POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Consistent management of dynamic policies in distributed IoT applications

Supervisors
Prof. dr. Fulvio Corno
Prof. dr. ir. Wouter Joosen
Dr. Bert Lagaisse
Dimitri Jonckers

Candidate
Susanna Aita

Academic Year 2018-2019

Ai miei genitori e
alle mie nonne

Summary

The Internet of Things (IoT) allows to introduce in the internet world billions of new devices. Together with new exciting functionalities, IoT leads to new management challenges. We are in the context of a smart building, in which the principal services offered, like access control for areas, climate control and fire detection, are controlled by IoT applications. These heterogeneous devices must be interconnected and must collaborate properly in order to provide services and support unexpected events or threats.
One of the main goals of this thesis is to achieve a mechanism that controls the system behaviour through policies. But mostly, we research how to guarantee dynamic and consistent adaptation of these policies throughout a distributed system. In this scenario, it is important to provide the possibility to change the system behaviour, in order to tackle different situations that can occur during the whole lifetime of a building. The consistency property is equally fundamental for a correct functioning of all the operations. Our challenge refers indeed to these problems: nowadays, the behaviours of the IoT devices are mainly defined upfront and the possibility to change them dynamically and, furthermore, in a consistent way, has not been well investigated so far.
After an introduction of the IoT technology, we describe the context of our work, the problem we aimed at solving and our goals. The core contribution of this thesis is the development of a flexible middleware, running on the different nodes of the system, able to handle dynamic policies in consistent way. In fact, our system is composed by nodes, divided in multiple tiers: the gateways, situated on different floors of a smart building, connected to some sensors, and a server that handles all the communications. In addition, the server governs the gateways behaviour triggering the transfer of new policies. At the gateway side these policies must be correctly managed at run-time. Our developed protocol guarantees that all the procedures are performed consistently without the need of restarting the processes. In this context, among all the possible use cases, we mainly focus on the encryption and decryption one. A node is in charge of the encryption of a message, while another one is responsible for the decryption. The server can trigger the update of a policy that contains a new encryption algorithm. Both of these nodes must update the policy consistently in order to exchange the message accurately. If something goes wrong, the system is not able to retrieve and process the message properly.
The validation tests illustrate that the temporary inconsistent situations that can occur are correctly managed. The performance tests show that our developed middleware has a limited impact in time, memory and on the network.

Acknowledgements

This thesis has been accomplished under the supervision of my promoters Prof. dr. ir. Wouter Joosen and Dr. Bert Lagaisse and my mentor Dimitri Jonckers at Katholieke Universiteit of Leuven. The most heartfelt thanks go to them that have supported me during all my thesis work, with willingness and cordiality, since my first arrival in Belgium. Thanks to their support I have been able to complete this thesis at my best.
I would sincerely like to thank my promoter at Politecnico di Torino Prof. Fulvio Corno for his help and assistance from Turin throughout the year.

Contents

1 Introduction 1

1.1 The Internet of Things paradigm . . . 1

1.2 Challenges and examples . . . 3

1.2.1 Applications domains . . . 3

1.2.2 Smart building management . . . 4

1.3 Policy based management . . . 6

1.4 The challenges of distributed policy management . . . 7

1.5 Goal . . . 7

1.6 Contribution . . . 8

1.7 Approach and results . . . 8

1.8 Structure of this thesis . . . 9

2 Background knowledge and related work 10

2.1 Enabling Technologies . . . 10

2.2 Policy enforcement approach . . . 11

2.2.1 Characterization of the policies . . . 11

2.2.2 Policy Systems: rule engines . . . 12

2.2.3 A Business Rules Management System: Drools . . . 13

2.3 VersaSense Micro Plug-and-Play (MicroPnP) . . . 14

2.4 Related work . . . 15

2.4.1 Significant technologies and techniques . . . 15

2.4.2 Dynamic Change Management: quiescence . . . 16

2.4.3 A low disruptive alternative: tranquillity . . . 17

2.4.4 CaPI: a component and policy-based approach . . . 18

2.4.5 Dynamic and selective combination of extensions in component-based applications . . . 19

2.4.6 Comparison and comments . . . 20

3 Problem elaboration, analysis and requirements 22
3.1 Use cases . . . 22
3.1.1 Access control . . . 22
3.1.2 Climate and light control . . . 23
3.1.3 Air quality, fire detection and evacuation . . . 23
3.1.4 Encryption and Decryption functionality . . . 24
3.2 Problem analysis and context diagram . . . 24
3.3 Goals and metrics . . . 26
3.4 Requirements analysis . . . 27
3.4.1 Functional requirements . . . 27
3.4.2 Non-functional requirements . . . 28

4 Dynamic and consistent policy updates 29
4.1 Architecture and design overview . . . 29
4.2 Java RMI communication protocol . . . 30
4.3 Distributed encryption/decryption protocol: a challenging consistency example . . . 32
4.3.1 Design of the standard protocol . . . 33
4.3.2 Design of the updating protocol . . . 34
4.3.3 Multiple senders and multiple receivers . . . 35
4.4 Management protocol for stateful policies . . . 36

5 Proof of Concept (PoC) and implementation 38
5.1 PoC structure overview . . . 38
5.2 PoC development . . . 39
5.2.1 Versasense's Micro Plug-and Play for the gateway implementation . . . 41
5.2.2 The procedure for a dynamic update of policies . . . 41
5.2.3 Distributed and consistent protocol implementation . . . 42
5.2.4 Implementation of stateful policy . . . 45

6 Validation 47
6.1 Effective dynamic update of policies . . . 47
6.2 Achievement of consistent updates . . . 47
6.3 Validation tests . . . 49
6.3.1 Standard scenario . . . 50

6.3.2 First problematic scenario: delay on the sender 51
6.3.3 Second problematic scenario: delay of an update at receiver side 52
6.3.4 Third problematic scenario: delay on message transmission 53

7 Evaluation and discussion 54
7.1 Failure model 54
7.2 Time overhead 55
7.3 Memory and network overhead 63
7.4 Limitations and future work 64

8 Conclusion 65

Bibliography 67

Chapter 1

Introduction

1.1 The Internet of Things paradigm

The Internet of Things (IoT) is an innovative paradigm that is radically changing the human society and life. The main idea of IoT is the introduction around us of heterogeneous devices (*things*) connected to a network. It provides worldwide access to information, interaction with different machines and with different stakeholders and cooperation in order to offer innovative and revolutionary services.

In this perspective, a new notion of interconnected and *smart* objects with computational capacity is replacing the conventional concept of the internet as infrastructure network that connects the end-users' terminals.
Therefore, the term *Internet of Things* includes: the global network used for the interconnection of smart objects, the set of technologies essential to achieve this purpose (e.g. sensors, actuators, cloud services, mobile phones) and the new services and applications that allow new opportunities in the most different fields that leverage these technologies [1].

The introduction of the IoT has an unquestionably impact in domestic and professional life, as in the private and public fields. Some examples of application scenarios could be domotics, e-health, automation, industrial manufacturing and intelligent transportation of people and goods.[2]
From 2008, there were already more devices connected to the Internet than people on Earth and some studies predict that by 2020 this number will grow beyond 50 billion, as shown in Figure 1.1 [3][4].

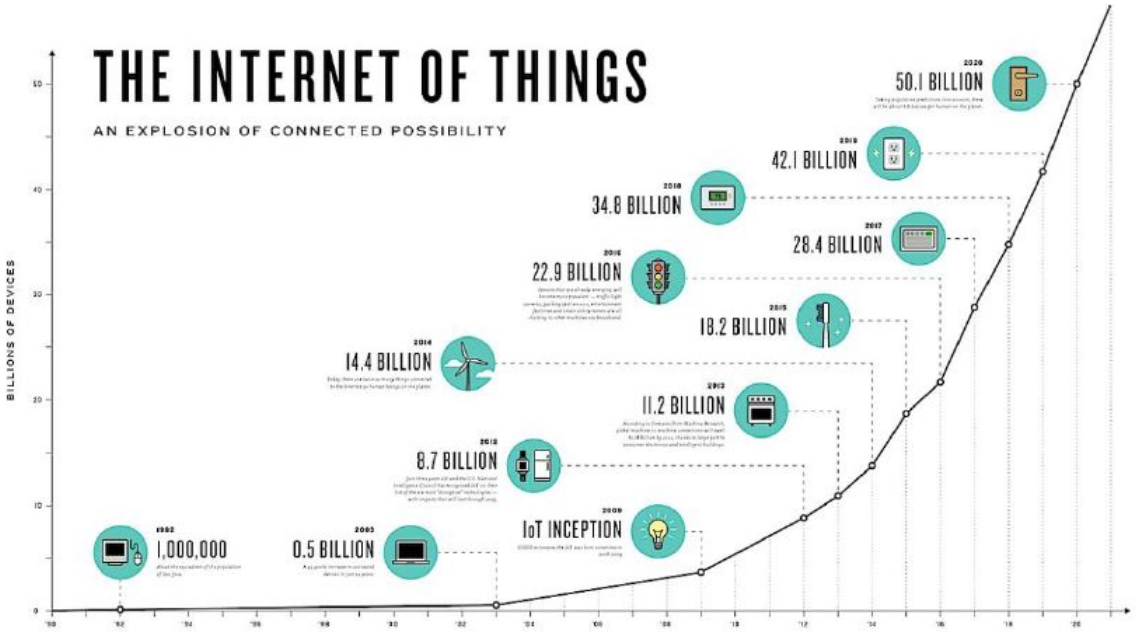

Figure 1.1: The growth of Internet of Things devices (Brandon, 2015)

In this context, it is not surprising that new risks, issues and concerns are arising together with these new exciting services.

The most related one to this thesis and so the main focus is the management challenge that such high number of interconnected devices arise. A flexible middleware infrastructure is needed in order to manage technical concerns in a dynamic and distributed environment. Self-healing and adaptation and are very important characteristics in order to tackle ordinary and unexpected events which are very likely in this context.

Due to the high level of devices' heterogeneity, the traditional communication schema, security countermeasures and privacy enforcement are not suitable for IoT devices, also because of their limitation in computing power, memory, energy and speed.

Another important issue is the lack of standardization and full deployment of the IoT paradigm. There are various integration efforts but they are not standardized in a complete framework. More and more devices are becoming interconnected with different communication stacks and different complexity. Hence, it is becoming complicated to manage trusted communication, connection or the authentication of system access [2].

In order to give a description of the IoT paradigm from different points of view, it is woth mentioning the interesting article in [3] that analyses briefly the subject of IoT from the technological, social, economic and cultural points of view:

Technological point of view. By making use of the huge amount of data, or *Big Data*, collected by sensors, IoT can be seen as a global network that can connect

virtual and real devices. The article mentions some important characteristics of these new technologies like omnipresence, ultra-connectivity and difficult control over this massive quantity of objects in the network.

Social and economic point of view. With the introduction of smart cities, smart industries and assisted living, IoT is generating an eco-system denoted as Smart Life. Together with new exciting services and with an inevitable change in all organizations' environment, this introduction of smart applications in our lives leads to new risks in security and in ensuring private life. The IoT is seen by most of European governments as an important factor of growth and economic innovation that must be standardized in order to improve services, security and data protection.

Cultural point of view. Some studies rise the problem of the secondary effect of internet usage, such as becoming too dependent, isolated, losing concentration abilities, imagination and cognitive skills.

We are going towards the internet of the future that will be considerably different from the internet we are used to. It moves the interactions between devices and people at a virtual level in both professional and social life.

As mentioned before, this thesis mainly deals with the distributed aspect of the IoT that allows to leverage on the various functionalities of heterogeneous connected devices, to spread the workload of the system in different machines and to control the whole architecture at different levels. In this scenario issues related to how to send, store, represent, search, interconnect and organize information generated by the IoT devices become very challenging. Therefore, a middleware between the technological level and the applications plays a key role in the correct management of the infrastructure.

1.2 Challenges and examples

The actual goal of this thesis is to examine the possibility to employ dynamic policies in the management of the IoT systems. The basic idea of the IoT is to enable enhanced environments like buildings, campuses or cities, sharing resources and creating new exciting services.
It is made possible by the interaction among heterogeneous devices that cooperate to reach common goals.

1.2.1 Applications domains

Thanks to its capability to connect disparate objects of our daily lives, the IoT has introduced several applications and smart services. Some of them are [5],[4]:

1. **Smart cities**: nowadays it is possible to automate and manage a whole city via internet, such as controlling traffic signals, monitoring pollution levels and seismic vibration buildings, tracking vehicle parking, surveillance of public areas and waste management.

2. **Domestic applications**: currently people can control many of the house tasks without the need of being near the devices. For example, controlling and adapting the consumption of electricity or water, monitoring details like room temperature, noise, oxygen or dust levels or turning ON/OFF devices using a smartphone.

3. **Connected industry**: industrial automation is made possible by the introduction of IoT. Nowadays it is easier to monitor energy consumption and storage conditions along the supply chain. A smart product management system can track products for traceability aims, control the rotation of products in the shelves and control the warehouses. In the agricultural sector IoT can, for example, improve the consumption of water by select the right level of irrigation.

4. **Logistics and Transportation**: new technologies arise in this field such as Machine to Machine (M2M) that allows the communication between cars in order to reduce the possibilities of incidents. It also provides the technology to keep track of the public transport system, in order to control, for example, the densities of pedestrian and traffic and to identify damages in the road.

In this thesis we focus on the domestic applications but in the broad scenario of a smart building and its management. The following section will introduce it in detail.

1.2.2 Smart building management

In particular we are in the context of a smart building in which new and existing devices are integrated into the IoT platform.
This platform, shown in Figure 1.2 taken from the paper in [6], has a typical IoT architecture with devices like temperature sensors and controllers for access, window, elevator and fire detector all connected to the gateways. These gateways are connected to a local service back-end, which manages one deployment domain (one building) and the interfaces for connecting with stakeholders and external systems. Applications are instantiated and activated in the local service back-end and in the involved gateways. For example, the Figure 1.2 below shows that an app instance of the physical access control application is running at both gateways. GW1 can govern access to a door with badge reader, while GW2 has a door with a numeric keypad for users to authenticate.
Different stakeholders utilise the services offered by the local service back-end: building maintenance managers, platform administrators and building security managers.

The integration between devices and subsystems and the possibility to unify the access to them through the IoT system allows users to share resources between applications and tenants. Furthermore this communication results in efficiency gains such as discover the optimal climate control schedule, the optimal energy usage and detect alarmed or emergency situations thanks to the data collected at different points.

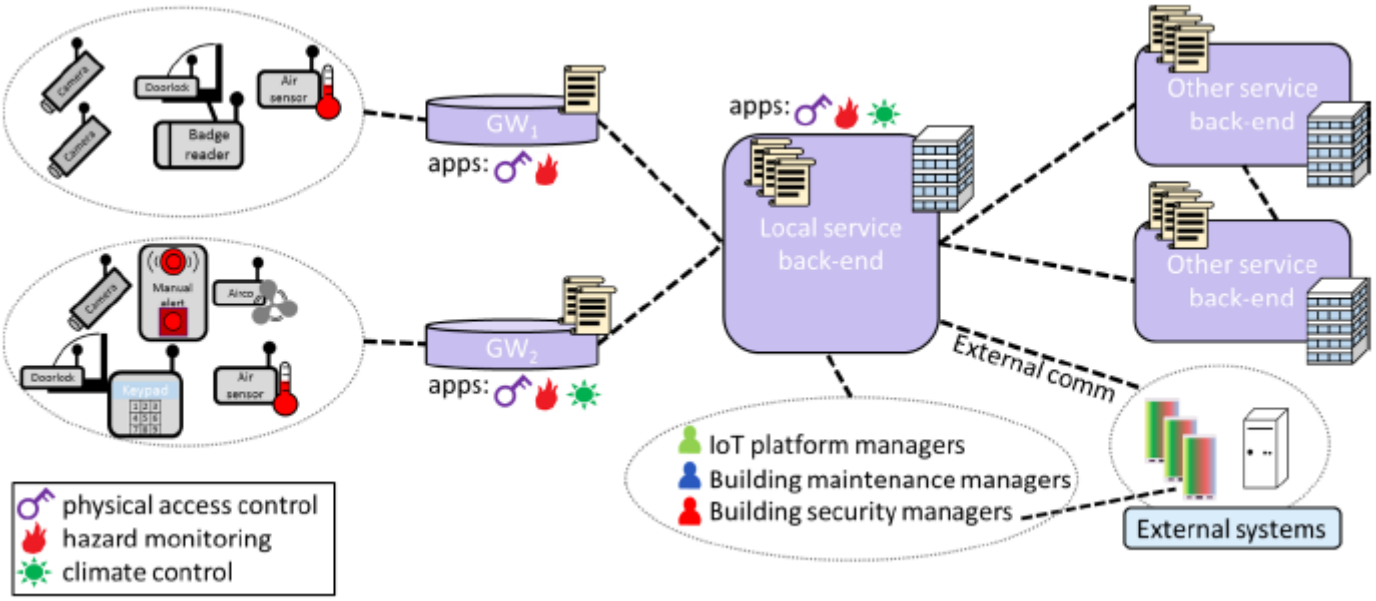

Figure 1.2: Architectural view of the IoT platform

As shown in Figure 1.2 the smart building is a large Cyber-Physical System (CPS) composed by the integration of systems such as lighting, heating, ventilation, air-conditioning (HVAC), access control, e.g. locks and motion detection, IT network and IT infrastructure devices and so on.
It is worth mentioning some use cases correlated to a smart building:

- (Physical) Access control: it manages access to private areas in the building. It controls locks and monitor presence in the various locations. For this purpose, in the mentioned Figure we could see a badge reader connected to a door-lock and some cameras, managed by GW1 and keypad that allows the entrance connected by GW2.
- Climate and light control: it regulates the temperature in the rooms and monitors the level of comfort. Based on the measurement of the light it can establish the right level without wasting energy. In the Figure, connected to GW2, is present air-conditioning sensors linked to a fan.
- Air quality, fire detection and evacuation: it can analyse the characteristics of the air condition and can raise an alarm in case of dangerous results found. Air sensors are connected to both gateway in the Figure.
- Encryption and decryption functionality: due to the huge amount of data exchange an application able to correctly encrypt and decrypt all these message is required in order to accomplish the security requirements.

In this work, we will focus mainly on the management of dynamic policies for IoT systems. All the mentioned functionalities offered by an IoT platform can be governed by policies and these must be implemented and enforced on various

devices and with different granularities.
In particular we are interested in implementing dynamic policies in a distributed IoT system and in handling them in a consistent and coherent way throughout the platform.

We have mentioned several times the term policy without gave an accurate description of it. Therefore, the following Section 1.3 gives a general overview and includes some example of the use of policies in this context.

1.3 Policy based management

Policy-based management is, in general, a technology that can simplify the handling of networks and distributed systems. In this thesis, we are more focus in this latter task. Thanks to this approach, also managers and directors, not expert in IT, can manage the different assets of a distributed system in a uncomplicated and flexible way by deploying and enforcing a set of policies that rule their behaviour. Policies are collection of rules that are independent from the technology below. They aim at enhancing the hard-coded functionality of the systems without modifying the underlying implementation. This allows to change the systems actions without the need to interrupt the running operations. [7],[8],[9].

