University of Namur
Faculty of Computer Science
Academic year 2021–2022

Linda in the fog

Maxime Béver

Supervisor : Antonio Brogi, Stefano Forti

Promoter : ____________________ (Signature for approval of the deposit - EAR art. 40)
Jean-Marie Jacquet

Thesis presented in partial fulfilment of the requirements for the degree of
Master in Computer Science

Acknowledgment

I would like to thank Antonio Brogi who invited me to come and write this thesis at the University of Pisa. He was also very present during meetings in the search for a subject, first of all, and then in the elaboration of this master thesis.

I would also like to thank Stefano Forti for his help and advice in writing the thesis.

I also wish to thank Lucas Embrechts in particular for his proofreading of the document and his comments.

Finally, I thank my promoter Jean-marie jacquet for his comments during the progress of the work and for his proofreading of the document.

Abstract

The amount of data sent to data centers is constantly increasing. Additionally, the number of 'Internet of Things' sensors generating data is also increasing. To manage this data, the fog computing paradigm has been invented. The aim of fog computing is to create a new layer of data management before sending it to the cloud. Furthermore, and in addition to avoiding sending too much data to the cloud, the fog layer will decrease the response time of requests that could already be implemented within this new layer. For these requests to be efficient, the coordination of the message is particularly important. Accordingly, a coordination language could be the solution. In this work, the different criteria for implementing a coordination language in fog computing are explained. These criteria concern security, scalability, performance, management of low-resource devices, data placement policy and data persistency. A proof-of-concept of these criteria is presented, and its development is explained. To prove the effectiveness of this proof-of-concept, performance tests were performed, and different use cases applying the implementation in various contexts are presented.

Keywords : fog computing, coordination language, edge computing, Linda, Rust

Résumé

La quantité de données envoyées aux datacenters est en constante augmentation. En outre, le nombre de capteurs de l'"Internet of Things" générant des données augmente également. Pour gérer ces données, le paradigme de fog computing a été inventé. L'objectif du fog computing est de créer une nouvelle couche de gestion des données avant de les envoyer dans le cloud. En outre, en plus d'éviter d'envoyer trop de données vers le cloud, la couche du fog computing diminuera le temps de réponse des requêtes qui pourraient déjà être implémentées dans cette nouvelle couche. Pour que ces requêtes soient efficaces, la coordination des messages est particulièrement importante. En conséquence, un langage de coordination pourrait être la solution. Dans ce travail, les différents critères d'implémentation d'un langage de coordination dans le fog computing sont expliqués. Ces critères concernent la sécurité, l'évolutivité, les performances, la gestion des dispositifs ayant peu de ressources de calculs, la politique de placement des données et la persistance des données. Une preuve de concept de ces critères est présentée, et son développement est expliqué. Pour prouver l'efficacité de cette preuve de concept, des tests de performance ont été effectués, et différents cas d'utilisation dans divers contextes sont présentés.

Mots-clés : fog computing, langage de coordination, edge computing, Linda, Rust

Contents

Chapter 1

Introduction

1.1 Context

Nowadays, an enormous amount of data is generated by Internet of Things (IoT) sensors and sent to the cloud. This extensive data generation, along with the ever-growing cloud computing, runs the risk to saturate the clouds with data. In addition to overloading the clouds, this also creates a huge amount of bandwidth transmissions. To avoid this situation, the fog computing paradigm was born. The aim of fog computing is to add a layer between the sensors and the data center. This layer is often geographically closer to the sensors and therefore avoids exorbitant bandwidth usage.

In the fog computing paradigm, numerous devices and/or sensors with limited resources produce the data. This data is then sent to the new layer which will initially process the data before sending it to the cloud. Therefore, communication between these different devices is extremely important.

It is in this context that coordination languages can help fog computing manage the data being sent into the fog as well as the error cases where a device is no longer available.

1.2 Objective of the thesis

The objective of this thesis is to identify the features that should be provided by a coordination language targeting fog computing scenarios and to design, implement, and assess a proof-of-concept (PoC) of such a language.

To that end, a secondary objective of this thesis is to define criteria for each feature for implementing a coordination language in the fog continuum. Once these criteria are defined, an implementation is developed that meets these criteria to create an effortless way to use a coordination language in the fog. The performance of the PoC is assessed, and different applications are written to illustrate where it can be used.

