data stored. However, databases might be a bottleneck due to excessive accesses or queries, and to avoid it a Synchronization Server might be used to synchronize the database queries, highly reducing the load of the system, providing an asynchronous way of communicating with the database, enabling an in-memory change of users or objects of the game [28].

2.3.3 Smart Grids

An energy grid is defined as network composed by several systems interconnected that has the capability of deliver electricity from suppliers (power plants) to consumers (houses, buildings,

factories, etc.) [29]. Former energy grids have evolved and a new concept has emerged, the new concept of smart grids. From this first definition, it is implicit the importance of distributed systems in the designing and deployment of smart grid concept. The NIST Framework identified 75 standards that are useful in the smart grids context, providing a high-level conceptual [30].

Smart grids are the future of electrical grids, evolving from a unidirectional production, transmission, distribution and consumption pipeline, from production plants (production domain), to a much more complex system where every actor of the grid can producer or consume energy (consumption domain) at the same time, store energy or exchange energy with other actors.

From several architectural solutions that were proposed, where systems seems to be the key role of the concept, with smart grids the energy grid is able to interact directly with the final user, having the ability to interact and control appliances, such as washing machines or heat pumps, in order to provide a more efficient energy consumption [31].

A Smart grid is a very complex network of systems interacting with each other pursuing the same goal of providing energy efficiency [32]. The adoption of the smart grid concept brings new challenges, such as the low-level communication technologies [33], issues with the Distributed Energy Resources [33] or electrical vehicles [34].

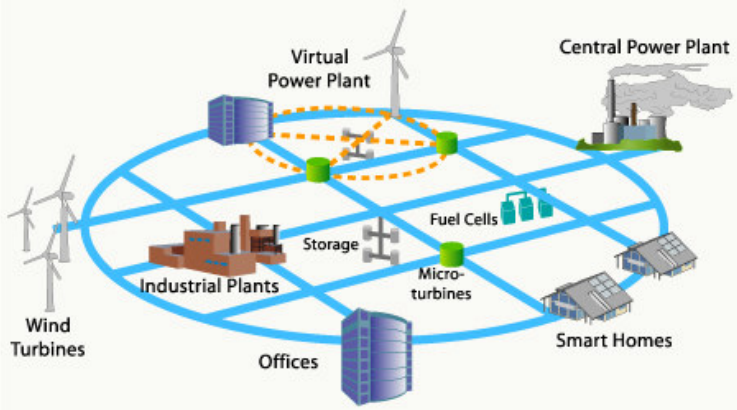

Figure 9 - Example of a Smart Grid Use-case [35]

Even with multiple definitions, a smart grid is generally defined as a merge between the traditional energy distribution network and the capability of having a multi-directional (instead of the unidirectional in former energy grids) way of communication. Giving the fact that the communication between entities flows from both sides, the grid will be able to sense, monitor and exchange information about the energy consumption [36].

Smart grids have a common characteristic that relies on embedded devices, sensors and actuators that are responsible for managing energy and controlling appliances in users' houses, usually through a gateway. This means that gateways are one way to centralize communications in the smart grid architecture. Basically the goal is to have gateways that are capable of

managing a subset of embedded devices, using adequate protocols, and that are connected to the internet in order to communicate with the grid services [37].

Lots of efforts have been made to implement the smart grid concept in many Europen countries, however several issues must be addressed to achieve all the advantages of a smart grid. These issues are related to the need to change and adapt existing infrastructures to support the integration of digital systems, support interaction between all the entities that make part of the grid (e.g. for demand response aplications), the non-existence of a virtual energy market, etc.

According to a study regarding the Convergence to the European energy policy in European countries [38], the installation of smart meters in countries, green commitments or distributed generation of energy are examples of challenges that each country faces that were compared. This study proved that the north countries of Europe, such as Finland, Norway and Sweden are making a big effort to design and implement a virtual market of energy capable of managing the energy exchanges among those countries [38].

In a distributed system like a smart grid, one of the challenges is the heterogeneity among all the systems that compose the smart grid, such as the protocols used for communication, exposed interfaces of each system, or even programming languages in which they are implemented.

To get through this challenge, standards seem to be one of the best ways to share knowledge among developers, sharing, for example, best practices which leads to an economic efficiency too [39]. Standards are what makes easier the integration of systems, enhancing the interoperability of those systems, exposing an “interface-like” equal to all systems that uses it.

According to Dr. W. Charlton Adams, Jr. [40], “In order for the Smart Grid to be successful, there needs to be a set of well-established standards in place that all industries and organizations involved can utilize.". Standards are the critical for the evolution of smart grids since it gives the possibility to ensure the compatibility and interconnection of systems implemented worldwide.

In most distributed systems, even more in complex ones like smart grids, standards are very important in order to provide interoperability and easier integration of systems. Instead of modelling systems in a plain formatted way, with standards is possible to define a more structured and hierarchical system. Also to establish the communication between systems becomes faster and convenient, since every one speaks a common language. Finally, in terms of cost, standards tens to reduce the cost of installation, configuration and maintenance of devices.

Follows a brief description of some of the current standards that can ensure the implementation and interoperability to smart grids.

Common Information Model

Common Information Model [41], or CIM, is an open-standard that aims to provide a unified and consistent way of how information exchanged between distributed systems is viewed, in order to any CIM-compliant system be able to retrieve that information [42].

The CIM Standard is composed by the CIM specification [43] and CIM schema [44], where the specification describes the language, naming, Meta Schema (formal definition of the model, defining terms used to express the model and their usage and semantics) and mapping techniques to other management models such as Simple Network Management Protocol Management Information Bases (SNMP MIBs) and Distributed Management Task Force Management Information Format (DMTF MIFs).

IEC61850

The Communication networks and systems in substations standard (IEC61850) has been defined by the International Electrotechnical Commissions's (IEC) [45] and aims the design of electrical substation automation systems [46]. Since abstract data models define it, it can be mapped to a large number of protocols, such as Manufacturing Message Specifications (MMS), Sampled Measured Values (SMV) and in a near future Web Services.

The core components of the IEC61850 are an object mode that describes the information available through an abstraction definition of services, data and Common Data Class and independent of underlying protocols; a specification of the communication among intelligent electronic devices existent in a substation automation system; and a configuration language that allows the exchange of configuration information.

The IEC 61850 is divided into ten different standards, from the Basic Principles to Glossary, General Requirements, System and Project Management, Communication Requirements, Substation Automation System Configuration, Basic Communication Structure and Conformance testing.

The advantages of using this standard are that it supports a comprehensive set of substation functions and it is easy to design, specific, configure, setup and maintain. It also provides a high performant multi-cast messaging communications and it is extensible and flexible which allows systems evolution.

2.3.4 Smart Cities

The concept of smart cities does not have yet a clear definition but according to Robert G. Hollands it could be defined as the "utilization of networked infrastructure to improve economic and political efficiency and enable social, cultural and urban development" [47]. In this context, the term infrastructure might be defined as the existent elements in a city, such as business services, housing, leisure and lifestyle services, and ICTs (mobile and fixed phones, satellite TVs, computer networks, e-commerce, internet services) [48].

The development and implementation of smart cities bring a large number of challenges to the traditional way everyone see a city and the infrastructures it holds since an uncountable number of devices are interconnected and communication with each other to provide valuable information to achieve efficiency in several elements that compose a city. Smart cities can be characterized as a huge number of devices and services that are interconnected (distributed systems), providing easy access to them and bringing efficiency in several aspects.

The Department for Business Innovation & Skill from United Kingdom Government, delivered in 2013 a background paper where they present some of the challenges that the concept of smart cities will bring to their cities, which may be generalized to every country in the world [49].

One of the challenges is the rapid evolution and usage of technology lead to unemployment in all sectors, mainly in younger people, which means that an economic restructuring is needed to support this changes. The increasing number of population moving to cities from more rural zones is also a challenge implying lots of changes in how the housing and the transport sectors are managed. Due to crisis, city authorities are getting lower budgets that makes harder to follow the proper response to the new changes [49].

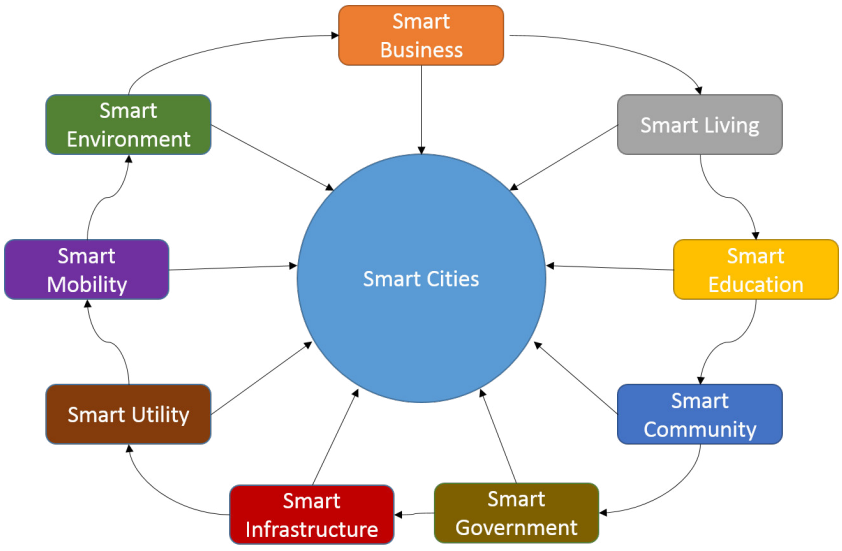

Figure 10 - Smart Cities impact

Figure 10 presents a set of sectors where smart cities have impact.

In terms of environment, smart cities will have impact in areas like the green pollution control or the climate change adaption, aiming to reduce the energy waste (using smart meters) and reducing emissions. Smart cities will affect the business industry regarding sustainability, information communication technologies and a smart economy. The innovation on how people deal with daily task using new equipment and the appearance of new ways to handle or monitor health problems in an ageing population [49] is another value that smart cities bring. In the sector of transports, electric vehicles and the dynamic control of traffic are also examples of

how smart cities will help people's life. Smart cities will also bring new ways to handle the education and the easier access to government services will help to create a happy and proactive community.

2.3.5 Cyber Physical Systems

Cyber physical systems have multiple ways to be defined but commonly it is defined as set of components in a computer, both hardware and software, which work together to achieve and a common goal [50].

The term microprocessor is usually associated to personal computers, however, even if it is not totally wrong, microprocessors are used in almost every device or appliance we daily use. Most recent cars can have more than 50 microprocessors [51], to control and monitor transmissions through electronically controlled gearboxes, brakes through anti-lock systems, electric windows, etc.

Microwaves ovens, that we use almost every day, are another example of an appliance that is controlled by this type of systems and we are normally not aware of it [50]. Embedded systems are widely used in many other areas, from home automation to control appliances, entertainment devices like smartphones or tablets, wearables for entertainment or medical purposes in order to diagnose or monitor some body aspects, such as heartbeat frequency, levels on diabetics.

Figure 11 presents a set of devices that are examples of cyber physical systems.

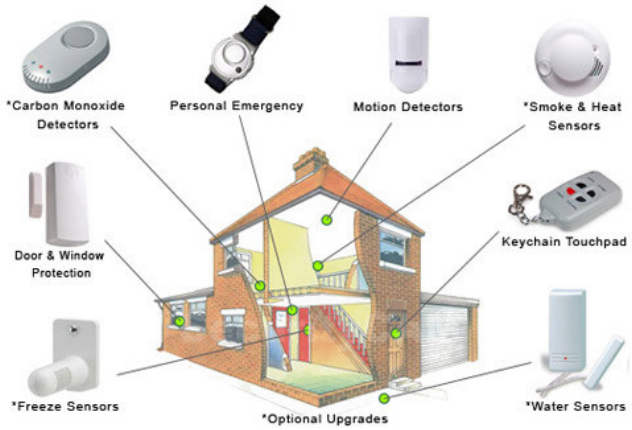

Figure 11 - Categories of Cyber Physical Systems [52]

In the context of smart grids and smart cities, cyber physical systems play an important role. Smart meters, sensors or actuators are examples of this systems which provide relevant information about the energy consumption (smart meters) helping to bring more efficiency to energy usage, motion detection, smoke detection, temperature, humidity and pressure in an area (sensors), or the control of appliances as washing machines, lightning or heaters (actuators). Sensors and actuators are similar, but sensors can only provide information about

the environment they are integrated whilst actuators beside the capability of gather information about the environment it is also capable of controlling appliances where they are integrated.

Once again, cyber physical systems are an example of distributed systems (real-time distributed systems or multimedia systems) [53], since several microprocessors can be distributed within a device, belonging to the same system, but providing information and actuating in specific functions.

2.3.6 Cloud-based Systems

"Cloud" computing became relevant when Google announced that they would work on the concept. Since then, lots of efforts were made in terms of virtualization, distributed computing, grid computing, networking and obviously web and software services [54]. The Cloud computing concept relies in a Service-oriented Architecture [55] providing a large number of orchestrated functions, through services.

Cloud services utilization is growing in popularity since the advent of the Cloud computing. Figure 12 depicts the evolution since the appearance of the World Wide Web until the appearance of the Cloud computing concept.

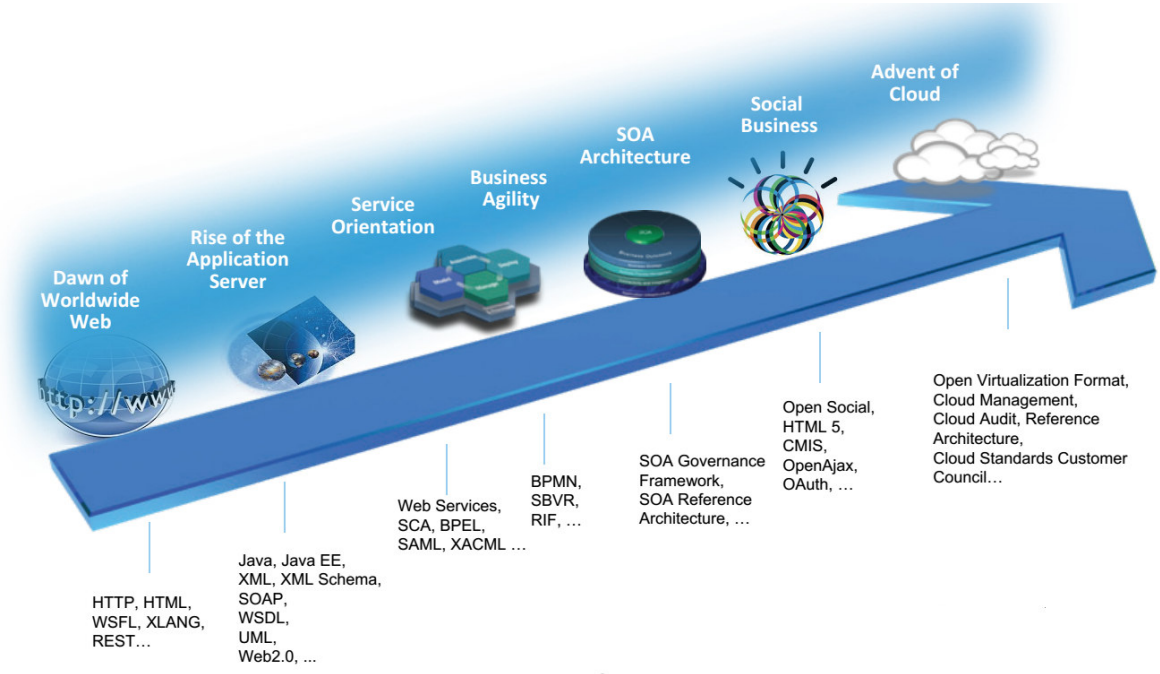

Figure 12 - Evolution of Systems [56]

In this Cloud distributed systems model, data and computation are no longer mandatory in all devices, but rather somewhere in a "cloud", though the existence of virtual computers in data centers worldwide.

This new model changed the way distributed systems were designed, offering not only huge processing capabilities in third-party virtualized computers, but also in terms of Cloud storage services, such as Dropbox [57] or Box [58], providing dozens or hundreds of gigabytes of storage in the Cloud for free.

The large amounts of storage services available in the Cloud is an extremely important advantage in terms of distributed systems, since it allows tons of information, from documents to multimedia elements, like videos of photos, to be shared among Cloud systems, making them accessible from anywhere.

3 Distributed Systems Middleware

Distributed systems brought several issues to the design of system architectures, mainly regarding network and communication between components of the system. As part of the distributed systems, middleware is a piece of software which is responsible for enabling the communication and cooperation between the components of the architecture [32].

This software layer aims to provide an abstract manner for developers to integrate enterprise systems without having the knowledge of which vendors implemented them, for how long or the protocols they use to exchange information [59]. The usage of a middleware-based architecture aims to solve network issues that might occur in distributed systems, aiming to enhance performance, scalability, reliability, security, mobility, quality of service (QoS) and multicasting.

In terms of performance, using a middleware it is supposed to reduce the latency and data transfer rates. Regarding scalability, if the systems demands, it should be possible to add any kind of nodes to the system, without having problems with integration. In this kind of systems where thousands of messages may be exchanged between applications, there is a need to make sure that those messages arrive and in the correct order – reliability – and middleware technologies aim to increase it.

Security is always a subject of great importance in distributed systems, maintaining the consistency and privacy of the data exchanged. There are ways to ensure security using firewalls, encryption, Virtual Private Networks (VPNs) which already provides encryption.

Another issue is the possibility to move applications from one place to another without stopping the application and middleware technologies due to its capability of replication of nodes, tries to solve this problem. In large-scale distributed systems, applications have different demands, which means that must be a way to guarantee that the overall system fits their needs.

At last, but not least, multicasting. Multicasting is the ability that a system provides to enable the communication from one-to-many or many-to-many applications at the same time [60].

There are several middleware solutions for the integration that simplify the integration of larger distributed systems, such as Remote Procedure Call (RPC) oriented middleware, Transaction-oriented middleware (TOM), Object-Oriented/Component middleware (OOCM) and finally Message-Oriented middleware [32].

Figure 13 presents a set of middleware types.

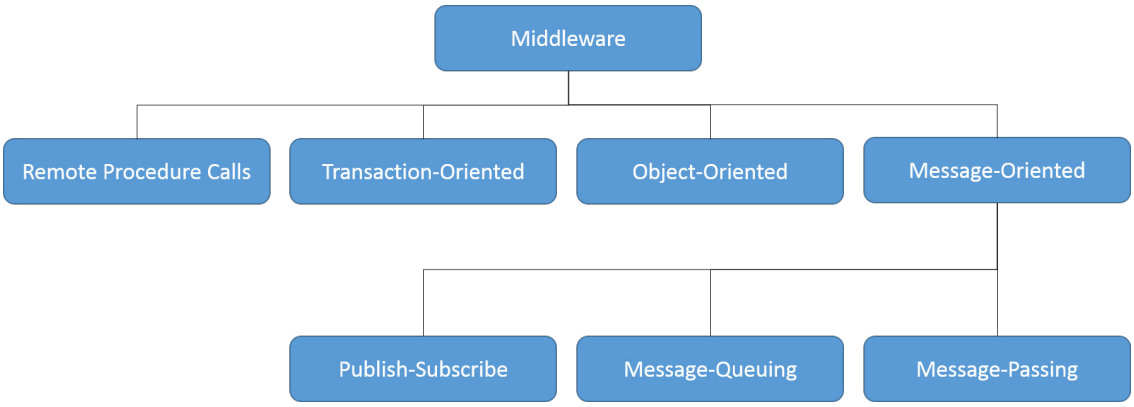

Figure 13 - Middleware Types

A Remote Procedure Call oriented middleware is characterized by a set of functionalities and infrastructures that enables the invocation, in a synchronous way, of procedures from remote systems. In cases where this type of middleware is used, most of the times client-server architectures, any system makes a call to a procedure in another remote system and waits for the answer (synchronous communication) [32]. It is possible though, to develop a multi-threaded system to simulate an asynchronous communication, however this kind of systems tend to make more difficult to systems to scale and usually presents a low-fault tolerance capability [61].

Figure 14 shows two examples of a RPC oriented middleware architectures.

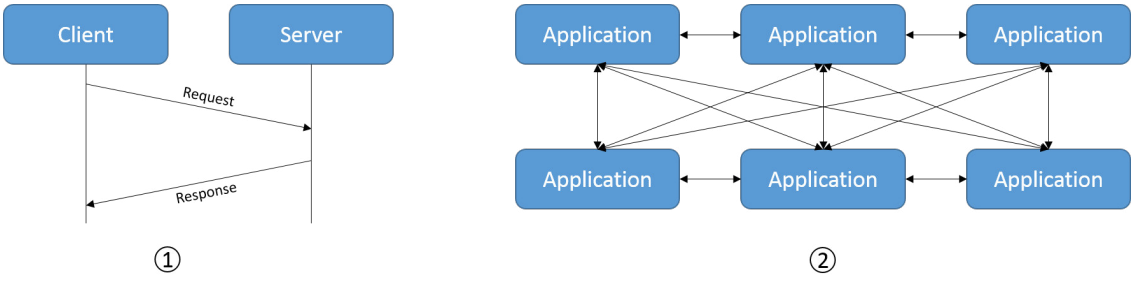

Figure 14 - Remote Procedure Call Oriented Middleware

In distributed systems where it is needed to provide a reliable transaction of operations, for example in cases in which databases are a central piece of the architecture, the Transaction-

oriented middleware (TOM) fits perfectly [32]. Comparing to the RPC oriented middleware, the TOM provides not only synchronous, but also an asynchronous communication between systems, facilitating the integration of any system with database management systems [32].

However, Transaction-oriented middleware due to excessive control and redundancy of the data exchange, to ensure safe operations, it has issues regarding scalability both in the volumes of data exchange and the number of interaction applications.

Figure 15 shows an example of a Transaction-Oriented middleware.

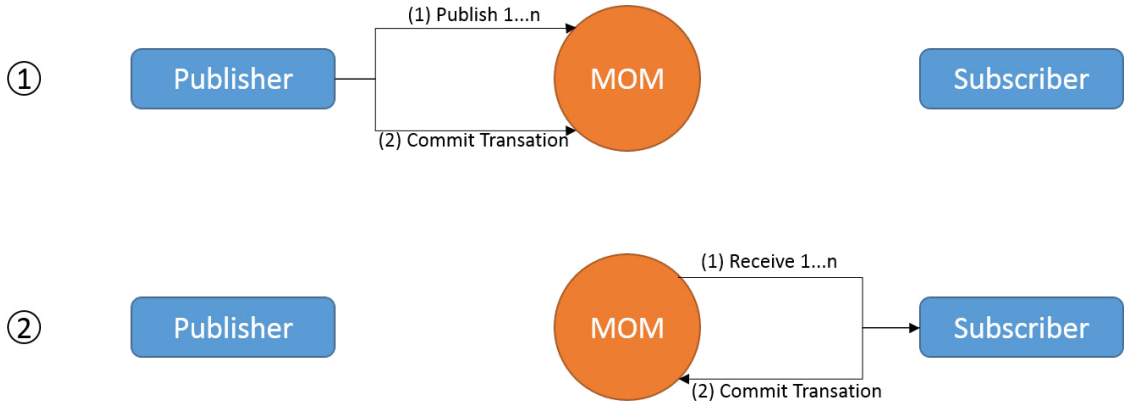

Figure 15 - Transaction-Oriented Middleware

The Object-Oriented or Component middleware is a flexible type of middleware that can be described as an evolution of the RPC model, extending the RPC features and adding some new features based on the object-oriented programming paradigm, such as object references, inheritance or exceptions [32]. It uses a synchronous communication though, and lacks of performance when used in some cases like event-based system.

Figure 16 depicts an example of an Object-Oriented/Component middleware architecture.

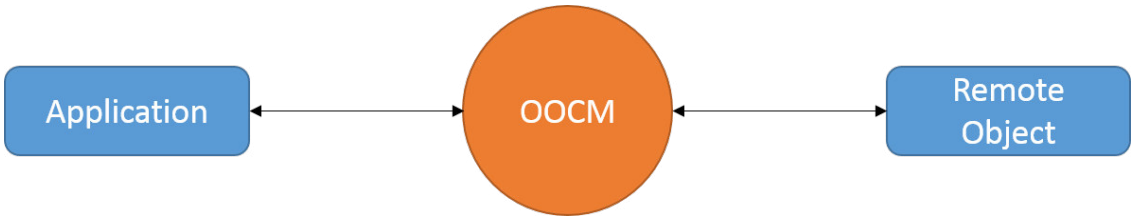

Figure 16 - Object-Oriented/Component Middleware

Finally, a Message-oriented middleware (MOM) is essentially a type of middleware that allows the exchange of messages between applications in a distributed system [32]. According to Edward Curry, a MOM can be defined as "*any middleware infrastructure that provides messaging capabilities*" [62].

Figure 17 depicts an example of a Message-Oriented Middleware architecture.

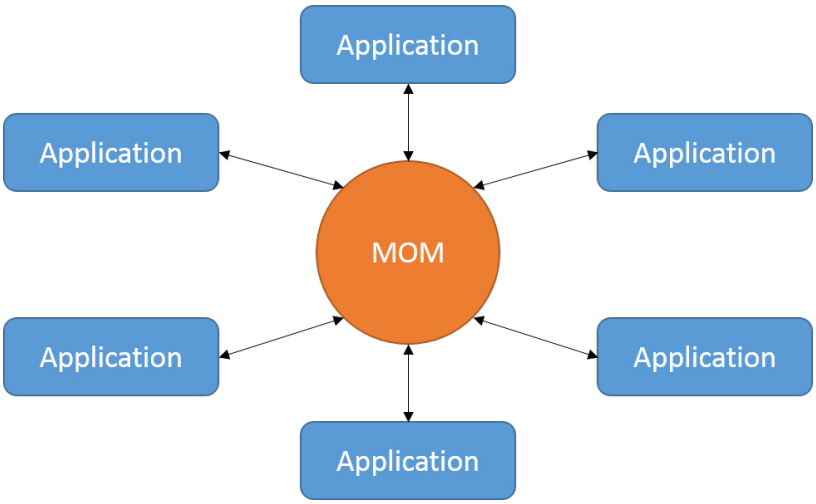

Figure 17 - Message-Oriented Middleware

In a MOM system, each client is able to send or receive message other clients using one or more servers that act as communication intermediaries and route message from one client to another. Generally, a MOM system works like a peer-to-peer relationship between the clients in a synchronous or asynchronous way. One of the most important characteristics of this type of middleware is that it provides that in any system, a client may be changed without affecting other clients/systems, due to its loose coupling among applications [62].