Generally, the policies can be categorized into three types: pure event driven {Event-Condition-Action ECA}, periodic check {[E]CA} and obligation or imposed decision {[EC]A}. In this thesis, we are mainly dealing with the first type.
We add below some examples of security and platform management policies, feasible in our smart building context:

- Access control fire doors: "Only the hazard monitoring application is allowed to send commands to the fire doors."

```
EVENT: Command to fire door device
CONDITION: Source command = hazard monitoring app
ACTION: Allow command (grant)
```

- Access control cameras: "No application other than the access control application is allowed to send commands to a camera that is being used by the access control application."

```
EVENT: Command to camera device D
CONDITION: Source command different from access
   control app
ACTION: Block command
```

- Monitoring of hazard monitoring application devices: "If any IoT device used by the hazard monitoring app has not been heard of for 5 minutes, log the event and raise a low-priority alert for the building maintenance manager."

```
EVENT: No heartbeat of device_x used by hazard
   monitoring application in last 5 minutes
CONDITION: No planned outage
ACTION: Log event AND High-priority unavailability
   alert to local back-end
```

In order to handle and execute business rules at run-time within a program environment, we can leverage on different software called Policy System that has been developed. In particular, we use Drools that processes data, facts and rules and produces results using a rule engine.

1.4 The challenges of distributed policy management

The opportunities enabled by IoT, together with new exciting services, cause new management challenges.

This thesis will examine how to express the new requirements as policies and how to enforce them in a distributed and dynamic IoT context, by using RedHat Drools, a general and non ad-hoc Policy System.
In the IoT context the possibility to implement dynamic and event-based policies has not been well investigated. The components' behaviour is mainly defined upfront, hard-coded in the architecture and allows limited changes.

Furthermore, the consistent dynamic updates of the policies over the distributed system is still a challenge. In this scenario of dynamic changes, it is fundamental to provide a protocol that guarantees consistency, coherency and robustness among the nodes of the system, before, during and after every run-time changes. For example, referring to the use cases in Subsection 1.2.2 a lack of consistency in the access control policies can cause a free entrance in locations that must be protected. An inconsistent management of evacuation policy can cause panic and misleading situations. As another example, in a scenario that involves the encryption and decryption of messages, a lack of consistency and robustness can lead to an incorrect retrieval of information that would cause unexpected results.

1.5 Goal

The key goal of this thesis is to research the feasibility of supporting dynamic policies, made by a set of rules, and to develop an event-based middleware platform to support these distributed and dynamic policies in an IoT systems.
The main challenging goal is to achieve consistent distributed adaptation of dynamic policies. Therefore we divide the middleware in two key subsystems: a

functional and a management level. The first is a distributed level that guarantees the communication between the different nodes, the second one is in charge of the correct distribution of the policies, it is responsible for deploying updates of the policy sets over the distributed nodes, it should replace policies consistently on multiple machines and handle the queue of events when changes are required.
We will perform measurements about the performances, about the computational load and about the time requested to carry out all the phases involved in the dynamic update of the policies.
We will investigate if the already existing policy engines like Drools are suitable for this task and their advantages compared to hard-coded rules.

1.6 Contribution

The main contributions of this thesis is to support dynamic policies in a distributed IoT architecture giving the possibility to change them in case of need. Moreover, we guarantee consistency through all the nodes of the system during and after this phase.
We researched and managed to build an event-based middleware platform able to deal with the different IoT services and applications through the use of policies and their dynamic updates. We succeed in developing a protocol that is:

1. Fast
2. Consistent through all the components of the IoT system
3. Robust
4. Well-performing
5. Valuable in an IoT systems

1.7 Approach and results

As better described in Section 2.4, we took inspiration from already investigated solutions, to come up with a protocol for an IoT system that performs consistent and fast update of policies.
In particular the notion of *tranquillity* in [10], approved to be a sufficient condition for updatability, is the closest to our implementation. During the upload of a new policy the devices become passive and cannot be engaged in a transaction, initiate a new one and process any requests.

Therefore, the main results of this thesis are:

- Achieve the usage of dynamic policies in an IoT context.
- Distribute and change them in a consistent way through all the nodes.

- Propose a design for the overall architecture that consists in multiple and different devices in which the qualities listed above are guaranteed.
- Validate and evaluate our achievements in a Proof of Concept.

Together with the main contributions we achieve some other results that are in the scope of this thesis.

- Literature review focused on dynamic and consistent update of system.
- Adapting these mechanism to the IoT context.
- Description of the use cases for a smart building setting that requires the usage and update of policies.
- Description and actuation of Drools as Policy System in the different devices.
- Research the use of Java RMI as a communication protocol

We validate our contributions and results in the encryption/decryption use case described in Section 3.1 and explained in details in Section 4.3.
It involves the encryption of messages at one side of the system and the following decryption at another side. A policy defines which algorithm they must use for performing the actions. Since the two sides must necessarily agree with the algorithm used, this example, better that any other, highlights if the consistent property is guaranteed: if they are not aligned with the use of the same algorithm the protocol fails in correctly retrieving the messages.

1.8 Structure of this thesis

This first Chapter of the thesis have presented an introduction of the IoT technology followed by the context of our work. Furthermore, we have described the problem we aim at solving, our goals and contributions.

The rest of this thesis is structured as follows: In Chapter 2 we introduce the background knowledge, enabling technologies and related work that have contributed to the achievement of our goals. In Chapter 3 we give an overview of the use cases that allow to better explain the challenges we want to achieve. Furthermore, we perform a more detailed analysis of the problem, of the goal and on the requirements, considering the background knowledge presented in the previous Chapter. The Chapter 4 presents our solution for the challenges defined before. In Chapter 5 we describe our implementation for the Proof on Concept (Poc). The Chapter 6 and the Chapter 7 present respectively the validation and evaluation test of our PoC and finally, in Chapter 8 we draw the conclusion of our accomplishments.

Chapter 2

Background knowledge and related work

In this Chapter we present the knowledge required to better understand the problem we want to solve, the goal we aim at achieving and our accomplished work.
In Section 2.1 we describe the current technologies that enable the Internet of Things paradigm. In Section 2.2 we introduce the policy enforcement approach and the policy systems available, in particular the one that we employ: Drools. In Section 2.3 we present the device used for developing our prototype as an example of a real IoT platform. Finally, in Section 2.4 we discuss about and compare four papers from which we find inspiration for the development of the consistent protocol among multiple nodes.

2.1 Enabling Technologies

The term *Internet of Things* was originally first presented in 1999 by Kevin Ashton, the founder of MIT auto-identification centre. Ashton has claimed:
"The Internet of Things has the potential to change the world, just as the internet did. Maybe even more so." [4].
We have already empathized the huge and increasing number of devices connected to the internet and in addition, with the involving of software, hardware, installation costs and management services, IoT becomes the most significant device market that will add $1.7 trillion in value to the global economy in 2019 [4].
In 2012 IoT was defined by ITU as: *"a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on, existing and evolving, interoperable information and communication technologies"* [11].

Nowadays, the technological development allows to include in the IoT paradigm the comprehensive infrastructure that interconnect physical and virtual devices, such as Wireless Sensors Networks (WSNs) and Cloud Computing [4].
Communication protocols and wireless mediums have an essential function in the IoT system. They are the enabling technologies that allow the various devices to

exchange their data over the network and most of them represent a low cost and simple deployment solution for IoT.

It is worth mentioning the most commonly used nowadays [5],[4].
RFID technology utilizes radio frequency for the transmission of data and thanks to the miniature size of the RFID tags they can be employed in any area.

The *IEEE 802.11*, most known as **Wi-Fi**, is a world wide used wireless medium that provides a way to send and receive signals, commands and data and it is characterized by simplicity and low installation and maintenance costs.

Bluetooth is an acclaimed standard among mobile phones in which the new technologies are "Bluetooth Low Energy" (BLE) and "Bluetooth smart" that are widely used particularly among the wearable electronic devices.

Another important communication protocol for IoT is **ZigBee**. It is suitable for applications that need more powerful battery, secure networking devices and a low rate for data.

Besides, **Near Field Communication** (NFC) is designed for a safe, wireless and bilateral communication of devices situated in a very short distance. As example, it is used for performing contactless payment transactions and for accessing digital contents.

Furthermore, some of the most common IoT protocols are: **6LoWPAN**, IPv6 over Low Power Wireless Personal Area Network, and **MQTT**, Message Queuing Telemetry Transport. The former is based on the IP standard so it can be connected straight to other IP networks, it supports low bandwidth, different network topologies, power consumption, mobility and unreliability [4]. The latter is a lightweight messaging protocol characterized by small size, low power usage and ease of implementation.

2.2 Policy enforcement approach

The policy enforcement is a mechanism able to force an application to follow a set of defined actions. In the following subsections we describe the nature of the policies and two of the most common rule engines: Drools and Esper.

2.2.1 Characterization of the policies

The policies can be seen as a set of operating rules established for performing actions, maintaining order and organization on data and on the operations.
They support separation of concerns and modularization of rules that need a dedicated suitable paradigm, next to the hard-coded implementation of the application behaviour.
In a distributed IoT context these policies allows interoperability and standardization, that are essential in the IoT development.
They might change in a consistent way through all the environments, they are spread in more than one device and in more then one kind of device (like gateway

and server). Furthermore, they are dynamic, they must encapsulate configurable solutions and they might change themselves in case of unexpected events or in order to tune the performance of the applications.
In our building management context we mainly focus on monitoring the environment values and provide a way to correctly handle the data transferred.
In general, the policies can be used for a variety of concerns listed here:

1. **Security**:
    - Access control
    - Encryption/decryption of the data
    - Security obligations
    - Handling security incidents and data anomalies
    - Availability
2. **Platform management**:
    - Performance
    - Updating
    - Maintenance
3. **Specific application policies**
    - Supporting app policy enforcement
    - Meta-application policies

2.2.2 Policy Systems: rule engines

A rule engine is a software system that executes business rules in a program environment at run-time. In this thesis we use Drools. It is an open source project written in the Java programming language and it is an example of Business Rule Management System (BRMS). It consists in a rule engine which produces output after the processing of facts and rules. It is possible to quickly introduce inexpensive changes thanks to its centralization of business logic. Drools also provides an efficiency writing of the rules in an human-understandable way [12].

Another available software for the management of rules is Esper. It is used in Complex Event Processing (CEP) and in event series analysis available for Java. Esper provides a language called EPL, similar to SQL, that allows an easy configuration of patterns and rules that will be applied on the data stream.
Esper is suitabe for applications that process large amount of incoming data. Esper filters and analyses events in various ways and respond to conditions of interest in real-time [13].

2.2.3 A Business Rules Management System: Drools

As introduced in Section 2.2.2, Drools is a Business Rules Management System (BRMS). This term relates to a software able to manage all the complexity and variety of decision logic, like its definition, deployment, execution and monitoring. We can define this logic as business rules, conditional statements and requirements that are exploited by an organization or an enterprise to define which actions have to be performed in a system or an application in different circumstances [14].

Specifically, Drools is and open source project written in Java supported by JBoss and Red Hat, Inc. The most important element in the Drools architecture is the Business Rules Engine (BRE). It processes objects (called *Facts*) and produces output as a result of rules applications. The Rule Engine is used to define "WHAT" to do and not "HOW" to do it [15].

Several are the advantages of a Rule Engine: it allows to bring closer the technical and management teams in a company, thanks to the syntactical simplicity of its rules that can be written in an human-understandable form.
It keeps separated the business logic (that resides in the rules) and the data (that resides in the object). It guarantees speed and scalability and has tool integration such as Eclipse [12],[15].

The central elements in Drools are the rules. A rule is a piece of code that define what action has to be performed due to a precise condition: " *WHEN* a condition is true *THEN* take this action". If the *when* part is fulfilled, the *then* part is executed.

The Figure 2.1 below shows an example:

```
rule "Temperature high check"
    when
        $t : TemperatureReading(toDouble(contents.data[0].value) > 50.0)
    then
        System.out.println( "Alarmed situation! HIGH temperature from sensor"
                            + $t.getContents().name);
end
```

Figure 2.1: A Drools rule

This simple rule is fired when the field `contents.data[0].value` of the `TemperatureReading` Java Object is greather than 50 and as result it prints a message about which sensor detected this temperature.

To fire rules on given data, we need to instantiate the framework provided classes with information about the location of rules file and the *Facts*. The latter are the data on which the rules will act upon. From Java perspective, Facts are the POJO (Plain Old Java Object).

We use the `KIE API` library to set up this framework [16].
First, we have to set the `KieFileSystem` that provides the container in which we define the Drools resources like the rule files. Next, we set the `KieContainer` that is a container for all the `KieBases` (the repositories of the rules) of a given `KieModule` (a container of all the resources necessary to define a set of `KieBases`). We perfom

the `buildAll()` method invoked on `KieBuilder` that builds all the resources and ties them to `KieBase`.
Lastly, the rules are fired by opening a `KieSession` and call the method `fireAllRule()`. The `KieSession` is the prevailing way to interact with the Drools engine. A KieSession allows the application to establish an iterative communication with the Drools engine. The reasoning process may be triggered multiple times for the same set of data. After the application finishes using the session, it must call the `dispose()` method in order to free used memory and the resources [17].

The power of Drools consists in the fact that the rules are not hard-coded and therefore we can change the policy files in the `KieFileSystem` at run-time, without the need to recompile the project. In this way we can modify the files while the application keeps running with the only carefulness of maintaining the consistency of these changes throughout the system.

2.3 VersaSense Micro Plug-and-Play (MicroPnP)

VersaSensese Micro Plug-and-Play is the IoT platform used to create the prototype of this thesis. It addresses the complexity of creating, deploying and configuring applications for the IoT. It provides a zero-configuration, that means no manual operations or interventions are required to set up the communication, and it reduces the efforts of installation, integration and management of the IoT systems.
This platform realizes a secure, low-power and trustworthy networking, a solution that easily integrates third-party sensing peripherals with already existing applications and on-line cloud services for the storage of the data collected [18].
This platform is used in the Industrial field of IoT, namely in harsh environments in which these devices are supposed to operate and acquire processes data over long periods.

Figure 2.2: The MicroPnP platform

The VersaSense MicroPnP consists in gateways connected to motes with pluggable sensors and actuators. Referring to the Figure 1.2 we could match two of these gateway as GW1 and GW2 and the motes at the level of sensors that provides information to perform hazard monitoring and climate control.
We use this technology as an integral part of our prototype to test our protocol in a real IoT system.
Every sensors provide to the associated gateway data about temperature, humidity, level of light, buzzer activation and motion detection every few seconds.
The gateway itself hosts our developed Java programming that fires policies in place about the value received from the sensors and sends them back to the server.
This mechanism allows to prove the feasibility of using the Drools engine on a real IoT system.
The Section 5.2.1 presents the functionality of the system in more details.

2.4 Related work

In this Section we discuss some significant technologies and techniques related to the dynamic update of components, that already exist in more generic related work beyond IoT.

2.4.1 Significant technologies and techniques

To implement the best procedure for handling the update of the policies in the IoT context, we took inspiration from these works introduced in this Section.

We mention four papers that analyse and propose models in order to address one of the major challenge in the distributed systems context: the presence of evolutionary changes. They refer to those kind of changes that may require modification of functions and applications already present in the system or require the adding of new features. Specifically, they cannot be predicted at the time the system is developed.
Additionally, the presented papers focus on the importance of consistency before, during and after run-time changes.

The first one presents the notion of *quiescence*[19], the second tries to optimize the solution proposed before with the introduction of the *tranquillity* criterion. [10]
The third paper presents an ad-hoc solution for the development, management and reconfiguration of evolving functional requirements of Wireless Sensor Networks [20].
The fourth one presents a distributed application composed by interchangeable components that can be integrated in a core and semi-complete system when a new functionality is required [21].

2.4.2 Dynamic Change Management: quiescence

The paper in [19] presents a model for dynamic change management in a system, based on separations of concerns: functional concerns and structural configuration concerns (interconnections and components). They investigated the possibility of developing a system that must be sufficiently flexible to allow dynamic changes, without interrupting the unaffected parts. These changes need to be managed and controlled.
To interface between the functional and structural view of the system they identify a separate configuration management. It provides means for specifying and performing changes.
Another architectural element, the change management, provides facilities to controlling the consistency of changes through the application. The changes refer only to the structure of the system.

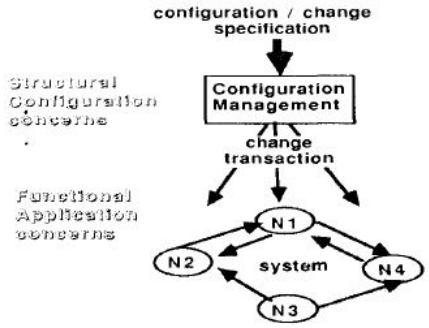

Figure 2.3: System configuration and change management

In particular, the objective of the change management are: specify changes in terms of system structure and not define them at too low-level, specify the changes declaratively so that it is the configuration management that determines the ordering of the changes not the user, the algorithms protocols and states of application must be independent and, finally, it must provide always a consistent state and affect only the nodes involving in the change without compromise the executions of the others.

The term they present as fundamental for the correct management of changes is *quiescence*. The change management waits for the node that needs a change to reach a *quiescent* state, that means no communications with the nodes involved nor with the environment.

In order to perform changes starting from a consistent state and ending to another consistent state, the management system must be able to check the configuration state for each node (active or passive). This information is provided by an interface with the application. Then, when a change occurs, being in a passive state for a node is not sufficient for guaranteeing the consistency. It only defines that a node cannot initiate any new transaction but it can accept and serve transactions requested from other nodes.
Instead, the quiescent state is required for changes. In this state a node is passive

plus, it cannot accept and serve requests, so this property involves the other nodes as well.

The definition of quiescence claims that a node is in this state if [19]:

1. *It is not currently engaged in a transaction that it initiated.*
2. *It will not initiate new transactions.*
3. *It is not currently engaged in servicing a transaction.*
4. *No transactions have been or will be initiated by other nodes that require service from this node.*

This is a strong requirement that makes the configuration state of a node consistent and frozen. It means that it does not contain partial result from transaction neither its state can change after new transactions.

2.4.3 A low disruptive alternative: tranquillity

In [10] the authors relax the condition of *quiescence* presented in the previous paper. They states that this condition results in a serious interruption of the updating application. To solve this problem, they introduce a weaker alternative state for the nodes of a system that ensures consistency during the update of a running system: *tranquillity*.
Even if is not proven that tranquillity can be reached in finite time, experiments show that a short time frame is nearly always enough for reaching this state.
The tranquillity notion is based on two facts: first, a node can be safely replaced if a transaction in which it is involved has already passed through it or it will pass in the future. Second, all nodes have to be black boxes by design in order to improve decoupling and re-usability of the system components. All the participants of a transaction either are the starter or are adjacent nodes. Other nodes can be not involved in the transaction since the starter does not know about their existence, but can be involved in following sub-transactions.

Therefore the authors come up with the definition of tranquillity as a proper status for updatability. They claim that a node N is in a tranquil state if [10]:

1. *It is not currently engaged in a transaction that it initiated.*
2. *It will not initiate new transactions.*
3. *It is not actively processing a request.*
4. *None of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.*

Quiescence entails tranquillity but not the other way around.
Tranquillity does not require that nodes connected to a changing node N cannot initiate a new transaction that includes it. Only node N must become passive, the other nodes have just to conclude active requests to it, they do not have to be completely passivated.

They prove that the tranquillity is sufficient for the updating of nodes' status.
An important drawback of this criterion is that the tranquillity state has not the guarantee to be reached in a bounded time. This could happen when a node is used in an infinite repetition of interposed transactions. This situation must require the use of quiescence because also the other nodes involved must be passivated. Therefore, any system that uses the tranquillity criterion for the updates must implement a mechanism that implements quiescence in case tranquillity cannot be reached.
Another drawback is the fact that this is not a stable criterion. All interaction with a tranquil node must be prevented to preserve the state.
Instead, in the case of quiescence, since it lead to the passivation of all nodes directly or indirectly linked to the changing node, the stability is always guaranteed.