The criteria defined must allow the implementation to meet the various criteria that an application using fog computing should have. Therefore, the aim is twofold: to create an implementation that would retain (i) the advantages of fog computing and (ii) the advantages of coordination languages. Naturally, however, by mixing these two paradigms, their respective disadvantages are also mixed, and thus, the implementation will also take these into consideration.

1.3 Outline of the thesis

Before looking at the criteria for the implementation, it was first necessary to determine why using a coordination language in the cloud was a promising idea. To do this, a thorough review of the literature concerning coordination languages was needed, in turn allowing us to determine the main advantages of implementing a coordination language in fog computing. The next step was to find the criteria for an ideal implementation in the fog continuum. For this purpose, it was necessary to read the literature on fog computing to learn how exactly it works. Once the criteria were established, an analysis of existing coordination language implementations was conducted to determine the shortcomings of each. This also allowed us to determine if any of these coordination languages were extensible by following the criteria. Thereafter, the development of the implementation could begin. Once this was completed, performance tests were written, and various applications designed for the implementation were developed.

The content of this thesis is organised as follows.

Chapter 2 describes the state of the art. It includes an introduction to coordination languages as well as an explanation of what fog computing and edge computing are. Finally, the state of the art and a review of different implementations of coordination languages tailored to fog computing are provided.

Chapter 3 explains the different criteria necessary for implementing a coordination language in fog computing. Two different implementations of a coordination language in fog computing, JSpace and Rustupolis, are also described.

Chapter 4 describes the implementation and how it was developed. First, the Rust programming language used to create the Proof-of-Concept (PoC) is explained. Then, the main concepts of the language are explained and the code of the Rustupolis library is analysed. This library is extended to create the PoC. Lastly, the implementation of the different criteria are explained.

Chapter 5 describes two applications created from the PoC. The first describes an application whereby the tuple space server will be installed on different devices such as a phone or Raspberry Pi. The second application will present a use case where traffic lights are connected and managed from different tuple spaces.

Chapter 6 describes the conclusion of the work and proposes reflexions for potential future work. This chapter also presents an assessment of the contributions of the work presented herein.

Chapter 2

State of the art

2.1 Introduction

To begin this document, the state of the art concerning the different topics of the problem is presented. It starts with a presentation of the literature on coordination languages. First, generative communication is presented and thereafter one of the first coordination languages named Linda is examined. As Linda has laid the foundations for several coordination languages, an analysis of it is particularly relevant and interesting. The last part of the section on coordination languages focuses on the languages born after Linda and explains their differences. Secondly, the state of the art of the fog continuum is discussed. This part starts by introducing fog computing, then presents edge computing, and thereafter explores the differences between these two paradigms. Finally, the existing coordination languages for fog computing are reviewed, highlighting both their advantages and disadvantages.

2.2 Linda coordination language

2.2.1 Generative communication

Before discussing the Linda programming language, the generative communication paradigm must be introduced. Generally, the communication between two clients is done by the message passing paradigm. This allows the transfer of messages directly between the two clients. This paradigm is mainly used for a concurrent language program whereby multiple processes are multiplexed on one machine, not distributed over many. Accordingly, processes need not execute in wholly disjoint address spaces, and they may use shared variables or a generalisation of conventional parameter passing such as the monitor call to communicate [1]. The paradigm is represented in the following Figure 2.1. Client A sends a message or request to client B who receives the message and processes it. Client B then sends a response back to client A and so forth.

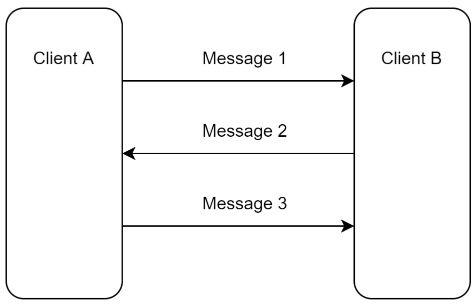


Figure 2.1: Message passing paradigm.

In distributed languages, the concurrent processes execute on separate machines and always in disjoint address spaces. They may therefore communicate with each other only by sending and receiving messages that are carried between address spaces by a runtime communication system [1]. It is in this context that generative communication emerged and became the basis of the distributed programming language called Linda.

2.2.2 Linda

Linda is a distributed coordination language that has been developed originally for the SBN network computer [2]. Linda stood out from other distributed languages because the language was fully distributed in space and time.