Comparing to the types of middleware describe above, a Message-oriented middleware provides both synchronous and asynchronous communication mechanisms. Other features like data transformation of message contents to fit the receiving application, parallel processing of messages and the support of several levels of priority are also provided by the type of middleware [32].

To simplify the integration of larger distributed systems, message-oriented middleware, also known as MOMs, became a central piece of this architectures taking advantage of an asynchronous way of communication, providing at the same time a reliable, scalable and robust implementation, making it an excellent approach for several types of systems.

Message-oriented middleware support more than one communication paradigms, as it is depicted in Figure 13. It supports paradigms such as Message Passing, Message Queuing and Publish-Subscribe.

The Message Passing paradigm allows the direct interaction between applications through a communication channel. The setup and configuration of this channel is responsibility of the MOM, however it does not hides each client identity, since the each client is defined explicitly [32].

Another paradigm supported by a MOM is Message Queuing. In this case, the communication is made through the use of message queues. Once again, the MOM is responsible for facilitate the creation, access and management of queues. Message queues are the entities that are

responsible for receive and dispatch messages to all clients that are subscribe to it. The dispatch of messages is made in a First in First Out (FIFO) manner [32]. Message loss in the system is prevented through the use of message storage in queues [62].

At last, Publish-Subscribe is another paradigm supported by a Message-oriented middleware and it is an extension of the Message Queuing paradigm. In event-based systems, this paradigm defines that a message is treated as an event that is exchanged between clients, being published by some clients and received by other clients that are eventually subscribed to that type of event [32]. It can have multiple configurations, depending on the context where the system is inserted: topic-oriented; content-oriented; and data-oriented.

If a client wants to receive all messages exchanged, it can subscribe to a topic (message queue) receiving all messages received in that queue – topic-oriented. If a client wants to receive only some types of messages, a filtering mechanism can be applied to the topic-oriented configuration, providing message inspection and delivering of only the messages that match the type of messages the client needs – content-oriented. Finally, data-oriented interaction is the configuration where data structures are shared among clients and whenever a structured is changed, the new structured is sent (published) to all clients subscribed to it [32].

In a Publish-Subscribe paradigm, a client that publishes messages (publisher) is not aware of the clients that will receive messages (consumers) through subscriptions. In a system using a Publish-Subscribe paradigm within a Message-oriented middleware is possible to achieve a high decoupled architecture since the clients do not need to know where and who are the other clients It interacts with (space decoupling), if clients are available or receive messages (time decoupling) and in terms of synchronization, it allows that subscribers might be notified when message are ready (synchronization decoupling) [32].

There are several protocols that may be used to implement Message-oriented middleware, however only the Advanced Message Queuing Protocol (AMQP) and the Extensible Messaging and Presence Protocol (XMPP) are described in detail, due to this thesis context. A brief description of the Data Distribution Service is also given for context purposes.

Other protocols however, such as the proprietary IBM protocol for the WebSphere MQ (IBM) [63], the Simple (or Streaming) Text Oriented Messaging Protocol (STOMP) [64] or the Message Queue Telemetry Transport (MQTT) even if this is an example of a machine-to-machine (M2M) protocol [65], could also be described in this section.

3.1 Advanced Message Queueing Protocol (AMQP)

The Advanced Message Queueing Protocol (AMQP) is an open standard application layer, which allows messages exchange between applications or organizations. This means that AMQP is a protocol for message-oriented middleware (MOM).

This section was based in the documentation of the AMQP protocol specification provided in RabbitMQ official website [66], the guide provided by Michael Klishin and Chris Duncan [67] and the Alvaro Videla and Jason J.W. Williams book entitled "RabbitMQ In Action – Distributed Message for Everyone" [68].

Since the 1970s, there was an effort to provide solutions to solve the integration of incompatible systems. Messaging solutions like the IBM WebSphere were extremely expensive and for that reason it was not the perfect solution. Furthermore, vendors started to create their own protocols, which resulted making even more difficult to integrate those systems.

Taking these issues into account, AMQP was designed to be an open standard for messaging middleware and to enable the interoperability among several technologies and platforms.

The most important units of this protocol are the message brokers. Message brokers act as bridge between two applications, receiving messages from producers (applications that send messages to the broker) and routing those received messages to the correct consumers (or applications that receive and handle publishers' messages).

The AMQP protocol was used to act as the message bus to coordinate and enable the communication between all the components of the ENCOURAGE Project architecture. The advantages that this protocol provides are explained in more detail in Section 4 Design and Implementation of Components on the ENCOURAGE Architecture.

The AMQP protocol consists of several entities, such as producers, consumers, exchanges, bindings and queues. Each entity of the architecture plays a different role and when acting together as one, enables the communication and cooperation between two or more applications. Producers are applications that are compliant with the protocol, and that publish messages into the broker. Messages are then published to an exchange in the broker.

The exchanges are responsible for the delivery or routing of copies of the messages to queues, received from the publishers. The routing of messages from the exchanges to queues follows some pre-defined rules, called bindings. A binding is the rule that defines the connection of a queue with a specific exchange. At last, queues are the storage elements of the architecture, with the capability of storing up to millions of messages at the same time.

Messages may carry several types of attributes as metadata in which some of the attributes can be used by the broker for routing purposes, however the content of the message is completely ignored by the broker and can only be used by applications that received the message. For instance, a message may be defined as persistent or not, which means that if the message is persistent, and is published into a persistent exchange and is routed to a persistent queue, the message will be store in disk, otherwise the message will be only in memory and will not be recovered if the message broker fails.

To ensure that messages are exchanged even if the network where the broker relies fails, AMQP provides message acknowledgements techniques. When a consumer receives a message, it

automatically notifies the broker that it received the message, however this notification might be sent only after the message processing, to avoid acknowledgements of messages that are, for instance, malformed. Since the broker waits for the confirmation from the consumer, the message remains in the queue until the consumer notifies the broker for that messages or a group of messages.

After the notification, the broker will remove completely the message from the queue. One of the goals of AMQP is to provide fault tolerance techniques, so if a message cannot be routed to any queue, for whatever reason, publishers when are publishing messages, may pass some specific parameters, choosing if the non-routed message is return to the publisher, completely removed or, using some extensions, order the broker to send that message to a "dead letter queue".

Figure 18 presents a basic idea of how the AMQP protocol works as bridging applications, using the internal entities it is composed by.

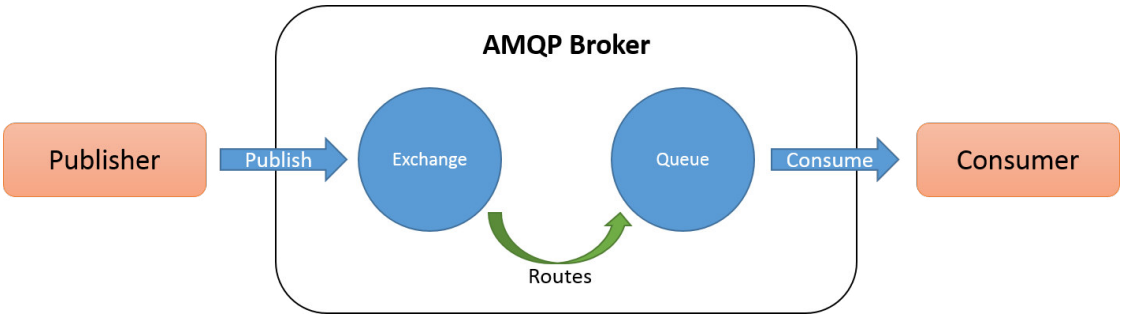

Figure 18 - Advanced Message Queueing Protocol Architecture

AMQP Consumers

The AMQP consumers are the applications that consume messages from queues. There are two types of consumption of messages, either by pushing messages or fetching messages as it is needed. A consumer when feels the need to consume messages from a specific queue, it can subscribe to that queue and from that moment, every message that arrives to that exchange will be routed to the consumer.

AMQP Producers

Similar to AMQP Consumers, the AMQP Producers are applications that publish messages into the broker, more precisely into exchanges. On the message creation, the publisher might define a set of metadata attributes in order to change the behavior of the broker.

AMQP Connections

The AMQP protocol relies on TCP IP connections to ensure the reliable delivery of messages. Due to the using as the underlying connections the TCP IP, it is supposed to have long-lived connections. Since AMQP aims to provide security, all connections use authentication and can be also protected with TLS (SSL) encryption. In cases when an application needs no more to be

connected, it should disconnect gracefully, avoiding the abruptly interruption of the TCP connection.

AMQP Channels

AMQP channels function in the protocol may be defined as the unit that carries message from and to clients to the broker. There are cases when applications need more than one connection with the broker and to avoid having many connections kept alive at the same time, AMQP channels are as "lightweight connections that share a single TCP connection", meaning that an AMQP connection might be composed by several AMQP channels at the same time. Doing that, only one connection needs to be kept alive whilst each channel is able to share that connection and perform different tasks at the same time. Each channels is set an identifier that can be used by applications to keep track of what is happening on a specific channel or what callbacks to invoke.

Figure 19 graphically represents how an AMQP Connection can be, embracing multiple AMQP Channels at a time.

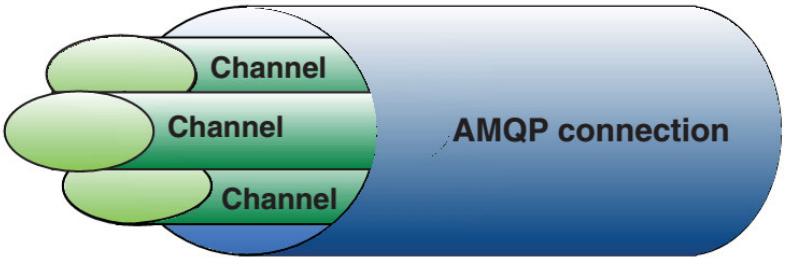

Figure 19 - AMQP Connection with multiple AMQP Channels [68]

AMQP Virtual Hosts (vHosts)

The concept of virtualization is also present and is provided by the AMQP protocol. To enable the separation of subsystems, AMQP provides a feature named virtual hosts. This features allows the creation of multiple subsystems embracing different groups of users, different exchanges and different queues. To be more precise, creating multiple subsystems means that a broker can be composed by several (sub)brokers.

AMQP Exchanges and Exchanges Types

In the AMQP protocol, several Exchanges are defined to accomplish multiple purposes. Exchanges are the entities to where the messages are published by clients (publishers). Once messages are published into Exchanges, they are routed to zero or more queues depending on the type of Exchange or rules applied when a queue is bound to an Exchange. There some broker exclusive Exchanges that are identified by a prefix in the name of the Exchange, as "amq.". The types of exchanges are: Default Exchange; Direct Exchange; Fanout Exchange; Topic Exchange; and Headers Exchange.

There are a number of attributes that can be defined when an Exchange is declared in the broker, such as the name of the Exchange, the durability that defines if the Exchange will survive when the broker is restarted (if not they are transient Exchanges), auto-delete to determine if the Exchange is deleted immediately if no other queue need it, and other arguments that are broker-dependent.

A Default Exchange basically is a Direct Exchange and it has no name. This Exchange is automatically declared by the broker and is very useful for basic applications since every time an automatically queue is created, it will be bound to that Exchange and the routing key will be the same as the queue name.

A Direct Exchange dispatches all messages to queues based on message routing keys. When a queue is bound to a Direct Exchange with a certain routing key, every time a message arrives that Exchange with a rouging key equal to the one used for the bind, the message is immediately routed. This kind if Exchanges are mostly used for unicast situations but it can be used to multicast though. If several consumers are connected to the Exchange with the same routing key, messages dispatch is load balanced in a round-robin manner.

Figure 20 represents how a Direct Exchange works.

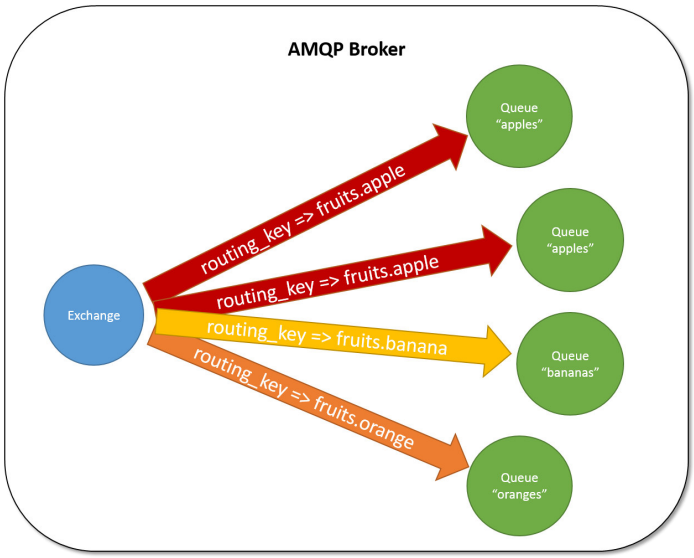

Figure 20 - AMQP Direct Exchange

In a Fanout Exchange, a copy of all messages that are published into the Exchange are routed to every queue that is bound to that queue, independently of the routing key used. This type of Exchange is widely used to broadcast messages to multiple subscribers, for example in cases of Massive Multiplayer Online Games (MMOGs). With Fanout Exchanges, events or configurations might be delivered to multiple systems in near real-time in a variety of large-scale distributed systems. This type of Exchange is represented in Figure 21.

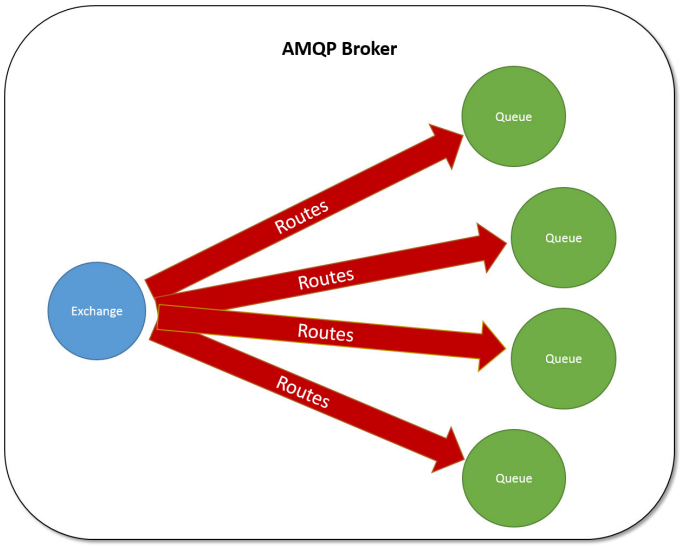

Figure 21 - AMQP Fanout Exchange

The Topic Exchange is an Exchange that provides features to filter messages that are published into the Exchange. This type of Exchange is used to routes messages to one or more queues, depending on the routing key that was used to bind the queue to the Exchange. Whenever a message is published into the Exchange, it will compare the message routing key with the routing keys used to bind queues to the Exchange and if they match the message is immediately routed to one or more queues. It can be used for orchestrate different types of services in the cloud or the parallel processing by workers that handle certain tasks.

Figure 22 represents how a Topic Exchange might work within the AMQP Protocol.

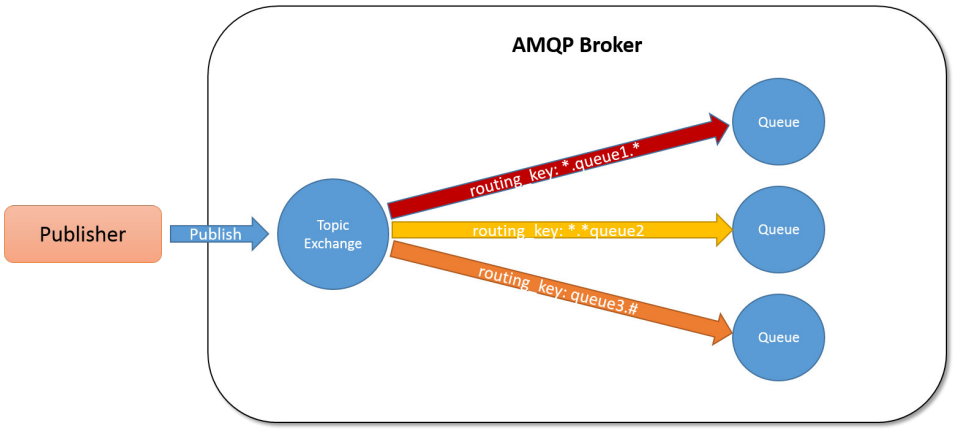

Figure 22 - AMQP Topic Exchange

Finally, the Headers Exchange may be described as a Direct Exchange with extra features. While Direct Exchanges route messages based on a routing key, in which a routing key is only a string, the Headers Exchange is able to route messages based on the attributes present in the headers of the message. The headers might be strings as well, but they can be integers or in some cases hashes or dictionaries.

AMQP Queues

In AMQP, Queues are the entities that are responsible for storing and keep messages until they are consumed by other applications. As Exchanges, Queues have also attributes that can be defined when they are declared. Some of the attributes are common the attributes of Exchanges. Similarly to Exchanges, there are broker exclusive Queues that are identified by a prefix in the Queue name as “amq.”.

Each Queue can be declared with a name that identifies the Queue in the broker. If an application does not set the name attribute, the broker will randomly give a name to the Queue. A durable attribute also exists to define if the Queue persists or not (transient Queues) after broker restarts. An auto-delete to specify if the Queue is deleted immediately after the last consumer disconnects. Queues have an extra attribute that can be defined, the exclusive, which means that the Queue can only be used by one consumer (usually by the one that declared it) and it is immediately deleted after that consumer disconnects.

An important question is that if both an Exchange and a Queue are marked as durable, in which they survive to broker restarts, that does not mean that messages will also survive that restart. In fact, for a message survive a broker restart, all three entities must be defined as durable, the Exchange, the Queue and the message itself.

AMQP Bindings

The Bindings were approached when Exchange types were described and are the rules that are used as part of the routing process made by Exchanges. Bindings can be viewed as the filters that helps select which messages an application will receive in a specific Queue. This feature allows the definition of complex situations that would be harder to implement if publishers sent messages directly to Queues.

AMQP Message Attributes and Payload

Messages are the entities that are exchange between applications and may hold a set of attributes as well. Some attributes of messages are: Content-type and Content-encoding that are commonly used to inform consumers of the message encoding (JSON, XML, etc.); Routing-key; Delivery mode to set if the message is persistent or not; Message priority to enable prioritization of message dispatching; Message publishing timestamp for example to define when the message was created; Expiration period to determine whenever a message should be dropped; and a Publisher application id which identified the entity the sent the message.

Messages that are defined as persistent may lead to a decrease of performance on the system since every persistent message is store in the disk. It is important to refer that a message is stored in the disk only if a combination of durable Exchange, durable Queue and Persistent message is verified.

AMQP Broker Implementations

The AMQP protocol is fully implemented by the known broker RabbitMQ. Another broker that implements the protocol is the Qpid Apache Project which aims to provide features like multi-platform support, message queuing, fast routing and distribution of messages, clustering and federation. Although Qpid seems to be quite a good choice, only RabbitMQ is approached since it was the broker used in the ENCOURAGE Project, described in detail in Section 3.

- RabbitMQ Broker

The RabbitMQ is a message broker software that aims to provide robust messaging for applications and is was developed to be easy to use and fit the purpose of being could scale and multi-platform. It is an open source software under the Mozilla Public License. RabbitMQ is written in Erlang which "*is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance*" [69].

The RabbitMQ broker is the leading implementation of the AMQP protocol, however using some adapters it supports several other protocols, such as XMPP, SMTP, STOMP or HTTP. It is also important to note that RabbitMQ is widely supported by a fully active community that helps to improve and evolve the broker.

3.2 Extensible Messaging Presence Protocol (XMPP)

The Extensible Messaging and Presence Protocol (XMPP) [70] is a protocol for near-real-time messaging, presence, and request-response services [71] that was specified by the Internet Engineering Task Force (IEFT) [72]. Originally it was developed in the Jabber open source community. The XMPP uses the Extensible Markup Language (XML) as the base format for the message exchange. Essentially, XMPP is a protocol that aims to provide an infrastructure to allow the exchange of small pieces of XML among entities in close real-time [73].

This section was mostly based in the documentation of the XMPP protocol specification [71], and two books "XMPP: The Definitive Guide – Building RealTime Applications with Jabber Technologies" [73] and "Professional XMPP Programming with JavaScript and jQuery" [74].

Due to XMPP provided features, it has been used to build large-scale distributed systems, Internet gaming platforms (i.e. MMOGs), search engines and video or audio conferences. The huge usage of XMPP in a large number of applications that are created or being created demonstrates how flexible, versatile and powerful this protocol can be [74].

The XMPP protocol is for Message-oriented middleware systems, like the AMQP, and is categorized as a message passing paradigm, since the identity of clients are always known by other clients. In fact, even if XMPP is used for peer-to-peer, or client-server architectures, it can

be extended with new features through the definition of XMPP Extensions (XEPs), which are needed to enable XMPP to be used in very large number of different scenarios, such as multi-user chat rooms (XEP-0045 [75]) or the publish-subscribe (XEP-0060 [76]) paradigm.

Several XEPs were defined and can be used to extend new features to the XMPP protocol, for example to used HTTP over XMPP, XMPP over Bosh, Service Discovery and a set of extensions to support Internet of Things (IoT) scenarios [77].

The XMPP is the protocol used in the Arrowhead Project that will be discussed in detail in Section 5 Design and Implementation of Components of the Arrowhead Project, to provide an infrastructure that is flexible, highly scalable, robust and capable of handling a multicasting environment, where multiple clients are able to communicate with multiple clients at the same time. In the case of the Arrowhead Project, a specific XEP (XEP-0332: HTTP over XMPP transport [78]) was implemented to support an HTTP over XMPP [78] communication, providing a similar Representational State Transfer (REST) communication between clients, over the XMPP protocol.

The XMPP protocol usually is used for scenarios of decentralized client-server architectures [73], such as the World Wide Web or email services, however it can be used for peer-to-peer communication between two servers or two clients. The usage of XML on exchanged messages in XMPP derives from the large knowledge that developers have in this message format and the interoperability that it allows since most applications use XML as the base format [74]. Each client in uniquely identified by a Jabber Identification (JID) [32].

The communication in an XMPP-based architecture is usually composed by clients and servers and is made possible through the use of streams. In a normal communication, between two clients, there are two streams, one for each direction of the communication. Messages are then passed through steams, the XMPP Stanzas that carry the content of the message from one client to another or to a server [32] [74].

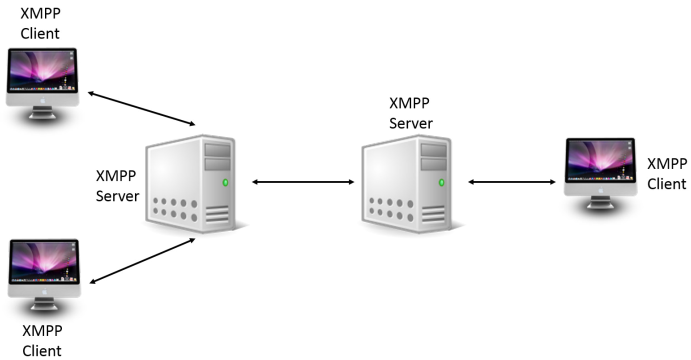

Figure 23 - XMPP Network Example

Figure 23 shows an example of how a communication is established between clients and servers in the XMPP Protocol. In this figure each client is connected to a server that is consequently connected with other servers.

When one client intents to communicate with another client, first it sends a message to the server it is connected to. The server will then take care of the message routing to the correct client. To ensure security, a server after receiving a message from one client, finds the correct server where the other client that will receive the message, is connected. Once it finds the server, the server will send the message to the correct server, which will then forward the message to the final client.

Given the fact that each server communicates directly to the correct server where the client which will receive the message, it ensure security in communications since it avoids, for example, spamming that might occur if the message was forward by other intermediary servers [74].

The XMPP protocol also provides security and authentication mechanisms, using the encryption of messages through the usage of the Transport Layer Security (TLS) and the Simple Authentication and Security Layer (SASL), respectively [74].

The XMPP network

The XMPP network can be described as the combination of different actors, namely servers, clients, components and server plugins. Each one of this actors may be created and modified by developers [74].

- Servers

Servers are the entities present in an XMPP network that are responsible for routing stanzas from one client to another, not only when clients are connected to the same server, but also when clients are connected to servers distributed [73] [74].

A group of servers that communicate with each other, form an XMPP network, whilst public XMPP servers that are interconnected form a federated XMPP network. There are several public XMPP servers available on the Internet, in which nay client can connect [73] [74].

The XMPP servers are intended to allow clients to connect to them, although it is possible for a developer to implement some applications or services that use the server-to-server type of communication in order to provide a more improved efficiency in communications, reducing the routing overhead on the server [73] [74].