The authors present DRACO, a general purpose middleware platform in which they implement a prototype of the system. They observe components in the tranquil state and, when it is not possible, fallen back to quiescence state.
The components in DRACO are Java classes (that match the nodes of the previous work) with connectors (edges) that provide interactions among them.
Similar to the previous paper, the architecture of the DRACO middleware platform consists in module for the management of different activities: component manager, schedule manager, message manager, module manager and connector manager.

2.4.4 CaPI: a component and policy-based approach

The policy-based systems have already been investigated in the context of Wireless Sensor Networks (WSN) [20],[22] and in the context of Body Sensor Networks (BSN) [23]. In particular, in [20], they propose the Component and Policy Infrastructure (CaPI), a reconfigurable middleware platform aims at overcoming the limits of using policies just for provide application functionality, extending the use of them also for entirely manage the changing of functional and behavioural concerns.

The authors focus on the longevity, large-scale and resource constraints of the contemporary wireless sensor networks. These characteristics demand possibly frequent reconfigurations to satisfy changing requirements, high-level of abstraction and efficient management of the networks.

CaPI provides two different programming abstractions: *policies* and *components*. The former are implemented to support personalization and management of concerns at run-time. These policies are lightweight and built with an effective and expressive language, independent from platform. They follow the Event-Condition-Action (ECA) model, like Drools.
The latter are sections of software functionality that allow access to features at low-level of different hardware platform and operating systems. Their functions

are described by a type and an identifier and their connections are event-based.

CaPI also provides a flexible reconfiguration model that manages dynamic evolution of both functional and non-functional concerns, adding, removing and recomposing new components and policies at run-time with minimal overhead.

The authors highlight the necessity of WSN's dynamic reconfiguration to optimize system behaviour in case of changes in environmental conditions and new application requirements that are impossible to anticipate. Furthermore, another reason for reconfiguration is the heterogeneity of software because of different platforms, operating systems and languages.

As far as policies are concerned, CaPI implements a run-time adaptable policy infrastructure that consists in:

1. *Configurable enforcement points*: Policies are enforced in distinct site. When an event traverses one of these sites, it is sent to the policy engine with some information that will be used for making the decision about what policies need to be enforced.

2. *Adaptable policy engine*: The architecture offers dynamic installation and dismissal of policies and extension with new functionality at run-time as well.

3. *Policy repository*: It computes and assigns the required memory for the policies and stores the corresponding metadata like status and enforcement points.

2.4.5 Dynamic and selective combination of extensions in component-based applications

In this paper [21] the authors present the concept of component framework. It is a semi-complete software architecture composed by components that can be selectively parametrized and interchanged within the core functionalities.
In such system we could replace a component with a more appropriate one for a specific task or add a new one due to new requirements requested by the end users.

The issue raised, however, concerns the nature of the components and their interactions that are not easily changeable due to consistency and interoperability problems that can occur.
They propose a component-based system made by the dynamic combinations of extensions implemented as mixin-like wrappers on a minimal core system. They simultaneously and dynamically adjust components and interactions when changes take place.

In order to look at the presented approach in our IoT context, we can match their concept of component with our concept of policy.
The authors described three main challenges involved in the introduction of consistent changes within the system:

1. Modular customization: adding a new extension should not influence other components not involved in the change. Besides, each change should, in accordance with the separation of concerns principle, be dynamically implemented

or removed as a disjointed module that does not require change in the core platform and in the code.

2. System-wide refinement: if some changes concerns to a refinement of the core system, they should be performed as an atomic and consistent operation. This is because it involves a system-wide refining of various core modules simultaneously, due to their interactions.

3. Context-sensitive extension: they point out the importance of contextual properties, such as specific service operations requested by the clients that require run-time changes to the core system.

They implement extensions as one or more wrappers that support modular customization of applications through the encapsulation of the code needed for implementing that new functionality. It is possible to construct multiple wrapper chains to support client-specific extensions. The wrapper-based system is inspired by and supports Role Object [24] and Decorator [25] design patterns. The former dynamically assigns to a core component new service interface that allows to widen the component type, the latter permits to add or remove functionalities to an existing object without impacting its structure.

This idea of multiple wrapper chains combined together for realizing the extension leads to consistency and scalability problems. The first problem refers to the complexity of maintaining consistency in case of dynamic aggregation of extension. The second problem concerns the issue for the objects to preserve the references to the chains in order to invoke the methods they offer. The wrappers themselves may need to refer to other wrapper chains as well.
They solve these problems developing *Lasagne*, an architecture independent from the platform, to be implemented on top of any object-oriented programming language based on classes, that supports the correct integration of extensions.

2.4.6 Comparison and comments

All the papers described in the previous Sections contribute, to different degrees, to inspiring the middleware and protocol implemented for this thesis. They also gave a comprehensive overview of existing models and technologies in the context of distributed systems. In this Section we made a comparison among them and we focus on the main characteristics that has been exploited.

The paper in Section 2.4.2 was written in the nineties and it is probably one of the first researches in the area of the evolutionary changes in the distributed systems. It contributes in giving a primary idea about the different kind of states for the nodes and how to correctly manage them.

This work was continued and completed with a more recent one, almost 20 years later, described in Section 2.4.3. The authors present an alternative state (*tranquil* state) for the node that is changing. It is the most similar to the one implemented in the prototype of this thesis: when a new policy file is uploaded, the two gateways end in a state in which they cannot manage new events and a queue stores them until they will become active again.

In [20] the focus is on the Wireless Sensor Network. They develop a sophisticated middleware based on components and policies. The requirements of supporting efficient reconfigurations and management of long-lived platforms are the same of ours in the IoT context, but what we are investigated in this thesis is the possibility of using already existing technologies, without realizing an ad-hoc architecture.

The fourth illustrated paper describes an architecture similar to the one presented in this thesis: a core system at the basis that provides the main functionalities and, separately, the possibility to add extension (components or policies for us) to provide new services or changes in the behaviour of the system. The issue of consistency is common in both context. They tackle this problem decentralizing the wrapper composition logic in a *composition policy* external from the rest of the system. It is sent, together with the stream of messages, and will be managed by interceptors that guarantee system-wide consistency.
In our prototype the consistency is guaranteed by the implementation of the protocol itself.
However, even if the overall architectures are close to each other, this approach is more oriented to adding functionality to the programming language and classes, while we are more oriented to the use of Drools that is above the code.

Chapter 3

Problem elaboration, analysis and requirements

In this Chapter we give an overview of the use cases used for putting into context our problems and our challenges. We then introduce the context diagram of our middleware and the goals that we want to achieve. In particular, in Section 3.1 we discuss the use cases that allow to better focus on the problem we aim at solving. In Section 3.2 we provide an high-level description of the principal components and layers that constitute our context diagram, on which we base our problem analysis. In Section 3.3 we describe the goals already introduced in the previous Section 1.5, in more details, taking into account the context diagram of the system just described and the prior knowledge presented in the previous Chapter. Lastly, in Section 3.4 we perform a requirements analysis.

3.1 Use cases

This Section presents different use cases that could be deployed in the context of our smart building and could take advantage of the policy approach. They allow to identify and clarify the major problems we want to solve and to organize the system requirements. The use cases presented are a set of possible functions and sequences of interactions between systems and users in our environment and related to our goals.

3.1.1 Access control

Access control is a security technique that regulates the usage of resources, it establishes who has access to which asset. In a smart building, the access control application manages, for example, the entrance to secured locations. It achieves this by controlling locks of doors that give access to protected area through a badge, an access card or a keypad. It can also monitor the presence in these areas by using devices such as motion detection sensors.
We define policies to establish the level of security of different locations and give the possibility to change it in case of need. These policies must be always consistent

in every part of the building in order to avoid the situation of having private areas unattended.
Others policies could require a fast update such as the one that detects attempt to unauthorised access, attempt to incriminate others for theft, espionage or damage through stealing someone's access card.

3.1.2 Climate and light control

By the integration of Heating, Ventilation and Air Conditioning (HVAC), the climate control application monitors and controls the climate conditions in the areas of the building through the activation of policies. By reading the sensors' values, they can heat up or cool down a room in order to maintain a comfortable temperature. Windows actuators can be useful for this purpose as well.
The light control application monitors and controls the lighting conditions. For example, the corresponding policies can turn lights on and off based on: preferences, the time of day, human presence and external light. Hence, they can also reduce wasted energy due to lights being on unnecessarily.
Possible policies could also be used for checking the manipulation of climate or light conditions. An attacker could be interested in causing monetary loss or in creating inconvenience or panic by manipulating climate conditions or in espionage on room occupancy. In this scenario is important that the policies are dynamic and interchangeable in order to face new and unexpected threats. The consistency property is also valuable in order to define equal suspected conditions that trigger an alarm through all the different areas.

3.1.3 Air quality, fire detection and evacuation

This application monitors the environment for dangerous or inconvenient conditions and can raise an alarm in case of a detected hazard. Air quality is monitored by sensing concentration of specific compounds. In case of an emergency, this application shows the most efficient escape routes at several locations throughout the building. Routes can be shown on displays or use alternative signs such as lighting up arrows. The application takes into account presence and topology information to avoid congestion and ensure a swift evacuation of the building. It could be triggered by other applications, such as the previously described climate control application.
The policies that rule and tune this application should have fast and consistent updates for all the areas of the building. A bad collaboration among the devices caused by inconsistency in the policies can lead to some problems such as a fake setting of the alarms, a wrong escape route calculations or theft, e.g. by causing a fake alert that opens the windows to freshen the air.

3.1.4 Encryption and Decryption functionality

It is clear that the amount of data and signals exchanged in order to provide the services described above is huge. Additionally, all these communications are wireless and, therefore, this leads to serious threats of eavesdropping or, worse, tampering. Since these data control the functionality of the assets in a living environment, a misbehaviour can endanger the life of the inhabitants and workers.
We develop a protocol able to support a secure exchange of data through the encryption and decryption of the messages. By means of policies, we also provide the possibility to modify the algorithm with which the messages are encrypted or decrypted, preserving the consistency at both sides. We present it in detail in Section 4.3.

3.2 Problem analysis and context diagram

As already highlighted, the main goal of this thesis is to accomplish a distributed system governed by dynamic policies, in which the consistency's property is guaranteed. The overall model of this proposed system consists in different components: one back-end server connected to multiple gateways, connected, in turn, to some sensors. The latter transmit to the connected gateways data about the environmental state every few seconds.
The architecture of this proposed model will be described in the Chapter 4.

Going into detail about every components involved, the inner diagram can be conceptually divided in two main layers:

1. An underlying collaborative architecture that provides the communication functionalities: it establishes which are the overall data and information that the devices make available to one another how the data exchange is executed. This architecture is fixed and is independent from the management level above. It is generic layer thanks to which the communication between server and gateways are performed, for example the gateways send to the server all data they collect from the sensors (temperature, data to encrypt/decrypt and other measurements) and the server can send to the gateways the commands for requesting particular measurements during an unexpected situation or it can transmit a new policy file.
   In that way, the policies, managed by the management layer above, will really control the behaviour of the system deciding which data to take into account from the total made available.

2. A management layer that controls the implementation and upload of the policy files.
   At the server side, this layer uses the communication channels of the collaborative architecture to trigger the transfer of a new policy file and it can send other metadata that guarantee the correct and uniform implementation of them. At gateway side, it is in charge of sending metadata for the same purpose as well plus, it works as a protection level that reasons about the

messages arrived and manages inconsistent situations that can occur. In addition, it handle all the steps for uploading correctly the new policy file once arrived.

Around these two main layers there are the level of rules application and the level of the Policy System from which the set of rules are fired and selected.
The Figure in 3.1 shows a schema of the described model.

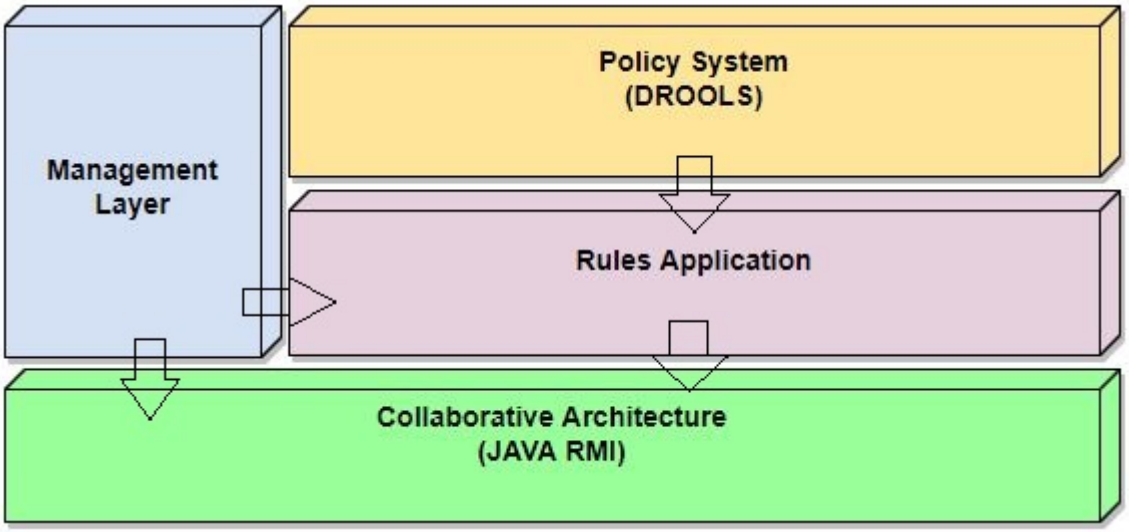

Figure 3.1: Context diagram

Through the communication protocols of the collaborative architecture the gateways can send the data to the server and with the rule application layer they take decision that depends on the policy file selected at that moment.
Similar actions performs the server, it collects and fires rules on the global data received from all the gateways plus, it can trigger the upload of new policy files on the overall system.
Indeed, the main role of the back-end server is to control over the policy's executions on the gateways and to receive data and results from them. Moreover, the back-end server can trigger an update or a change of the policy files to the gateways. This is a critical step because it involves changing policies at different places and at run-time, during the continuous acquisition of new data.

Every component of the overall architecture has their own policy files, the back-end server controls all of them in order to guarantee distributivity and consistency of the execution.
The policies implemented are divided into stateless and stateful. The former are the simpler type, they only trigger an action based on the current data without additional analysis. A possible update only requires a correct management of the queue of events.
The latter demand more attention. They maintain and update a state during their execution with the consequence that a potential change of them will require the correct management of this state. It must be correctly and efficiently preserved. In Section 4.4 we propose a solution for handling this eventuality.

The policies are modelled as Drools files. As introduced in the paragraph 2.2.2, these files are composed by rules as well as some resource declarations like imports, globals, functions and attributes that are assigned and used by the rules.
Therefore, all components include a Drools file (with .dlr extension) that contains the rules in the form:

```
rule < rulename >
<attribute> <value>
when <conditions>
then <actions>
end
```

These files are the changeable part of the system that stay on a well-known and declared run-time.

3.3 Goals and metrics

As our model is divided in two main levels, the goals and objectives of the thesis involve the investigation of two aspects of the system:

- At the management layer we want to investigate the achievement of a policy-based system with determinate characteristics.
  This is the main focus of the thesis. We want to show that this layer allows to deploy and upload policy files in multiple places in a way that is consistent, coherent, fast, robust, versatile and automatic.
  We use the notion of quiescence and tranquillity shown in Section 2.4 to come up with a protocol that enforces all these requirements.
  The principal questions that we want to answer to are: if we are able to effectively get dynamic updates of policies, if these updates are well-performing, if they are distributed overall the system, if they are update consistently in a current processing system and if these policies can carry a state that will be correctly managed.

  We demonstrate that the protocol works as intended, that changes are possible at run-time without restarting the application, that no messages are lost during the updating phase, that after a reasonable amount of time all the components come back operational and that inconsistent situations that can occur are correctly managed.

- At the underlying communication layer we want to examine the criticality of the implementation of Drools infrastructure on a real IoT gateway, in particular on the VersaSense gateway introduced in Section 2.3.
  We want to demonstrate that even if Drools was not designed to these kind of devices it is a feasible option in the construction of IoT system governed by policies. We want to prove that the time and memory overheads do not prevent the success of the application developed.
  Furthermore, we want to show that even if the VersaSense's gateway has less

power capacity, less speed and less memory than a ordinary laptop, all the steps involved in the realization of the protocol are carried out in a reasonable amount of time.

Together with the Drools infrastructure, we analyse and prove the functionality of Java RMI as communication mechanism chosen.
The metrics used to measure these requirements are the milliseconds needed to realize all the operations at the real gateway side compared to the same functions realized by an ordinary laptop. We want also to measure the extra memory intended for the supportive structures used for storing and managing the helpful metadata.

3.4 Requirements analysis

In this Section we perform a requirements analysis in which are summarized our goals, seen as functional and non-functional requirements of our middleware.

3.4.1 Functional requirements

The first functional requirement concerns the policies themselves. The development of a policy-based system is one of the main focuses of this thesis, therefore we want to achieve a platform able to work with policies and based on policies.

A correct communication between the different components must be guaranteed in order to have a consistent state assured every time. It is required to correctly handle the queue of events that need to be evaluated during the updating phase, to avoid losing events or management messages and to prevent the misunderstanding of commands.
In particular requirements of consistency, coherency and speed are the most important ones. For example if we upload a new policy file at the gateway side that raises an alarm if it is reached a temperature of 60 degrees, the similar policy must change consistently at the back-end server side.
Another example is that a message in the queue of events that has been encrypted with a specific algorithm, it must be decrypted with the same algorithm even in case of intermediate changes of policies.

In case of more complex policies that carry a state, our middleware must be able to preserve it during the update phase.

A further requirement concerns the underlying collaborative architecture: it is declared and well-known and it must be as general as possible in order to accommodate every requests and commands demanded by in the policy files.

The overall middleware composed by the architecture and the Policy System must provide a degree of flexibility adequate for being compatible with heterogeneous devices. In addition, scalability is also required to handle a variety of multiple devices.

3.4.2 Non-functional requirements

In our platform, the non-functional requirements are basically the performances and acceptable time, memory and network overheads.
The impact of the Policy System running on the platform must be reasonable. Equally, the impact of the additional structures needed for memorizing all the metadata must be sustainable.
Furthermore, we aim at providing a middleware that must offer the fulfilment of the policy updating phase in a short amount of time, in order to decrease the possibilities of inconsistency as much as possible.

Chapter 4

Dynamic and consistent policy updates

This Chapter presents the architecture of the system designed and developed for accomplishing the dynamic and consistent updates of policies in a distributed IoT scenario. Section 4.1 introduces the description of the architecture, Section 4.2 gives an overview of the components and mechanisms of Java RMI, the technology chosen as communication channel. The following Sections describe the solution's key points. In 4.3 is explained the use case we focus on and we describe the steps of our protocol that achieves a consistent and dynamic policy update. In 4.4 we discuss the possibility of having to deal with stateful policies.

4.1 Architecture and design overview

The inner model of the components shown in Section 3.2 underlines the presence of two main layers on which we base the inner architecture of the system:

- The collaborative architecture is the operating level thanks to which any kind of communications between the different devices are made possible. This is achieved by the usage of Java RMI technology, explained in detail in Section 4.2. It provides channels for file transfer and channels for sending management messages.
- The management level is the protocol implemented for achieving the correct execution of the rules. It is able to satisfy all the requirement described in Section 3.4.

In order to present our solution, we refer to a global architecture that consists in one back-end server connected to some gateways. It represents an example of an IoT platform. The gateways are connected to some sensors that continuously provide some environmental information, such as temperature readings, movement detection, level of light and activation of the buzzer, to the gateway they are connected to. All these data are modelled as events.
Furthermore, the gateways and the server host their own policy files, thanks to

which they can fire rules on data in place.
The architecture described is shown in the Figure 4.1.