Another quality of the language is that it is more expressive than other languages, while at the same time being simpler [1]. Linda allows the communication of tuples through a tuple space. The tuples are the equivalent of the messages or the process that a client wants to send to another client in the message passing paradigm. The tuples are ordered lists of typed data. For example, the tuple *("foo", 562, 45.4)* contains three fields. The tuple space, also known as ‘blackboard‘, is the environment which manages the tuples. The tuple space can be seen as a collection of tuples. This communication model allows persistent broadcast communication and is persistent because the tuple sent by client A does not depend on the client B. The tuple model contrasts a normal communication channel whereby client A sends a message to a client B and if client B is not available, client A’s message will be lost. In Linda, however, the tuples are sent to the tuple space and stored within it. That also means that the tuple sent by client A is accessible to all clients that have access to the tuple space. Thus, this type of communication is called broadcast. According to [1], the tuple communication model is said to be generative because, until it is explicitly withdrawn, the tuple generated by client A has an independent existence in the tuple space.

To realise the different operations on the tuple space, Linda has three primitives. Thanks to this small number of primitives, Linda offers an abstraction for accessing shared memory and a simple communication interface. To retrieve a tuple in the tuple space, Linda uses the principle of pattern matching.

- *out(t)* adds an occurrence of the tuple t to the tuple space.
- *read(p)* reads the pattern p and searches if there is a tuple that matches the pattern in the

tuple space. If there is no corresponding tuple, the operation stays blocked until a tuple matching the pattern is added to the space. If the pattern matches, the tuple is sent back without removing it from the tuple space.

- *in(p)* the operation is the same as *read* but removes one occurrence of the tuple matching the pattern.

Linda also has another operation called *eval* that is used to spawn a process. This operation, though included in the original Linda model, tends to be excluded from most other implementations in favour of more traditional process creation mechanisms, which is usually largely influenced by the host implementation language [3].

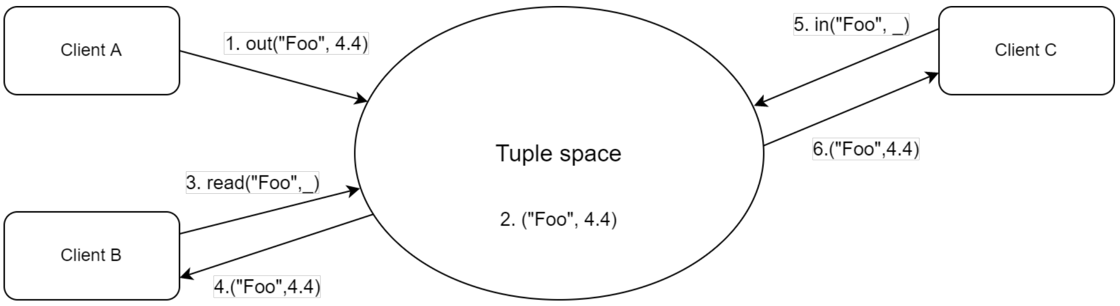


Figure 2.2: Example of communication with Linda

Figure 2.2 illustrates an example of communication between three clients. In point 1, client A adds a tuple to the tuple space. This tuple is added to the tuple space in point 2. Client B in point 3 tries to read a tuple with the pattern of a tuple starting by "foo". The _ value functions as a wild card, matching against any given value in the corresponding field of a tuple. The tuple space finds a tuple matching this pattern and sends a copy of the tuple back to point 4. In point 5, client C asks to take out a tuple matching the pattern. The corresponding tuple is given back to client C and is removed from the tuple space in point 6.

This means that Linda's communication model permits:

- *time uncoupling*: a tuples' lifetime is independent of the producer process's lifetime,
- *destination uncoupling*: the creator of a tuple is not required to know the future use or the destination of that tuple [4],
- *space uncoupling*: communicating objects need to know a single interface, i.e. the operations over the tuple space [4]. This approach is also known as flow-of-objects and is in opposition to the method invocation, which requires many interfaces for the operations supplied by remote objects.

2.2.3 Coordination language

To create the best implementation of a tuple-based platform in fog/edge computing, the existing coordination language implementations have been studied. The following list contains implementations of existing coordination languages:

- *JavaSpaces* [5]: JavaSpaces is one of the first developed implementations of tuple spaces. A tuple is an instance of a Java class, and its fields are the public properties of the class. This means that tuples are restricted to contain only objects and not primitive values [6]. However, the repository is now archived.