There are several implementations, or servers, of the XMPP protocol, such as the eJabberd server, the Openfire server or Tigase. This three examples of server implementations are open source and widely used over the Internet, providing most of the XMPP features. They can be installed and executed in almost every platform, from Windows to Linux or even Mac OSX [74].

The eJabberd and Openfire servers will be briefly described in more detail further in this section.

- Clients

Clients in an XMPP network are entities that can connect to a server through the usage of the client-to-server protocol. This clients are usually applications, such as instant messaging, or in some cases automated applications, also known as bots [74]. This client applications usually connect and authenticate to servers anywhere that are responsible for handling client sessions or the roster (entity in the server that holds information of who is available in a friends list, in cases of instant messaging systems).

- Components

The XMPP components are entities that can be implemented by developers in order to add some other features to servers. Components also connect to XMPP servers, but it does not need the full SASL authentication mechanism, using most of the times a simple password to authenticate to servers.

Components communication with servers through the component protocol defined in the XEP-0114: Jabber Component Protocol [79]. This entities have their own unique identification within the server and are viewed externally as sub-servers.

The Components can also be configured and implemented to route stanzas in the server, allowing messaging transformation before it is delivered to the client destination.

- Plugins

The XMPP servers can be extended not only by Components, but also by other entities, the Plugins. However, while Components are external applications that can speak the same protocol of the servers, Plugins are usually written in the same programming language as the server and since they run inside server processes, they can modify the behavior or the server as well [73].

Since Plugins do not need to communication through any network socket, they can result, when needed, in high performant solutions.

The XMPP Addressing

Since the communication between XMPP clients and servers happens over a network, each client needs a unique identifier that can be used to locate the client or to forward a stanza to it. This unique identifier is called a JabberID, or only JID [73].

Usually the JID is composed by a domain for servers, and a pair of user and domain for clients. To avoid complex identifiers, XMPP relies on the Domain Name System (DNS) and it provides an easy way to identify any system through its JID. Follows an example of how a JID is composed:

(1) For servers: example.org

(2) For clients: user@example.org

In the client example, it represents a user that is registered in the server “example.org”.

The XMPP protocol provides another interesting feature in terms of addressing, the resources [73] [74]. When a client connects to an XMPP server, it can provide a resource or the server will automatically provide a random one based on that particular connection. This resource can be a normal string and it is attached to the end of the JID and it could be something like the following example:

(1) Client connected in his laptop: user@example.org/laptop

(2) Client connected in his smartphone: user@example.org/smartphone

This kind of configuration allows the applications to route stanzas to different devices that are connected simultaneously. With resources, connections might represent different devices in different locations, which means that users can be categorized by capabilities, presence. A great example of how resources can be useful is when an application receives notifications, but on laptops does not, this means that the stanzas regarding notifications were only routed to the connections that represent mobile devices.

The XMPP Stanzas

The XMPP Stanzas, or the communication primitives of XMPP [73], are part of the core of the XMPP protocol, and can be of several types: Message Stanzas, Presence Stanzas and Info/Query (IQ) Stanzas [74]. Each XMPP Stanza is used depending on the context of the application or scenario.

The three types of stanzas share a set of attributes that have the same meaning when used in all of them: the “id” attribute to identify the stanza for distinguish messages; the “from” attribute defines the address of the client which sent a stanza; the “to” attribute defines that address of the client destination, to where the message should be forward; and the “type” attribute defines if it is a message, presence or IQ stanza [74].

- Message Stanzas

Message Stanzas are used for the normal communication between two entities [74]. This type of stanzas are of a “fire-and-forget” type, which means that when an entity send a message to another, it does not receive any information to acknowledge if the message was received or not [73] [74].

Within Message Stanzas, the message can be defined as a “normal” message (by default), when a single message expecting or not acknowledge is sent, as a “chat” message, when the message is exchange for example in an instant messaging chat between two entities, as a “groupchat” if the message is exchanged within the context of a multi-user chat room, as a “headline” message to send alerts or notifications and does not expect a reply of any kind, and finally as an “error” message, if any error occurs after sending a message [73].

The content of a Message Stanza is inside two optional XML elements, the body and the thread elements. The body element is used to carry the actual message content that may be human-readable, whilst the thread might be used to correlate messages like emails, being each different thread within a unique conversation [74]. Follows an example of how an XMPP message can be.

```
<message from='cesar@isep.ipp.pt/office'
         to='rui@homenetwork.pt'
         type='chat'>
     <body>Hi there, how are you?!</body>
     <thread>4fd61b376fbc4950b9433f031a5595ab</thread>
</message>
```

- Presence Stanzas

The Presence Stanzas are the type of messages that allows the XMPP protocol to provide availability features to applications. Basically this type of messages is used to notify or inform other entities that it is available/online or not/offline. In some cases users can even define custom messages, informing other entities that is, for example away or listen to music [74].

Presence Stanzas can be of different types like Message Stanzas: Normal Presence Stanzas, Extending Presence stanzas, Presence Subscriptions or Directed Presence. Providing different types of Presence Stanzas, the XMPP protocol presents itself as a high flexible technology for integrating different kinds of applications [73] [74].

In Normal Presence Stanzas a user can inform the server that it is available or not. It is possible also to send some extra elements in the presence message, informing if it is available but is not supposed to be disturbed and a status message to be displayed, if wanted, in the client's destination [74].

The Extending Presence Stanzas are used to add some other extra information to the Normal Presence Stanzas. With this type of Stanzas additional information might be sent, for example to inform what music the client is listening for, and it can be used to display that information to other clients in chats or other applications [74].

Another type of Presence Stanzas are the Subscription Presence Stanzas. This type of Stanzas allows several configurations regarding availability of entities in the XMPP server. In cases where Presence Subscription is used, the server broadcasts the presence information of all contacts that have a subscription to a specific user. It is important to refer that those subscriptions and unidirectional, which means that if a user subscribes to another users' presence, only the user that requested the subscription will receive information about the presence of the other user. To establish a bidirectional subscription, each user should request the presence subscription to the other client [73] [74].

Follows some examples of Presence Stanzas. The first example shows how a user requests the presence subscription to another user, and the second example shows the message received when a subscription was accepted and established. The third example shows how a user can

notify the server that he is available from that moment and the fourth example shows how the user performs the opposite action, informing the server that he is not available anymore. The last example shows how a user informs the server that he is away and the reason for being away is that he is taking a bath [73] [74].

```
<presence from='cesar@isep.ipp.pt/office'
          to='rui@homenetwork.pt'
          type='subscribe'/>

<presence from='rui@homenetwork.pt/home'
          to='cesar@isep.ipp.pt/office'
          type='subscribed'/>

<presence/>

<presence type='unavailable'/>

<presence>
       <show>away</show>
       <status>Taking bath</status>
</presence>
```

At last, there are the Directed Presence Stanzas. In this kind of Stanzas, are like normal Presence Stanzas but they are not sent to the server, they are sent directly to another user or entity instead. This type of Presence Stanzas might be useful to notify other entities when users or applications forget to notify explicitly other entities that they are no longer available.

- Info/Query (IQ) Stanzas

Finally, the XMPP protocol provides the Info/Query or only IQ Stanzas. This type of Stanzas is used to provide an infrastructure for a request-response type of communication, similar to the HTTP methods GET, POST and PUT [74]. IQ Stanzas unlike Message Stanzas can contain only one payload [74].

The IQ Stanzas, like the other Stanzas may have different types: “get”, “set”, “result” or “error”.

An IQ Stanza of type “get” is used to request some information, and the “set” type is used to request or provide some information [74]. The “result” and “error” types are responses to the “get” and “set” types. Whenever an IQ Stanza of type “get” or “set” is sent, it must be responded with another IQ Stanza of type “result” or “error” depending on the result of the request [73] [74].

Follows an example of how an IQ Stanza may be created and used by a user to get the list of contacts it has on his roster, sending an IQ Stanza of type “get” to the server. Then the server replies with the list of contacts in the roster.

```
<iq from="cesar@isep.ipp.pt/office"
    id="rr82a1z7"
    to="cesar@isep.ipp.pt"
    type="get">
 <query xmlns="jabber:iq:roster"/>
</iq>
```

```
<iq from="cesar@isep.ipp.pt"
    id="rr82a1z7"
    to="cesar@isep.ipp.pt/office"
    type="result">
 <query xmlns="jabber:iq:roster">
    <item jid="girlfriend@girlfriendhouse.pt"/>
    <item jid="father@jobnetwork.pt"/>
    <item jid="mother@homenetwork.pt"/>
    <item jid="friend@friendnetwork.pt"/>
 </query>
</iq>
```

XMPP Broker Implementations

- ejabberd

Ejabberd is an open source Jabber/XMPP server and it is the most used implementation of the XMPP protocol [80] under the GNU General Public License (GPLv2). It is written in the Erlang [69] language and it claims to be a stable, massively scalable and infinitely extensible server for the XMPP protocol that can be used for several different scenarios, such as mobile messaging, social networks, massively multiplayer online gaming, among many others [80].

The ejabberd server if a modern, fault-tolerance server, manageable, highly versatile and very modular, providing almost every extension of the XMPP protocol. Ejabberd was made to be compliant which means that it aims to be interoperable. In terms of security, the ejabberd server provides several solutions, mainly using TLS encryption of messages and SASL for authentication of clients [80].

- Openfire

The Openfire [81] is also an open source XMPP server written in Java under the license of the Open Source Apache License. Openfire is an instant messaging server, easy to install, setup and use. Like ejabberd, the Openfire server provides a rock solid solution, claiming to be highly secure and with a high performance.

3.3 Data Distribution Service (DDS)

The Data Distribution System for Real-Time Systems (DDS) [82] is a standard that was defined by the Object Management Group (OMG) organization and claims to be a high performance middleware for predictable distribution of data between applications [32]. This standard is mainly used to implement the Publish-Subscribe model for communications in real-time and embedded systems and provides a set of QoS policies.

The DDS is a data-oriented middleware based on the Data Centric Publish-Subscribe (DCPS) [32] model where it implements a distributed peer-to-peer architecture providing a reliable and efficient communication among applications [83]. DDS provides high interoperability among heterogeneous systems through the existence of a Global Data Space (GDS), where multiple applications publish (publishers) messages into the GDS, and other applications (subscribers) are able to access the same GDS and subscribe to the information they are interested [83]. Every time a publisher changes data in the DDS, the DDS is responsible for propagating that changed data to all subscribers [32].

In DDS, the data available in the GDS follows a data model based on structures. Each structure is identified by a topic that uniquely identify the structure (when a client needs to change some data, it uses the topic to deliver the data to the correct DDS server) and a type that provides structural information that is used by the middleware to perform actions on that data (used by a client to perform a preliminary check on the data) [32]. A set of QoS policies is provided by DDS aiming the guarantee of data delivery, real-time systems performance, bandwidth reservation, redundancy and data persistence.

Figure 24 presents how DDS architecture is defined and works.

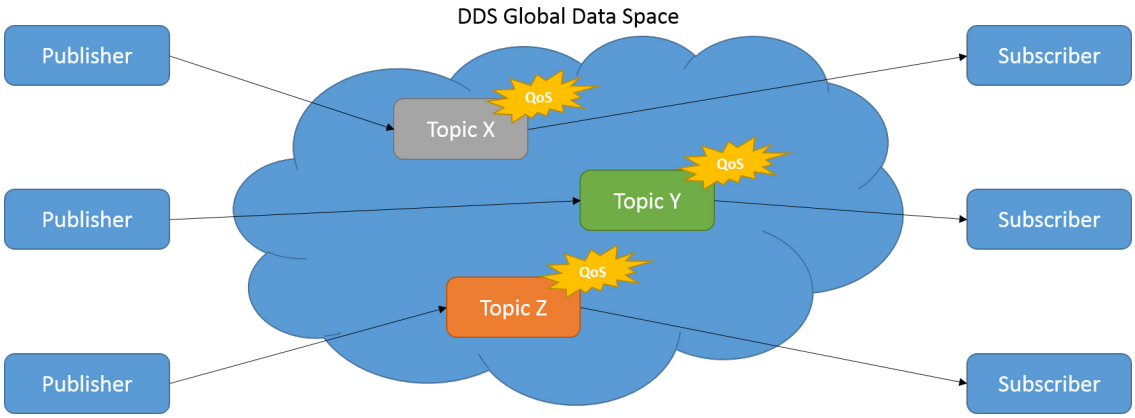

Figure 24 - Data Distribution Service Architecture

The DDS is characterized by providing an overall decoupled data-centric publish-subscribe paradigm. A high decoupled architecture between publishers and subscribers is provided in space since both may be located anywhere. Neither publishers nor subscribers are obligated to be available at any time, so DDS provides is high decoupled architecture in time. Publishers and subscribers may be implemented in different languages and in multiple operating systems or deployed in different hardware since DDS also provides a high-decoupled architecture in terms of platform [83].

4 Design and Implementation of Components on the ENCOURAGE Architecture

The ENCOURAGE – acronym for Embedded iNtelligent COntrols for bUildings with Renewable generAtion and storage – project aims to develop an interoperable platform where all the components are linked through the usage of a middleware-based architecture. Using the ENCOURAGE platform, it is supposed to save up to 20% of energy consumption. The proposed architecture provides scalable, performant and reliable communication between storage, consumption or generation devices in the same building or in different buildings. This type of architecture proved to be useful in the ecosystem of the smart grid.

One of the goals that the ENCOURAGE project aims to pursue, is to achieve a platform that provides a performant monitoring of all devices in a smart grid scenario. This means that near real-time information about the status of each device should be provided. This collected information might be presented to the end users through several technologies, such as social networks or web interfaces.

Initially, in order to control and coordinate actions with larger subsystems, such as HVACs (Heating, Ventilating, and Air Conditioning systems), lightning, renewable energy generation, thermal storage, energy saving, among others, a set of supervisory control strategies were developed, the latter being the strategies to orchestrate the operations of different subsystems in a Cell. Some examples of operations are the HVAC systems, lightning or renewable energy generation, among others.

The SC will schedule energy-consuming appliances in a Cell, taking into account the energy produced locally by the user. The SC will be focused either on supply side (local generation control), demand side (load management), or combination of both (energy management).

This feature serves the optimization of the energy consumption, since collected data can enable correlating the occupants comfort, the energy costs and the environmental impacts with other important things like peoples' habits, weather conditions, status of appliances, the local generation and storage of energy and market conditions.

Another important part of the project is the concept of an intelligent gateway, which holds embedded logic supporting inter-building energy exchange. This subsystem will be responsible for the communication between, for example, buildings in order to negotiate the usage of electricity produced locally.

Finally, the platform maintains a virtual representation of every real-world device that is connected to the platform, with the current status of each of them updated in near-real time, to enhance the monitoring and diagnostics capabilities. By providing a reliable and systematic monitoring of the information exchanged between all components, the overall performance is controlled and diagnosed before something happens, which will result in a sustained long-term energy savings. The rest of this section provides details, and an analysis, of the described architecture.

The project developed four pilots, the first two designed for campus and non-residential places, and the main goal is to provide a well designed and implemented platform that could be used in any system. Near the end of the project two more pilots tested the platform in residential buildings, and the platform collected data from sensors and made informed decisions to change the state of appliances.

When I was first involved in the ENCOURAGE Project, the architecture was already defined, and I was in charge of designing the routing structure of the middleware to enable the communication between all the components, and also the implementation of some core components of the architecture.

In Section 3.1, a brief description of each component of the architecture is given to explain the context and the features of each component. However, the focus of the description will be on the components where I gave my contribution at the time I was involved in the project, thus the Virtual Devices Module, the Database Handler and the libraries for the Encoding and Decoding of messages and the Rabbit Manager library. An in-depth and more technical description of each component and library, along with their design, is provided. A brief discussion on some tests performed is given as well.

4.1 Architecture Overview

The ENCOURAGE Platform was designed to be a highly scalable architecture, to be fast and fault tolerant, and to be as decoupled as possible in order to fulfil the needs for current and future large scale Smart Grid applications, allowing the control of thousands of houses, each one with tens of devices providing or consuming information.

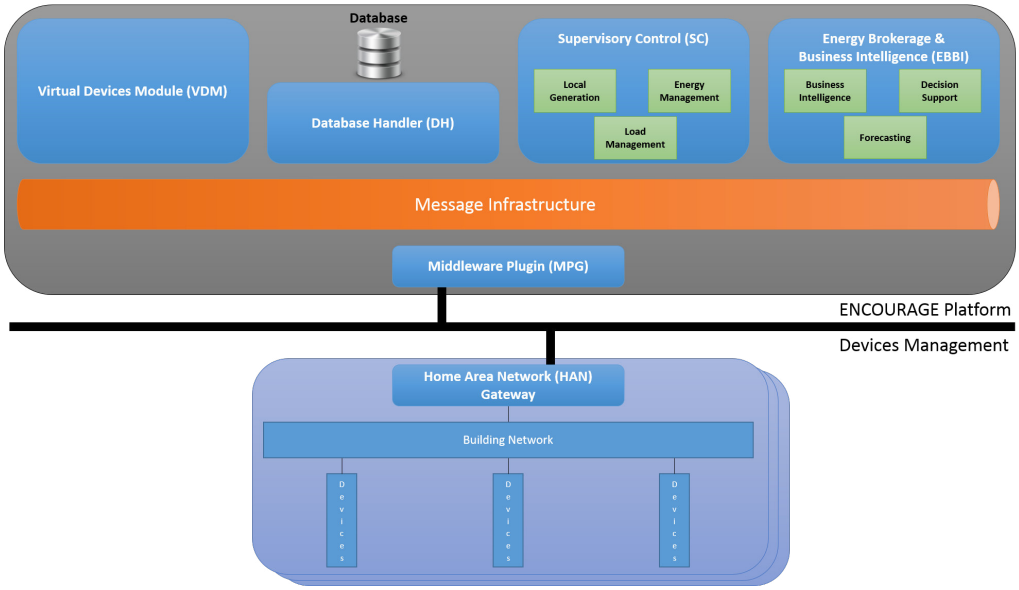

Figure 25 - ENCOURAGE Architecture

To fulfill those requirements, Figure 25 presents the final architecture for the ENCOURAGE Platform, it shows its main components and the messaging infrastructure, which was used to establish the communication between the components in an abstract way. The interaction between the architecture components and systems from the outside world is also presented through the representation of the Home Area Networks and its devices connected to the so-called Middleware Plugins, which are the components of the architecture that are responsible for handle the data collected in the gateways and transform the data to be compliant with the standard (CIM). This components are also responsible to act as the bridge between external systems and the ENCOURAGE platform.

The middleware layer presented in the architecture, in this case a RabbitMQ broker, is responsible for ensuring the communication between all components of the architecture, such as the Virtual Devices Module, the Supervisory Control, the Energy Brokerage & Business Intelligence and Middleware Plugins. The middleware allows the integration and communication with external applications, and it is responsible for the internal platform communication as well. The middleware uses a distributed publish/subscribe pattern, providing a transparent (in 2.1 Distributed Systems) implementation of distributed applications.

Current home automation and smart building infrastructures are commonly developed and implemented as separate systems with their own user interfaces and gateways (e.g. HVAC systems, water/gas/energy metering systems, etc.). The ENCOURAGE Project also allows to have multiple gateways in each house or building, each responsible for a subset of the HAN devices. It is thus possible to support multiple gateways to take care of different tasks such as energy production, energy consumption or home automation devices control, or just merge devices from different vendors, supported through the gateways of each vendor.

The gateways may be aggregated into a single logical entity, named Cell, representing a house or single building. A Cell represents an entity that pertains to the same stakeholder and that is economically independent from other Cells. Figure 26 depicts an example regarding how a Cell may be composed by, with some appliances and gateways connected together and to the ENCOURAGE platform.

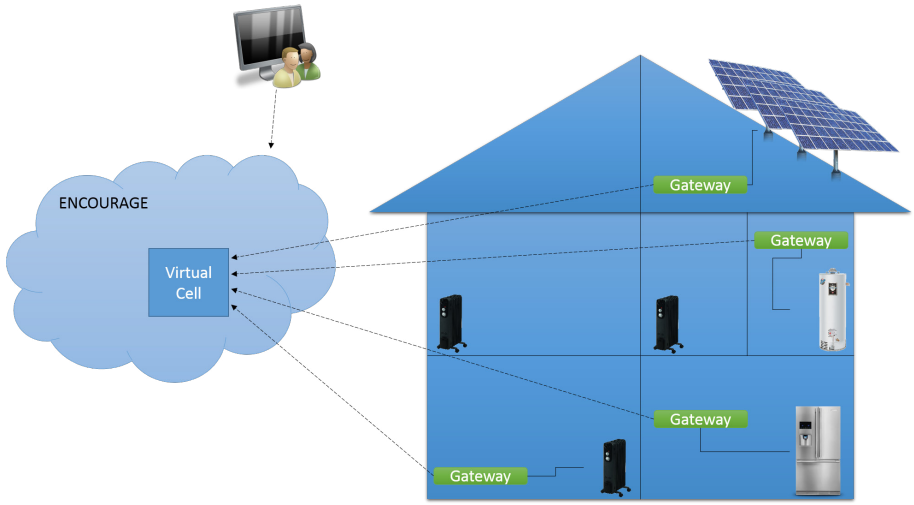

Figure 26 - ENCOURAGE Cell

Furthermore, several Cells may be grouped into a single MacroCell entity, in cases where the combined energy management might be needed, enabling the energy exchange between Cells inside a MacroCell. The ENCOURAGE platform may collect data from an entire MacroCell and also send to the entire MacroCell a set of commands to change the state of some appliances. The MacroCells may be defined as a group of cooperating users that are connected through the distribution domain.

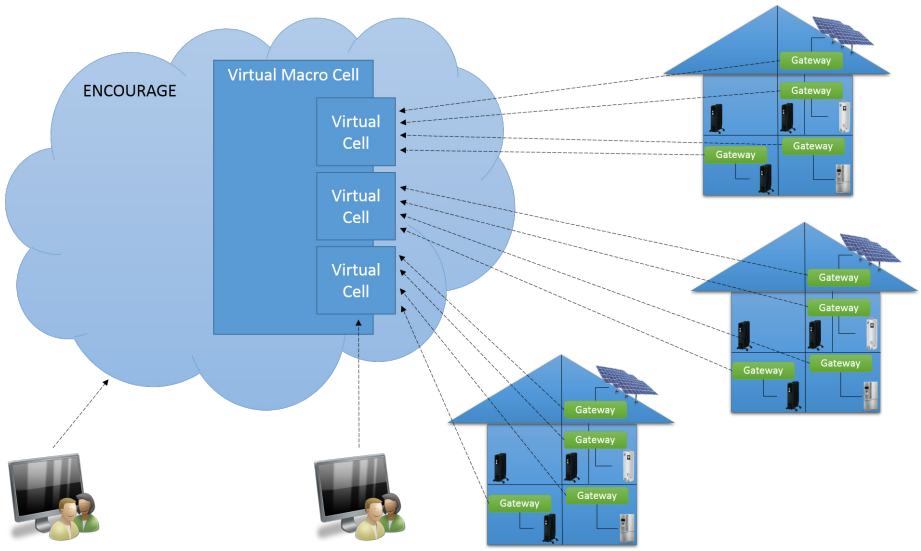

Figure 27 - ENCOURAGE MacroCell

In terms of the energy market, the ENCOURAGE Project has also the capability to provide extended market opportunities by providing energy management capabilities through the Supervisory Control and the Energy Brokerage & Business Intelligence components.

In this subsection a brief explanation of all the components presented in the architecture is given, with each component explained in a different subsection, providing a definition, main features and usability of each component within the project architecture.

4.1.1 Virtual Devices Module

The Virtual Devices Module (VDM) is the central component for data management in the ENCOURAGE architecture, being responsible for several and crucial tasks within the ENCOURAGE platform. The VDM acts as message router, caches state of the system, and enables the permanence of the historical data of the system.

The VDM is responsible for establishing the correct connections between modules. For that purpose, it parses messages incoming from the message infrastructure and forwards the messages to the components that are interested in the information. Moreover, Middleware Plugin components, which are explained later in this section, do not keep information of which devices they handle, and the VDM is in charge of keeping track of the correspondence between the gateways and devices that the MPG controls, allowing it to route commands to the correct devices.

Another important feature of the Virtual Devices Module is that since it keeps track of everything that is exchanged through the middleware, and of the current status of each device in the smart grid. Whenever the VDM is interrogated regarding current status of a device, for example a temperature sensor, it will answer the query from its current internal cache, without the need to contact the device installed in a HAN.

Finally, the VDM stores all messages into the ENCOURAGE database. To avoid database performance issues and to separate implementations, the VDM uses another complementary component, called Database Handler. The communication with the Database Handler component is established by the ENCOURAGE middleware as well. This component will be described in more details later in this section, since it was one of the main contributions to the project.

4.1.2 Database Handler

The information, contained in all messages exchanged between the components of the architecture is stored for monitoring or future analysis. In order to achieve that, the Virtual Devices Module is capable of handling those messages, parse the information and then store it into the ENCOURAGE database, through the usage of a so called Database Handler component, also depicted in Figure 25.

Most of the times, the interaction between systems and databases, and databases themselves, represents a challenge faced by developers. To avoid a possible bottleneck of the whole platform in terms of performance and processing capabilities, the Database Handler component is implemented as an external component with respect to the Virtual Devices Module, allowing it to store information in a database.

The Database Handler can be defined as a bridge between the VDM and the ENCOURAGE database, extending database functionalities and reducing the VDM processing capabilities. This component of the architecture has all the capabilities to consume messages from the message broker, in this particular case a RabbitMQ broker, and store them into the database, without interfere with the other components of the architecture. This provides a separation between the events processing and handling from the database features.

This approach of separating the database functionalities from the actual processing modules of the architecture provides a highly decoupled and scalable design of the architecture, allowing to change the database type and database location or even database replication without interfering with the platform operation.