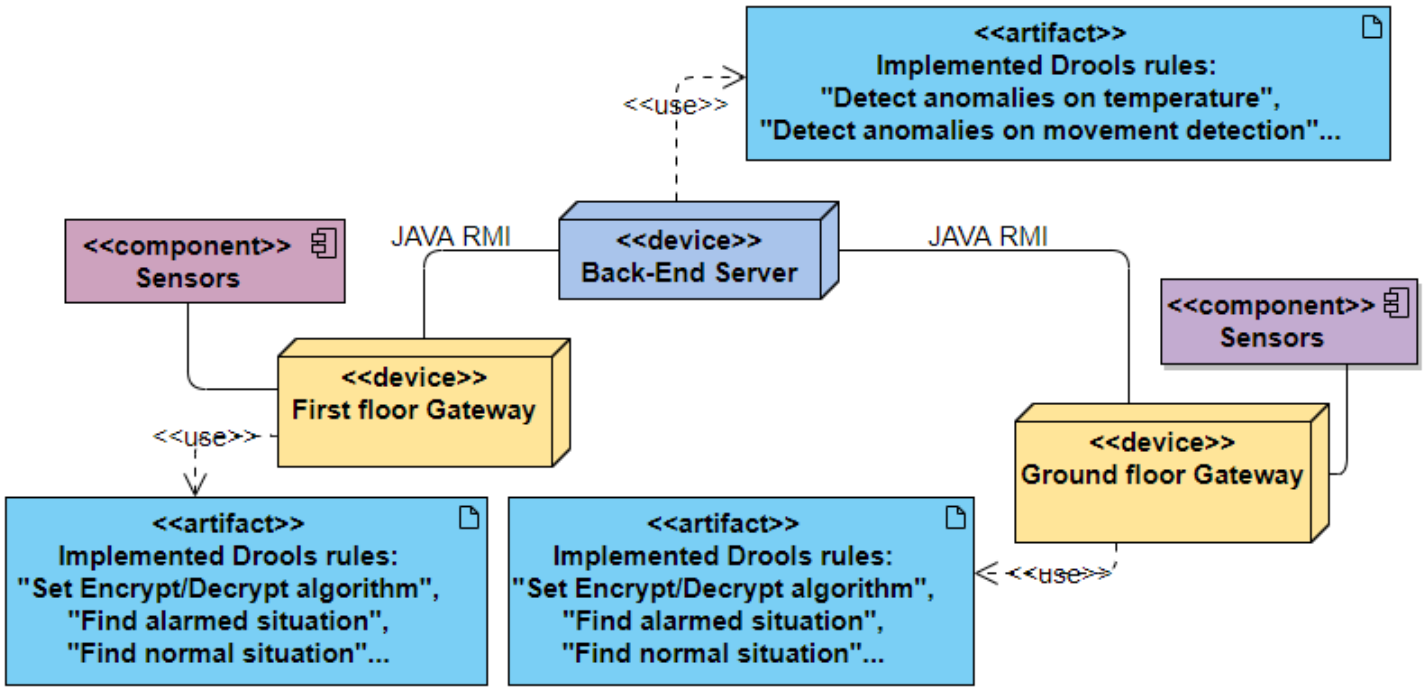

Figure 4.1: Global architecture: deployment diagram

Next Sections describe in details the choices for the realization of the architecture and the design of the solution proposed.

4.2 Java RMI communication protocol

The communication is established through Java RMI technology. Since our focus is on building a distributed middleware, through Java RMI we achieve remote communication between the components involved.

The general RMI scenario consists in [26]:

1. A remote object that is created by the server and a reference of that object that is made accessible for the client through a registry.

2. The client makes a request for this remote object of the server, in order to be able to invoke its methods.

We instantiate different remote objects, in the server and in the gateways (clients), in order to made methods available for each side [27].
The server creates a remote object and makes it available for the clients. In our

architecture we run the RMI registry on the server. With the methods offered by Java RMI API, a stub for the RMI registry is obtained by the server. Subsequently, the server registers its object with the RMI registry calling the `bind()` or `rebind()` methods. On the client side, to invoke this remote object, it is necessary to fetch it from the registry using the `lookup()` method [26].

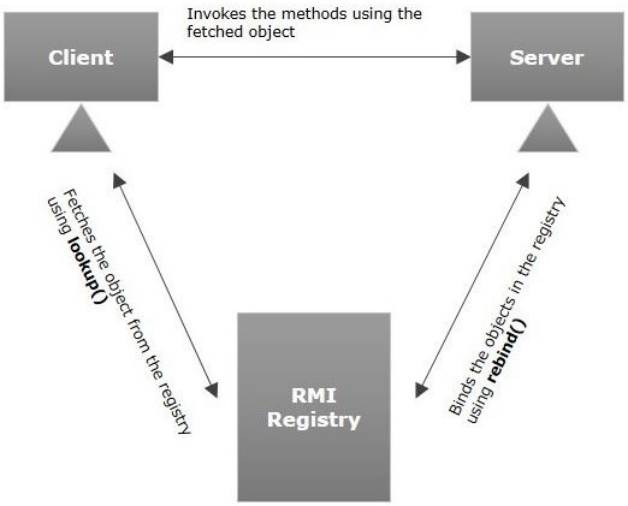

Figure 4.2: Java RMI components

After this procedure, the clients (in our prototype the gateways) can start the interaction with the server. Since we are interested in peer-to-peer communications, in which the server can also invoke methods on the client side, the clients, in turn, create their own remote objects and export them to the server side.

In this paragraph, we present some specific RMI classes and objects we created in our middleware architecture. In particular, we describe the procedure followed by the climate control system, in which the corresponding policy requires analysis and exchange of the temperature values. We include all the steps for realizing the communications through Java RMI within our implementation.
First of all, as described previously, the server calls the `bind()` method for sharing its remote object, hence the gateways can call the `lookup()` method for retrieving it. After that, both gateways share their own remote objects with the server in order to put in place a bilateral communication, as you can see in Figure 4.3, with the server's method `registerClient()`.
At this point the server can call the client method `createThread()`, that triggers the creation of a thread at the client side. This thread will manage the data received from the sensors.
After, both the clients handle the collected data and send them to the server with `sendEvent()` method. It will use them to compute aggregate values, to perform operations and to calculate statistics. If the server detects some unexpected values, it can request an immediate transmission with the `c.sendAllTemperature()` method.
The sequence diagram of the protocol described is shown in the Figure 4.3.

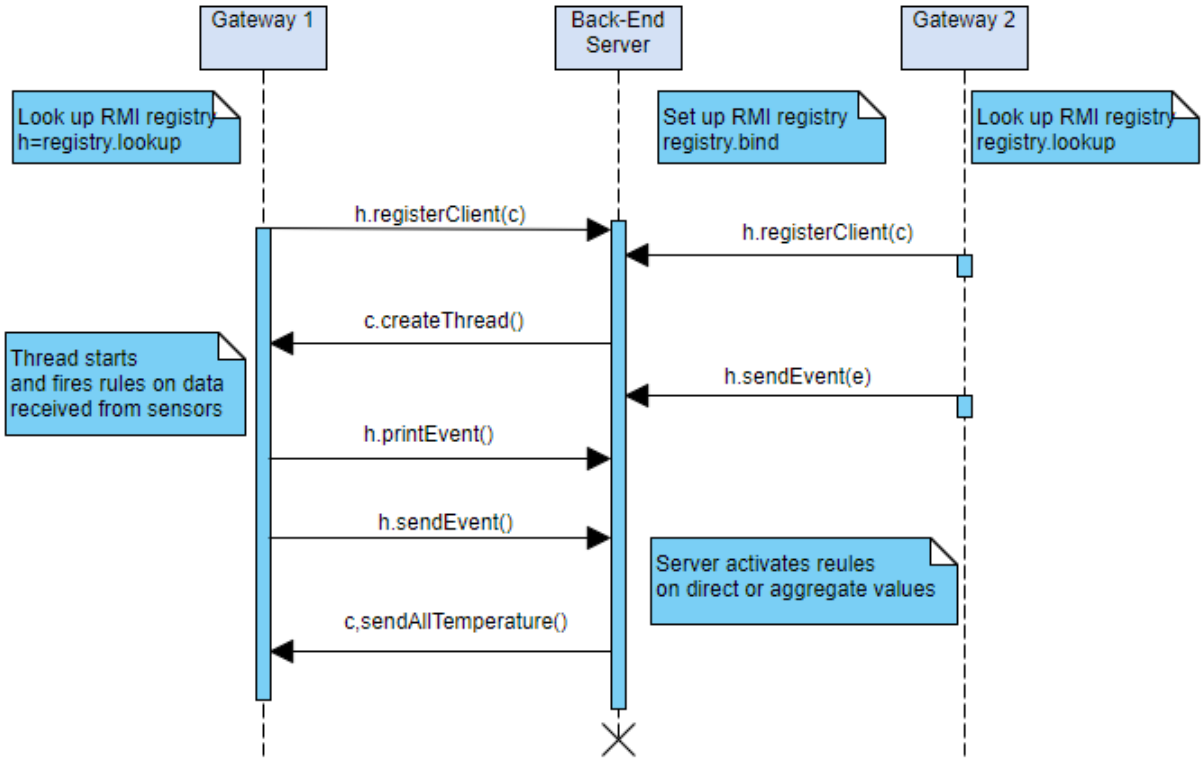

Figure 4.3: Example of a Java RMI communication

In our developed Proof of Concept, described in Section 5.2.3 of Chapter 5, we introduce specific RMI classes and objects fundamental to achieve the distributed communications in our chosen use case.

4.3 Distributed encryption/decryption protocol: a challenging consistency example

This Section presents our solution that addresses the possibility of implementing dynamic policy at run-time and solves the consistency problem among the nodes, during these policy updates. In order to emphasize the importance of consistency among the distributed policies, we focus our research on a particular use case: the encryption and decryption of messages. We present which steps our developed protocol must follow when we upload a policy that requests encryption of data at one side and the consecutive decryption at another side. This, more than any other use case, clearly shows if the consistent properties are achieved.

In the IoT context we have already accentuated the importance of security and privacy of the information stored and exchanged between different nodes of an IoT system.
A possible use case related to the confidentiality of data through a system is an encryption and decryption process: it consists of the encryption of messages at one node of the system and the corresponding decryption at another node.

This schema requires some careful and precise steps of the protocol that guarantee consistency between the two nodes involved, in order to correctly transfer and retrieve the information.
Moreover, in an IoT system it is useful to provide the possibility to change the algorithm used for encrypting at any time it is necessary and, accordingly, the same algorithm must be immediately used at the other side. A message encrypted with a specific version must be necessarily decrypted with the same version.

4.3.1 Design of the standard protocol

The standard protocol for a single sender and receiver, that represents how the middleware handles the message flow, is shown in the blue diagram below, in the Figure 4.4. The yellow boxes, even though they are actions always performed by the decryption gateway, do not refer to the standard part of the protocol, but to the updating part, described in next Section.
We present a diagram that can be generalized for multiple senders and multiple receivers as well. It will be explained in more details in Section 4.3.3.
As already mentioned, it is worth noting that these operations can be similarly performed for other use cases, where a sender and a receiver of events are involved. For this reason, we call the two gateways generically sender and receiver gateway.

Basically, the protocol is straightforward: at the beginning both sender and receiver gateways create and store a new `KIESession` (see Section 2.2.3) together with the number of the corresponding version of the policy file the `KIESession` is associated with. After this, the sender gateway checks if the state of the system is not frozen, that means no ongoing updating phase. If it is not the case, it retrieves new data, it keeps track of the number of data that it is sending, it fires the rule to encrypt them (or more generally, to handle them) and lastly, sends them to the server.

The latter, after a request, sends the data to the receiver gateway, that, in turn, checks if there is data available and if the upload phase is not in progress. Then, it fires rules to decrypt (or handle) this data if it owns the correct version indicated on the data themselves. Furthermore, it maintains a counter that indicates how many messages it has decrypted so far and with which version. This information is crucial for the success of the protocol.
Every time it checks if it has received all data belonging to a specific version for every stored `KIESession`. If this is the case, it disposes the corresponding `KIESession`. These two last steps actually belong to the updating steps described in the next Section. We want to highlight that all these operations are performed when the updating procedure is not running.

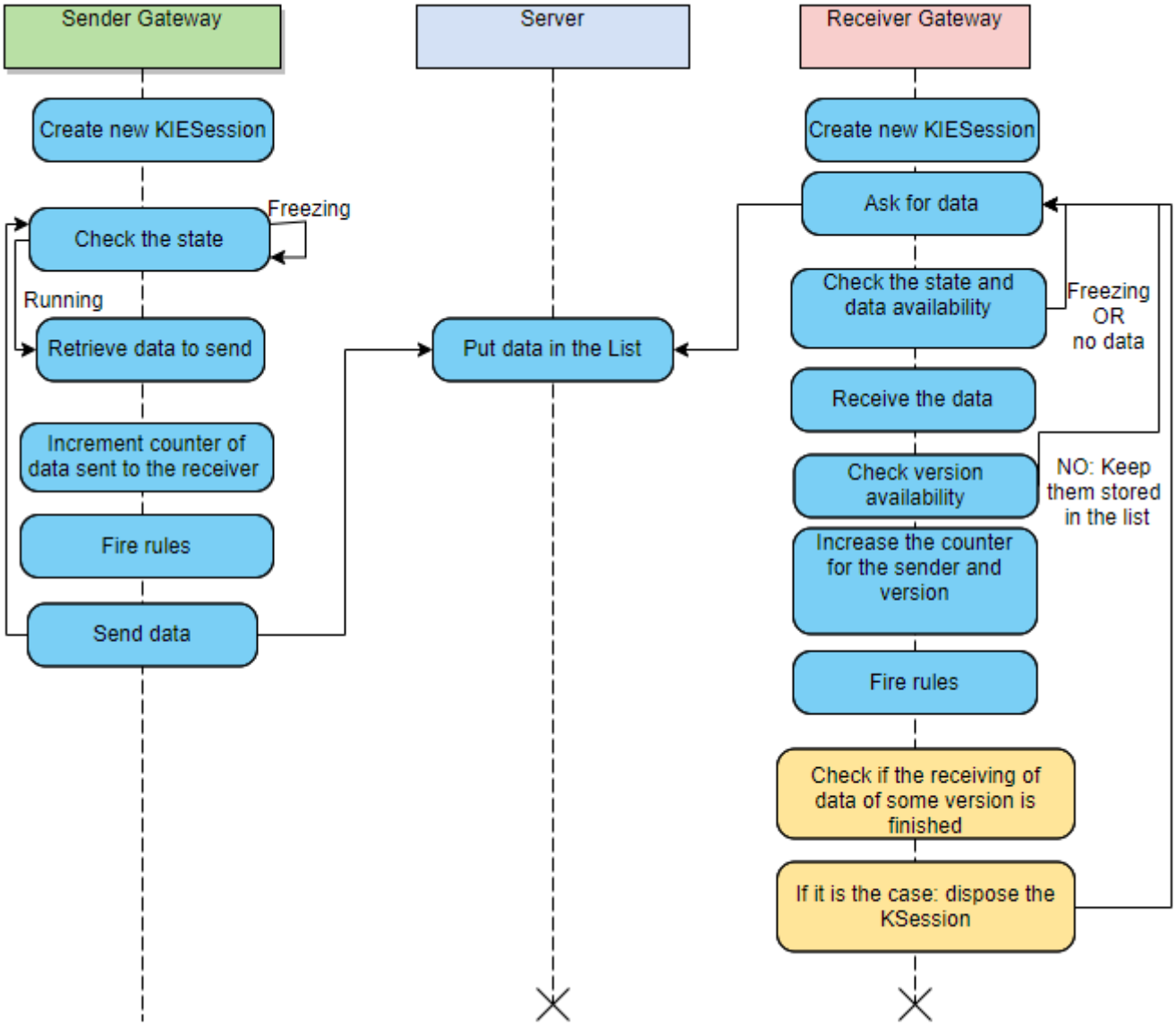

Figure 4.4: Standard protocol sequence diagram

4.3.2 Design of the updating protocol

The instant at which the server triggers a version update of the encryption algorithm imposes the need for a carefully designed updating protocol. In our case it consists of the upload of a new policy file with a defined version. For a correct result, it is fundamental that data encrypted using a certain version must be decrypted using the same version.
Due to a possible overload of the network, in a real scenario, even if the server communicates the change to both sides at the same time, it may happen that the two gateways are not perfectly synchronized with the same version currently in use. This may lead to some inconsistency problems that we will analyse in the following Chapters.

The Figure 4.5 shows the diagram of this updating phase that, again, can be generalized for other use cases. First of all, the server sends the new policy file with the associated version number to both sides. The sender, after waiting that the already processed messages have been sent, freezes the system. The receiver freezes the operations as well. It means that the communications are blocked and no message can be sent, received or processed. Then, the two gateways instantiate a new `KIESession` with the corresponding file and version number.

At the sender side we can safely dispose the previous `KIESession` because we are sure that no message will be generated with that version any more. It subsequently sends to the server, and this latter will then forward to the receiver, a piece of information very important for a successful outcome of the consistent protocol: the total number of messages that has been elaborated with the just disposed version. Lastly, it restarts the standard process.

At the receiver side, in order to avoid inconsistency problem, we cannot immediately dispose the old `KIESession` because some data in transit can still require the old policy file.
Therefore, back in the standard flow, the receiver checks every time if it has received all the messages processed with a certain version that it still stores. It compares its own counter with the total number received from the sender. If they are equal, it can dispose the `KIESession` without any risk. It has to perform this check every time because it cannot predict when the information from the sender will be available.

The Section 5.2.3 will describe the implementation of this protocol in details.

4.3.3 Multiple senders and multiple receivers

We have just described a scenario that consists in one single sender and one single receiver. Actually, in a distributed context, multiple senders and receivers can perform respectively the encryption and decryption phases. In addition to the previous steps, every sender gateways must store how many data they have sent to which receiver for each version. In the same way, the receivers must store how many data they have received by which sender and with which version so far.

During the updating phase, the senders send to the different receivers their own code, the version of the old policy file that they are replacing and the total number of data that they have sent, processed with this old version of the algorithm.
The receivers will store and use all these data in order to correctly handle the messages. In the standard flow they can perform the dispose of an old `KIESession` only if they are sure that from no senders they will receive some messages that belong to the old version any more.

The technicality of this protocol is also presented in the Section 5.2.3.