- *GigaSpace* [7]: GigaSpace is a commercial implementation of tuple spaces. It is an extension of JavaSpaces. User applications should interact with the server to create and use their own tuple space [6]. The primary areas where GigaSpace is applied are those concerned with big data analytics. However, Gigaspace does not have a public repository for its project.

- *Tupleware* [3]: Tupleware is specifically designed for array-based applications. The array is decomposed into several parts, and each of the parts can be processed in parallel. It is aimed at developing a scalable distributed tuple space with good performances on a computing cluster [3]. It also aims to provide simple programming facilities to deal with both distributed and centralised tuple spaces. [6] The project repository is written in Java and has not been updated for 12 years.

- *MozartSpaces* [8]: MozartSpaces is a Java implementation of the space-based approach. MozartSpaces uses the XVSM language. MozartSpaces also provides transactional support and a role-based access control model.

- *TuCSoN* [9]: TuCSoN is a coordination model adopting Linda as its core but extends Linda in several ways, for instance by adopting nested tuples (expressed as first-order logic terms), adding primitives, and replacing tuple spaces with tuple centers programmable in the ReSpecT language.

- *pSpace* [10]: pSpace is a project between the Technical University of Denmark and the University of Camerino. The goal of the project is to develop a support for programming distributed applications with tuple spaces. The project is implemented in different languages like Java, C#, Go, JavaScript, and Swift.

- *TuSOW* [11]: As discussed in the state-of-the-art section 2.4, the project is built in the fog and has deficiencies concerning security. Another point of concern is that the current implementation of the project is written in Kotlin and in Java.

- *LighTS* [12]: LighTS provides a flexible framework that makes it easy to extend the tuple space in many ways, including changing the back-end implementation and redefining the matching semantics.

- *Klaim / X-klaim* [13] [4]: Kernel Language for Agents Interaction and Mobility is an extension of Linda. The aim of Klaim is focussed on process mobility, meaning that processes, as any data, can be moved from one locality to another [6]. The processes can be executed in any localities. The implementation of Klaim is called Klava and is written in Java. Klaim supports multiple tuple spaces and operates with explicit localities where processes and tuples are allocated. Thanks to this system of localities, a system of access controls exists in Klaim.

- *Rustupolis* [14]: As its name suggests, Rustupolis is an implementation of Linda written in the language Rust. It implements the primitives of Linda and the pattern matching that Linda uses. The implementation manages concurrent access to tuples and is multithreaded.

2.3 Fog computing continuum

The Internet of things (IoT) concept was created in 1999 and was originally intended for factories and production lines [15]. The concept was then exported to other areas such as transport, health, and housing. Accordingly, being able to automate systems through sensors that measure data saves time and often money, as the tasks are no longer managed by humans [16].

According to Cisco's annual Internet report for the period 2018-2023 [17], the number of machine-to-machine (M2M) connections from IoT devices is expected to increase by 19% annually, resulting in 14.7 billion connections by the year 2023. This means that the number of IoT devices and applications is going to become more immense compared to today, consequently generating a problematic amount of data sent to the cloud. In fact, in the current situation, a large proportion of generated data is sent to the cloud, resulting in a high latency. This slowness makes it impossible to design applications that require fast response times.

Researchers have developed new paradigms to avoid sending this data to the cloud and to process it closer to the source. These paradigms extend the cloud and enable IoT applications to be deployed in the proximity of sensors, adding new benefits like fast response times and better security and privacy [18]. The two paradigms are fog computing and edge computing. In the rest of this chapter, each paradigm, and the difference between them is explained.