This component will be described in more details later in this section, since it was another of my contributions to the project.

4.1.3 Middleware Plugin

The goal of the ENCOURAGE Platform is to interact with external entities, applications or systems. This can be a complex task, in terms of interoperability, due to different protocols or different data formats that these external entities may use in their communications. Middleware Plugins are the components of the architecture that are responsible for acting as a bridge between these entities and the ENCOURAGE Platform.

The Middleware Plugins are capable of performing several operations, such as the parse of messages encoded in custom formats, sent by entities, applications or systems that want to interact with the ENCOURAGE services. It is also capable of translating those messages into the common language (Common Information Model) that is supported inside the ENCOURAGE Platform. This transformation is possible through the usage of the encoding and decoding library, which was another contribution that I gave to the project.

Since the Middleware Plugin acts as a bridge between the outside world and the ENCOURAGE Platform, it is also capable of communication with the RabbitMQ broker. This communication is established through the usage of the RabbitManager library, which is also described in more detail in this section since it is also a contribution to the project.

At last, the Middleware Plugins establishes a bidirectional communication between external entities and the ENCOURAGE Platform. It is also capable of forwarding messages consumed from the middleware, sent by the Virtual Devices Module, to the correct devices through the

gateways. Messages consumed from the middleware are in the CIM format, so the Middleware Plugins once again transform those messages into the external entities specific formats and forward them.

4.1.4 Supervisory Control

The Supervisory Control (SC) component is responsible for the control of each device (in this case actuators) within each house or building and it has the capability of issuing commands to order changes on appliances that devices are connected to. The issued commands are sent to devices through the ENCOURAGE middleware.

Three modules constitute the Supervisory Control: the Local Generation, the Energy Management and the Load Management. The Load Generation module is responsible for the local energy production by renewable resources. The Energy Management module is responsible for the execution of the energy brokerage plan, acquired by working together with the Energy Brokerage component of the architecture. At last, the Load Management module of the Supervisory Control is responsible for enabling the demand side management.

4.1.5 Energy Brokerage & Business Intelligence

The Energy Brokerage & Business Intelligence (EBBI) is the architecture component responsible for the management of the participation of Cells or MacroCells on the energy brokerage. Possible retrofits, equipment replacements and other capital investment actions are also features covered and provided by the EBBI component.

The three different modules support the EBBI component features: the Forecasting, the Decision Support and the Business Intelligence modules. The Forecasting module is the module responsible for predicting the amount of energy that will be consumed or produced during a certain period of time, based on historical data and real-time data (for example, consuming data from the ENCOURAGE middleware).

The Decision Support for Energy Brokerage module is responsible for making decisions on the energy exchange, taking into account the exchanged energy between buildings that are over producing energy and the ones that are under producing energy, trying to maintain a certain balance between producers and consumers of energy, enhancing the energy consumption.

Finally, the Business Intelligence module (BI) is responsible for the preparation and delivery of reports, Key Performance Indicators (KPIs) and non-real time alerts, which may be used to analyze the performance and operation of the whole ENCOURAGE platform.

4.1.6 Complex Event Processor

Events are not necessarily simple and independent. In some cases, multiple events may define and generate more complex events. These events are called complex events and may be generated by the Complex Event Process component, present on the ENCOURAGE architecture, through the processing and correlation of events in real-time. The configuration of this component is allowed through the supply of so called “rules”, which are then used by the component to process and correlate incoming events into complex outgoing events.

An example of this type of events are alarm systems where multiple events and conditions must match and be evaluated as true at the same time to trigger the actual alarm. This information may be collected by the Complex Event Processor from multiple external systems, consuming it from the ENCOURAGE middleware infrastructure. Furthermore, this information (events) is compiled, processed and correlated to generate new events that eventually will be forwarded, as commands, to the external systems or actuators.

4.1.7 Home Area Network Gateways

The ENCOURAGE platform provides a middleware that acts as a bridge, not only linking all the components of the architecture, but also establishing a communication between all the components of the architecture and the outside world. Each house or building may have one or several gateways, which orchestrate the functioning of the devices located in houses or buildings.

The gateways collect raw data from devices they control/handle and send it to the Middleware Plugins, which translate that data into CIM and then pushes it into the ENCOURAGE middleware. The gateways are also capable of receiving commands from Middleware Plugins and redirect those commands to the specific actuators in houses or buildings they control.

The communication between the gateways and devices may be established following different protocols, such as the Smart Energy Profile of ZigBee (SEP), and it is the responsibility of who implements the gateway to setup everything in order to correctly communicate with the devices it wants to monitor or manage.

4.1.8 Devices

The devices are the endpoint of the architecture design, where two types of devices are distinguished. Sensors and actuators are the two types of devices.

Sensors are small pieces of hardware which are able collect information from the environment where they are inserted and passing that collected information to the system, when request by, for example, the HAN Gateways.

Some examples of these types of devices are, among many others, the temperature sensors to collect information about the temperature of some room or area within a house or building; humidity sensors to collect information about the humidity of a place; proximity sensors to detect objects that are closer to the sensor; or motion sensors which can detect objects that are moving near the sensor, used for example in alarm systems.

Actuators are much similar to sensors, however these kinds of devices are not only capable of collecting data from the environment, but they are also capable of performing some changes on some appliances. Actuators give the possibility to systems to send commands that are then executed, applying a change in the state of some appliances.

An example may be a washing machine that has an actuator, which receives commands to start or stop a specific program for that appliance.

4.2 Routing Structure

In the ENCOURAGE Platform applications communicate through a middleware that acts as a message bus. In this project, the RabbitMQ broker was used, which is an implementation of the AMQP protocol. To provide the communication between all the applications, a set of queues and exchanges were defined.

Figure 28 presents the defined routing structure for the ENCOURAGE Platform with all the applications that are connected to the middleware and all the queues and exchanges that are used by them to enable the communication. Figure 28 also presents all the bindings between queues and exchanges within the middleware.

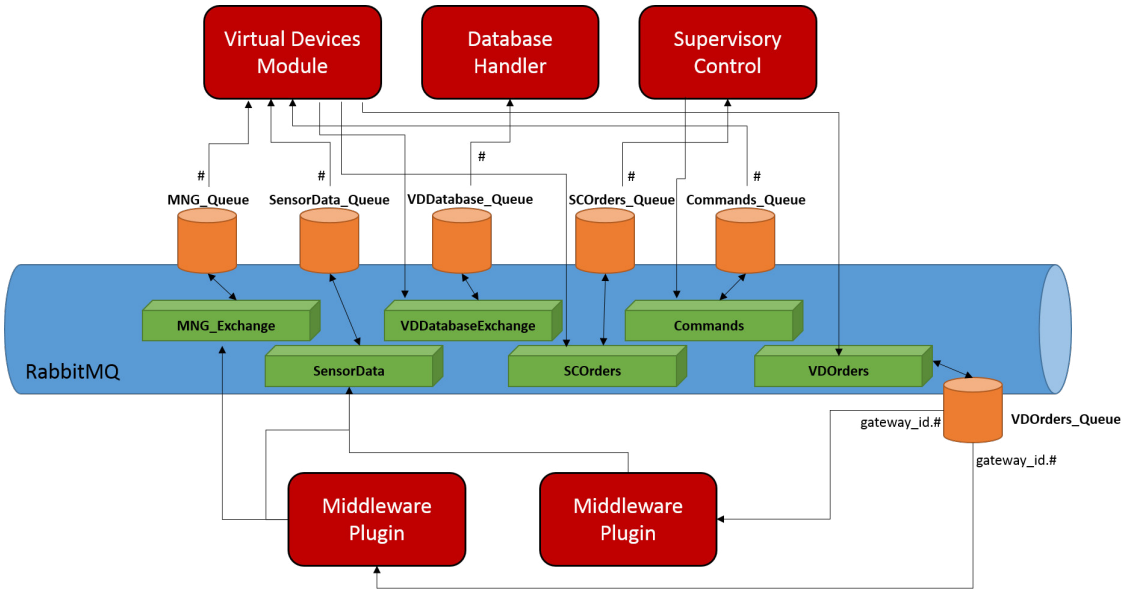

Figure 28 - Routing Structure of ENCOURAGE Middleware

The Virtual Devices Module consumes from all the queues created in RabbitMQ server and it pushes messages into some exchanges, such as the VDDatabaseExchange, the SCOrders exchange and the VDOrders exchange. Each exchange has a queue bound to it, such as the VDDatabase_Queue that has a binding to the VDDatabaseExchange and the SensorData_Queue is bound to the SensorData exchange. All bindings use a wildcard as the routing key, since no filtering of messages is needed in this case.

The Virtual Devices Module is responsible for the routing of commands from the ENCOURAGE Platform, generated by the Supervisory Control, to the correct devices. To do that, the VDM uses the VDOrders exchange, which is used by the MPGs to consume those commands.

There is also another exchange, the MNG_Exchange, which is used by any other application that pretends to create, modify or delete some entities on the configuration of the VDM itself, such as MacroCells, Cells, Devices, etc. The Virtual Devices Module consumes messages from the MNG_Queue, which is bound to the MNG_Exchange using a wildcard as he routing key for the binding, since it wants to consume everything that arrives to that exchange.

The DatabaseHandler application consumes messages from the VDDatabase_Queue, which is consequently bound to the VDDatabaseExchange using a wildcard as the routing key as well. The Virtual Devices Module pushes SQL statements to the VDDatabaseExchange that is responsible for the forwarding of those messages to the VDDatabase_Queue. Since the routing key used for the binding between the queue and exchange, all messages published to this exchange are forward to the specific queue.

The Supervisory Control application interacts with two exchanges and one queue to receive messages from the VDM. A SCOrders exchange was defined, along with a SCOrders_Queue, which is bound to the SCOrders exchange. When VDM pushes messages to the SCOrders exchange, this exchange forwards those messages to the SCOrders_Queue that are consumed by the Supervisory Control application as soon as possible.

On the other hand, when the Supervisory Control application has any command to send to any device, it generates a message with a command and publishes it to the Commands exchange. The binding between the SCOrders exchange and the SCOrders_Queue uses a wildcard as the routing key, so any message that arrive to this exchange are forward to the SCOrders_Queue queue.

Middleware Plugins are responsible for aggregate data sent by the gateways in houses or buildings. Then it transforms the raw data received into CIM messages, using the Encoding & Decoding library, and publishes them to the SensorData exchange. MPGs are also responsible for the configuration of the entities that are virtualized on the VDM. To accomplish that, MPGs generate configuration messages with entities it pretends to virtualize and publishes them to the MNG_Exchange.

Finally, MPGs are also capable of consuming commands from the ENCOURAGE Platform and forward them to the correct devices. To do that it consumes messages from the

VDOders_Queue, which is bound to the VDOrders exchange using specific routing keys to filter what messages it intends to receive.

The routing key is built based on the gateways that the MPGs are controlling, and are formed as “gateway_id.#”. The gateway_id represents the gateway that is controlling the device that is supposed to receive the command. The wildcard means that all messages for devices controlled by that specific gateway are receive by the MPG.

Most of the applications are using a wildcard as the routing key since there is no need to filter messages in this case. However, one example of how routing keys can be used is for example the Supervisory Control. If there is an instance of the Supervisory Control application that pretends to control only some buildings or houses, it can filter the messages to consume using a specific house or device. Assuming that the Supervisory Control wants to control devices from MacroCell “A”, and only from Cell “B”, it needs to use a routing key as “A.B.#” and it will consume only messages from all the devices inside Cell B.

4.3 Virtual Devices Module

The Virtual Devices Module (VDM), considered in this thesis as the central component of the ENCOURAGE architecture, is responsible for several and crucial tasks within ENCOURAGE platform, such as the virtualization of all the physical entities that are connected or were connected to the ENCOURAGE platform, the configuration (create, delete and modify) of the virtualized entities, keeping track of every data that is exchanged inside the ENCOURAGE middleware, storing that information into ENCOURAGE database, and routing the messages or commands to the correct components.

Figure 29 depicts the internal architecture of the Virtual Devices Module (VDM) and the communication with the ENCOURAGE middleware’s Exchanges and Queues. The VDM consumes from all the queues in the middleware and publishes to the Exchanges related to other components of the overall architecture, thus implementing routing functions.

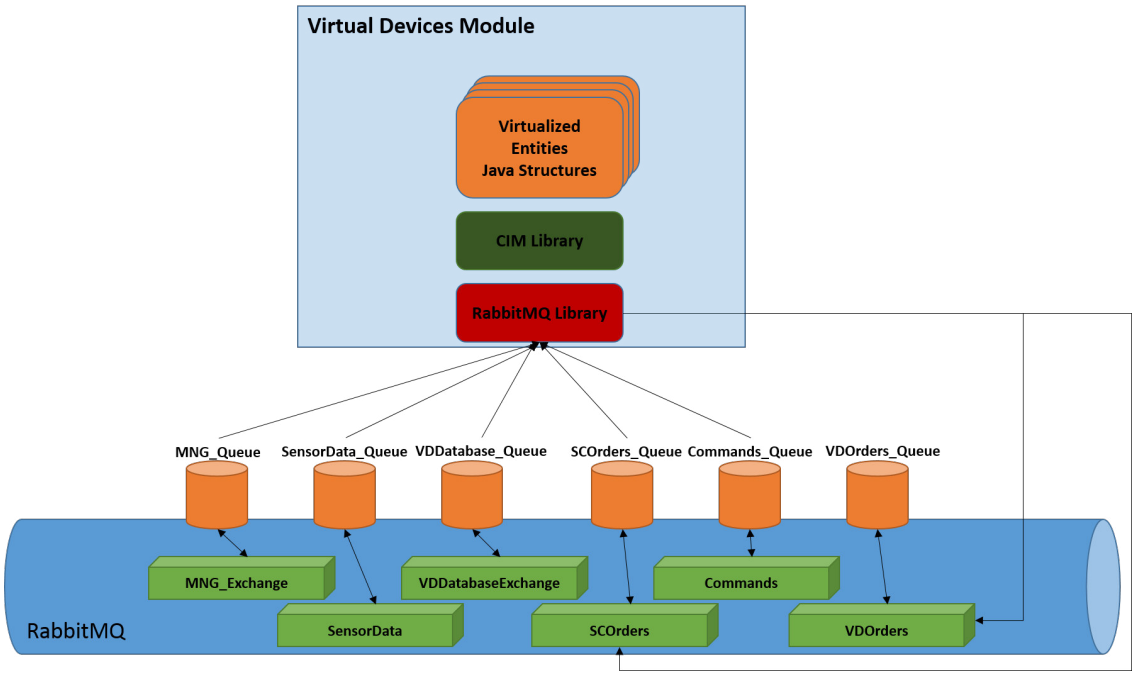

Figure 29 - Virtual Devices Module Architecture

This implementation of the VDM component uses the RabbitManager library to communicate with the middleware, which was implemented and used by all the partners that contributed to the project. This library will be discussed and explained in detail further in this chapter. The goal of the RabbitManager library was to facilitate the integration and communication of any system implemented in Java and using the RabbitMQ server as message broker.

Since the VDM is responsible for the virtualization of all the physical entities that are connected to the ENCOURAGE platform and are being controlled or monitored by it the current state of the "world" must be known by the component. This configuration can be provided at any time, both at component's boot and at runtime, through a specialized exchange called "MNG_Exchange".

The configuration is possible through the usage of an XML file, structured following the hierarchy defined in the ENCOURAGE Project. This configuration may contain information about a complete MacroCell and its composing entities, a single Cell or even a single device. Each element of the configuration file represents an entity in the real world that needs to be virtualized, which means that it is an entity that would be monitored or controlled by the ENCOURAGE platform.

Some of the entities considered into an ENCOURAGE Middleware are organized in a tree structure, with MacroCells containing Cells that contain Rooms. Each Room may contain Appliances that contain ShadowDevices. Each device is related to a Manufacturer and a Gateway.

Assuming that a new MacroCell or a new device has to be added to the platform for monitoring or control, two examples of how a configuration file can be sent to the Virtual Devices Module through the ENCOURAGE middleware are given.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<root>
    <!-- MacroCells -->
    <ConfMacroCell macrocell_id="DKD" macrocell_desc="DANISH DEMONTRATOR"
region="DK" />

    <!-- Cells -->
    <ConfCell cell_id="DKDEM07" macrocell_id="DKD"
cell_latitud="56.9730630000" cell_longitud="9.9595370000"
cell_surface="96.00" cell_volumen="228.00"
cell_building_shade_coefficient="0" cell_wall_insulation_thickness="0"
cell_desc="JADEVEJ 14" />
    <ConfCell cell_id="DKDEM08" macrocell_id="DKD"
cell_latitud="56.9728620000" cell_longitud="9.9595240000"
cell_surface="96.00" cell_volumen="228.00"
cell_building_shade_coefficient="0" cell_wall_insulation_thickness="0"
cell_desc="JADEVEJ 16" />
    <ConfCell cell_id="DKDEM09" macrocell_id="DKD"
cell_latitud="57.0045120000" cell_longitud="9.7367230000"
cell_surface="169.00" cell_volumen="405.60"
cell_building_shade_coefficient="0" cell_wall_insulation_thickness="0"
cell_desc="Vestervej" />

    <!-- Rooms -->
    <ConfRoom room_id="UPCTC0101" cell_id="DKDEM07" room_surface="19.00"
room_volume="59.00" room_max_ocup="1" room_desc="Student's appartment" />
    <ConfRoom room_id="DKDEM0101" cell_id="DKDEM07" room_surface="96.00"
room_volume="228.00" room_max_ocup="1" room_desc="1 adult &gt; 90 years
retired" />
    <ConfRoom room_id="DKDEM0201" cell_id="DKDEM08" room_surface="96.00"
room_volume="228.00" room_max_ocup="2" room_desc="2 adult &gt; 70 years
retired" />
    <ConfRoom room_id="DKDEM0301" cell_id="DKDEM09" room_surface="96.00"
room_volume="228.00" room_max_ocup="1" room_desc="1 adult &gt; 60 years
retired" />
    <ConfRoom room_id="DKDEM0401" cell_id="DKDEM09" room_surface="96.00"
room_volume="228.00" room_max_ocup="1" room_desc="1 adult &gt; 70 years
retired" />

    <!-- Manufacturers -->
    <ConfManufacturer manufacturer_id="" manufacturer_make=""
manufacturer_model="" manufacturer_firmware="" manufacturer_desc="" />

    <!-- Gateways -->


    <!-- Appliances -->
    <ConfAppliance appliance_id="ADKDJ010101" appliance_power="4"
subcategory_id="F1" appliance_desc="Solar panel Inverter"
appliance_investment="0" room_id="DKDEM07"
appliance_activacion_date="19001010" appliance_leaving_date="90001231" />
    <ConfAppliance appliance_id="ADKDJ0101010" appliance_power="1"
subcategory_id="TOT" appliance_desc="Total" appliance_investment="0"
room_id="DKDEM08" appliance_activacion_date="19001010"
appliance_leaving_date="90001231" />
```

```
    <!-- Devices -->
    <ConfShadowDevice device_id="DKDJ010101" device_desc="Solar panel
Inverter status" appliance_id="ADKDJ010101" manufacturer_id=""
device_output="false" gateway_id="" />
    <ConfShadowDevice device_id="DKDJ010102" device_desc="Floor heating
Energy" appliance_id="ADKDJ010101" manufacturer_id="" device_output="false"
gateway_id="" />
    <ConfShadowDevice device_id="DKDJ010103" device_desc="Electrical meter
consumption from grid Energy from grid" appliance_id="ADKDJ0101010"
manufacturer_id="" device_output="false" gateway_id="" />
</root>
```

This first example shows how to create the virtualization of a new MacroCell that is already, or pretends in the future, interacting with the ENCOURAGE Platform. As it depicts, first the MacroCell is defined, giving a set of attributes to the "ConfMacroCell" element, defining all the needed information to that MacroCell. Then, a set of Cells, the "ConfCell", that are common to the MacroCell are also presented, each one of them defining a set of attributes as well. Each Cell must define a "macrocell_id" attribute, defining which MacroCell that Cell belongs to.

Usually, each Cell represents a house or a building, so after Cells, the next element to define are rooms, with the element "ConfRooms", which also needs to define a "cell_id" attribute, defining to which cell this room belongs. Following the hierarchy defined on ENCOURAGE, under rooms we can define for example, gateways, through "ConfGateway".

The gateway must also specify the "room_id" attribute, to define which room this gateway belongs to. Each gateway may belong to different manufacturers, and it is possible to define a new manufacturer, which gets stored into the database.

In each room, several appliances may exist, from computers, heaters or any type of sensors. To virtualize each appliance, we can send in this message, one "ConfAppliance". One appliance usually exists inside some room, so each appliance must specify the "room_id" attribute defining to which room it belongs.

Finally, the devices are defined using a "ShadowDevice" element. A "ShadowDevice" represents a real device in the real world, such as sensors or actuators, which can be present in some appliances. Since devices belong to appliances, each "ShadowDevice" must specify the "appliance_id" attribute, defining to which appliance the device is related to.

It is also possible to send the configuration of individual elements of the real world, such as a new appliance or a new device. To be able to do that, a single element message can be sent to the VDM. Follows an example of a message to configure a new existent device related to a specific appliance.