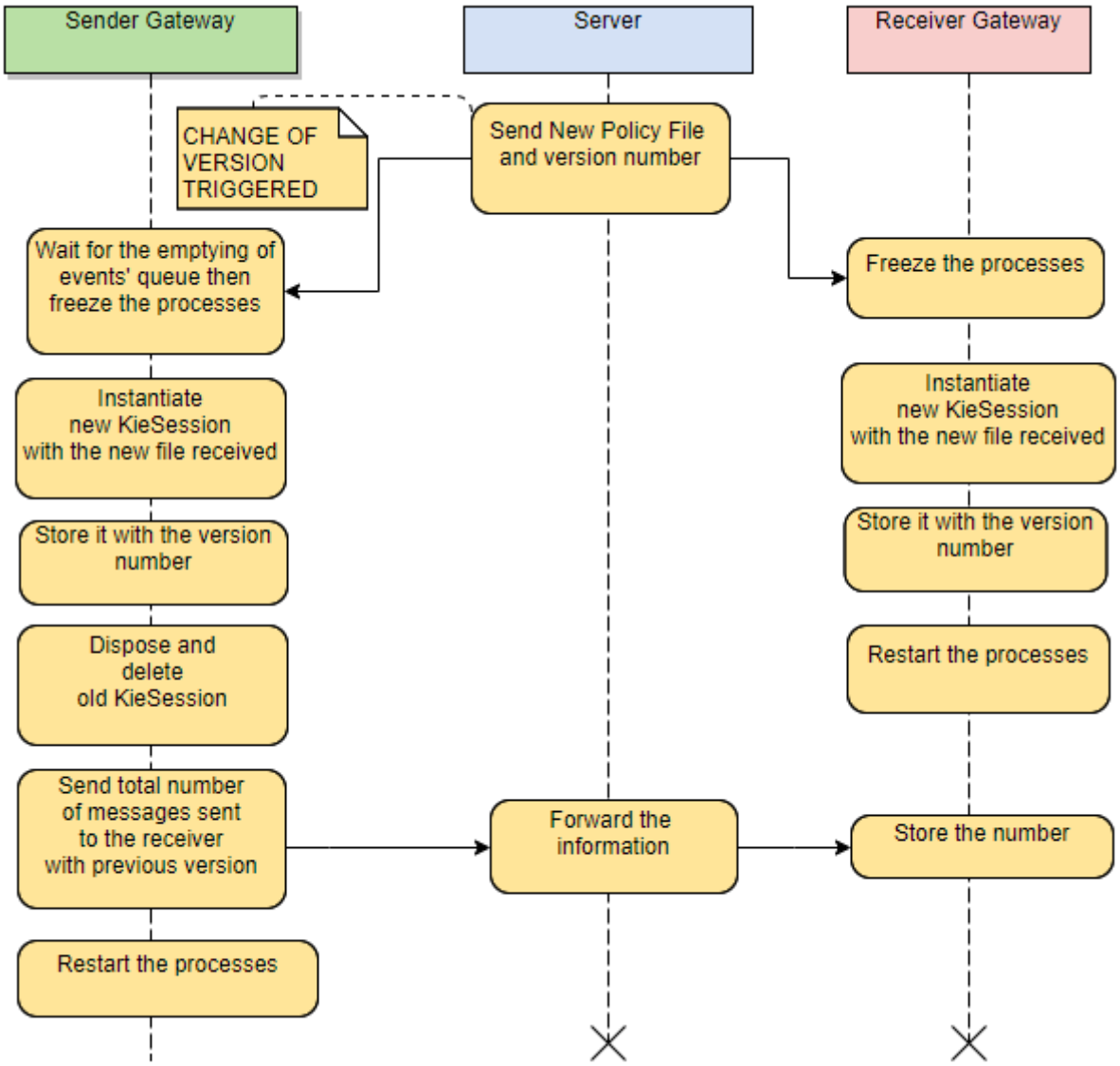

Figure 4.5: Updating protocol sequence diagram

4.4 Management protocol for stateful policies

In the presented IoT scenario controlled by policies, it is useful to define a specific kind of policy that requires not only the current data but also some previous knowledges in order to perform decisions. This kind of policy are so called *stateful policies*.

The stateful policies can offer a more accurate support for the decisions, based on the current state and on some previous ones about a certain condition. For example, this is the case of the motion detection policy. It detects the presence of people in a room and it must know if it has detected a new movement or the continuation of a prior one. The same goes for a policy that maintains a counter about how many times some actions have been triggered. For example how many people have required access to a specific area.

The procedure is different for the *stateless policies*. Any data passed through them are transitory and they will not be required any more in the future. For this reason the former kind of policy is more complex to manage.

Keeping states becomes problematic when an update of a policy is required. We analyse two different solutions for maintaining the correct state in case of the update of a new policy file.

The first one consists in storing the value of the states inside the policy files themselves. It is a more complicated solution because it involves some changes in the `KieSession`. However, on the other hand, it turns out to be handier for the developer, since there is less constraints to follow.
Indeed, the second solution exploits a Java class to store all the states' values, hence a developer must refer always to this class for managing a state.

The detail of the implementation will be explained in the Subsection 5.2.4 of Chapter 5.

Chapter 5

Proof of Concept (PoC) and implementation

This Chapter describes the key points of our Proof of Concept and its implementation. The Section 5.1 gives an overview of the structure. The Section 5.2 goes into details of the PoC development. The Subsection 5.2.1 presents the VersaSense's MicroPnP, the platform used for the PoC realization. The Subsection 5.2.2 explains how we realize the exchange of policy files between the nodes. The Subsection 5.2.3 describes the implementation's technicality of the protocol developed as a solution of our principal use case, already introduced in Section 4.3. Finally, the Subsection 5.2.4 describes in detail how we correctly manage the stateful policies.

5.1 PoC structure overview

Based on the architecture and design illustrated in the Chapter 4, a prototype implementation has been developed. This prototype includes all the mechanisms and steps of the protocol described before, crated to support this thesis and to compute the measurements.

It is composed by the following components:

- Back-end server: it is the core of the architecture. It accepts the connections to the gateways, it allows the communication among the different nodes in the system, it collects all information sent by the gateways and fire policies on the data received. With an appropriate User Interface the server triggers the upload of a new policy file that will be sent to the gateways.
- Gateway1: it is connected to two sensors and receives data about the environmental conditions. It requests the connection to the server, it collects data from the sensors, fires rules on them and forwards to the server.
  Furthermore, it plays the role of the sender that transmits the encrypted data to the receiver.
- Gateway2: it is connected to two sensors and it performs the same actions of the Gateway1. Additionally, it is the receiver that executes the decryption phase of the protocol.

5.2 PoC development

Our Proof of Concept has been developed using Java programming language together with RedHat Drools version 7.15 as Policy System.
It is composed by three different parts, one for each component mentioned in the previous Section: the server and the two gateways.
This Section describes the development's details, in particular how the most important technologies adopted, Java RMI and the Policy System Drools, are used in these three parts of our prototype.

As far as the communication channel are concerned, we use the java.rmi library of Java RMI technology.
As Java RMI requires, we define the remote interfaces of the server and of the two gateways that contain the methods that they provide. They are called `ServerInterface.java`, `ClientInterface.java` and `Client1Interface.java`.
These interfaces extend the predefined interface **Remote** which belongs to the java.rmi.Remote package.
Then, for each part, we develop the class that implements the methods defined in the corresponding interface: `ServerImplementation.java`, `ClientImplementation.java` and `Client1Implementation.java`.

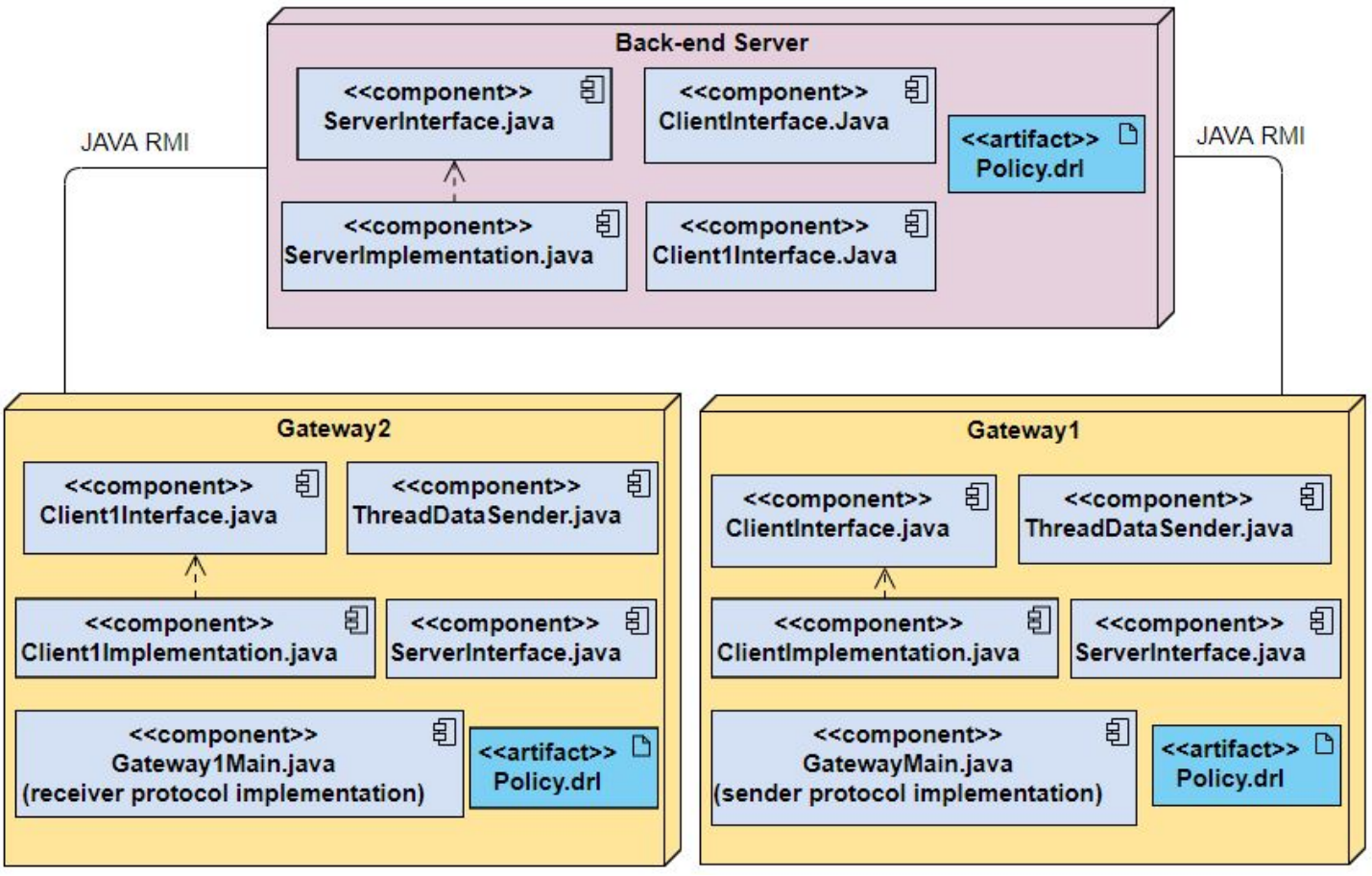

Figure 5.1: Prototype component diagram

In addition, the two gateways need to create a separate thread (`ThreadDataSender.java`), in order to split the two main tasks that must be performed simultaneously. The main flow receives and stores the environmental data from the sensors and made them available for the thread in a shared list. The thread, in turn, takes these data, processes them, fires rules on them and forwards to the server. Furthermore, these threads are also in charge of blocking any elaborations of the data and performing the operations for the upload of a new policy file when it is requested.
The Figure 5.1 shown the overall prototype's implementation.

At the server side, the prototype provides a basic Graphical User Interface (GUI) (shown in Figure 5.2) that runs on a different thread. Through this, the user can trigger the upload of a new policy file at the gateway side.

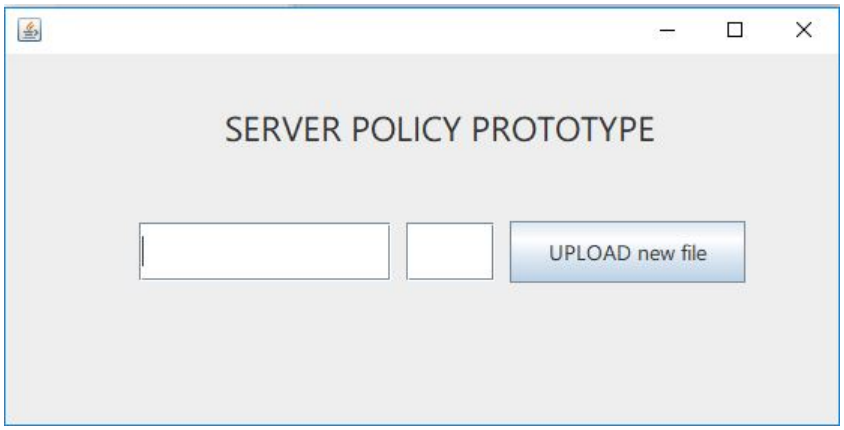

Figure 5.2: Prototype Graphical User Interface

At the same time, another thread creates an instance of the remote object's implementation class (`ServerImplementation`), exports it and binds that instance to a name (`Server`) in the Java RMI registry. These are the fundamental steps of Java RMI protocol as explained in Section 4.2. At this point the server is ready to accept the gateways' connections.

```
ServerInterface stub =
(ServerInterface)UnicastRemoteObject.exportObject(new ServerImplementation(),
0);
Registry registry = LocateRegistry.getRegistry(1099);
registry.bind("Server", stub);
```

As soon as a remote object is registered, the gateways can search for the object by name, in order to obtain a reference to the remote object and call its methods.

```
   Registry registry = LocateRegistry.getRegistry("172.16.0.3", 1099);
h = (ServerInterface) registry.lookup("Server");
```

The gateway obtains the stub for the registry by calling the static `LocateRegistry.getRegistry`
This method returns a reference to the remote object Registry on the host specified by its IP address and on the port number 1099.

Then, the gateway calls the method lookup on the registry to retrieve the stub for the remote object [28]. After these steps the gateways can call the methods provided by the server.
Similar procedures are required at the other side for establish a bidirectional communication.

As far as the Drools technology are concerned, in Section 2.2.2 we have already described the main features of the `KIE API` library used for implementing Drools on the prototype. We dedicate the Subsection 5.2.3 for describing in detail how we exploit the power of this technology in the encryption/decryption use case.

5.2.1 Versasense's Micro Plug-and Play for the gateway implementation

While the server part of the prototype runs on the laptop, the gateway part runs on a real gateway: the Versasense's Micro Plug-and Play (MicroPnP) gateway. Once this part is tested and validated on the laptop, we proceed with uploading it on the gateway. It provides its own Wi-Fi network to which we connect the laptop. We establish an SSH connection using PuTTY, that is a software terminal emulator that can control the running of the prototype and allows the file transfer.
As already mentioned, the use of Versasense's Micro Plug-and Play platform allows to test our protocol in a real IoT setting and to check if the consistency feature is preserved. Furthermore, with its devices we analyse the functionality of the technologies used and we perform all the tests and measurements for the validation and evaluation part.

About the hardware specification, this device presents: 1 Gb of RAM and around 10 GB of flash memory.

5.2.2 The procedure for a dynamic update of policies

All the devices involved in the prototype start with a default policy file already present in memory. It is uploaded in the `KieSession` when this latter is built, as one of the very first operations performed for setting up the environment.

As already mentioned, every time it is needed, we give the possibility to trigger the update of a new policy that can change the behaviour of the system's components. Since a policy consists in a set of rules collected in a file, we achieve this by transferring a chosen file from the server to the gateways. The set of available files that a user, operating on the server, can send to the gateways is already present in the server disk.

For achieving the transfer of files we use the `java.io` package that allows to accomplish input and output [29] of data in Java. It works with the concept of stream, a flow of data with a defined source and destination. The server uses

the classes `FileInputStream`, provided by the package, that accepts as parameter of the constructor the name of the file we want to transmit. With the method `read(byte[] r)`, the bytes of the specified file are read into an array. The server will then use it as parameter to call the remote method of the gateway `receiveFile`. This method accepts as parameter also the name of the file, the number of the bytes to transmit and the number of the policy version. At the gateway side, this method creates a new file and instantiates an object of the `FileOutputStream` class that allows the writing of the received bytes. It uses the `write(byte[] r)` method to write the stream of received bytes on the just created file. In this way, the policy file is transmitted properly.

Together with the file itself, another important piece of information that they exchange is the number of policy version of this file. Once the file has been received by the gateway, it can call the update procedure that will instantiate a new `KieSession`. The gateway adds a new element in a map that will contain this new file as value and the specified number version as key.

In the next Section we describe the details about the encryption and decryption protocol that uses these procedures to exchange the policy files between the nodes. This is a fundamental step to achieve an effective dynamic update of policies.

5.2.3 Distributed and consistent protocol implementation

In this Subsection we give some implementation details of the protocol presented in Section 4.3. We highlight the key points that allow to achieve the consistency among the data and the operations during every phases of the protocol.

In our prototype we use two gateways for the encryption and decryption phases respectively and the server in between, that allows the communication within the two. Since the transfer of the data is very fast within our prototype, in order to introduce a more accurate reflection of the reality, the server stores the data received from the encryption gateway for few seconds, before sending them to the decryption gateway.

First of all, we define the kind of data our protocol aim at exchanging. They are defined by five elements: the message we want to encrypt (or in general to process), the version, the details about the source and the destination (sender and receiver) and a counter:

`EncryptString(String message, int version, String sender, String receiver, int counter)`.

Standard phase

As already described in Section 4.3, the standard phase of the protocol consists in the encryption of a message by one gateway, the consequent transmission and the decryption by another gateway. In order to work properly, both gateways store the currently version number of the algorithm in use for encrypting and decrypting the messages in a variable `flagVersion`.

As already said, different versions of the algorithm correspond to different policy files. To keep track of all these files and in order to use the right one, the gateways build one `KieSession` for each version. Both gateways use a map that stores the version number as key and the corresponding `KieSession`, that includes a policy file, as value: `Map<Integer, KieSession> ksessionVer`.

Another important operation performed by both sides is keeping track of how many messages the gateways effectively exchanged. For this purpose, both gateways instantiate another map that store as key the *version number* and as value a *counter* that progressively increases based on how many messages they has respectively sent or received of that particular version. They are respectively:
`Map<String, Integer> dataSent` and `Map<String, Integer> dataReceived`.
They will be referred with these names in this Section.

At the start of the protocol, before any other operations, the sender (or encryption) gateway checks the state of the process (active or frozen) looking at the value of the `flagUpdate` variable. If it is equal to 0, it means that the process is active and it can generate the data to send filling the version field with the content of the `flagVersion` variable. After, the sender processes the message firing a rule that simply prints a line indicating if the version of the Drools file used corresponds to the `version` field of the message. It then increments the value in the `dataSent` map corresponding to the current version used. Sequentially, it transmits the message to the server, using the remote method provided by the server: `h.sendString(message)`.

At the other side, the gateway that performs the decryption calls the remote method of the server `h.sendStringForDecryption()` that allows to receive the data and inserts them on a list. To send the data, the server calls the remote method of the receiver `c1.sendToDecrypt(message)`. As for the sender, before starting the processing of any message, the receiver must check if the process state is active, looking at the `flagUpdate` variable. If so, the receiver retrieves the `version` field of the message and uses it to get the correct `KieSession` from the map and fires the rules (i.e. performs the decryption):
`ksessionVer.get(data.getVersion()).fireAllRules()`.
Again, it simply consists in detecting the conformity between the `version` field of the message and the version of the Drools file used. Thereafter, it increments the corresponding value in the `dataReceived` map.

During the decryption phase, since we have to consider the chance of inconsistency between the two gateways, it is necessary to check, for every data in the list, if the map of the receiver already contains the `KieSession` with version needed by the new data received, before performing any actions:
`if(ksessionVer.containsKey(data.getVersion())) {...}`
If the requested version of the `KieSession` is available, the decryption gateway can fire the rules. Then, it deletes the already processed message from the list in order to avoid multiple rule evaluations on the same data.
If the map does not contain the requested version, the gateway just loops in the `while`, during which the list stores all data that are arriving. Indeed, the list works as a buffer when the gateway waits for the uploading of the new version.
When the requested version becomes available all the events in the list will be

processed and, successively, deleted.

Updating phase

The delicate updating phase starts when the server sends a new policy file to the gateways. They perform the procedure described in Subsection 5.2.2.
Next, the sender gateway processes all the messages already present in the queue and interrupts the creating and the sending of new data. It assigns the value 1 at the `flagUpdate` variable, that means that all the processes are frozen.
After that, it instantiates a new `KieSession` with the new policy file just received from the server and stores it in the map with the associated version. Next, it sends to the server, that will in turn forward to the receiver, the total number of data encrypted with the previous version just replaced, using the remote method of the server `h.numberEvent()`. At this moment, it can safely dispose the old `KieSession`, because it is sure that it will never use it any more.
Then, the gateway assigns the value 0 at the `flagUpdate` and starts again the procedures of encryption and sending of data with the new version.