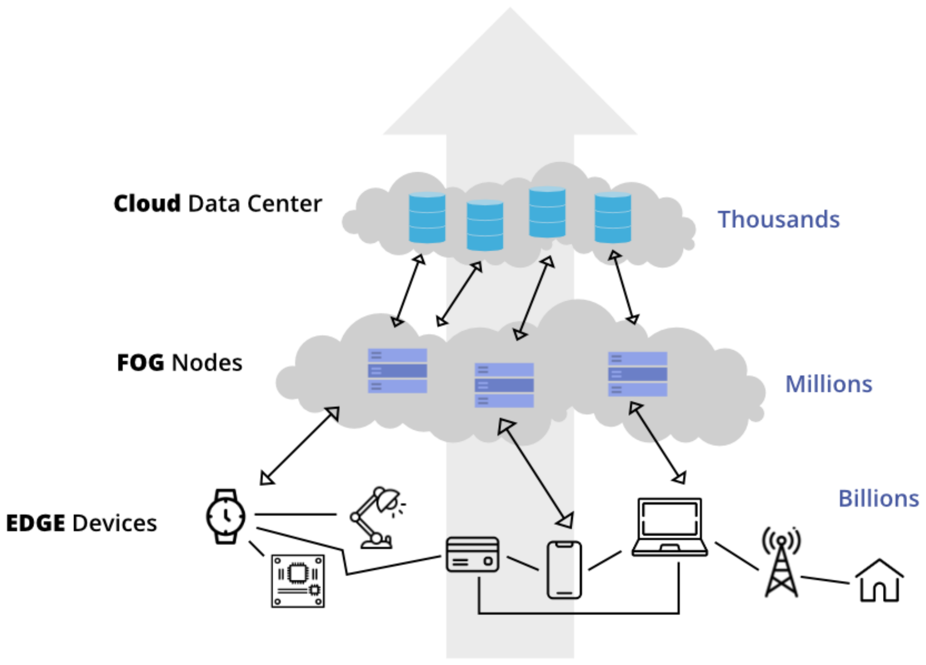


Figure 2.3: Schematic diagram of fog computing [19]

Figure 2.3 shows the different layers that can be found in the fog computing continuum. The first layer (EDGE) holds all IOT sensors and devices that produce data. The second layer

(FOG) consists of the fog nodes that serve as an intermediate layer between the cloud and the edge devices layer. The third layer (CLOUD) represents the cloud and the various data centers in which the data is stored.

2.3.1 Fog computing paradigm

According to [20], the definition of fog computing is: *"Fog computing is a geographically distributed computing architecture with a resource pool consisting of one or more ubiquitously connected heterogeneous devices (including edge devices) at the edge of the network and is not exclusively seamlessly backed by cloud services, to collaboratively provide elastic computation, storage, and communication (and many other new services and tasks) in isolated environments to a large scale of clients in proximity"*

The main goal of the fog computing paradigms are to bring the computational resources (i.e., storage, networking, and processing) closer to the edge of the network. Consequently, this reduces the reliance on cloud-based environments while simultaneously decreasing latency and network congestion [18]. Furthermore, it enforces privacy by processing the data "near" the user.

The nodes in the fog computing layer are servers that use their resources to make a first treatment of the generated data. The process that needs a large amount of computing power is left to the cloud. Actually, the fog node provides limited computing resources as opposed to the cloud where resources are unlimited. The fact that the data is first processed in the fog also means that raw data is not sent to the cloud. This can therefore also protect the privacy of users. For example, a common feature of applications using the cloud is to upload photos or videos to the cloud. One could imagine a fog node that would first process this content by optimising the resolution for example [21].

Thus, the use of fog nodes allows us to overcome limitations of the cloud paradigm [18]:

- *Latency constraints*: The proximity of the fog node to the IoT device allows for very low latency. This is particularly important in the development of real time applications, whereby real-time decision-making is not possible when analytics are performed on a distant cloud. [22]
- *Network bandwidth constraints*: Nowadays, the increasing quantity of data generated at the edge layer creates a bottleneck in the speed of data transport. For example, a Boeing 787 generates about five Gigabytes of data every second [23], but the bandwidth between the plane and the satellite or base station on the ground is not large enough for data transmission [21]. The first process that the fog node performs on the data avoids sending raw data to the cloud. This reduces the amount of data sent to the cloud and means that most of the data produced will never be transmitted to the cloud. However, a balance must be found between the amount of data sent to the cloud and the data deleted by the first processing performed by the nodes.
- *Resource constraint devices*: As the fog nodes are closer to the device, the node can perform computational tasks that were processed by the edge device. The goal of this is to reduce energy consumption and life-cycle costs by discharging some computation parts of the application from such restricted devices to nearby fog nodes [22].
- *Increased availability*: The fog nodes do not need connectivity to the cloud to process the

data. Consequently, the application is less dependent on the cloud and more available as it can be replaced more easily than the cloud. This means that an application using fog computing nodes must be able to continue working when a node is no longer available in the network. This also depends on how the node network handles the persistence of data and services.