```
<!-- Devices -->
    <ConfShadowDevice device_id="DKDJ010101" device_desc="Solar panel
Inverter status" appliance_id="ADKDJ010101" manufacturer_id=""
device_output="false" gateway_id="" />
```

The presented examples show how to configure and virtualize real world elements inside the ENCOURAGE platform. The relation between elements is given specifying a specific attribute defining the parent element. This kind of relation and independency of elements when a configuration message is built, provides a high level of configuration and customization. Different elements may be configured and associated to already existing elements.

In cases that a new configuration, for example of a device, is sent to the VDM and it defines that it device is related to a non-existent appliance or gateway, the VDM receives the message, stores it into the database but signals and warns that the new device is missing some information or the parent element doesn't exist. This warning allows to alert who sent the configuration that with the current information it is not possible have a complete control and monitoring of that device, since it doesn't know to where messages should be forward.

From the moment that a configuration is sent and handled, the VDM has in memory all structures it needs to represent each entity and is now able to start receiving and handling messages from those entities. All messages that are pushed into the "SensorData" Exchange are consumed and handled by the VDM.

All messages exchanged within the ENCOURAGE Platform are in the Common Information Model (CIM) format, so the VDM is able to parse and transform those formatted messages into the structures virtualizing the real world entities.

To transform and parse messages in the CIM format, the VDM uses the Encoding & Decoding library, which is explained further in this chapter. Each message that is consumed from the "SensorData" Exchange is parsed, using the JAXB framework, and new objects are created to be manipulated by the VDM. The JAXB framework allows the definition of bindings between XML elements and Java classes, allowing to parse and instantiate entire XML documents to Java objects and Java classes to XML documents. Fields that contain the information it brings compose a message, and the fields will be described in the Encoding & Decoding library.

Having extracted the information from the messages received, the VDM updates the current values on the structures in memory, keeping always the last value and other information as status of every real entity. The idea to keep that information in memory gives the possibility to ask the VDM at any time for the current values of any entity, and VDM will be able to provide the information in the shortest time possible, without querying the devices located in the HANs.

The Enconding & Decoding library is also capable of generating SQL messages from the extracted information of received messages. Those statements are forwarded to the Database Handler component through the "VDDatabaseHandler_Exchange", also present in the middleware. Pushing database communications and accesses to another component of the architecture, reduces the processing time of each messages handled in the VDM.

In cases that the database crashes or is down for some reason, the VDM keeps working without any problem, and SQL messages are kept inside the "VDDatabaseHandler_Queue" in the RabbitMQ broker. To keep messages in queues until any application consumes it, the VDM

marks those messages as persistent. Since both "VDDatabaseHandler_Exchange" and "VDDatabaseHandler_Queue" are also durable, messages are then kept in queue until they are consumed. More information on these mechanisms can be found in the Advanced Message Queueing Protocol section.

In the next section the Database Handler component is described in more detail.

4.4 Database Handler

The Database Handler component is responsible for inserting, updating and deleting data from and to the ENCOURAGE database. Similar to the VDM, the Database Handler uses the RabbitManager library to interact with the ENCOURAGE middleware, through the VDDatabaseQueue, which is bound to the VDDatabaseExchange. Both Exchange and Queue are configured as durable in order to keep messages stored until the component consumes them.

In Figure 30, the internal architecture of the Database Handler is provided, along with the interaction with the ENCOURAGE middleware and its structures.

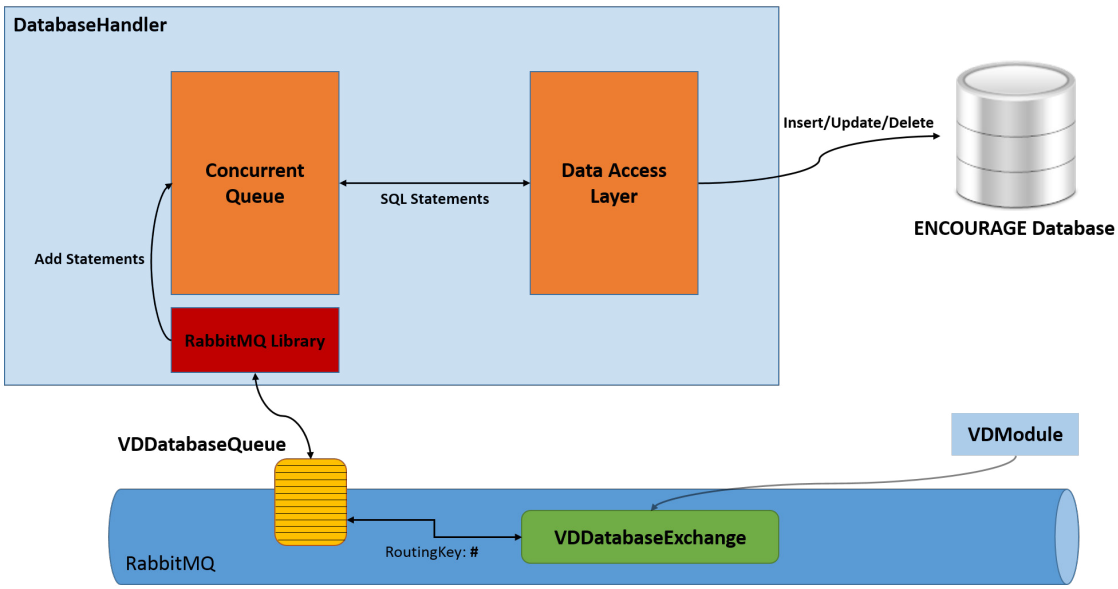

Figure 30 - Database Handler Architecture

This component connects with the middleware and consumes from the VDDatabaseQueue. This queue is bound to the VDDatabaseExchange exchange, using a wildcard as the routing key. This means that all messages that are pushed into that exchange are forward to the bound queue, without any filtering. The messages received by this component are simple strings with the SQL statements generated by the Encoding & Decoding library in the VDM component.

Internally, the component consumes messages from the RabbitMQ queue and adds them into a concurrent queue shared with a worker thread called Database Access Layer. Then the

Database Access Layer module of the Database Handler pulls messages, one by one, from the concurrent queue and executes the SQL statements in the database. To ensure the functioning of the component, the concurrent queue is used to allow both RabbitMQ consumer threads and Database Access Layer module to access the queue, avoiding concurrent accesses to the same Java structure at the same time.

The concurrent queue could be removed and the component could handle one message at a time, but the AMQP protocol assumes that it is better to have queues getting emptied as fast as possible. This architecture allows the component to consume messages as soon as they arrive to the queue, maintaining them in memory to be handled by the component.

The Database Handler doesn't have any logic and message processing, it just consumes messages and executes the contained SQL statements. This component is extremely important to reduce the VDM processing and to remove a possible database bottleneck on the system. It also ensures that all messages containing SQL statements are stored into the database and kept in the middleware queue if the component fails.

4.5 Encoding & Decoding Library

The content of all the messages that are exchanged within the ENCOURAGE middleware must be Common Information Model compliant, particularly on IEC 61968. However there are cases where messages are not in this format, such as the configuration of virtualized entities or the messages used to execute performance tests on the platform. Apart from that, all messages are encoded in the XML format.

This subsection is devoted at explaining the types of CIM messages that are used in the project, and some UML class diagrams are given to better explain how they are structured for this specific case. Two types of messages will be detailed, namely MeterReadings and EndDeviceControls.

Figure 31 depicts the class diagram for the MeterReadings CIM messages. This kind of message is used to encode data received by the MPGs from several gateways. In this messages, data regarding the environment, electricity measurements, weather data or any kind of forecasting, are respectively encoded.

Each MeterReadings message is composed by one Meter object, describing the entity it refers to, being a MacroCell, Cell, Device, etc. Zero or more EndDeviceEvents and Readings objects also compose a MeterReadings message. The EndDeviceEvents contains general information regarding events, such as alarms or state changes of appliances, and can be described in plain text. The information contained in the Readings message represents values sensed by the sensors, which can be of numerical or Boolean type.

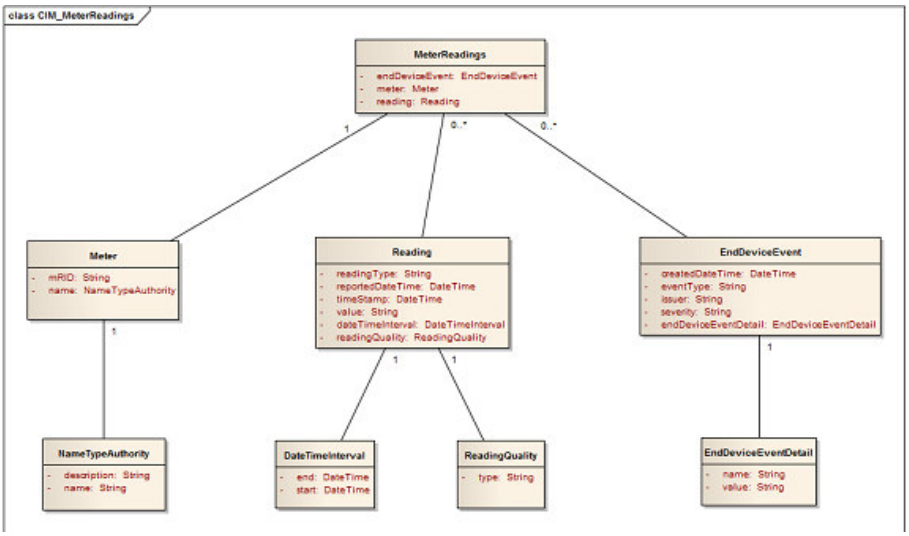

Figure 31 - MeterReadings Class Diagram

The CIM message of type EndDeviceControls has its class diagram given in Figure 32. This type of message is used to set values onto other modules. The Supervisory Control component, which is capable of generating command messages to send to devices, usually actuators, is one of the components that it uses the EndDeviceControls messages to send commands.

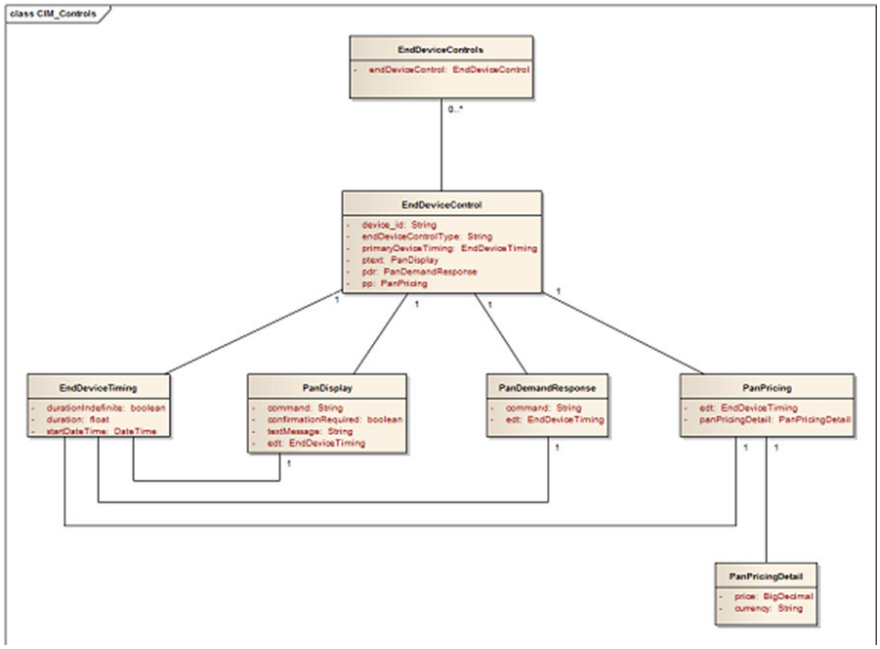

Figure 32 - EndDeviceControls Class Diagram

For example, the Supervisory Control can create this type of messages to switch on/off the devices, and to present custom text messages to end users through user interface installed on HAN devices' displays. Another example is related to the computation performed by the EB&BI indicating how much energy should be consumed by some appliance, which can be sent to the Supervisory Control encoded in this type of message.

4.6 RabbitManager Library

The RabbitManager library was one of the most important contributions to the project and it was developed to facilitate the integration of any Java application that aimed to access and communicate with the ENCOURAGE Platform. The main goal of this library was to provide an easy way to communicate with the RabbitMQ broker. Its requirements were to provide all the necessary features to publish or consume messages from the middleware.

It was a priority for the library to provide an easy way to configure the publishers and consumers that were supposed to be created, to configure the needed connections and channels and the configuration of the exchanges and queues it would use. It was created a configuration file that allowed the configuration of several attributes, such as the number of messages each consumer would consume at a time (prefetch_count) or if the messages produced by the publishers are persistent or not (message persistency).

This configuration file may be missing, since the RabbitManager library would configure itself using default values. In our case, we decided to set a default prefect count attribute with a value of 20, which means that every consumer would retrieve 20 messages at a time from the queue it is consuming. Another configurable attributed is the QoSEnable, which orders the RabbitMQ broker to dispatch messages in a round-robin manner among all the consumers connected.

In terms of message persistency, for performance purposes, we decided to set non-persistent messages as default behavior, since every time a persistent message is published into the middleware the broker writes the message to the disk, and it would reduce the performance of the broker when the number of messages published gets high.

The development of the RabbitManager library was articulated through four main generations, looking for configurable and high performant communication with the middleware. This evolution process will be described in this subsection. All the approaches were developed and tested using a full VDM application, which led to find other issues to be solved in both RabbitManager library and the VDM architectures. The results of the performed tests are discussed in the Performance Tests within this chapter.

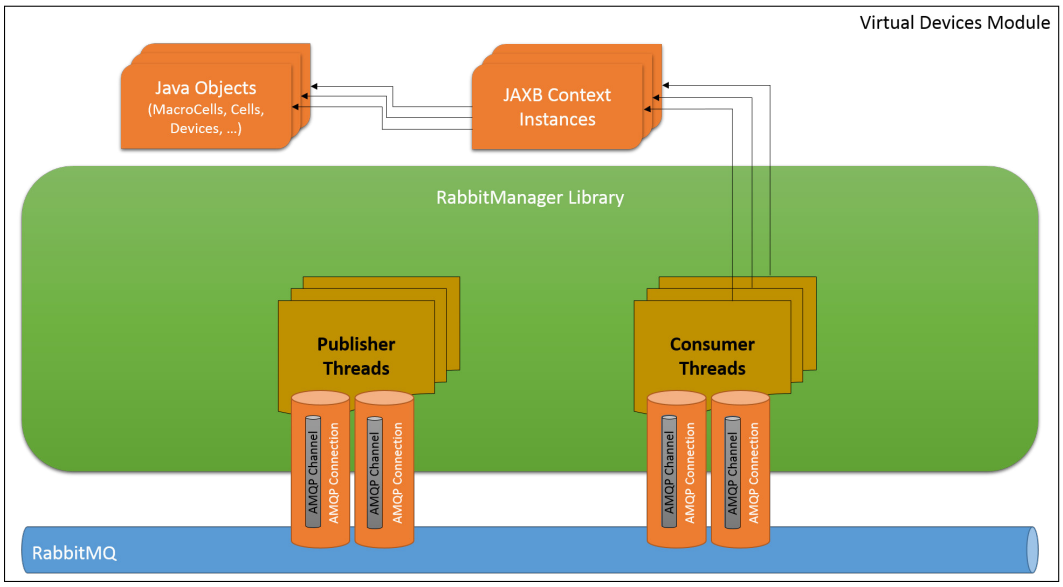

Figure 33 - RabbitManager Dynamic Design

Figure 33 depicts the first approach to the RabbitManager library. In this approach, using the RabbitManager library it was possible to create publishers and consumers of the RabbitMQ broker. Each publisher and each consumer instantiated a new thread to enable the concurrent creation and execution of several publishers and consumers at the same time, in a single application.

Furthermore, each publisher and consumer communicated with the RabbitMQ broker by creating a new AMQP Connection with a single AMQP Channel. Whenever a publisher or consumer was not needed anymore, both AMQP Channels and Connections were correctly closed and the publisher or consumer thread terminated.

This approach led to some issues. First, the VDM was creating a JAXB Context instance every time a new message was consumed and handled. Since the class loader for the JAXB Context instantiation took a lot of time, it affected the time needed to process each message, creating a delay if several messages arrive at the same time.

Thus, since each publisher and consumer created a new thread, in cases where a huge number of publishers and consumers were created, it affected in the CPU load of the machine and several problems with the thread handling. The threads created by publishers were created only to publish messages, but consumers created a different thread for each message handling.

To solve this problem, the internal architecture of the RabbitManager library was improved, resulting in a new architecture that solved some of the issues we had faced. This new approach is depicted in Figure 34.

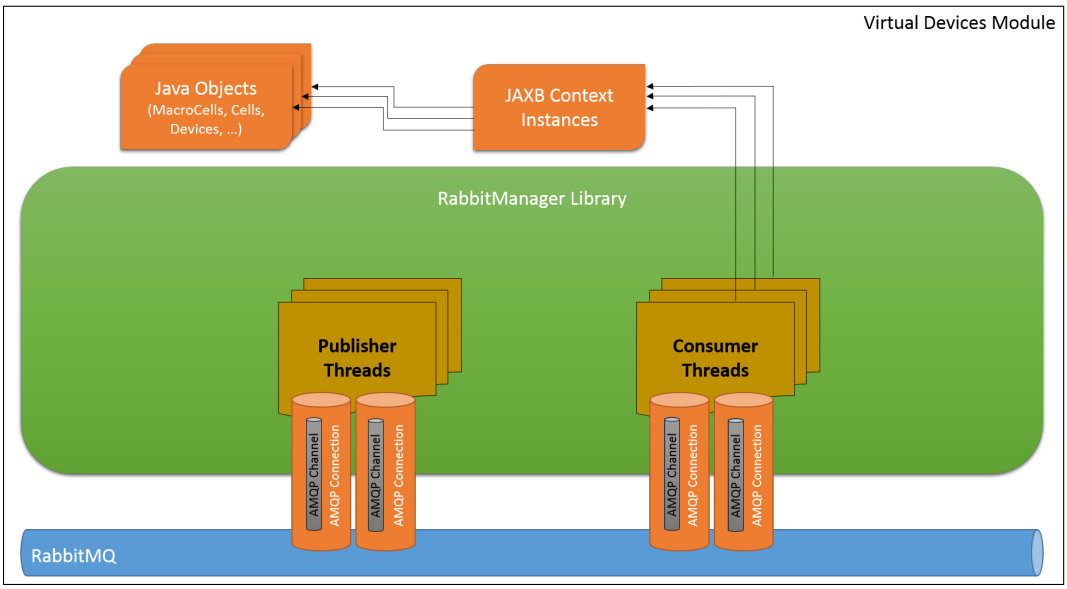

Figure 34 - RabbitManager Massively Multi-threaded Design

The new architecture was similar to the previous one, except the fact that the JAXB Context instantiation became a single instance shared among all the threads that were handling messages. The JAXB framework is thread-safe, allowing multiple threads to concurrently use the same JAXB Context instance.

This reduced the time for the handling of each message, however the huge number of threads instantiated by each publisher and consumer persisted. In cases where a consumer receives for example 20 messages at the same time, 20 threads had to be created, which resulted in high loads of CPU and thread handling issues.

Considering that each publisher instantiated a new thread, if 10 messages were published, this means that 10 publishers were created and consequently 10 threads establishing AMQP Connections were also created. The same problem was faced in terms of consumers, where each consumer was a new thread establishing the AMQP Connection and thus several new threads to consume and handle each message consumed.

This issue led us to create a new architecture, this time using a Thread Pool to control the number of created threads, and not to create any more thread at runtime. In terms of the AMQP Connections and Channels, there were also some improvements regarding the way how Connections and Channels were used. This new architecture is depicted in Figure 35.

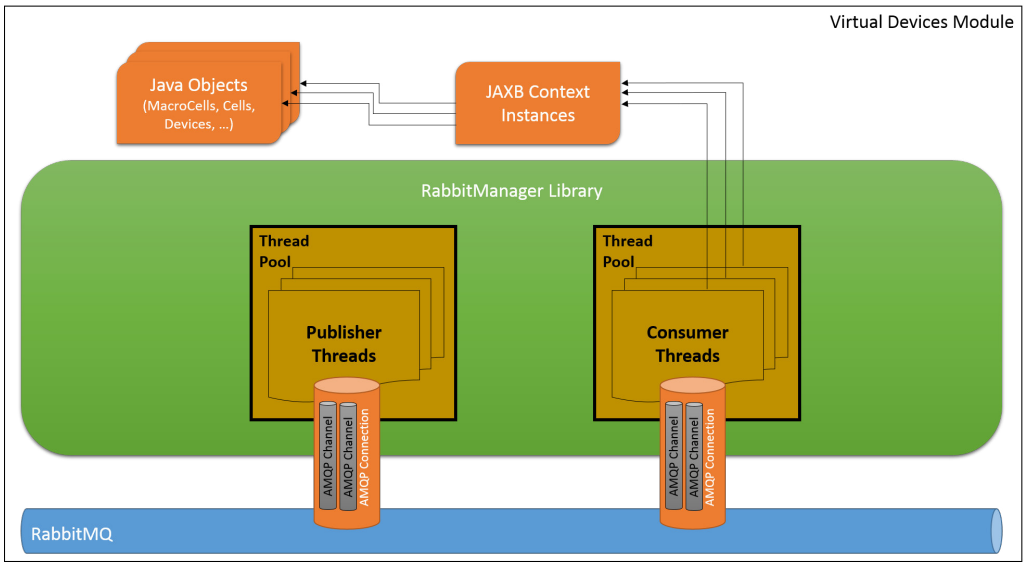

Figure 35 - RabbitManager Multiple Channel Design

This solution brought a more stable version and more controlled in terms of CPU load and monitoring.

Furthermore, a new way to use AMQP Connections and Channels was developed and tested. In previous versions of the library, each publisher and consumer created their own AMQP Connections with a single Channel, but this approach reduces the performance when a huge number of publishers and consumers are created or deleted, not only in terms of CPU load but also regarding network load.

In following generation, to provide a more stable and reliable library, we implemented only one AMQP Connection for all publishers, and one AMQP Connection for each consumer. This solution reduced the number of connections established between the VDM application and the middleware. Each AMQP Connection embraces multiple AMQP Channels, something similar to a fiber optical cable, giving us the opportunity to replace different connections for multiple channels.

In terms of network delay, a normal AMQP Connection takes two round-trips to be fully established. AMQP Channels take only one round-trip, which means that replacing multiple connections bearing one channel each with multiple channels inside a single connection was a step further toward an efficient solution. The AMQP Channels are also thread-safe and enable a bi-directional communication with the middleware.

With the new approach, several issues were solved from the previous architectures of the library. In terms of the VDM application and message handling, multiple JAXB Context instances were replaced by a single and shared instance, reducing the time needed to handle a single message. Regarding thread issues, the usage of a Thread Pool allowed to have a full control in managing the number of threads that were created and executed.

Finally, this architecture provided a simpler and more performant way to deal with the creation of multiple AMQP Connections with a single AMQP Channel. Instead, a few number of AMQP Connections were created and several AMQP Channels were created within the same AMQP Connection, taking advantage of the thread-safe capability provided by the AMQP Channels. The network load was also reduced giving the fact that few connections were needed to establish the communication between applications and the middleware.

A final small but equally important improvement was implemented in the final generation of the library, whose structure is depicted in Figure 36.

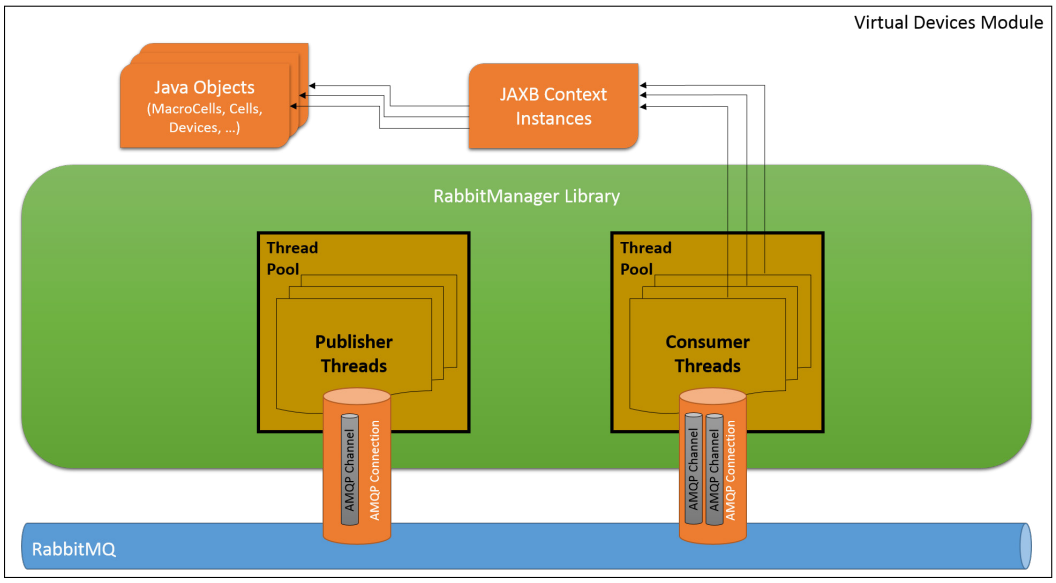

Figure 36 - RabbitManager Final Architecture

With the previous configuration, should a publisher publish several messages at a time, it needs to create an AMQP Channel for each message, reducing the performance of the application and increasing the load on the network. To solve these issues, a single AMQP Connection and a single AMQP Channel were created for each publisher, allowing it to publish all messages over an already opened channel.

In this final architecture we faced the best performance and we provided to our users an easy, configurable, reliable and high performant library. It could be used in any Java application and would provide an easy way to interact with a RabbitMQ broker. In fact, this library was used by all partners of the project ENCOURAGE and in all project pilots, and accelerated the integration of other applications with the ENCOURAGE Platform.

The evolution of the presented approaches was possible due to extensive performance tests that were performed within a reduced, yet realistic scenario. A brief explanation of the performance tests is given in the next subsection, providing information about the performance of the library in a simulated scenario, the obtained results and conclusions drawn from the tests.

4.7 Performance Tests

In this section, the performance tests, the environment in which they were performed, the analysis and the results are explained in more detail. These tests were performed as long as the Rabbit Manager library was implemented, resulting in a more robust, reliable and easy-to-use library that was further used by almost all partners in the ENCOURAGE Project to integrate their own systems with the ENCOURAGE Middleware, a RabbitMQ broker.

The performance tests were performed in a virtual machine with 2GB of RAM and 2.4GHz CPU.

Table 1 depicts the test cases that were executed.

Table 1 - ENCOURAGE Performance Tests

<table>
<tr><th>Number of messages</th><th>Interval between messages</th><th>Prefetch count</th><th>Persistency</th></tr>
<tr><td rowspan="5">10000</td><td>0ms</td><td rowspan="3">0</td><td rowspan="3">Yes</td></tr>
<tr><td>30ms</td></tr>
<tr><td>35ms</td></tr>
<tr><td>50ms</td><td rowspan="2">20</td><td rowspan="2">No</td></tr>
<tr><td>300ms</td></tr>
</table>

The main goal of this tests was to analyze and infer if the library was capable of handling a huge number of messages and how much time it would take to pass through all the components of the ENCOURAGE architecture, simulating a real scenario. In these tests a message was generated in the CIM format in a MPG, which was responsible for pushing those messages into the RabbitMQ broker to be forward to the Virtual Devices Module.

As soon as a message arrives to the Virtual Devices Module, it parses the message, stores the information contained in the message into the database, pushing the SQL statement to the Database Handler module through the broker. Then the VDM passes the received message to a simulated Supervisory Control, which for tests purposes waits for an estimated time of 80ms to simulate the SC processing.

The SC then is responsible to push a command to the RabbitMQ broker to be forward to the VDM again, which is then sent to a specific MPG (the one that sent the message). The VDM is the only module in the architecture that is capable of keeping track of who is sending messages and is capable of route the commands issued by the SC to the correct MPG.

The described exchange of messages can be summarized as: after a message being generated by a MPG, sent to the VDM, where it is parsed, stored and forward to the SC, the SC processes the message and issues a command that is forward to the VDM once again. The VDM then