At the other side, the upload of the file and the instantiation of a new `KieSession` works the same way. Right after that, the receiver restarts the standard process.
Clearly, the only difference is that for the receiver gateway the disposal of the previous `KieSession` is a bit more tricky than for the sender gateway.
It cannot be sure than every data of the prior version have already been received. This circumstance can occur when the network is delayed for overloading or any other problems. As already mentioned, to tackle this eventuality, during the standard phase, the receiver gateway progressively counts the number of messages received and it stores the value in the `dataReceived` map. Besides, it instantiates another map to store the number of total messages sent by the sender during the update phase. These two map have the same key (the *version number*). Instead, as value, the first map as the increasing number of messages received so far from the sender. The second one as a value that stays null until the sender will send the total number of messages processed with that version.
In the standard phase, the receiver compares these two values in the maps for every stored key.
If these numbers are equal for a certain key, the receiver is confident that no data have been left behind and we can safely dispose that particular `KieSession`.

Multiple senders and multiple receivers case

In the previous Subsection we have explained a simpler case that considers one single sender and receiver. We introduces in this Subsection the implementation of our PoC in case of multiple senders and receivers.

The overall protocol is basically the same with some additions in order to manage the multiplicity of senders/receivers.
A sender gateway must store not only how many data has sent with which version, but also to which receiver. Therefore, we instantiate a map:
`Map<String, Integer> dataSent` that contains *code of the receiver + the version*

as key and *total number of data sent* as value.
For example: `dataSent.put("R01,"+1, 5);`.
This map is similar to the one instantiated in the previous scenario, with the addition of the code of the receiver in the key.

During the upload of a new version, the sender gateway sends to the receivers its own code, the version that it is replacing and the total number of data that it has sent with this version of the algorithm.
A receiver will use all these data in order to keep track of the messages received. It instantiates two different maps: one for counting progressively and storing the number of elements received from a sender of a particular version and the second one for maintaining the number received by the sender regarding the total number of messages sent. The key are the same for both maps (*code of the sender + version*), while, for the first map, the value increases any time a new data is received from a specific sender and for the second it stays null until the sender will send the value. Again, these two maps are similar to the previous case, with the extra information about the code of the sender in the key.

Furthermore, the receiver must know how many senders there are in total that can use a specific version. The receiver can dispose a `KieSession` only when for all the senders that have used a specific version the values of the two maps are equal.

5.2.4 Implementation of stateful policy

As introduced in Subsection 4.4, we research how to handle the stateful policies in a consistent way. We analyse two procedures with their advantages and drawbacks.

The first one consists in the instantiation of a class, called `Fact`, inside the Drools file itself (.drl). It simply has a name and fields in which it stores the different states. Inside a rule, we can define a new object of this class or modify an existing one with the required values. The example below shows a `Fact` in a policy that stores the states about the previous movement and the activation of the camera:

```
declare States
movementState :  int
cameraState:  int
end
```

In this way we do not need the support of a specific Java class but when the replacing of the file is required, we need to correctly retrieve this object with all its fields and store it in the new updated policy file. As we can see, the update phase is more delicate than before and require more attention.

In the second procedure analysed, every component of the PoC presents a class called `StatePolicy`. It includes as a field a map with a *string* for both key and value. It is used to store the name of the state that we want to preserve and the corresponding value. Inside the Drools file, every time a rule, that is involved in a stateful policy, is fired the Drools engine can access this map and retrieve the value

of the requested state. Then, it can use it or modify it. Since this class is defined and stored at the Java level and not at the Drools level, it is independent from the replacing of the Drools policy files. We are sure that the data cannot be lost during their updates.
Even if this property makes easier all the update procedures, the main drawback is that it puts constraints on the development of the Drools files. We must always refer to this class if we want to implement a stateful policy.

In our PoC, we choose to implement the second solution because consists in a better trade-off between simplicity and efficiency.

Chapter 6

Validation

In this Chapter we demonstrate the achievements of the goals that we set in Section 3.3. In Section 6.1 we prove that we have effectively achieved dynamic updates of policies throughout the system at run-time. In Section 6.2 we illustrate that the consistency property and the preserving of the state are always guaranteed during the operations. Finally, in Section 6.3 we perform some tests on the standard and on some problematic scenarios. We demonstrate the correct management of some temporary inconsistent conditions that can happen in different situations.

6.1 Effective dynamic update of policies

In the Subsection 5.2.2 of Chapter 5 we described the procedure chosen for the transfer of a policy file and, therefore, for the update of the policy in use. It is worth noting that we can easily achieve an effective dynamic update of policies thanks to the capabilities of Drools to operate with simple files. It would not be possible in a system that manages policies using another approach, for example that defines hard-coded policies in the system or uses some specific formats.
Drools, instead, allows to handle the policies using the file streams, a straightforward and well-known procedure that we can therefore leverage.

Furthermore, the `Kie API` allows to instantiate and dispose `KieSession` in a dynamic way at run-time, without the need of restarting the application.

The combination of these two functionalities allows us to accomplish efficiently our first goal and requirement for our middleware.

6.2 Achievement of consistent updates

As presented in the previous Chapter, all the mechanisms involved in the correct execution of the application, occur simultaneously during the continuous management of the messages. Clearly, these actions take time and resources. Therefore, another objective has been to find a way to correctly handle all the operations that running at the same time and, most important, to preserve the consistency among them. This is another important step and another goal achieved.

We achieve an accurate and consistent update of the policies file through the actuation of a straightforward protocol. It takes care of all the steps and combine them in a coherent way. The most important concept is that all the devices involved need to keep track of the number and type of data exchanged. These are easy-reachable but also fundamental pieces of information that lie at the heart of the consistency achievement. Unquestionably, this consistency property that we guarantee comes at cost of reserving memory for the additional data, managing them and send them through the network. We consider it as a good trade-off between the functionalities offered and the effort made to obtained them.

In the more frequent context of a large scale system, it can often happen that a gateway is not reachable any more due to network congestion or any infrastructure problems. This issue leads to an interruption in the flow of messages from the problematic node to the others and to a temporary arrest of its functionalities. For how our protocol is designed, even if the main functions are interrupted, the gateway keeps on working in a consistent way, until the situation will recover and it will restart to work properly.
During the update phase, thanks to the additional piece of information that the messages carry with them, even if the problematic gateway has not received the policy file from the server yet, it is impossible that it uses the incorrect policy to process a message. Before performing any operations, the gateway checks if the version of the requested policy file is available in its memory. If it does not have received the requested policy file from the server yet, it simply stores the messages as they are and waits for the recovering of the network connection that will guarantee the receiving of the new policy file.

Besides the consistency among the operations, we also research and achieve the consistency among the single policy itself. In particular, to those kind of policies called *stateful policies* that need a careful and specific processing. Their decisions are not based only on the current value of some variable, but also on some previous ones that need to be stored and maintained: the state. Our protocol allows to preserve this state consistently even in case of update of the policy file.
We have experimented two different approaches. The first one is based on the functionalities offered by the `Kie API`, while the second one leverage the use of an extra Java class to store the values of the state. We analyse the trade-off of these two approaches. The former preserves the state inside the Drools file but leads to a more complex management during the uploading phase of the protocol. The latter does not need any additional procedures during the update process but establishes some constraints to the developer of the policies. Indeed, the class in charge of storing the state, even if is as general as possible, is determined upfront and cannot be changed.

6.3 Validation tests

In this Section we show the validation tests that we have performed to demonstrate the effective achievement of our main goal: the consistent management of dynamic policies throughout the nodes and the system's operations. We still refer to the encryption and decryption use case. We first describe the standard scenario in which no problematic situations occur. Then, we introduce three problematic scenarios that could generate issues. They refer to those situations in which the receiving of a new policy file happens during an intermediate phase, when the processing of a message has already started but not finished yet.

During the execution of our application, we log the timestamps in which the principal steps occur. They are indicated in capital letter in the sequence diagrams below, used for describing the scenarios. The meanings are listed in Table 6.1.

A	Standard phase: Creation of a new message by the sender
B	Standard phase: Encryption performed and sending to the server
C	Standard phase: Message arrived at the server
D	Standard phase: Message sent to the receiver
E	Standard phase: Message received by the receiver from the server
F	Standard phase: Decryption of message
G	Update phase: The server starts sending a new policy file
H	Update phase: Transmission completed to both sides
$\mathbf{I_S}$	Update phase: The sender completes the update
$\mathbf{I_R}$	Update phase: The receiver completes the update

Table 6.1: Table of the principal steps

6.3.1 Standard scenario

First of all, as a first experiment, we log the points in time when the different events happen in a scenario that do not present any hazardous situations, problems or delays. In the Figure 6.1 we plot the sequence diagram performed by the node involved: the sender, the receiver and the server in between. We collect the timestamps in Table 6.2. As can be seen from the sequence diagram, the start of update phase occurs when both the sender and the receiver have already terminated the elaboration of a message. In this case, we notice that also in the Table the point in time G, when the update phase starts, follows the points in time B and F, that indicate the time in which the nodes end the management of the message. That means that the queues of events at both sides are empty and the inconsistent situations are avoided. A message that leaves the sender elaborated with a particular version of a policy will be processed at the receiver side with exactly the same version.

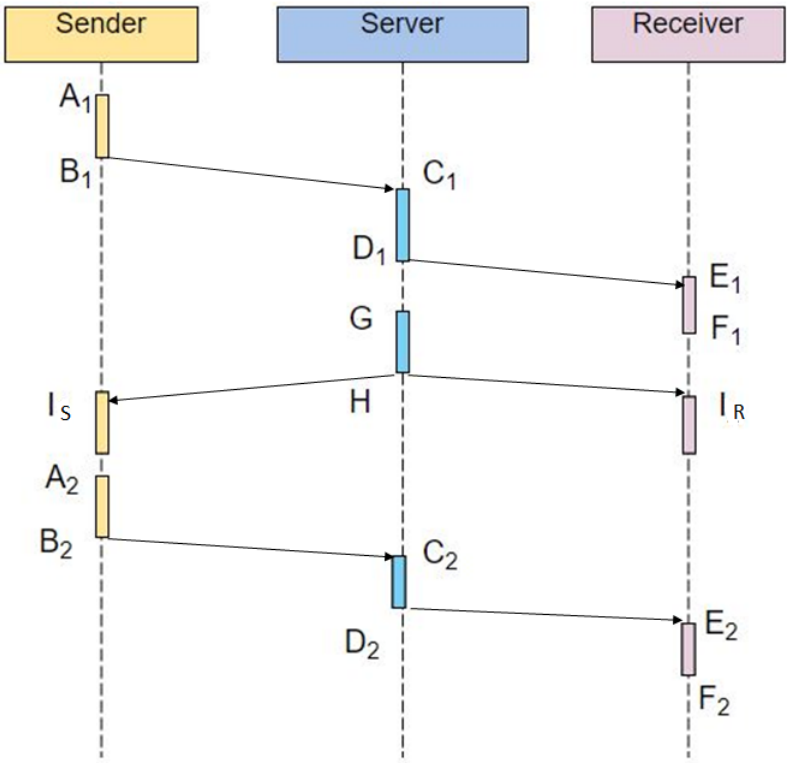

Figure 6.1: Sequence diagram of the standard scenario

$\mathbf{A_1}$	14:46:48.257
$\mathbf{B_1}$	14:46:48.319
$\mathbf{C_1}$	14:46:48.319
$\mathbf{D_1}$	14:46:48.321
$\mathbf{E_1}$	14:46:48.321
$\mathbf{F_1}$	14:46:48.372
G	14:47:02.411
H	14:47:02.761
$\mathbf{I_S}$	14:47:02.548
$\mathbf{I_R}$	14:47:02.761
$\mathbf{A_2}$	14:47:08.319
$\mathbf{B_2}$	14:47:08.324
$\mathbf{C_2}$	14:47:08.323
$\mathbf{D_2}$	14:47:08.324
$\mathbf{E_2}$	14:47:08.324
$\mathbf{F_2}$	14:47:08.327

Table 6.2: Table of the timestamp of the standard scenario

6.3.2 First problematic scenario: delay on the sender

In a real system, a possible scenario that can occur is the update of a policy file during the elaboration of a message at the sender side. In particular an update can happen between the creation of a message (point A) and its encryption and sending (point B). This is a critical situation because the sender has already created a message and it presumes that it will be encrypted with the current policy file not the new one that is arriving.
In order to validate our protocol in this problematic scenario, we induce a delay of 20 seconds between the points in time A and B. In this time frame we trigger the update of a new policy file and we analyse the actions performed.
The Figure 6.2 shows the sequence diagram of the described scenario. We first insert an ordinary exchange of a message and the beginning of a new one (A_2). Then, at point G, an update of a policy is triggered. As it is possible to read in the Table 6.3, it happens after the time A_2, in which the sender creates a message, but before the time B_2, in which the same message will be encrypted and sent. During the execution we observed that at point B_2, that occurs after the point in time I_1, the message is encrypted with the correct and new version, even though it was created before the update. The same happens at the receiver side, resulting in accurate retrieving of the data.
This correct management of the messages is preserved thanks to the extra information available for the nodes. The consistency is guaranteed by the fact that every message brings with it the number of version which it has to be processed with.

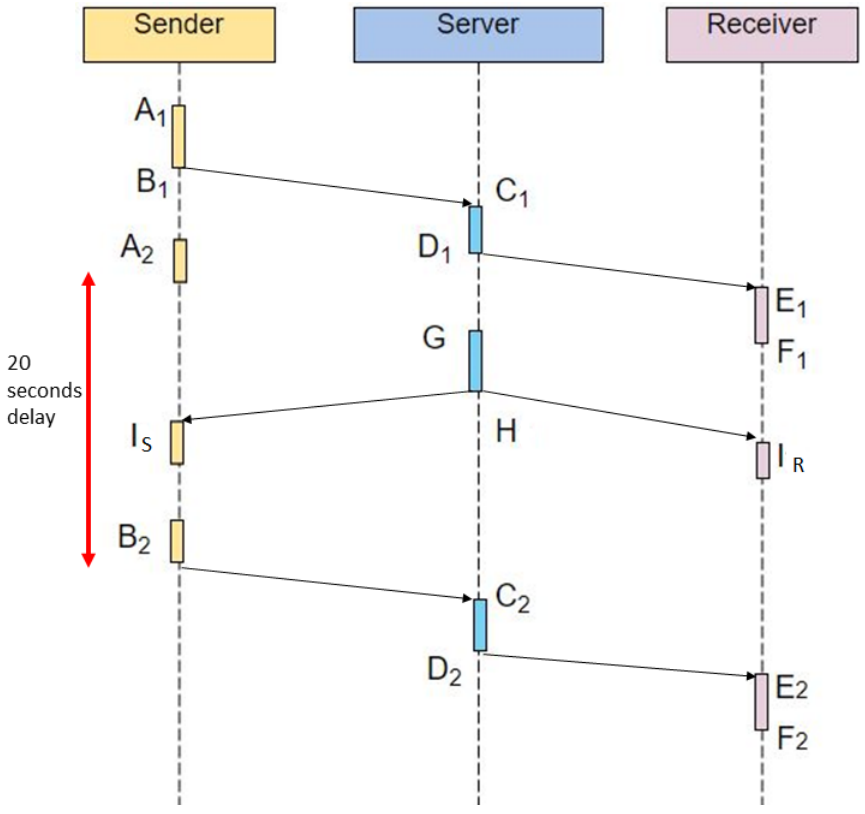

Figure 6.2: Sequence diagram of the first problematic scenario

A_1	17:08:01.512
B_1	17:08:21.819
C_1	17:08:21.819
D_1	17:08:21.826
E_1	17:08:21.826
F_1	17:08:21.974
A_2	17:08:31.821
G	17:08:39.451
H	17:08:41.090
I_S	17:08:41.018
I_R	17:08:41.090
B_2	17:08:51.827
C_2	17:08:51.827
D_2	17:08:51.829
E_2	17:08:51.829
F_2	17:08:51.831

Table 6.3: Table of the timestamp of the first problematic scenario

6.3.3 Second problematic scenario: delay of an update at receiver side

We analyse another problematic scenario that consists in a delay in the receiving of a new policy file at the receiver side. In particular, the sender receives immediately the new file from the server, while the receiver has to wait some time.
This experiment simulates the case in which a node in the system is temporarily not reachable, due to some connectivity problems or to an overload of the network. Similar to the previous scenario, for validating our protocol in this situation, we induce a delay of 20 seconds at the server side in the sending of the file to the receiver.
The sequence diagram in Figure 6.3 shows the events that occur. We split the transfer of the new policy file in two different moments. At H_1 the server immediately sends the file to the sender, but it sends it to the receiver at H_2, 20 seconds later. Meanwhile, the sender starts again the standard procedure of crating, encrypting and sending a message, but this time with the new policy file that the receiver does not own yet. Therefore, the points E and F cannot be performed consecutively and immediately one after the other, like in the standard scenario. When the receiver accepts the messages (point E_2), it detects that the version required is not available yet. For this reason, it stores them in a queue until the point in time H_2 occurs. At this moment, the receiver updates the policy and elaborates all the messages present in the queue. Like the previous scenario, we avoid inconsistency thanks to the extra piece of information, indicating which version the messages required.

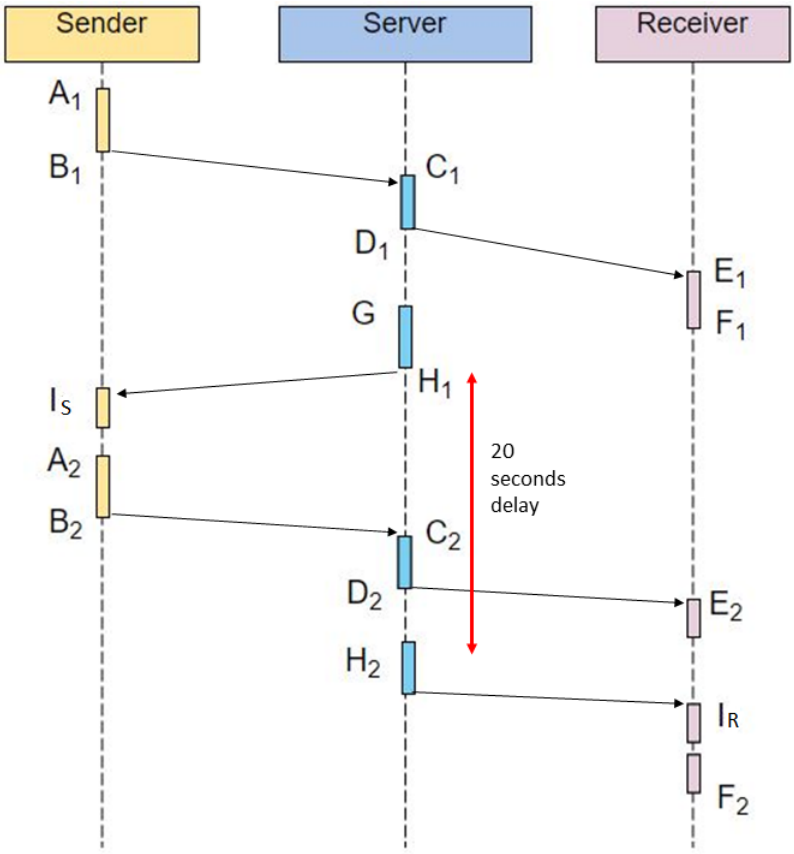

Figure 6.3: Sequence diagram of the second problematic scenario

$\mathbf{A_1}$	15:38:10.446
$\mathbf{B_1}$	15:38:10.521
$\mathbf{C_1}$	15:38:10.521
$\mathbf{D_1}$	15:38:10.524
$\mathbf{E_1}$	15:38:10.524
$\mathbf{F_1}$	15:38:10.578
$\mathbf{G}$	15:38:19.117
$\mathbf{H_1}$	15:38:19.308
$\mathbf{I_S}$	15:38:19.308
$\mathbf{A_2}$	15:38:20.522
$\mathbf{B_2}$	15:38:20.529
$\mathbf{C_2}$	15:38:20.529
$\mathbf{D_2}$	15:38:20.530
$\mathbf{E_2}$	15:38:10.530
$\mathbf{H_2}$	15:38:39.938
$\mathbf{I_R}$	15:38:39.938
$\mathbf{F_2}$	15:38:39.949

Table 6.4: Table of the timestamps of the second problematic scenario

6.3.4 Third problematic scenario: delay on message transmission

The third problematic scenario that we analyse presents a delay during the transmission of a message from the sender to the receiver. This test takes into account the case in which the communication get slower due to a network congestion. Specifically, a new message is created, encrypted and sent by the sender but it will be received by the receiver after some time, not immediately. Moreover, we want to observe if the consistency is guaranteed. We trigger the update of a new policy file during this delay and we analyse if the message in transit will still be correctly managed.
The sequence diagram in Figure 6.4 illustrates the events happening in this scenario and the corresponding Table 6.5 shows the timestamps collected during the experiment.
In order to mimic a delay in the transmission of a message, we divide the point in time B in two: B' refers to the encryption phase only, while B" to the sending of the message. The message that leaves the sender at time B_2" has been encrypted with the old policy at time B_1" because the update happens only after, at time I_1. Therefore, the message reaches the receiver at time E_2 when the update has already happened at time I_2". Since the receiver does not dispose the old version until every corresponding messages has been received, it detects the version needed by the message and retrieves the right file to process it.