- *Better security and privacy.*: In the fog computing paradigm, the fog node can process the personal data of the users and avoids sending this data to the cloud. The data are treated locally, and no data is stored on a remote cloud.
- *Energy consumption of the data center*: The energy consumption of data centers is constantly increasing due to the number of applications using the cloud. One way to reduce this consumption is to perform trivial tasks on edge nodes without significant energy implications. [22]
- *Collaboration between edge*: In the current cloud paradigm, the stakeholders that send their data to the cloud rarely share it with each other due to privacy concerns and the difficulty of transporting data from one cloud to another. As these are often distant, the cost of data transport is too high. Thus, the collaboration between stakeholders is blocked. The node in fog computing acts as a small data center which can be part of the concept of collaborative edge. The concept is therefore to connect the edge of different geographically 'close' stakeholders with the aim of sharing data. One of the promising future applications is connected to the health sectors, whereby the edge of a hospital can share data with doctors, pharmacies, or even insurance companies [21].

One of the most popular examples of fog computing is the traffic light system in a city which require synchronisation with low latency to avoid traffic congestion [18]. To illustrate this example, Figure 2.4 explains the infrastructure.

The device layer is the layer where the sensors are. These sensors generate data about the numbers of cars that are waiting at the red light, the number of cars that cross the intersection, the number of pedestrians that are waiting to cross the road, etc. All the data are sent to fog layer 1, which processes the data and sends it to fog layer 2. Additionally, fog layer 1 has the task of managing the lights of the intersection.

Fog layer 2 receives the data from various fog layer 1s. The task of fog layer 2 is to make another filtering process on the data before sending it to fog layer 3. Additionally, fog layer 2 has the task of coordinating the different intersections of the districts.

Fog layer 3 manages all the districts of the city, receives the data of all the fog layer 2s of the city, and then processes this data before sending it to the cloud. This example uses three different layers of computing in the aim of reducing the latency.

In [24], the authors develop a PoC of a network of wearable cognitive assistants and, proved that the response time when using their application with cloudlet was superior to when using the cloud. In [25], researchers built a platform to run a face recognition application and, by moving computation from cloud to the edge, the response time was reduced from 900 to 169 ms.

Another popular example of fog computing applications is the case of smart home systems. In a smart home, various sensors monitor every aspect of the home, for example the room temperature, home security, shutters, and other objects that are connected to the network. In this scenario, a server (or fog node) is managing all the sensors. Thus, one way of building the server is to use EgdeOS [26] which is an operating system specialised for the IOT. The EdgeOS allows for the easy management of sensors in the home and the data can be processed locally.