parses the command sent by the SC, stores it and forwards it to the correct MPG. It is important to notice that all the message exchanges were made through the RabbitMQ broker.

Figure 37, Figure 38, Figure 39, Figure 40 and Figure 41 depict the results obtained with the following setup:

- Prefetch count: 0
- Delivery Mode: Non persistent
- Interval between messages published: 0ms, 30ms, 35ms, 50ms and 300ms, respectively;

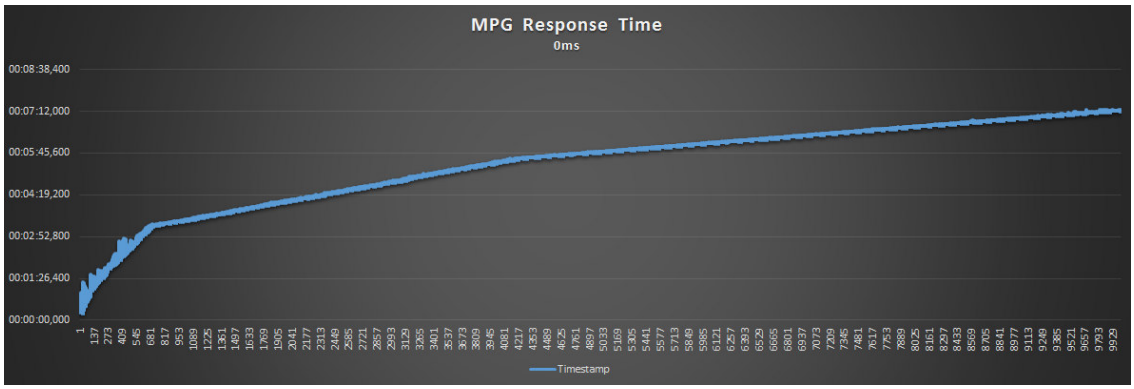

Figure 37 - Interval- 0ms | PrefetchCount - 0 | Delivery Mode - Non Persistent

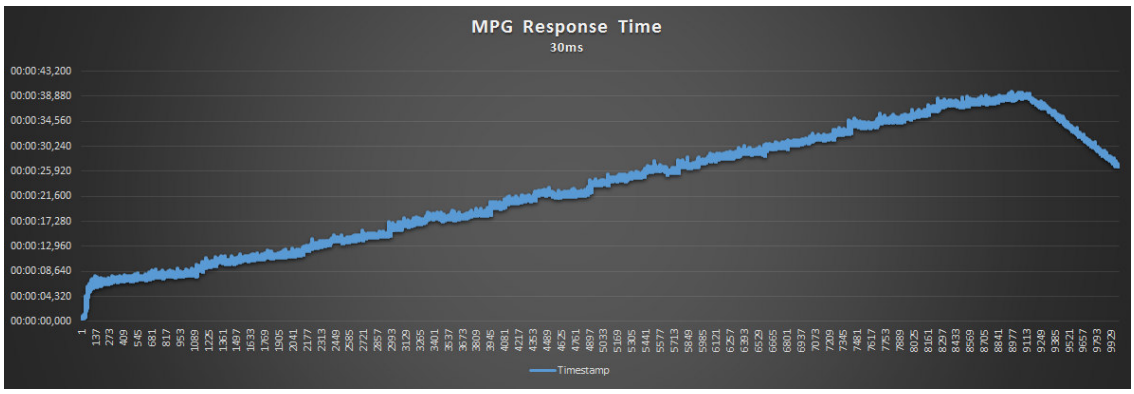

Figure 38 - Interval- 30ms | PrefetchCount - 0 | Delivery Mode - Non Persistent

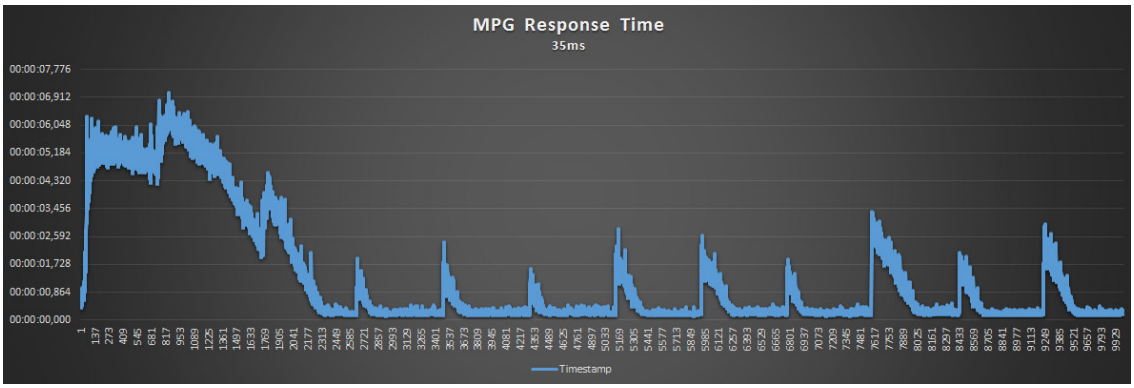

Figure 39 - Interval- 35ms | PrefetchCount - 0 | Delivery Mode - Non Persistent

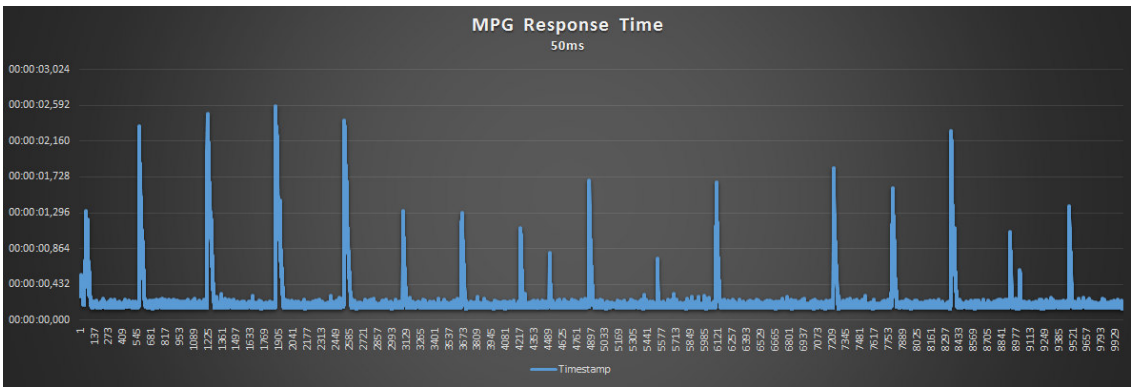

Figure 40 - Interval- 50ms | PrefetchCount - 0 | Delivery Mode - Non Persistent

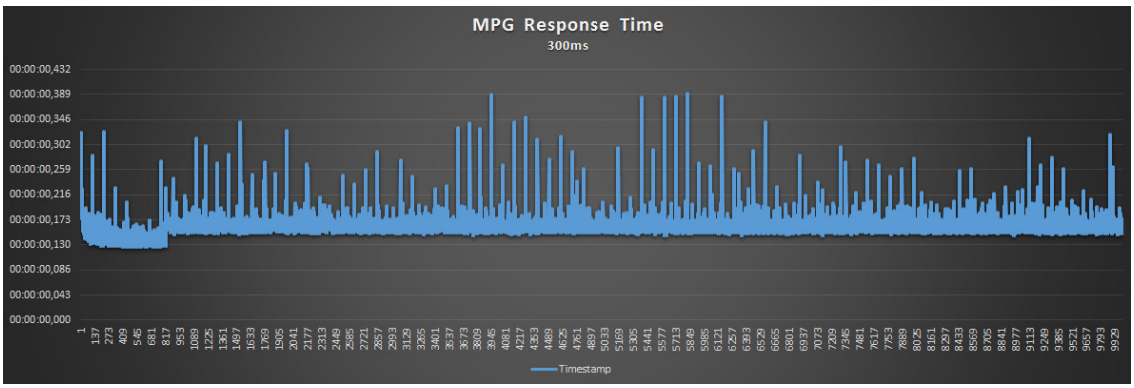

Figure 41 - Interval- 300ms | PrefetchCount - 0 | Delivery Mode - Non Persistent

It was noticed that when messages were marked as persistent, which means that the RabbitMQ broker saves the messages on disk, the system degrades its performance significantly, so for that reason the results are not presented here, since the system performance was very low. The same reason applies to the prefetch count. When the tests were executed with a prefetch count of 20, each consumer consumes batches of 20 messages at once, reducing the number of messages in the broker but impacting the systems performance, since it keeps consumed messages in memory.

It was concluded that the systems starts to degrade if messages were published with an interval less than 35 milliseconds, passing from increasing significantly the amount of time each message took since it is published by the MPG and the correlated command arrives the same MPG.

These tests were important to determine if the proposed ENCOURAGE architecture using a RabbitMQ broker as the message bus is capable of handling a huge number of messages in a robust, reliable and scalable manner. This means that a single Virtual Devices Module instance and a single Supervisory Control are capable of handling a huge number of devices largely distributed in several cells in the ENCOURAGE architecture.

In the next chapter, the Arrowhead Project is approached.

5 Design and Implementation of Components of the Arrowhead Project

The Arrowhead Project aims to develop a way to facilitate the cooperation between embedded systems, a highly scalable architecture, and aims to promote high levels of interoperability among different systems. With these objectives in mind, it present both similarities and differences to the ENCOURAGE Project. While the ENCOURAGE Project aims to enable one architecture where several systems communicate through a message bus, the Arrowhead Project aims to develop a service-oriented framework enabling multiple architectures and supporting several common functionalities such as to enable the registry and lookup of systems/services, the authorization, the authentication and even the orchestration of systems/services.

This section aims to provide an architectural design, implementation and interfaces between components of the ARROWHEAD Pilot for the Virtual Market of Energy – Generation 1. The main goal of the pilot is to demonstrate the flex-offer [85] concept, which has the goal of balancing energy demand and supply, through for example, the synchronization of consumption with production of energy. The balancing is achieved through shifting consumption loads to periods of the when energy is cheaper.

The flex-offer approach aims to reduce the energy costs of industrial or domestic installations, and it helps flattening out energy consumption, refraining, or at least minimizing, the usage and the negative impact of the peaking power plants. Several entities communicate through a so-called Virtual Market of Energy, which allows the scheduling of energy consumption or production. The flex-offers represent the flexibilities and schedules of energy of systems involved through formal data structures.

The flex-offer approach involves energy consumers, aggregators, and a so-called flexibility market. Sensors collect energy consumption data from Distributed Energy Resource (DERs).

Then, this data is transformed into device-specific energy requests that are encoded using flex-offers. Flex-offers contain information about the consumption profile of the device, their time and the amount of flexibility in energy consumption.

Aggregators aggregate micro flex-offers, i.e. flex-offers produced by appliances into macro flex-offers. These macro flex-offers are large enough to be submitted to an energy market and to comply with market regulations. The aggregation process can be performed by one of several aggregators. The aggregator choice may depend on several factors, e.g. type of appliance or its location. The aggregated flex-offers are then sent to a market. The market aims to minimize the total costs to the end user by scheduling the energy consumptions while respecting the constraints specified in the flex-offers (minimum/maximum power, earliest/latest start of energy consumption, etc.).

Finally, the plan with scheduled energy consumption is disaggregated by the aggregators to build up multiple individual DER plans, which are then sent to the respective DER controllers that drive the machines involved in an industrial process, the households or to an intelligent building.

In the following subsections, a general architecture overview of the pilot is given, providing some technical details on how the systems interact and cooperate. Then, a more detailed description of the components, which comprise the contributions related to this thesis, is also provided.

In the next subsection, a detailed description of the Arrowhead Framework is given, explaining each one of the composing elements of the framework, how they will work and what are they important to.

5.1 Architecture Overview

Figure 42 presents the Arrowhead Framework, which is the infrastructure responsible for supporting the interoperability among all the systems that need to communicate with other systems in an Arrowhead network. It follows a service-oriented architecture approach in which three main services are provided. These services are called the Core Services of the Arrowhead Framework, which provide the main functionalities to all systems that aim to enter an Arrowhead network. Those services are the Authorization, Authentication and Accounting; the Service Discovery and the Orchestration services.

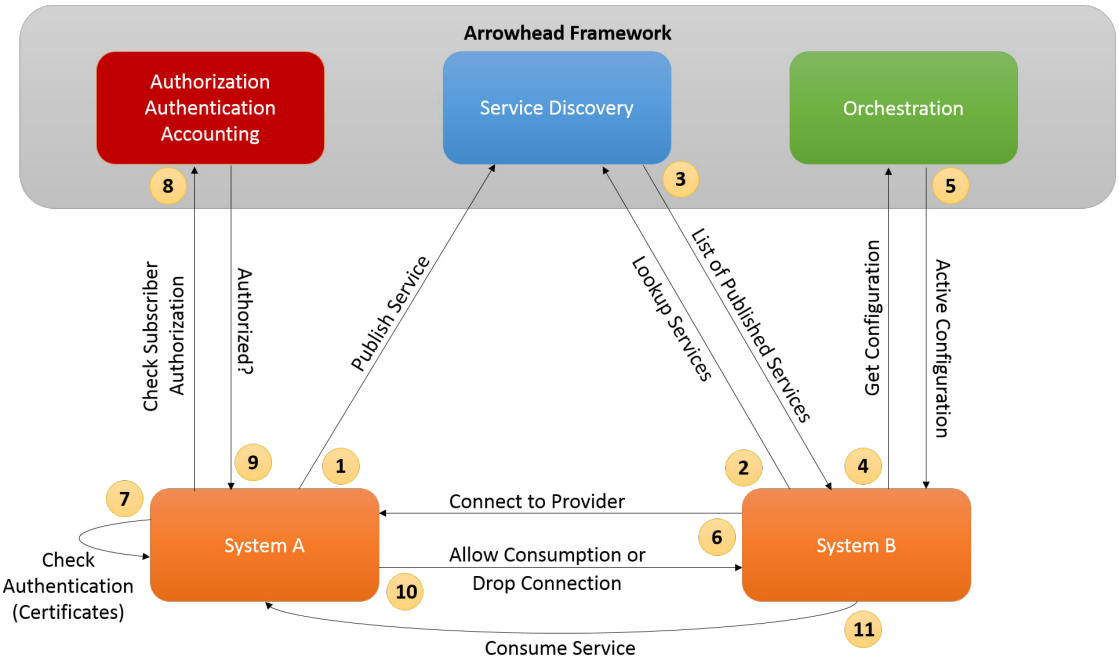

Figure 42 - Arrowhead Framework Architecture

These three Core Services allow systems to register their own services, lookup for other services (Service Discovery), authorize or authenticate their services (Authorization, Authentication and Accounting). An orchestration service (Orchestration) is also provided, facilitating the configuration of service connections, making it easier to connect to the most appropriate services.

This architecture is proposed to enhance the scalability of most automation applications, aiming to obtain high performance on the exchange of information between all services and the interoperability among all the interconnected services.

Each one of the Core Services exposes a set of REST interfaces (interfaces with other protocols (CoaP, MQTT) are under development), providing all the features that are needed to connect to the framework. This type of architecture allows systems to scale without major problems, facilitating the integration of new systems to the network, or the increase the complexity of systems in the network.

A basic use case of how a system may connect to the Arrowhead Framework and communicate with another system is depicted in Figure 42. Assuming a System A that aims to enter an Arrowhead network, it would need to publish (1) its own services (let's assume a Service X) through the Service Discovery Core Service, giving a service type as an identifier of the service. From this moment, the service provided by System A is discoverable for any other system that wants to consume that service, through specified service type provided at the time it was registered and published. Then, System B wants to communicate or consume from the service provided by the System A.

The System B needs to contact the Service Discovery Core Service (2) in order to lookup for service providers of Service X. The Service Discovery will then reply (3) with a set of service providers for that kind of service. At this moment, System B may request a specific configuration from the Orchestration Core Service (4). If any configuration exists, the Orchestration Service should provide information about what is the best service provider (5) for Service X (for instance based on the geolocation of System B).

If there isn't any configuration on the Orchestration, the System B should be able to choose one of the service providers of Service X, and should also already have all the information needed to contact that provider (addresses, protocols used, type of data returned, etc.). System B will then try to establish a connection (6) with System A (provider of Service X). The connection between the System B and the System A is validated using some certificates (7) on System A, allowing System A to accept or reject the connection request. If the certificates are valid, the System A should contact (8) the Authorization, Authentication and Accounting Core Service in order to validate if System B is authorized to consume the available service.

As soon as the System A completes all the verifications (9), it would accept (10) the connection request sent by System B, allowing it to start the consumption of Service X by the System A. At last, System B will start the consumption of the desired Service X.

Last, a Middleware, in this case an XMPP broker supports the communication of System A with System B. Each system represents an entity of the XMPP server, exchanging messages (flex-offer messages), providing a highly scalable, reliable and extremely fast communication among systems.

Since both systems were implemented following a REST interface, it was needed to implement an XMPP Extension, more specifically the XEP-0332 – HTTP over XMPP, enabling the integration with the XMPP broker. This approach is described in the following sections.

In the next subsections, a brief description of Arrowhead Framework is given, along with a brief description of each one of the three Core Services provided by the Arrowhead Framework.

5.1.1 Arrowhead Framework

The Arrowhead Framework is the most important part of the Arrowhead Project, since it allows the management and configuration of all the systems that are connected to an Arrowhead network.

The Arrowhead Framework, depicted in Figure 42, is composed by three main services, called the Core Services. Those services are the Authorization, Authentication and Accounting Core Service, the Service Discovery Core Service and the Orchestration Core Service.

Each one of the Core Services provides a set of functionalities, exposing a set of REST interfaces to which any system is able to connect and take advantage, in order to enter a specific

Arrowhead network and communicate with other systems in the network. Any system that is able to communicate with the Arrowhead Framework Core Services becomes an Arrowhead compliant system.

The main goals of the Arrowhead Framework is to allow all kinds of systems to provide and lookup for any kind of services, facilitating the orchestration of services among all the connected systems, and enhancing the security through a set of validations using certificates.

The Arrowhead Framework architecture follows a Service-oriented approach, enabling any kind of system to communicate with the Framework and register itself or lookup for other services, provided by any kind of system, aiming the high scalability of systems.

In the next subsections, a brief description of each one of the three Core Services is given.

5.1.2 Service Registry Core Service

The Service Discovery Core Service if the service provided by the Arrowhead Framework that enables systems to register their services or lookup for other services that are provided by other systems.

Whenever a system wants to publish its own service(s), it should contact the Service Discovery Core Service providing all the information needed to describe the services it wants to publish. This information is important in order to allow other systems to know the available services on the Arrowhead network.

The protocol used by the system provider, the type and formats of data provided by the service, the addresses and the service identifier are some examples of information that should be provided by the system that wants to publish its own service(s). This information will then be provided to other systems that are looking up for service providers of this type of service.

The main goals of this Core Service are to keep track of all services that are provided and registered on the Arrowhead network, being able to provide additional information about services or the system that provides the service, in order to allow the interaction among systems. The existence of this kind of Service Discovery service aims to enhance the scalability of a large-scale distributed system, keeping track of everything that is exposed and provided by each one of the interconnected systems.

5.1.3 Authorization/Authentication/Accounting Core Service

The Authorization, Authentication and Accounting Core Service is the Arrowhead Framework service that aims to support and facilitate the security procedures in any kind of network. It exposes a set of features to set rules to configure if a system is authorized to consume a service provided by another system, if a system is authenticated through a set of certificates, etc.

The authorization part of the Core Service may be used by a system to verify if some other system that wants to consume its own service is allowed or not. Then, if it is allowed, the authentication of the system that wants to consume the service may be verified through a set of certificates and key stores.

5.1.4 Orchestration Core Service

The Orchestration Core Service is the Arrowhead Framework service that is responsible for the creation and management of rules that define and control the way how systems interact with other systems in the network.

Basically, in the Orchestration Core Service it is possible to define rules specifying, for instance, the best service provider for a certain service consumer. This kind of rules may be defined based of the geo location of both service provider and consumer, network traffic load, current condition of the service provider (if it supports more connections from service consumers, etc.), among other features.

The usage of this Core Service allows the orchestration of service providers and consumers, facilitating the management and control among all the systems in terms of performance and scalability of the whole network.

5.1.5 Virtual Market of Energy

For managing flex-offers, it was proposed the usage of a general Virtual Market of Energy system, which is depicted in Figure 43, by providing a set of Service Oriented Architecture (SOA) interfaces, responsible for interconnecting several (existing and new) European Electricity Market Actors [85].

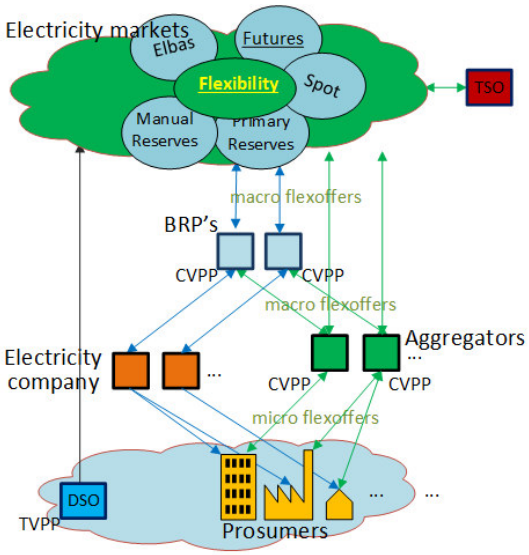

Figure 43 - Virtual Market of Energy [85]

The Prosumers or the Distributed Energy Resources (DERs) can both consume and produce electricity. Examples of Prosumers are residential houses, commercial buildings, manufacturing, and process industries. These generate flex-offers and consume schedules [85].

The Aggregators are specialized entities capable of aggregating several (micro) flex-offers from Prosumers into larger (macro) flex-offers. It is also capable of disaggregating (macro) flex-offer schedules, e.g., received from the Electricity Market. An aggregator might be an integrated part of a Balance Responsible Party (BRP) [85].

5.2 Aggregator

The Aggregator component is entrusted for acting as a bridge between the FlexOfferAgent and the Virtual Market of Energy. This section describes the interfaces and functionality of the Aggregator component.

This component is capable of executing the following actions:

- Receive (micro) flex-offer requests, sent by a FlexOfferAgent;
- Accept or reject flex-offers;
- Aggregate (micro) flex-offers into (macro) flex-offers;
- Disaggregate macro flex-offer schedules into micro flex-offer schedules;
- Send assigned (micro) flex-offer schedules to a FlexOfferAgent;
- Send flex-offers to and receive assigned flex-offer schedules from the Virtual Market of Energy (to be implemented in Generation 2);
- Request the power consumption and the state of a flex-offer (to be implemented in Generation 2).

Figure 44 presents a class diagram for the Aggregator component, containing all its classes.

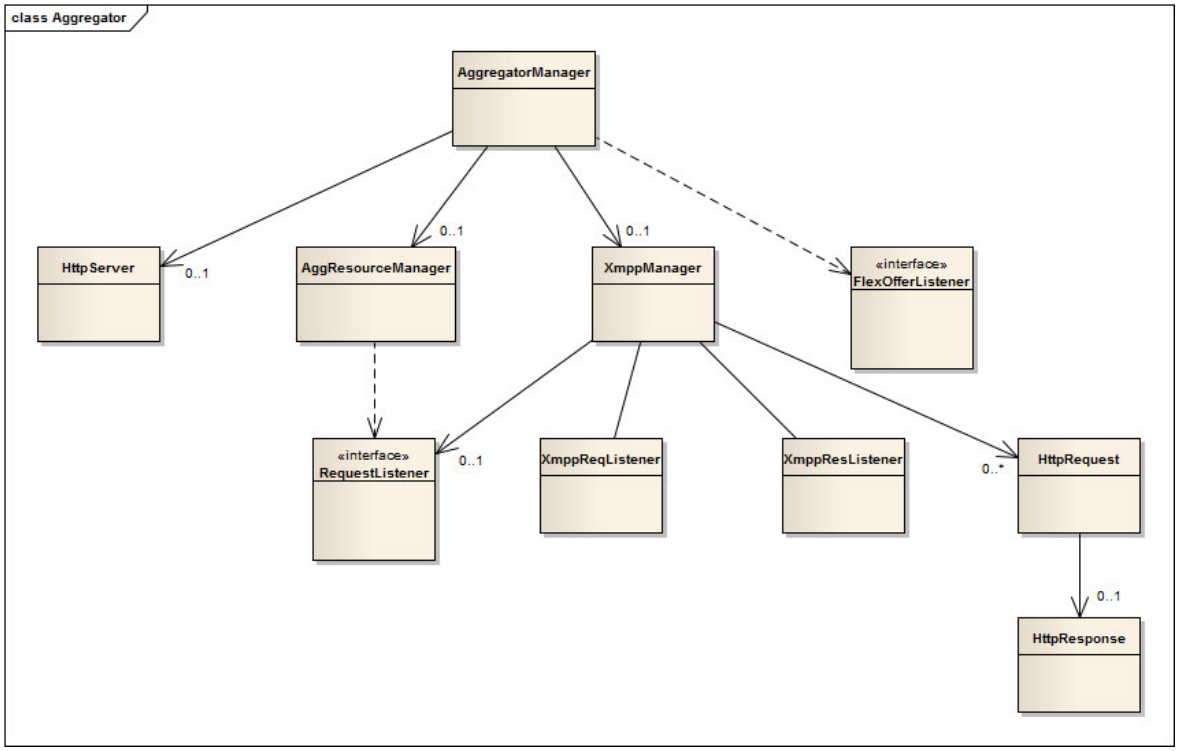

Figure 44 - Aggregator Class Diagram

Once an `AggregatorManager` instance is started, an `AggResourceManager` instance, an `XmppManager` instance, and an `HTTPServer` instance are created. The `AggregatorManager` instance handles the REST calls over XMPP, leveraging the other two classes. The `XmppManager` instance is responsible for enabling the communications between the `Aggregator` and the `FlexOfferManager` via the XMPP protocol. The `HTTPServer` takes care of encoding and interpreting the REST calls, which can also be performed by Web-based user interface.

At current state of development, the AggregatorManager component can listen for requests via both XMPP and HTTP protocols, for instance performed using a web browser.

Figure 45 presents a sequence diagram for the interactions between the `AggregatorManager`, `FlexOfferManager`, and Virtual Market of Energy.

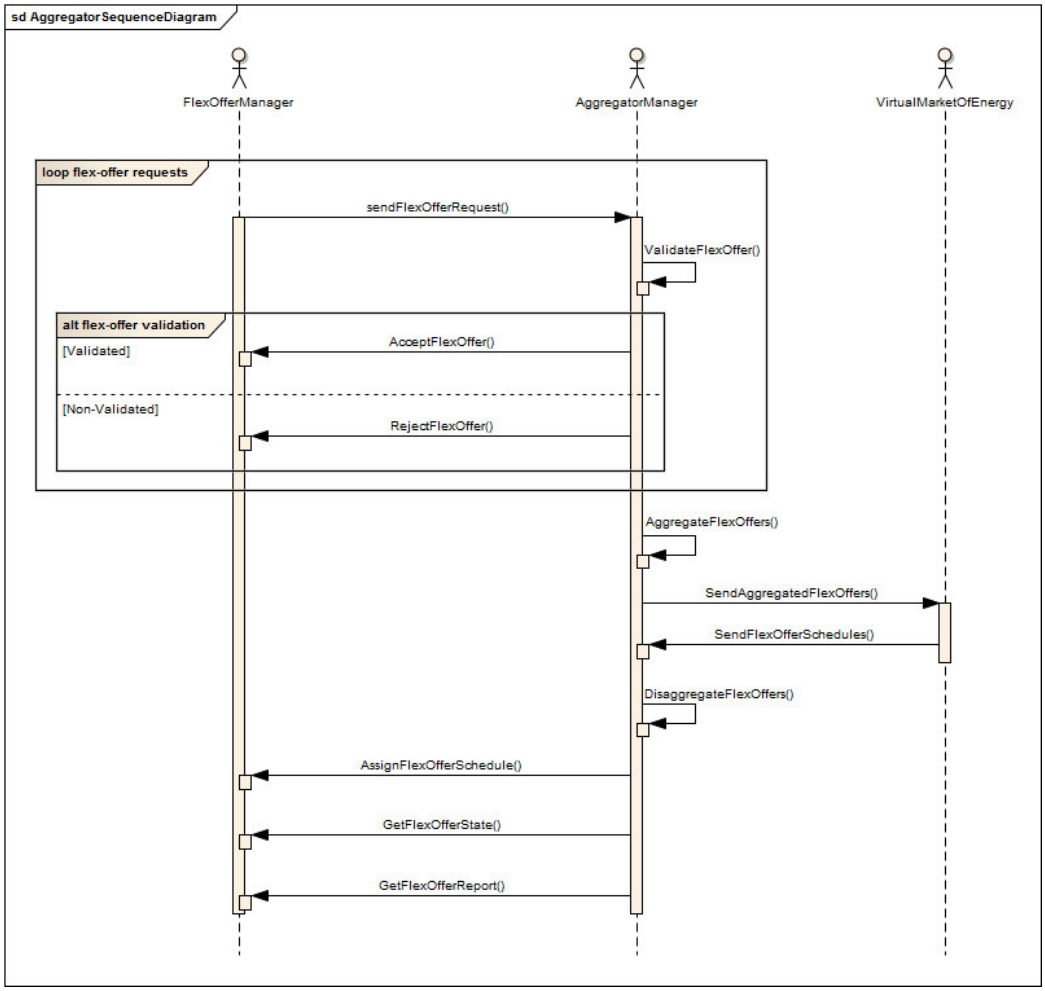

Figure 45 - Aggregator Sequence Diagram

A flex-offer request is created by the `FlexOfferManager` and sent to the `AggregatorManager` through an XMPP-REST based interface, using JAX-RS library. JAX-RS stands for "*Java API for RESTful Web Services*". JAX-RS is a Java programming language API which provides support for creating web services according to the REST architectural pattern, and its provides annotations to simplify the development and deployment of web service clients and endpoints.

A flex-offer request message consists of an XMPP IQ stanza, which is used as an envelope to the message containing an `HTTPRequest` object as payload of the stanza. The `HTTPRequest` contains a `FlexOffer` object inside. The FlexOfferManager creates these messages.

The Virtual Market of Energy replies to the `AggregatorManager` through the REST interface of the latter. The answer is a set of flex-offer schedules, encoded into a new XMPP IQ stanza containing a `HTTPResponse` object and based on a XMPP-REST interface. When the `AggregatorManager` receives the schedules, it is responsible for disaggregating them and sending the disaggregated schedules to the correct `FlexOfferManager`.

At any moment, the AggregatorManager may query the state of a flex-offer or the power consumption of a flex-offer execution. Both requests may be performed through an XMPP IQ message and an HTTPRequest, addressing a resource to retrieve data from a FlexOfferManager instance.

The communications with an AggregatorManager are performed through its interfaces, which are described in the following subsections.

5.2.1 REST Interfaces

The AggregatorManager component provides a set of interfaces on the four base resources that are provided, to be used according to the REST paradigm. For better understanding of how the resources are organized, Figure 46 presents a resource tree for the AggregatorManager.

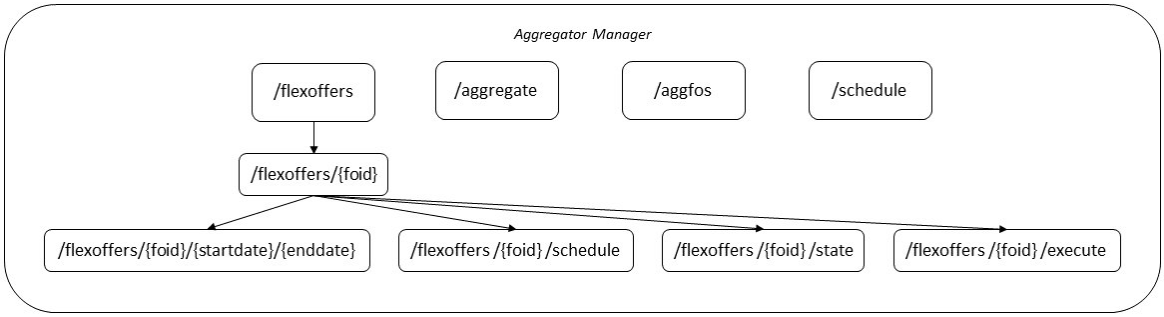

Figure 46 - Aggregator Manager Resource Tree

In this subsection, we will provide a more detailed description of each interface of the AggregatorManager component. Since the overall architecture assumes a REST-based architecture, a pointer (URL) to the resources needed to access those interfaces is also specified.

Each interface is related to a specific feature of the AggregatorManager. There are interfaces to operate on flex-offers, flex-offer schedules, flex-offer aggregation, etc.

1. Flex-offers management resources