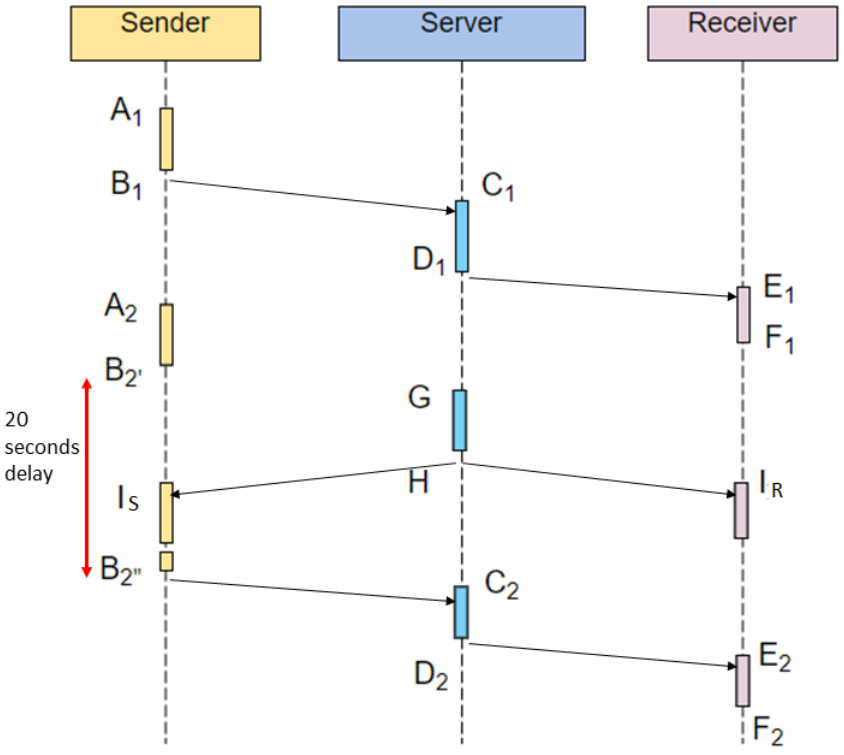

Figure 6.4: Sequence diagram of the third problematic scenario

A_1	13:27:51.209
B_1	13:28:11.343
C_1	13:28:11.343
D_1	13:28:11.349
E_1	13:28:11.349
F_1	13:28:11.547
A_2	13:28:21.345
B'_2	13:28:21.347
G	13:28:29.113
H	13:28:30.599
I_S	13:28:30.598
I_R	13:28:30.877
$B"_2$	13:28:41.350
C_2	13:28:41.350
D_2	13:28:41.352
E_2	13:28:41.352
F_2	13:28:41.355

Table 6.5: Table of the timestamps of the third problematic scenario

Chapter 7

Evaluation and discussion

In this Chapter we perform the evaluation of the designed protocol in the developed prototype. In Section 7.1 we discuss which failures our solution supports. In Section 7.2 and in Section 7.3, we provide some measurements and comparison about the overhead in time, in memory and on the network of the sender and receiver protocols, presented in Section 4.3, running on the laptop and on the MicroPnP VersaSense gateway. Lastly, in Section 7.4 we discuss some limitations arisen during our research and we give some suggestions for future work.

7.1 Failure model

Especially in distributed systems, in which multiple devices participate, there are several possible points of failure. In this Section we specify the failure model of the communication channel which our protocol can handle.

We insert in the Table 7.1 the general failures likely in a distributed scenario and we highlight which ones our communication channel support and which ones not.

Out of order delivery	✓
Delayed delivery	✓
Message loss	✗
Loss of power supply	✗

Table 7.1: Types of failures in a distributed system

The first three failures presented in the table are included in the more general network failure. In this case, the devices are still running but the communication channel is interrupted or does not work properly.
Our protocol supports the delivery of the messages in a wrong order because what is important for the success of our solution is the total number of messages received at the other side, not the specific order. For that reason, we assume that in our

solution every message arrives and, moreover, exactly once. In case of message loss, the sender will not send again the same message and, consequently, the receiver cannot dispose the corresponding `KIESession`, keeping it in memory. As far as the delayed delivery failure is concerned, we discuss extensively about it in the Chapter 6. This is the failure in which we more focus on and we demonstrate that it is supported by our solution.
A possible failure not supported in our protocol is the loss of power supply. We do not investigate the problem of the power consumption, even though we are aware that in some systems the gateways could be battery-powered, therefore limited in energy. In our research we assume that this problem does not exist and that the communication channels established are always operating.

7.2 Time overhead

In order to evaluate the time overhead in our solution, we perform some measurements regarding the crucial phases carried out by the Policy System Drools, in particular the use and the management of the policy files.

We use Box Plot diagrams to show our results. It is a method for graphically delineating a dataset based on five numbers: minimum, first quartile (Q1), median, third quartile (Q3) and maximum. In the diagram we plot a rectangle that traverses the first quartile to the third quartile. The line inside the rectangle shows the median and the lines above and below the box show the locations of the smallest and the highest values, outliers excluded.
In particular, the *first quartile (Q1/25th Percentile)* is the middle number between the smallest and the median numbers of the dataset, the second number is the *median (Q2/50th Percentile)*,the middle value, the third *third quartile (Q3/75th Percentile)* is the middle value between the median and the highest value of the dataset.
The Box Plot can also have lines (whiskers) that indicate the degree of dispersion and dots that represent the outliers.
To obtain their representation we need to calculate the *interquartile range (IQR)* that is the difference between the first and third quartiles. Thereafter, we compute the *maximum* as Q3+1,5*IQR and the *minimum* as Q1-1,5*IQR. These numbers are important in order to detect the outliers: they are the values either above the maximum or below the minimum and instead of being shown using the whiskers, outliers are shown as separately plotted points [30],[31].

The Figure 7.1 shows the Box Plots about the milliseconds (on the y axis) required by the server and gateway part of the prototype to set up the `KIESession` on the laptop and on the real gateway. As described in Section 2.2.3, this is a fundamental step in order to start the interaction with the Drools rule engine.
We performed approximately thirty evaluations. It is possible to see outliers in all of the Box Plots.

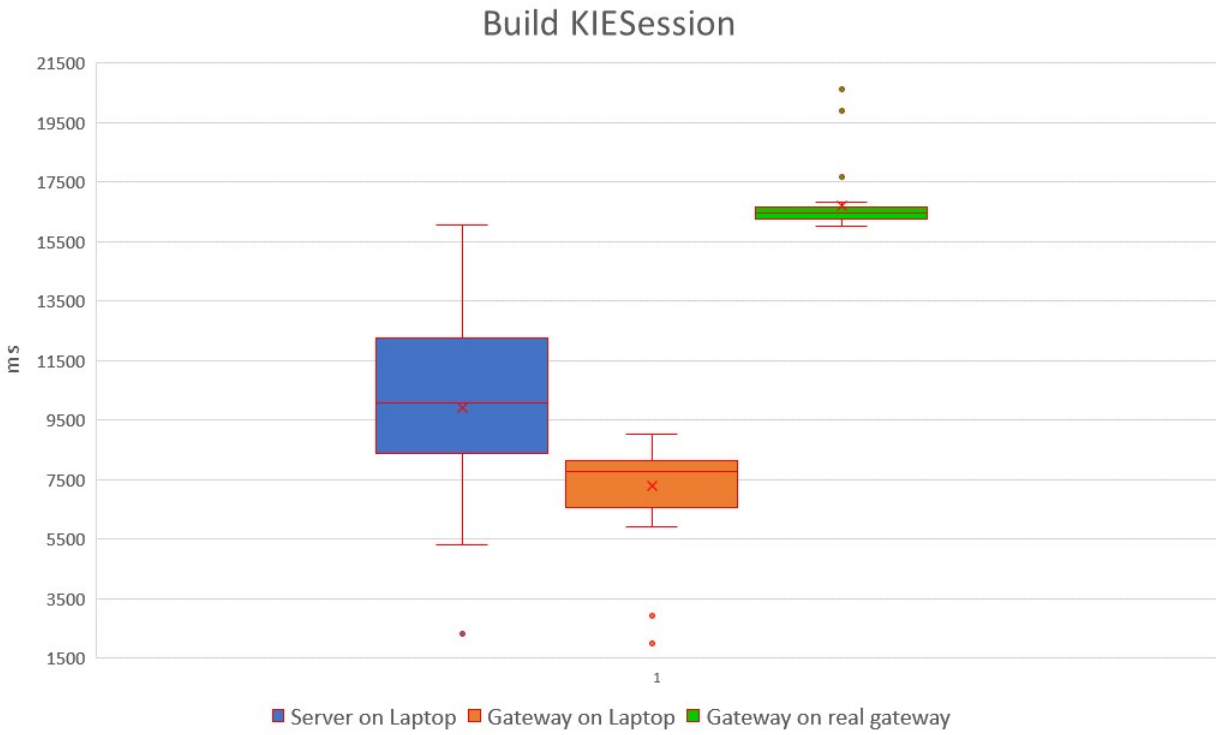

Figure 7.1: Box Plots that represent the time overhead for instantiating the `KIESession` on laptop for the server part and for the gateway part and on the real gateway for the gateway part of the PoC

	Server (ms)	Gateway (ms)	VersaSense Gateway (ms)
Smallest	2314	1976	16003
Q1	8384,25	6572	16261,8
Median	10059	7784,5	16449
Q3	12267,25	8118,5	16645,8
Highest	16034	9021	20635
Mean	9929,55	7283,82	16686,4
Range	13720	7045	4632
IQR	3883,5	1546,5	384
Minimum	2559	4252,25	15685,8
Maximum	18093	10438,3	17221,8

Table 7.2: Box Plots values of the server and gateway parts running on laptop and on real gateway that refer to the instantiation of a new `KIESession`

The Table 7.2 contains the values in milliseconds of our measurements that allows to draw the Box Plots diagram for the building of the `KIESession`. We added the mean among all the values and the range that is the difference between the highest and smallest values.

The most interesting Box Plots to compare in the Figure 7.1 are the orange and the green one, corresponding to the same piece of software running, respectively, on the laptop and on the real gateway. The VersaSense gateway is characterized by different cpu power, capacity and memory than the laptop. We notice than the amount of time needed is about 10 seconds more and the presence of outliers is more clear, with a marked distance from the median, probably due to a stronger variability in time of the internal processes.
We can also observe that the server part of the prototype (blue Box Plot) takes few seconds more than the gateway part running on the laptop, even if the lines of code used to build the `KIESession` are the same in both parts. A possible explanation could be that the server have to constantly managed the communications among the devices as well and that can overload the system.

Together with the setting up of the `KIESession`, another decisive step of the protocol is the update of a new policy file. As better explained in Section 2.2.3, this procedure consists in building a new `KIESession`, uploading the new file on it and, for the sender protocol, disposing the old one.

The Box Plots below, in Figure 7.2, draw the statistical values of the milliseconds required by the same procedure running on the laptop and on the real VersaSense gateway. We notice that this time the difference is about 2/3 seconds but in the real gateway we register more significant outliers, probably again due to a stronger variability.

Figure 7.2: Box Plots about the time overhead for update a policy file on laptop and on the VersaSense gateway

The Table 7.3 presents the data collected for plotting the corresponding Box Plots.

	Laptop (ms)	**VersaSense Gateway (ms)**
Smallest	140	214
Q1	174,25	264
Median	202	309,5
Q3	227,5	407,5
Highest	401	1027
Mean	211,7	381,33
Range	261	813
IQR	53,25	143,75
Minimum	94,37	48,75
Maximum	307,37	622,75

Table 7.3: Box Plots values about the update of a policy file on the gateway part on laptop and on the VersaSense gateway

Another important step is the firing of the rules. We evaluate the time needed to fire the rules on the data received according to the protocol. As described in Section 4.3, they are simple rules that check if the right version of the policy has been used to encrypt or decrypt the messages.

During the executions of our experiments we noticed that in both the gateway part of the prototype running on the laptop and on the real gateway, the milliseconds requested vary considerably based on the phase in which a rule is fired.
A rule takes around 0.2 and 0.4 seconds, respectively on laptop and on the real gateway, to be evaluated the very first time the application is running, 1 and 2 milliseconds the following evaluations of the same rule and around 10 millisecond and 40 milliseconds, the first time after a new policy file has been updated.

The Box Plots and table of statistical measurements are shown in Figure 7.3 and on the Table 7.4.

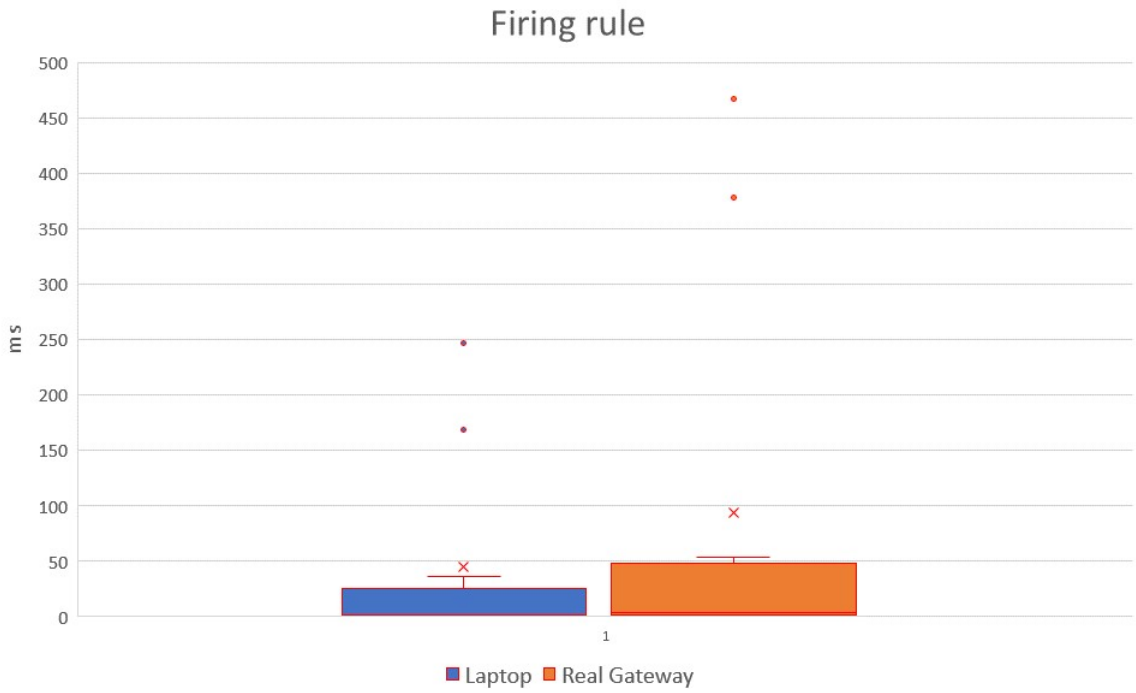

Figure 7.3: Box Plots about the time overhead for firing a rule on laptop and on real VersaSense gateway

	Laptop (ms)	**VersaSense Gateway (ms)**
Smallest	1	1
Q1	1	2
Median	2	4
Q3	26	48,5
Highest	247	468
Mean	45	93,93
Range	246	467
IQR	25	46,5
Minimum	61,5	68,75
Maximum	309,5	118,25

Table 7.4: Box plot values about firing of rules on the gateway part on laptop and on VersaSense gateway

The diagram on Figure 7.4 shows better the temporal evolution in time. The highest peaks (378 and 247 milliseconds) concern the very first running of the program, they are followed by the lower ones that regard the time required in the standard running, when the same rule is evaluated on another message. The middle peaks are referred to the first evaluation of a rule when a new policy file has just been updated.

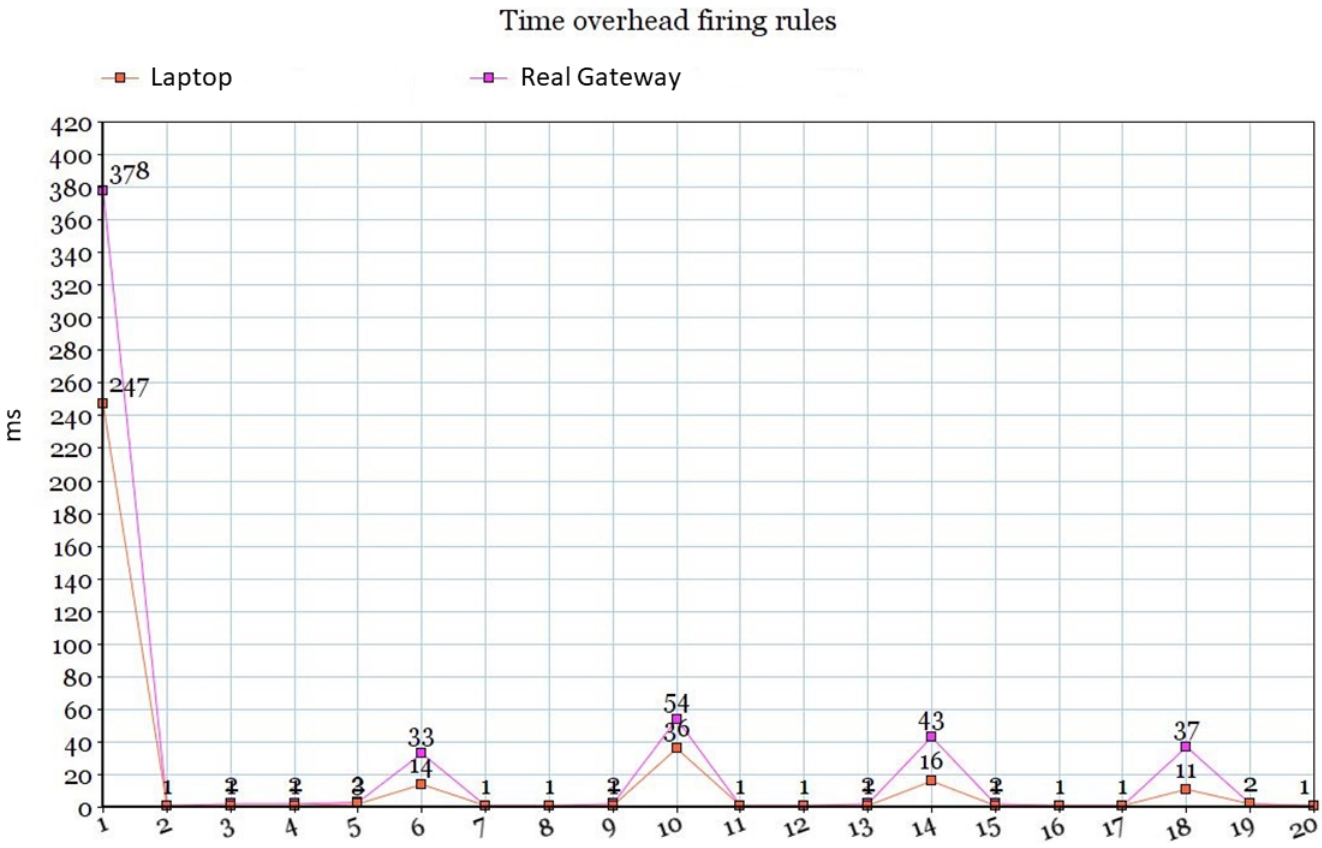

Figure 7.4: Temporal evolution of firing rules on laptop and on real VersaSense gateway. The y axis refers to the millisecond and the x axis to the number of the evaluation of a rule

Lastly, we present the overhead in time introduced by the operations of our sender and receiver protocols. We compute the difference between the amount of time required to exclusively fire the rules and the amount of extra time needed for performing all the other necessary operations. In particular, to set up the counter, the maps and to evaluate the `if` clauses as well, as described in Section 4.3.
Different operations are required at the two sides. They will be explained in the next paragraphs. We instantiate an inner and an outer timer respectively for performing the comparisons. We perform these measurements at the sender and receiver sides.