Table 2 - Flex-offer management resources

Resource	Verb	Method	Input	Type	Output
/flexoffers	GET	getFlexOffers	-	-	List of received flex-offers

The AggregatorManager supports the handling of flex-offers. The "getFlexOffers" interface method is accessible through a "/flexoffers" resource, using a GET request, and

retrieves a set of flex-offers. The `AggregatorManager` receives multiple flex-offer requests and stores them. This method returns everything on that list.

2. Flex-offer aggregation resources

Table 3 - Flex-offer aggregation resources

Resource	Verb	Method	Input	Type	Output
/aggregate	POST	triggerAggregation	-	-	Returns true if flex-offers were successfully aggregated

Normally, after receiving several flex-offer requests, the `AggregatorManager` aggregates them into a macro flex-offer. The The "`triggerAggregation`" is used to trigger the Aggregator to place a flex-offer into the market, without having to wait for receiving a fixed number of flex-offers from households. This is particularly useful for demonstration proposes or in cases when the issuers of the request require a fast answer.

The "`triggerAggregation`" interface method is accessible through sending a POST request on an "`/aggregate`" resource, and it is used to initiate the process of aggregating received flex-offers into a macro flex-offer, to be sent to the Virtual Market of Energy.

3. Aggregated flex-offers resources

Table 4 - Aggregated flex-offers resources

Resource	Verb	Method	Input	Type	Output
/aggfos	GET	getAggFlexOffers	-	-	List of aggregated flex-offers
/aggfos	DELETE	deleteFlexOffers	-	-	Returns true if flex-offers were successfully deleted

This interface is used to manipulate aggregated flex-offers. The method "`getAggFlexOffers`" is accessible using a GET request, and retrieves a list of aggregated flex-offers. To delete all flex-offers, the "`deleteFlexOffers`" interface method from the `AggregatorManager` must be used, which is invoked on the same "`/aggfos`" resource, but through a DELETE request.

4. Scheduled flex-offers

Table 5 - Scheduled flex-offers resources

Resource	Verb	Method	Input	Type	Output
/schedule	POST	triggerScheduling	-	-	Returns true if flex-offers were successfully scheduled

The "triggerScheduling" method is responsible for generating a schedule for each one of the aggregated flex-offers on the Aggregator and setting the state of each flex-offer as "Assigned" through a FlexOffer class method "setFlexOfferState". After all the flex-offers are disaggregated, the flex-offer schedules are ready to be sent to the FlexOfferManagers.

The "triggerScheduling" method may be accessed through the "/schedule" resource using a POST request, and the method allows to make a quick schedule of the flex-offers, and to trigger the process that disaggregates them. This interface is mostly used for demonstration purposes, to force the last part of the process to start immediately.

5.2.2 Java Interfaces

Apart from REST interfaces, the AggregatorManager has an AggIf Java interface, which allows the communication between modules implemented locally inside the AggregatorManager. A set of methods were implemented on the AggIf interface, whose description is be given in this section.

Table 6 - Aggregator Java interface

AggIf Java Interface			
Method	Input	Type	Output
getAggregatedFlexOffers	-	-	List of aggregated flex-offers
getAggregatedFlexOffer	flexOfferId	Integer	Aggregated flex-offer
generateId	-	-	Flex-offer id

The "*getAggregatedFlexOffers*" method is invoked to retrieve a list of aggregated flex-offers from the AggregatorManager. However, if a specific aggregated flex-offer is needed, the "*getAggregatedFlexOffer*" method should be invoked, using the desired flex-offer id as a parameter. Finally, the AggregatorManager component is responsible for assigning an id to each flex-offer that has been received. To assign an id to each flex-offer, the "*generateId*" method produces an Integer that will be used as the flex-offer id.

5.3 Flex-Offer Agent (FOA)

The FlexOfferManager is the component that is responsible for acting as a bridge between the FlexOfferAgent and the AggregatorManager. This component is entrusted with managing a set of DERs and thus it controls interactions with the physical devices.

Moreover, a `FlexOfferManager` is meant to be associated with an Arrowhead subsystem that advertises the services of the `FlexOfferManager` through the Arrowhead Service Discovery, and manages security. However, note that Arrowhead framework is not connected in this pilot generation, which is described later on this thesis..
Figure 47 presents a class diagram for the FlexOfferManager component.

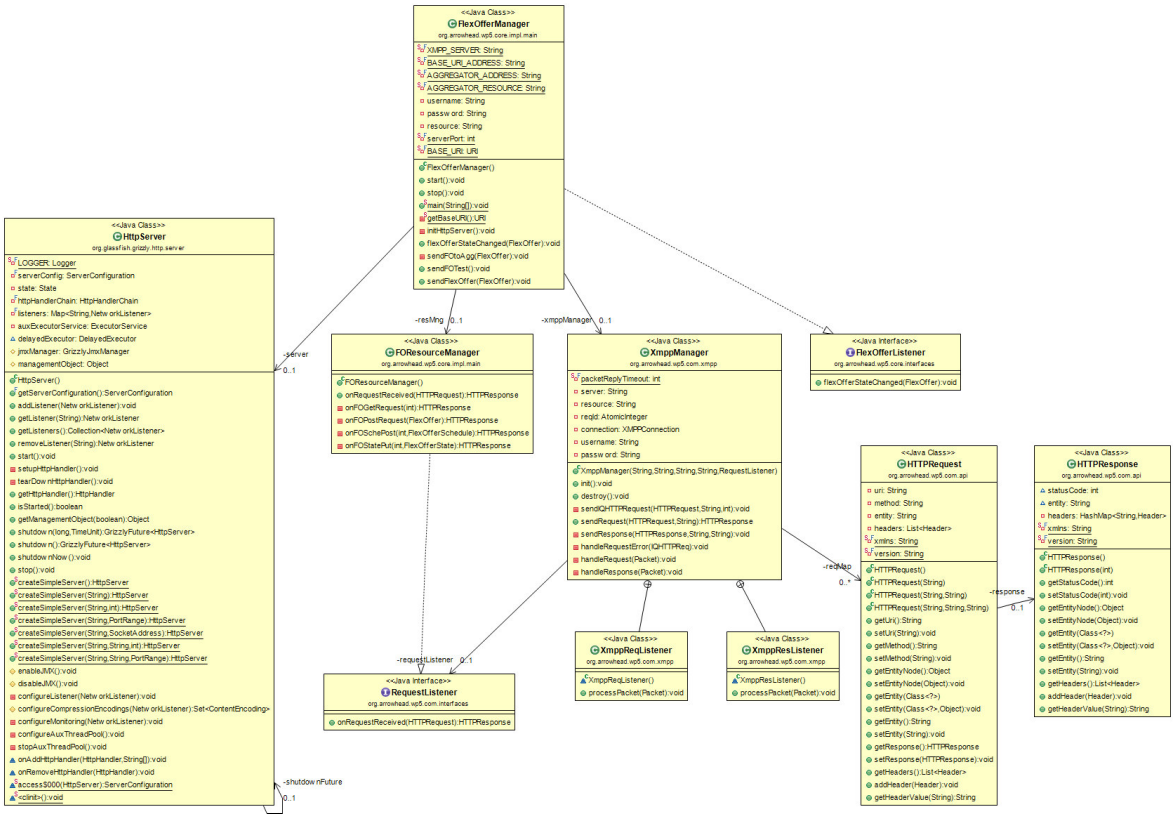

Figure 47 - FlexOfferManager Class Diagram

Once a `FlexOfferManager` is started, a `FlexOfferResourceManager` instance, an `XmppManager` instance and an `HTTPServer` instance are created. The `FlexOfferManager` instance handles the REST calls over XMPP, leveraging the other two classes. The `XmppManager` is responsible for enabling the communication between the `FlexOfferMAnager` and the `AggregatorManager`. The `HTTPServer` is responsible to interpret REST calls generated through HTTP request, for example from a Web-based user interface.

This component manages the flex-offers and all the DERs attached to the component, and it provides interfaces with a set of methods used to access each functionality. Like in the case of the `AggregatorManager`, all methods from this component are accessed in a REST-based manner. In this section we describe the methods of each interface.

5.3.1 REST Interfaces

The `FlexOfferManager` component provides a set of interfaces on the two resources (flex-offers and DERs) that are provided, to be used according to the REST paradigm. For the sake of clarity on how the resources are organized, Figure 48 and Figure 49 present the resource tree provided by the FlexOfferManager component. Two base resources are provided, enabling flex-offer and DER handling functionalities on the component. The "`/flexoffers`" base resource provides functionalities to generate flex-offers, assign flex-offers schedules, order the execution of flex-offers or retrieve flex-offers.

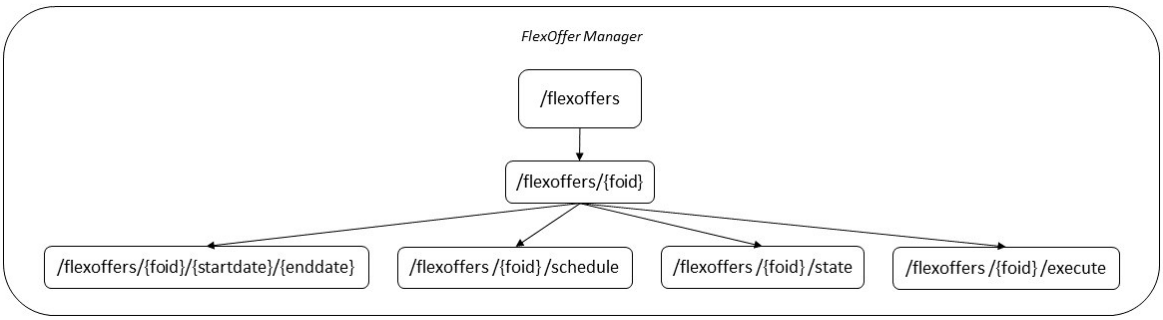

Figure 48 - Flex-offer Manager Resource Tree

The "`/ders`" base resource enables the FlexOfferManager to interact with the DERs attached to it. This resource allows the generation of DER programs or DER profiles retrieve information on the state of execution of flex-offers, etc.

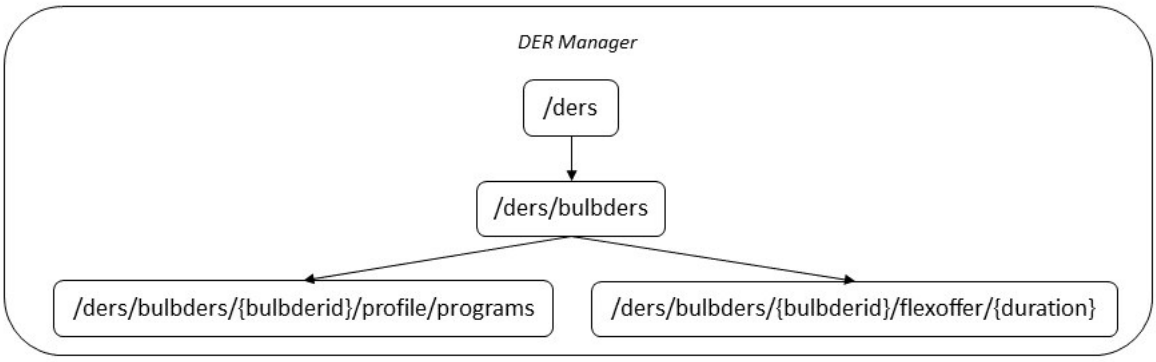

Figure 49 - DER Manager Resource Tree

A `FlexOfferManager` component provides two different interfaces, one to access flex-offer functionalities and another one to manage DERs attached to the component. Each interface is accessed via REST, using different resources and REST verbs combinations. A more detailed description of each interface, methods and how they can be accessed is given throughout this subsection.

1. Flex-offer management resources

Table 7 - Flex-offer management resources

Resource	Verb	Method	Input	Type	Output
/flexoffers	GET	getFlexOffers	-	-	List of flex-offers
/flexoffers	DELETE	deleteFlexOffers	-	-	Returns true if flex-offers were successfully deleted
/flexoffers/{foid}/{startdate}/{enddate}	GET	createFlexOfferResource	foid	Integer	True/false
			start date	String	
			enddate	String	
/flexoffers /{foid}/execute	GET	executeFlexOfferResource	foid	Integer	True/false
/flexoffers /{foid}	GET	getFlexOffer	foid	Integer	Flex-offer
/flexoffers /{foid}	PUT	setFlexOffer	foid	Integer	-
/flexoffers /{foid}	DELETE	deleteFlexOffer	foid	Integer	-
/flexoffers /{foid}/state	GET	getFlexOfferState	foid	Integer	Flex-offer State
/flexoffers /{foid}/state	PUT	setFlexOfferState	foid	Integer	-
/flexoffers /{foid}/schedule	GET	getFlexOfferSchedule	foid	Integer	Flex-offer Schedule
/flexoffers /{foid}/schedule	PUT	setFlexOfferSchedule	foid	Integer	-
/flexoffers /{foid}/schedule	DELETE	deleteFlexOfferSchedule	foid	Integer	-

To access methods on this interface, a base "`/flexoffers`" resource is provided. The interface method "`getFlexOffers`" is accessible through an "`/flexoffers`" resource, using a GET request, and retrieves a list of flex-offers. To delete all flex-offers, the "`deleteFlexOffers`" method may be invoked on the `FlexOfferManager`. The resource to invoke this method is the same as above ("`/flexoffers`") but in this case a DELETE request should be used.

The interface method "`createFlexOfferResource`" is used to command the generation of a new flex-offer, by using a GET request on the "`/flexoffers/{foid}/{startdate}/{enddate}`" resource, passing some

parameters like flex-offer id, start date and end date. The method "executeFlexOfferResource" is used by means of a GET request on a "`/flexoffers/{foid}/execute`" resource to order the execution of a flex-offer. The flex-offer id specifies which flex-offer's execution should be started.

To access a specific flex-offer, the method "`getFlexOffer`" is accessed through a GET request of the flex-offer id on a "`/flexoffers/{foid}`" resource. The method "`setFlexOffer`" is invoked to create a new flex-offer on the `FlexOfferManager`, by executing a PUT on the "`/flexoffers/{foid}`" resource, and passing the flex-offer id as a parameter to the resource. A flex-offer is deleted through the method "`deleteFlexOffer`", by using a DELETE request on the "`/flexoffers/{foid}`" resource with the flex-offer id as a parameter. To set or get the state of a flex-offer, GET and PUT requests are executed through the method "`getFlexOfferState`" on "`/flexoffers/{foid}/state`" resource, passing the flex-offer id as a parameter, and the new state in the case of the PUT request.

Regarding flex-offer schedules, three methods are available on this interface, to be used on the "`/flexoffers/{foid}/schedule`" resource. A GET request is used to retrieve a flex-offer schedule, by passing the flex-offer id as a parameter to the resource. A PUT request is used to create a new flex-offer, passing the flex-offer id as a parameter. Finally, to delete a flex-offer schedule, a DELETE request is used, passing the flex-offer id to be deleted as a parameter.

1. DER management resources

Table 8 lists all interface methods to be used while managing DER resources, together with the input parameters, and the output types. A more detailed description of each method and how it can be accessed is described in the following.

Table 8 - DER management resources

Resource	Verb	Method	Input	Type	Output
/ders	GET	getDers	-	-	AbstractDER Object
/ders /bulbders	GET	getBulbDers	-	-	List of Bulb DERs
/ders/bulbders/{bulbderid}/profile/programs	POST	createBulbProgram	bulbderid	Integer	Returns true if program was successfully created
/ders/ /bulbders/{bulbderid}/profile/programs	DELETE	deleteBulbPrograms	Bulbderir	Integer	Returns true if all bulb programs were successfully deleted

/ders/bulbders/{bulbderid}/profile/programs	GET	getBulbPrograms	bulbderid	Integer	Bulb program if It exists
/ders/ /bulbders/{bulbderid}/flexoffer/{duration}	GET	getWashingMachineFlexOffer	bulbderid	Integer	FlexOffer object if successfully created
			duration	Integer	

The interface method "`getDers`" is accessible through the "`/ders`" resource using a GET request, and it retrieves an `AbstractDER` object. To retrieve the list of all BulbDER instances, the "`getBuldDers`" interface method from the `FlexOfferManager` may be used through a GET request. To create a new program for a BulbDER, the "`createBulbProgram`" method can be invoked with a POST request, on a "`/ders/bulbders/{bulbderid}/profile/programs`" resource. A BulbDER id must be passed as a parameter to the resource. To delete a specific BuldDER program, the same resource can be used, but in this case the request must be of type DELETE. To retrieve all the programs from a specific BulbDER, a method "getBulbPrograms" is available and can be accessed using a GET request on a "`/ders/bulbders/{bulbderid}/profile/programs`" resource, passing the BulbDER id as a parameter to the resource. Finally to get a WashingMachine flex-offer, the "`getWashingMachineFlexOffer`" method can be invoked using a GET request, through a "`/ders/ /bulbders/{bulbderid}/flexoffer/{duration}`" resource, passing it a DER id and a duration as a parameter.

5.4 Virtual Market of Energy Pilot

In the context of the Arrowhead Project a pilot was implemented to demonstrate the usability of flex-offers within a Virtual Market of Energy. To demonstrate the flex-offer concept, the implement pilot consists on having a washing machine, built in Legos, simulating a real washing machine appliance, using a FlexOfferAgent, a so-called Washing Machine DER and a Washing Machine Controller.

The main goal of this pilot is to present a simple scenario where a user chooses an interval in which he wants to execute a program in the washing machine. Since the program and the interval for the program to be executed are defined, one or several flex-offers are generated and are then pushed to an Aggregator through a FlexOfferAgent. The Aggregator is then responsible to collect several flex-offers from several DERs that will be aggregated, creating a so-called macro flex-offer.

The Aggregator pushes the macro flex-offer to the Virtual Market of Energy, which is then responsible for giving the best interval for the chosen program to be executed. However, the

purpose of this pilot is to demonstrate the capabilities of the flex-offer concept, so the communication between the Aggregator and the Virtual Market of Energy is not implemented.

Figure 50 and Figure 51 depicts the prototype built for the pilot. This prototype is composed by a Raspberry Pi to execute the Washing Machine DER and the Washing Machine Controller, several electronic pieces and a Lego motor to control the “washing machine”. The Raspberry Pi, through the presented connections using GPIOs controls the motor.

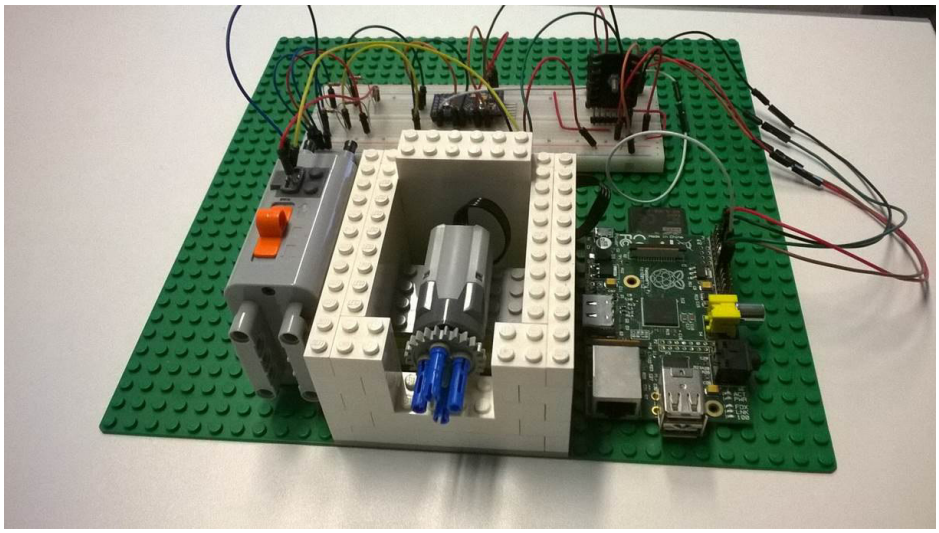

Figure 50 - Lego Washing Machine (1)

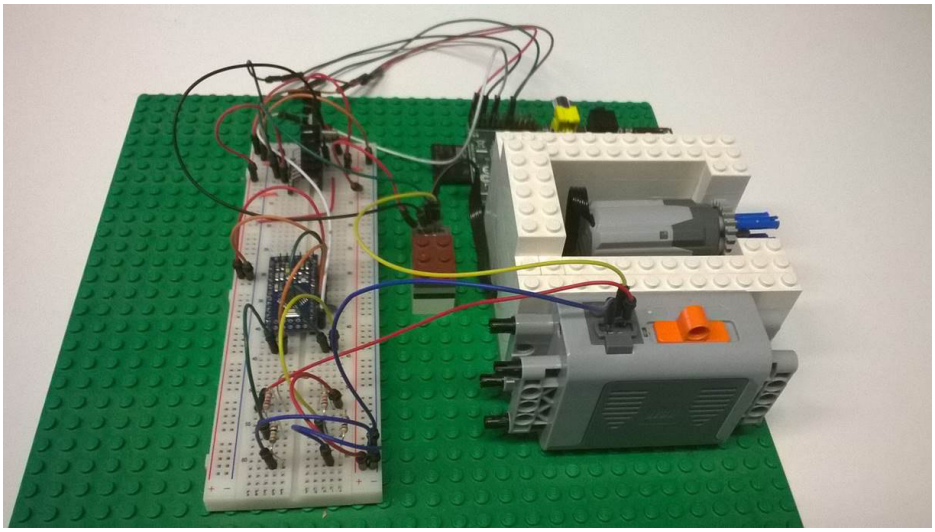

Figure 51 - Lego Washing Machine (2)

In this section, a brief description of the Washing Machine DER and the Washing Machine Controller is provided.

5.4.1 Washing Machine Distributed Energy Resource (DER)

For the washing machine prototype, a DER was implemented. This DERManager is responsible for generating flex-offers based on an existing profile. For demonstration purposes, two different programs were defined and assigned to the profile. Those two programs were created in order for the DER to be able of generating specific flex-offers for the washing machine prototype.

To generate flex-offers, a specific algorithm was implemented in the Washing Machine DERManager. This algorithm creates basic flex-offers compatible with the prototype, basically defining intervals where the motor should work or not. Every flex-offer is generated based on the programs and profile previously defined.

The Washing Machine DERManager component is capable of generate flex-offer for the Lego washing machine, receive and convert flex-offer schedules sent by the Aggregator, and assign the schedule to the Washing Machine Controller Manager.

5.4.2 Washing Machine Controller

The Controller component is manufacturer specific, which means that each controller implementation depends on the type of device, protocols used or types of commands a device receives. The Controller is also responsible for measuring the consumption.

The Washing Machine ControllerManager component is responsible for interacting with the real appliance, in this case being the Lego washing machine prototype. To enable this communication, the Lego motor was connected to the Raspberry Pi through GPIO pins. This setup allows the Controller application to execute commands on the motor. To this aim, an open source library named Pi4J was used. This library provides all the features needed to interact with real devices through a Raspberry Pi.

The ControllerManager component is responsible for establishing and managing the communication between the Raspberry Pi and the FlexOfferManager. Basically, the Controller can receive a schedule, generate a plan based on that schedule, and eventually execute it.

For this demonstration, a basic algorithm was implemented to generate and execute plans. For demonstration purposes, based on the duration defined for the schedule, the Controller generates a similar plan every time a schedule is received. Basically it receives the schedule and generates a plan (refer to AbstractPlan class) with 4 intervals of execution. First the motor works for 5 seconds to the right, and then pauses for 1 second.

After the pause, the motor works another 5 seconds to the left and pauses again 1 second. Finally it works two more times, 3 seconds to the right and then 3 seconds to the left, without pauses between them. As soon as the plan execution starts, another process on the Controller is started, and it is responsible for measuring the consumption of the motor.

6 Conclusions and Future Work

In the context of distributed systems, the traditional problem of communication and the novel problem of integration represent the most important - and the most difficult - part of their design and implementation. These difficulties become harder to overcome when designing large-scale distributed systems.

In this thesis, the main goal was to study and analyze Middleware solutions to provide an easier way to design architectures for large-scale distributed systems. The thesis couples general analysis with a practical approach that leads to the implementation of the solution over some real scenarios. Focus was given to middleware application to large-scale distributed systems, and deep insights on usage of the main communication protocols leveraged in this thesis (AMQP and XMPP).

The work described in this thesis was developed in the context of two European projects, the ENCOURAGE Project and the Arrowhead Project, which were essential for the study and analysis of real scenarios where Middleware solutions are useful and a good choice for these architectures.

In the context of both projects, some requirements were prioritized, such us the system performance of distributed operations, the Quality of Service for communications, its scalability and the interoperability, which allows these systems to interconnect with different technologies.

The research on Middleware technologies provided the needed knowledge to understand and explain why a solution might be better in one case or another, depending on the project requirements and the goals each project has to achieve. In both ENCOURAGE and Arrowhead project sections, these choices are explained considering the context of both projects.

In the ENCOURAGE project, the monitoring and controlling of the devices belonging to the large scale distributed system, such as sensors, actuators and energy readers, were achieved

successfully through the usage of the RabbitMQ broker, which allowed all systems to communicate with each other. Some performance tests were made, with the objective of showing its high performance, stability, scalability and availability achieved using a Middleware based architecture, in this particular case based on a RabbitMQ broker implementing the AMQP protocol.

The flex-offer concept, central to the implementation of virtual energy markets, was proved in the first generation pilot for the Arrowhead project, where two systems successfully exchanged flex-offer messages to simulate a real case scenario. In the scenario two system interacted, the first to provide its time flexibility for consumption energy, and the second for aggregating several flex-offers and sending them to an energy market. Furthermore, the integration with the Arrowhead Framework proved that the usage of a set of Core Services providing features like service discovering, authorisation, authentication, accounting and orchestration allows systems to find each other, coordinate services operations and initiate a safe and reliable communication with another service provider system of interest.

The intensive research, design and discussions of the architectures and the implementations achieved along with thesis, allowed me to develop knowledge in different areas and technologies that were almost unknown to me.

Follows a list of contributions of this thesis:

Easy to use and RabbitMQ Java library:

The RabbitMQ Java library (subchapter 4.6) may be considered the major contribution for the ENCOURAGE project (chapter 4), and consequently for this thesis as far as implementation is concerned. This library was fully implemented by me and was leveraged by all partners involved in the project to integrate their systems with a RabbitMQ message broker. One main characteristic of the library is that it facilitates to use and integrate with a new or existing Java project.

The different implementation steps of this library, described in detail in subsection 4.6 (the RabbitManager Library subchapter of the Implementation and Design of Components on the ENCOURAGE Architecture chapter), allowed the optimization of the library and most important, it allowed to perform several tests to determine the efficiency of the broker in different scenarios and the capability of the implemented modules for the project.

Implementation of key modules of the ENCOURAGE architecture:

The implementation of the Virtual Devices Module (subchapter 4.1.1) and the Database Handler (subchapter 4.1.2) modules of the ENCOURAGE architecture were also focal for this thesis. The Virtual Devices Module was the central module of the architecture, which keeps track of everything that happens within the architecture since every message exchange between all systems should pass for this module. This module virtualizes the real devices connected to the system and also allows the monitoring of those devices through a web-based interface. The

Database Handler provided a basic but efficient way to deal with the database that supported the ENCOURAGE project, acting almost like a load-balancer for the database.

Design of the architecture for the Arrowhead Project:

In the context of the Arrowhead project, my contributions were on the architectural and design parts. Several interactions and meetings with several partners involved in the project led to decision regarding the best architecture and technologies to use in the implementation of the Arrowhead Framework and applications.

In terms of the implementation of modules of the architecture, part of the flex-offer concept pilot (subchapter 5.4) was implement by me, particularly the Lego Washing machine prototype, which was used to demonstrate a simulated case of how the implemented systems could control and monitor an actual washing machine. I was responsible for the implementation of the pilot where a Lego motor was controlled by a Raspberry Pi through its GPIOs, applying on the Lego Washing Machine the result of a flex-offer schedule obtained after some flex-offer messages exchanged. This pilot is discussed in the Washing Machine Distributed Energy Resource (DER) and the Washing Machine Controller subchapters of the Implementation and Design of Components of the Arrowhead Project chapter (chapter 5).

Integration of systems with the Arrowhead Framework:

In the Arrowhead Project pilot, each system described and partially implemented was also integrated with the Arrowhead Framework Core Services, using a Service-Oriented Architecture supported over the XMPP protocol, which was used to establish communications between systems.

Technical reports, conference papers and a journal paper:

Finally, some technical reports and papers were published reporting the results of the work done during the time of this thesis:

- Conference and Workshop Papers
    - Luis Lino Ferreira, Laurynas Siksnys, Per Pedersen, Petr Stluka, Christos Chrysoulas, Thibaut Le Guilly, Michele Albano, Arne Skou, César Teixeira, Torben Pedersen, “Arrowhead Compliant Virtual Market of Energy”, explaining the advantages of the Virtual Market of Energy in the context of the Arrowhead project. [85]
- Journal Papers
    - César Teixeira, Michele Albano, Arne Skou, Lara Pérez Dueñas, Francesco Antonacci, Rodrigo Ferreira, Keld Lotzfeldt Pedersen, Sandra Scalari, “Convergence to the European Energy Policy in European countries: case

studies and comparison", comparing the several approaches to energy markets, green certificates, energy incentives, etc. in European countries. [38]

- Luis Lino Ferreira, Luis Miguel Pinho, Michele Albano, César Teixeira, "Adaptive offloading for infotainment systems", providing a service-oriented architecture pattern solution for an adaptable offloading mechanism, taking into account the QoS requirements of the applications. [84]

In the next subchapter, some ideas of what might be done in the future within the context of this thesis, related to the projects in which this thesis is based on.

6.1 Future Work

Although the ENCOURAGE project is officially finished, its software is still in use in real installations and in other projects. Nevertheless, it is possible to add new features and perform more work to improve the system and the knowledge about it. In the context of the Arrowhead project, a set of possible ideas to further develop the system are also described.

ENCOURAGE Project

Performance tests:

In terms of performance tests, taking into account that the performance tests were performed in a local scenario, I propose to portray some more performance tests in real scenarios with up to thousands of devices publishing their data, for both the ENCOURAGE general architecture, and for the RabbitMQ library that was implemented for the project.

Implementation optimizations:

The tests on performed on the RabbitMq library allowed to improve its performance but further improvements are possible making it possible for the ENCOURAGE middleware to process more requests with enhanced reliability. I foresee that the performance of the system can be improved by changing the threading mechanisms to better scale for multicore processors. Additionally, the system should also be able to dynamically scale in a distributed system. Improvement in performance can also be achieved by optimizing the parsing operations.

QoS features:

Taking into account the ENCOURAGE project, it would be interesting to make some efforts to add Quality of Service features to the ENCOURAGE communication architecture, seeking a reliable and stable architecture and more predictability on communication delays.

Arrowhead Project

Performance tests:

Regarding the Arrowhead project, since no performance tests were possible, it would be great to perform some tests in a simulated and local scenario, or within some real scenarios. This could be important to prove the performance of the system, its reliability and the concept of the Service-Oriented Architecture for embedded systems. It is important to note that this implementation is planned to be tested by the end of the project timeline.

Orchestration service:

The Orchestration Core Service in the Arrowhead Project is not fully used in the current implementation, and it would be important to harvest fully the advantages of the Orchestration service, for example to enable a Flex-Offer Agent to request information about the best configuration in terms of which Aggregator it should communicate, e.g. taking into account the geographical area and type of load. Nevertheless, it is important to note that at the time of writing for this thesis the Orchestration service was not fully developed.

Core services:

In the Arrowhead Project, besides the three described Core Services (Service Registry, Authorisation, Authentication and Accounting Service, and Orchestration Service), other Core Services were proposed, such as the Event Handler System and the QoS Service. The main goal of the Event Handler System is to provide a service in which it is possible to register all kinds of events that may happen within an Arrowhead Framework. The QoS Service might be helpful to establish a set of configurations to guarantee robust and reliable communications among all systems connected to the framework.

QoS features:

Taking into account the Arrowhead project, it would be interesting to make some efforts to add some Quality of Service features to the established communications, seeking a reliable and stable architecture and communications, trying to improve the predictability on communication delays.

7 Bibliografia

[1] G. Colouris, J. Dollimore e G. Blair, "Distributed Systems – Concept and Design 5th Edition", Morgan Kaufmann Publishers, pp. 1-33, 2012.

[2] N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann, 1996.

[3] A. Tanenbaum e M. Van Steen, “Distributed Systems: Principles and Paradigms, 2 Edition,” Prentice Hall, 2007, pp. 2-66.

[4] Carnot Institutes’ Information Communication Technologies and Micro Nano Technologies, “Smart networked objects and internet of things,” 2010.

[5] L. Atzori, A. Iera e G. Morabito, “The Internet of Things: A survey,” *Computer Networks,* vol. 54, nº 15, pp. 2787-2805, 2010.

[6] G. Kortuem, F. Kawsar, D. Fitton e V. Sundramoorthy, “Smart objects as building blocks for the Internet of things,” *Internet Computing, IEEE,* vol. 14, nº 1, pp. 44 - 51, 2009.

[7] D. Le-Phuoc, A. Polleres, M. Hauswirth, G. Tummarello e C. Morbidoni, “Rapid prototyping of semantic mash-ups through semantic web pipes,” *WWW '09 Proceedings of the 18th international conference on World wide web,* pp. 581-590, 2009.

[8] D. I. Wolinsky, A. Agrawa, P. O. Boykin, R. J. Davis, A. Ganguly, V. Paramygin, Y. P. Sheng e R. J. Figueiredo, “Design of Virtual Machine Sandboxes for Distributed Computing in Wide-area Overlays of Virtual Workstations,” p. 8, 2006.

[9] R. Figueiredo, P. Dinda e J. Fortes, “A case for grid computing on virtual machines”, Distributed Computing Systems, 2003. Proceedings. 23rd International Conference, pp. 550 - 559, 2003.

[10] D. Marshall, “Top 10 Benefits of Server Virtualization,” 2 November 2011. [Online]. Available: http://www.infoworld.com/article/2621446/server-virtualization/top-10-benefits-of-server-virtualization.html. [Accessed on 28 January 2015].

[11] B. Lampson, M. Abadi, M. Burrows e E. Wobber, “Authentication in Distributed Systems: Theory and Practice,” ACM Transactions on Computer Systems, vol. 25, pp. 165-182, 1992.

[12] P. Artigas e M. Ferdman, “Centralized vs Decentralized: Allocation in Distributed Systems,” 2000.

[13] G. M. Weiss e J. W. Lockhart, “A Comparison of Alternative Client/Server Architectures for Ubiquitous Mobile Sensor-Based Applications,” UbiComp '12 Proceedings of the 2012 ACM Conference on Ubiquitous Computing, pp. 721-724, 2012.

[14] Z. F. Chen, X. M. Zhang, Q. C. Chen, T. F. Ma, H. Wang e X. Feng, “Design and Implementation of an Elementary School Online Registration and Enrollment Position Allocation System Based on Three-Tier Architecture,” *Applied Mechanics and Materials,* pp. 1843-1848, August 2013.

[15] B. Xu e C. Lin, “An extended practical three-tier architecture based on middleware”, Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference, pp. 243 - 246, 2013.

[16] R. Nandakumar, K. K. Chintalapudi, V. Padmanabhan e R. Venkatesan, “Dhwani: Secure Peer-to-Peer Acoustic NFC”, SIGCOMM '13 Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, vol. 43, pp. 63-74, 2013.

[17] T. Erl, SOA: Principles of Service Design, Prentice Hall, 2007.

[18] C. Wu, B. Li e S. Zhao, “Characterizing Peer-to-Peer Streaming Flows,” *IEEE Journal on Selected Areas in Communications,* vol. 25, pp. 1612-1626, 2007.

[19] A. Yahyavi e B. Kemme, “Peer-to-peer architectures for massively multiplayer online games: A Survey,” *ACM Computing Surveys (CSUR),* vol. 46, nº 1, 2013.

[20] Y. Yusufa, A. Gunasekaranb e M. S. Abthorpec, “Enterprise information systems project implementation: A case study of ERP in Rolls-Royce,” *Internation Journal of Production Economics,* vol. 87, nº 3, p. 251–266, 2004.

[21] S. R. Magal e J. Word, Integrated Business Processes with ERP Systems, Wiley, 2012.

[22] “SAP ERP,” SAP, [Online]. Available: http://www.sap.com/portugal/pc/bp/erp.html. [Accessed on 5 February 2015].

[23] “SAP AG,” SAP, [Online]. Available: http://go.sap.com/index.html. [Accessed on 5 February 2015].

[24] NetSuite, “NetSuite ERP,” NetSuite Inc., 2015. [Online]. Available: http://www.netsuite.com/portal/products/netsuite/erp.shtml. [Accessed on 5 February 2015].

[25] Microsoft, “Microsoft Dynamics GP,” Microsoft, 2015. [Online]. Available: http://www.microsoft.com/en-us/dynamics/erp-gp-overview.aspx. [Accessed on 5 February 2015].

[26] “EverQuest Online,” Sony Online Entertainment, 989 Studios, [Online]. Available: https://www.everquest.com/home. [Accessed on 12 February 2015].

[27] “EVE Online,” CCP Games, 2015. [Online]. Available: https://www.eveonline.com/. [Accessed on 12 February 2015].

[28] “A Journey Into MMO Server Architecture,” Nexeon Technologies, Inc., 30 May 2013. [Online]. Available: http://www.mmorpg.com/blogs/FaceOfMankind/052013/25185_A-Journey-Into-MMO-Server-Architecture. [Accessed on 13 February 2015].

[29] S. M. Kaplan, “Smart Grid. Electrical Power Transmission: Background and Policy Issues,” 2009.

[30] National Institute of Standards and Technology, “NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release 3.0,” 2014.

[31] United States Department of Energy, “Smart Grid,” [Online]. Available: http://energy.gov/oe/services/technology-development/smart-grid.

[32] M. Albano, L. L. Ferreira, L. M. Pinho e A. R. Alkhawaja, “Message-oriented middleware for smart grids,” *Computer Standards & Interfaces,* vol. 38, pp. 133-143, 2015.

[33] V. Gungor, “A Survey on Smart Grid Potential Applications and Communication Requirements,” *IEEE Trans. on Industrial Informatics,* vol. 9, nº 1, pp. 28-42, 2013.

[34] F. Kennel, D. Gorges e S. Liu, “Energy Management for Smart Grids With Electric Vehicles Based on Hierarchical MPC,” *IEEE Trans. on Industrial Informatics,* vol. 9, nº 3, pp. 1528-1537, 2013.

[35] “Smart Grid Project,” Provincial Electricity Authority (PEA), 2014. [Online]. Available: http://www.powergenasia.com/conference/smartmeter.html. [Accessed on 2015].

[36] A. Bari, J. Jiang, W. Saas e A. Jaekel, “Challenges in the Smart Grid Applications: An Overview,” *International Journal of Distributed Sensor Networks,* p. 11, 2014.

[37] M. Albano, L. L. Ferreira e L. M. Pinho, “Convergence of Smart Grid ICT architectures for the last mile,” *Industrial Informatics, IEEE Transactions,* vol. 11, nº 1, pp. 187-197, 2015.

[38] C. Teixeira, M. Albano, A. Skou, L. P. Dueñas, F. Antonacci, R. Ferreira, K. L. Pedersen e S. Scalari, “Convergence to the European Energy Policy in European countries: case studies and comparison,” *Social Technologies Resarch Journal,* vol. 4, nº 1, pp. 7-18, 2014.

[39] “IEEE Standards Association,” IEEE, 2015. [Online]. Available: http://standards.ieee.org/.

[40] S. Browna, D. Pykea e P. Steenhof, “Electric vehicles: The role and importance of standards in an emerging market,” *Energy Policy,* vol. 38, nº 7, pp. 3797-3806, 2010.

[41] “Common Information Model Standard,” Distributed Management Task Force, Inc., 2015. [Online]. Available: http://www.dmtf.org/standards/cim. [Accessed on February 2015].

[42] J.J.Simmins, “The impact of PAP 8 on the Common Information Model (CIM),” Power Systems Conference and Exposition (PSCE), 2011 IEEE/PES, 2011.

[43] Distributed Management Task Force, Inc., “Common Information Model (CIM) Infrastructure Specification,” Distributed Management Task Force, Inc. (DMTF), 2005.

[44] Distributed Management Task Force, Inc., “Common Information Model (CIM) Schemas,” Distributed Management Task Force, Inc. (DMTF), 2015.

[45] “IEC Technical Committee,” IEC TC 57, 2015. [Online]. Available: http://tc57.iec.ch/index-tc57.html.

[46] “Core IEC Standards,” International Electrotechnical Commission, 2015. [Online]. Available: http://www.iec.ch/smartgrid/standards/. [Accessed on February 2015].

[47] R. G. Hollands, “Will the real smart city please stand up?,” *City: analysis of urban trends, culture, theory, policy, action,* vol. 12, nº 3, pp. 303-320, 2008.

[48] A. Caragliu, C. D. Bo e P. NijKamp, “Smart cities in Europe,” 3rd Central European Conference in Regional Science – CERS, 2009.

[49] Department for Business Innovation & Skills, “Smart Cities - Background Paper,” United Kingdom Government, 2013.

[50] M. Barr e A. Massa, Programming Embedded Systems: With C and GNU Development Tools, O'Reilly, 2007.

[51] S. Heath, "Embedded Systems Design", Newnes, 2003, pp. 1-30.

[52] EdgeFX, “Understanding of Embedded Systems,” EdgeFX Kits & Solutions, 2015. [Online]. Available: http://www.edgefxkits.com/blog/embedded-systems-with-applications/.

[53] H. Koptez, "Real-Time Systems: Design Principles for Distributed Embedded Applications", Springer, 2011.

[54] M. A. Vouk, “Cloud computing — Issues, research and implementations,” Information Technology Interfaces, 2008.

[55] Microsoft, “Chapter 1: Service Oriented Architecture (SOA),” Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb833022.aspx. [Accessed on February 2015].

[56] A. L. Diaz, “Lessons Learned: Business Agility through Open Standards & Cloud,” [Online]. Available: http://www.slideshare.net/angelluisdiaz/up2011-diazibm-cloudstandards.

[57] “Dropbox Official Website,” [Online]. Available: https://www.dropbox.com. [Accessed on April 2015].

[58] “Box Official Website,” [Online]. Available: https://www.box.com. [Accessed on April 2015].

[59] S. Ghosh, Distributed Systems: An Algorithmic Approach, Second Edition, CRC Press, 2014.

[60] L. Harte, "Introduction to Data Multicasting", Althos Publishing, 2008.

[61] K. Geihs, “Middleware Challenges Ahead,” pp. 24-31, 2001.

[62] E. Curry, “Message-oriented Middleware,” pp. 1-35, 2004.

[63] IBM, “IBM Websphere Message Broker,” IBM, [Online]. Available: http://www.ibm.com/software/integration/wbimessagebroker.

[64] “Simple (or Streaming) Text Oriented Messaging Protocol (STOMP),” [Online]. Available: https://stomp.github.io/.

[65] “Message Queue Telemetry Transport (MQTT),” [Online]. Available: http://mqtt.org/.

[66] “RabbitMQ – AMQP Concepts,” [Online]. Available: http://www.rabbitmq.com/tutorials/amqp-concepts.html.

[67] M. K. Duncan e C. , “AMQP 0.9.1 Model Explained,” [Online]. Available: http://www.rubydoc.info/github/ruby-amqp/amqp/master/file/docs/AMQP091ModelExplained.textile.

[68] A. Videla e J. J. W. Williams, "RabbitMQ in Action: Distributed messaging for everyone", Manning, 2012.

[69] “Erlang Official Website,” [Online]. Available: http://www.erlang.org/.

[70] “Extensible Messaging and Presence Protocol (XMPP) Official Website,” [Online]. Available: http://xmpp.org/.

[71] P. Saint-Andre, “Extensible Messaging and Presence Protocol (XMPP): Core,” [Online]. Available: http://www.ietf.org/rfc/rfc3920.txt.

[72] “International Engineering Task Force (IEFT) Official Website,” [Online]. Available: http://ieft.net/.

[73] P. Saint-Andre, K. Smith e R. Tronçon, "XMPP: The Definitive Guide - Building Real-Time Applications with Jabber Technologies", O'Reilly, 2009.

[74] J. Moffitt, Professional XMPP Programming with JavaScript and jQuery, Wrox, 2010.

[75] “XEP-0045: Multi-User Chat Specification,” [Online]. Available: http://xmpp.org/extensions/xep-0045.html.

[76] “XEP-0060: Publish-Subscribe Specification,” [Online]. Available: http://www.xmpp.org/extensions/xep-0060.html.

[77] “XMPP Extensions (XEPs) Official Website,” [Online]. Available: http://xmpp.org/xmpp-protocols/xmpp-extensions/.

[78] “XEP-0332: HTTP over XMPP transport,” [Online]. Available: http://xmpp.org/extensions/xep-0332.html.

[79] “XEP-0114: Jabber Component Protocol,” [Online]. Available: http://xmpp.org/extensions/xep-0114.html.

[80] “Ejabberd Official Website,” [Online]. Available: https://www.ejabberd.im. [Accessed on April 2015].

[81] “Ignite Realtime: Openfire Official Website,” [Online]. Available: http://www.igniterealtime.org/projects/openfire/.

[82] “Data Distribution Service for Real-Time Systems Specification,” Object Management Group, Inc. (OMG), 2005.

[83] J. Sanchez-Monederoa, J. Povedano-Molinab, J. M. Lopez-Vegab e J. M. Lopez-Solerb, “Bloom filter-based discovery protocol for DDS middleware,” *Journal of Parallel and Distributed Computing,* vol. 71, nº 10, pp. 1305-1317, 2011.

[84] L. L. Ferreira, L. M. Pinho, M. Albano e C. Teixeira, “Adaptive offloading for infotainment systems,” *SIGBED Review, ACM.,* vol. Volume 12, nº Issue 3, pp. pp 19-23, 2015.

[85] L. L. Ferreira, L. Siksnys, P. Pedersen, P. Stluka, C. Chrysoulas, T. Le Guilly, M. Albano, A. Skou, C. Teixeira e T. Pedersen, “Arrowhead Compliant Virtual Market of Energy,” *9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014). 19 to 21, Sep, 2014, Flexible And Interoperable Automation Systems, Barcelona, Spain,* 2014.