As far as the sender side is concerned, we calculate the overhead in time of the procedure to be followed in case of an update, compared to time requested to perform the standard loop, that consists in simply retrieving a new value and fire the rule on it.
The updating procedure requires, besides the update in the `KieSession` of the new file itself, the fetching from the map of the total number of messages that have to be send to a specific receiver, sending it to the server and change the `flagVersion` value.

Since these procedures are really fast, in order to make the difference discernible,

we execute these measurements in nanoseconds on both the laptop and on VersaSense gateway.
Comparing the means, we can see that the difference is about 1 order of magnitude.

The Figure 7.5 shows the relative Box Plots and the Table 7.5 presents the corresponding measurements.

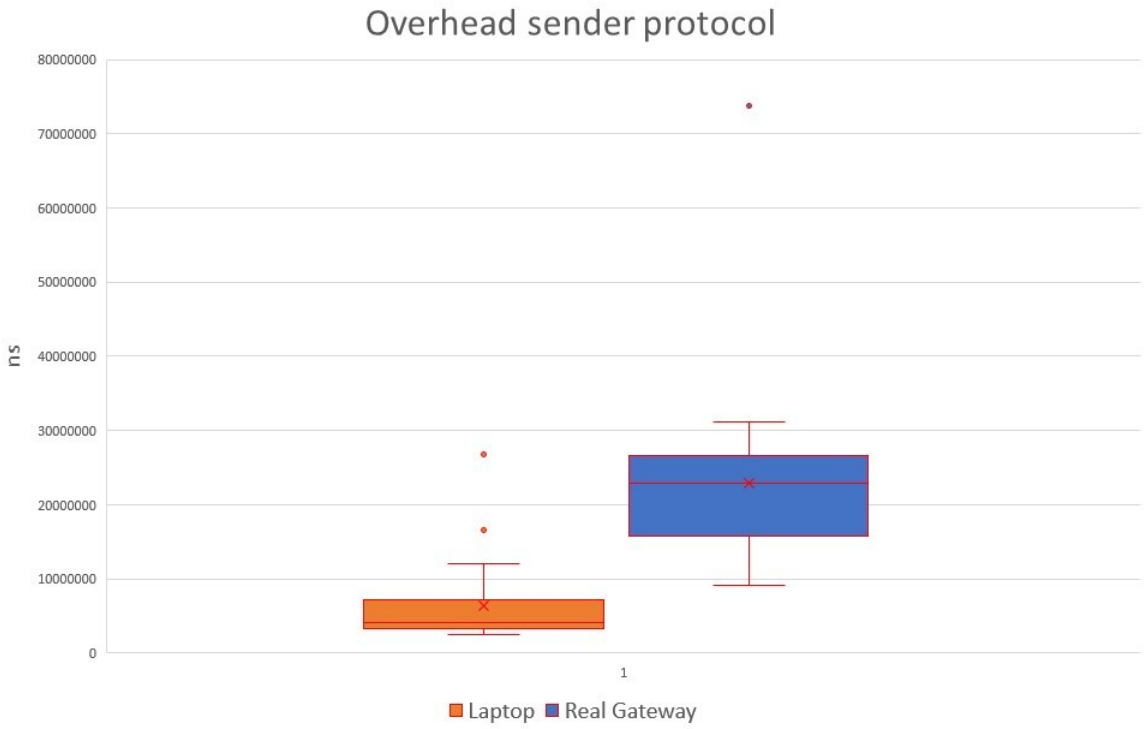

Figure 7.5: Box Plots about the time overhead of the sender protocol on laptop and VersaSense gateway

	Laptop (ns)	**VersaSense Gateway (ns)**
Smallest	2500924	9113071
Q1	3213824	15694732
Median	4017387	22901132
Q3	7125580	26676603
Highest	26857988	73839808
Mean	6397358,1	22897070,34
Range	24357064	64726737
IQR	3911756	10981871
Minimum	2653810	778074,5
Maximum	12993214	43149409,5

Table 7.5: Box Plots values about the overhead of sender protocol on the gateway part on laptop and on VersaSense gateway

Different is the case of the receiver side. As explained previously, its standard loop consists in checking the presence of available data in the queue and the if the `flagUpdating` is set to zero, it has to increment the counter in the map of the corresponding data received, fire the rule and check if there is the possibility to dispose some `KieSession` that will not be used any more in the future. We compare the time for this procedure with the evaluation of the rules only.

Again, we execute this measurements in nanoseconds on both the laptop and on VersaSense gateway. The results are shown and compared in the Box Plots on Figure 7.6 and the data are presented on the Table 7.6.

Even if the VersaSense gateway has less capacity than the laptop, we noticed a relative small difference in the time overhead. Due to a stronger variability the values of the real gateway are more sparse and leads to a mean of 1 order of magnitude larger.

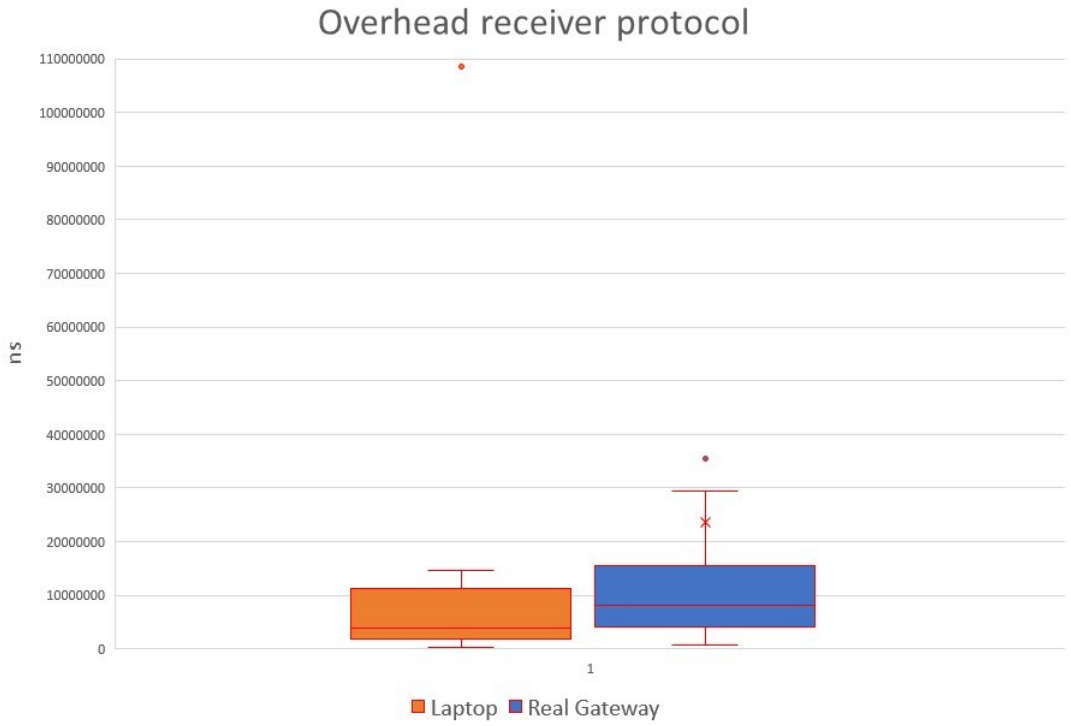

Figure 7.6: Box Plots about the time overhead of the receiver protocol on laptop and VersaSense gateway

	Laptop (ns)	VersaSense Gateway (ns)
Smallest	274982	783494
Q1	1780326	4075260
Median	3861720	8113232
Q3	11176324	1549832
Highest	1061682752	407378666
Mean	153258664,6	2366149,41
Range	1061407770	406595172
IQR	9395998	11419572
Minimum	12313671	13054098
Maximum	25270321	32624190

Table 7.6: Box Plots values about the overhead of the receiver protocol on the gateway part on laptop and on VersaSense gateway

Looking at the Box Plots and at the data in the Tables we do not detected a glaring difference between the two phases of the protocol. It could be explained due to the fact that even if the decryption phase requires more checks, they only consist in few CPU instructions that do not determine a substantial overhead.

7.3 Memory and network overhead

The memory overhead of the protocol consists in the additional structures needed for storing the messages exchanged and for preserving the information required for a correct management of them.
We give an estimation of these values using the big $\mathcal{O}$ notation that represents how a process' running space requirements grow as the input size grows.

All the components of the system store two queues of events modelled as arrays: one during the ordinary evaluation of the rules and another one that store the events arriving during the updating phase. They grow as $\mathcal{O}$*(number of message sent or received)*. Furthermore, all the gateways instantiate a map that stores the version number and the `KIESession` associated. It grows depends on the number of the versions used, so it is $\mathcal{O}$*(number of current versions)*. Therefore, it is linear to the number of version used at that moment.

In the more complex scenario of multiple senders and receivers, a sender (or encryption) gateway has also to maintain how many data it has sent to which receiver and with which version. For that reason, another map is instantiated that grows like $\mathcal{O}$*(number of the combinations of number of version + ID of receiver)*.
At the other side, the receiver (or decryption) gateway has to manage two maps for the counting of messages received that grow as $\mathcal{O}$*(number of the combinations of versions + ID of senders)*.

Is it also interesting to discuss the network overhead due to the number of extra messages required by the protocol. At the beginning, the server and the gateways

exchange data in order to set up the Java RMI communications. These procedures are performed only once at the very start of the application and do not lead to a substantial overhead.
During the standard execution of the protocol, the sender and the receiver exchange data that do not consist only in the message itself, but they also include the version for the right processing, a counter and the IDs of the sender and of the receiver. Therefore, the size of the messages grows, but only in the order of some bytes.
In the course of the updating phase, the server sends to the gateways the new file, together with the version number. After that, the sender sends the total amount of messages processed with the just disposed version. Again, these extra information are in the order of some bytes and thus, we do not detect a relevant network overhead.

7.4 Limitations and future work

In this Section, we want to make a critical reflection about our work and suggest interesting future researches.

First of all, we have developed a system in which all the communications must strictly pass through the back-end server. We have not implemented a direct communication between the gateways. Therefore, the server is a single point of failure for the system. It would be interesting to research this possibility in order to avoid congestion in the server communications and to make the system more interconnected. However, for how we designed our system architecture, a direct peer-to-peer communication between the gateways would be possible.
But, certainly, since the network would become more complex, also the procedure for guaranteeing consistency, that we achieved in our simpler scenario, must be adjusted and implemented accordingly. At the moment, for the update phase we rely on the back-end server to guarantee the consistency of the procedures.

Another suggestion for future work might be to integrate the sender and receiver parts of the protocol in a single one. With this implementation, it would be possible to make the gateways capable of performing both the roles on demand.

In Section 7.1, we discussed about the failures not supported in our solution. The problem of the message loss limits the reliability of our protocol. We have not implemented a procedure able to trigger the sending of a message again in case of a network failure.
Furthermore, we have not taken into account the battery limitation that some IoT devices can present. Even if we have evaluated the time, memory and network overheads, we have not examined the power consumption. Our protocol could be demanding to support for some unsophisticated devices. A possible solution would be implementing a mechanism to log all the activities of the devices every some minutes. In this case, even if one node loses the battery supply, the system would be able to restore the state, once the device will come back operational. Since the data structures to store are relatively simple, it would not consist in a big overhead. Advanced researches could investigate this problem and this proposed solution.

Chapter 8

Conclusion

This thesis presented a distributed and flexible middleware for supporting dynamic policies in a consistent way, in the context of the Internet of Things.

Our main focus was to achieve a mechanism that guarantees dynamic adaptation of policies in every nodes of a distributed system. After an introduction of the IoT paradigm, we described the context of our work, the problem we aimed at solving and our goals. Basically, we researched how to control the behaviour of a distributed system using policies and how to handle them consistently during all the operations. In particular, we worked in the context of a smart building which is composed of different devices, located at different floors, that must collaborate in order to offer their services and functionalities. It is important that all the nodes' behaviours are consistent with one another, in order to avoid hazardous events. Our system is distributed in the sense that the evaluation of a rules, included in a policy, is based on data coming from multiple nodes and, furthermore, the decision made by a node can have an effect on another one, at the other side of the system. Therefore, we spread the decisional capacity on more than one point. Nowadays, the behaviours of the IoT devices are mainly defined upfront and the possibility to change them dynamically and, moreover, in a consistent way has not been well investigated so far. Our challenge has therefore consisted in solving this problem.

We presented different use cases common in the context of a smart building. We decided to focus on the encryption and decryption functionalities because they allow to demonstrate if the consistency among the nodes is guaranteed, better than any other use case. Indeed, a message is correctly retrieved only if both the nodes, in charge of the encryption and decryption phases respectively, have accomplished all the operations properly.

Our solution is based on the Drools Policy System and on the KIE API that allow to enforce and manage policy files in an accessible way. A policy consists in a simple file that can be transmitted among the nodes and substituted at run-time. In this way we can modify the nodes behaviour by updating the policy files. Furthermore, we achieved a protocol that leverages these technologies and it is able to guarantee the consistent fulfilment of all the operations involved.

We developed our middleware as a Proof of Concept in order to validate and evaluate our protocol. We performed some validation tests for analysing if the consistency is effectively assured also in the problematic scenarios. In particular, when the request of the update of a policy file comes close to a processing of a message at any sides. The evaluation test, instead, involved the measurements about the time, memory and network overhead. We showed that the performances are acceptable for this context.

Finally, some limitations and possible extensions are presented. It could be interesting to make our protocol even more distributed, in which also the gateway themselves are directly connected with one another. Furthermore, a research about the energy consumption would help to make this protocol sustainable for any kind of IoT devices.

In conclusion, this thesis has contributed to research a mechanism to manage IoT platforms with the usage of policies. We demonstrated the feasibility of implementing dynamic policies and the possibility to distribute and replace them in a consistent way through all the devices involved.

Bibliography

[1] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges", Ad hoc networks, vol. 10, no. 7, 2012, pp. 1497–1516

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey", Computer networks, vol. 54, no. 15, 2010, pp. 2787–2805

[3] D. Popescul and M. Georgescu, "Security, privacy and trust in internet of things: A straight road?", 05 2015

[4] H. F. Atlam, R. J. Walters, and G. B. Wills, "Internet of things: State-of-the-art, challenges, applications, and open issues", Int. J. Intell. Comput. Res., vol. 9, no. 3, 2018, pp. 928–938

[5] P. Suresh, J. V. Daniel, V. Parthasarathy, and R. Aswathy, "A state of the art review on the internet of things (iot) history, technology and fields of deployment", 2014 International Conference on Science Engineering and Management Research (ICSEMR), 2014, pp. 1–8

[6] D. Jonckers, B. Lagaisse, and W. Joosen, "Expect the unexpected: Towards a middleware for policy adaptation in iot platforms", Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things, 2018, pp. 7–10

[7] R. Boutaba and S. Znaty, "Towards integrated network management: A domain/policy approach and its application to a high speed multi-network.", NOMS, 1994, pp. 777–789

[8] R. Boutaba and I. Aib, "Policy-based management: A historical perspective", Journal of Network and Systems Management, vol. 15, no. 4, 2007, pp. 447–480

[9] M. Sloman, "Policy driven management for distributed systems", Journal of network and Systems Management, vol. 2, no. 4, 1994, pp. 333–360

[10] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates", IEEE Transactions on Software Engineering, vol. 33, no. 12, 2007, pp. 856–868

[11] S. SECTOR and O. ITU, "Series y: Global information infrastructure, internet protocol aspects and next-generation networks next generation networks–frameworks and functional architecture models", International Telecommunication Union: Geneva, Switzerland, 2012

[12] "Introduction to drools." https://www.baeldung.com/drools/, 2018-11-2

[13] "Using drools vs. esper rules." https://www.zymr.com/using-drools-vs-esper-rules/, 2016-06-26

[14] HSG, "Business rule management system." https://www.hartmannsoftware.com/Blog/Enterprise-Rule-Applications/brms/

[15] "Drools tutorial." https://www.tutorialspoint.com/drools/index.htm/

[16] T. D. team, "Drools documentation." https://docs.jboss.org/drools/release/7.15.0.Final/drools-docs/html_single/index.html/

[17] T. D. team, "Interface kiesession." https://docs.jboss.org/drools/release/latestFinal/kie-api-javadoc/org/kie/api/runtime/KieSession.html/

[18] J. R. Nelson Matthys, "Micropnp the zero-configuration platform for wireless sensing & actuation."

[19] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management", IEEE Transactions on software engineering, vol. 16, no. 11, 1990, pp. 1293–1306

[20] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen, "A component and policy-based approach for efficient sensor network reconfiguration", Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on, 2012, pp. 53–60

[21] E. Truyen, B. Vanhaute, B. N. Jørgensen, W. Joosen, and P. Verbaeton, "Dynamic and selective combination of extensions in component-based applications", Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 233–242

[22] G. Russello, L. Mostarda, and N. Dulay, "A policy-based publish/subscribe middleware for sense-and-react applications", Journal of Systems and Software, vol. 84, no. 4, 2011, pp. 638–654

[23] Y. Zhu, S. L. Keoh, M. Sloman, and E. C. Lupu, "A lightweight policy system for body sensor networks", IEEE Transactions on Network and Service Management, vol. 6, no. 3, 2009, pp. 137–148

[24] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, "The role object pattern", Washington University Dept. of Computer Science, 1998

[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software", Design Patterns. massachusetts: Addison-Wesley Publishing Company, 1995

[26] C. a Web Page with no author, "Java rmi tutorial." https://www.tutorialspoint.com/java_rmi/java_rmi_introduction.html/

[27] C. a Web Page with no author, "Java platform standard ed. 7." https://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html/

[28] C. a Web Page with no author, "Getting started using java rmi." https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html/

[29] C. a Web Page with no author, "Java-files and i/o." https://www.tutorialspoint.com/java/java_files_io.html/

[30] Michael Galarnyk, http:/towardsdatascience.com/understanding-boxplots-5e2df7bcbd51/

[31] Roald Hoffmann, http://www.physics.csbsju.edu/stats/box2.html/