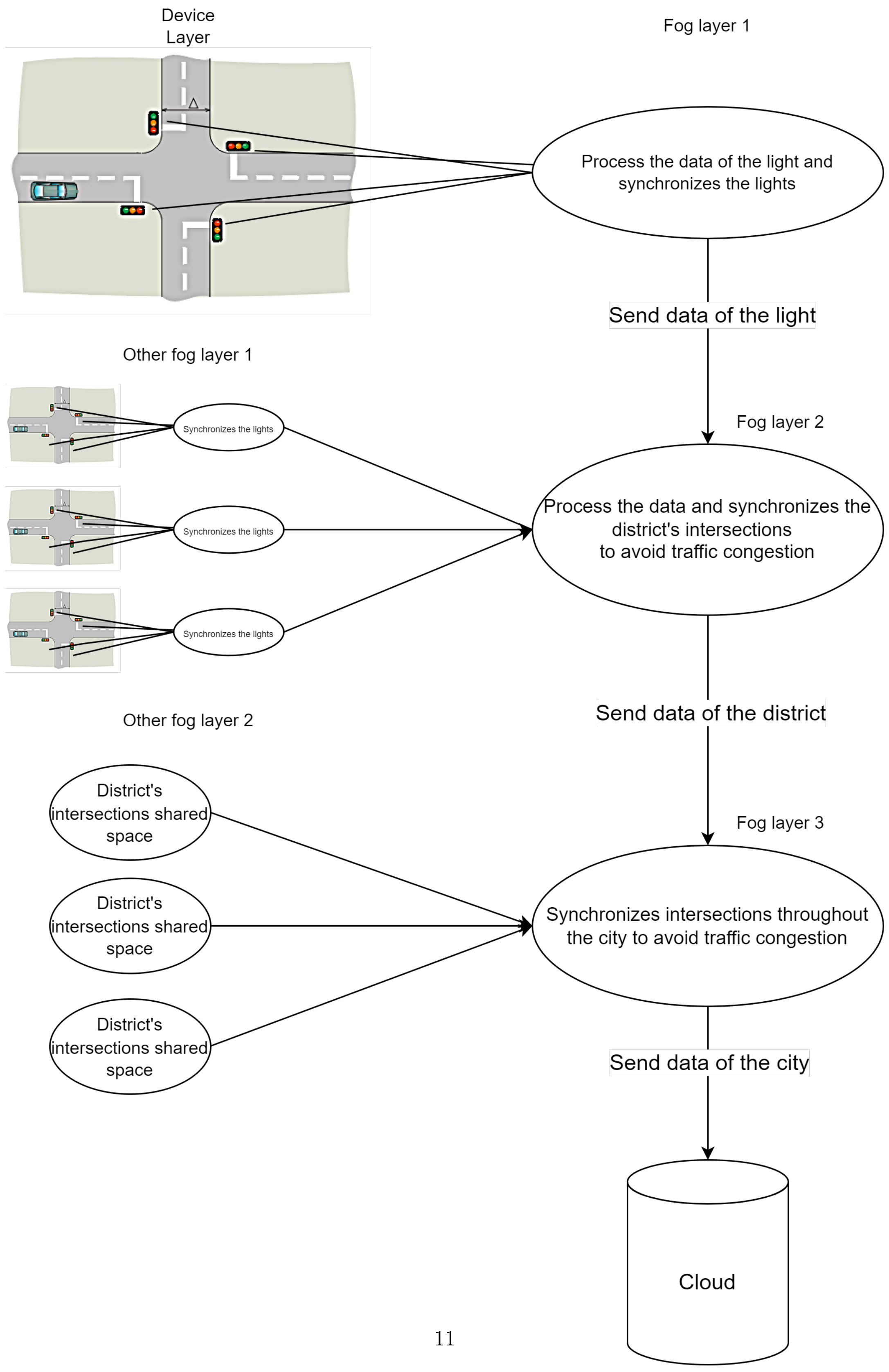

Device Layer
Fog layer 1
Process the data of the light and synchronizes the lights
Send data of the light
Other fog layer 1
Synchronizes the lights
Synchronizes the lights
Synchronizes the lights
Fog layer 2
Process the data and synchronizes the district's intersections to avoid traffic congestion
Send data of the district
Other fog layer 2
District's intersections shared space
District's intersections shared space
District's intersections shared space
Fog layer 3
Synchronizes intersections throughout the city to avoid traffic congestion
Send data of the city
Cloud

Figure 2.4: Example of the fog computing

2.3.2 Edge computing paradigm

Like fog computing, edge computing aims to bring computing resources closer to the edge of the network [18]. The main difference between the two paradigms is that the edge computing moves some computational resources to the resource constraint devices located at the logical extremes of a network [18]. The difference with the fog computing paradigm is the location in which the data is processed [18]. In the edge computing paradigm, the 'things' are not only data consumers, but also function as data producers [21]. The other difference between the two paradigms is the available computing power, which is less in edge computing as compared to fog computing. The edge computing paradigm will therefore be preferred for first elementary processing.

The smartphone of a user could become a device that serves the role of a server for an IoT device (Figure 2.5). Some part of an IoT application is deployed directly on the device on the edge of the network. Nodes are not just mobile phones. They can also be desktops, laptops, tablets, or any other device capable of processing data. Resultant, edge computing provides the opportunity to reduce latency and bandwidth waste even more than fog computing.

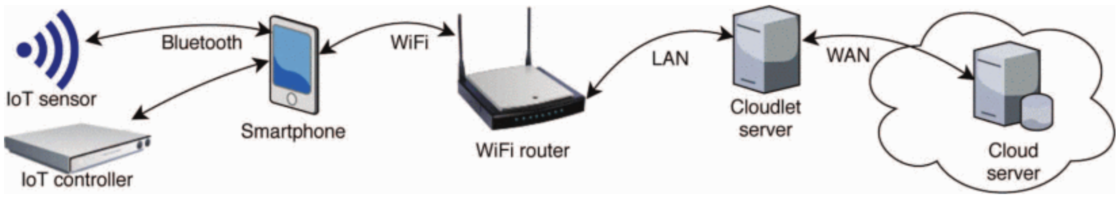


Figure 2.5: Example of edge computing [27]

An example of edge computing in practice is presented in article [18]. The authors present a solution whereby a person is wearing an ECG sensor. The sensor is connected to the smartphone of the person via Bluetooth. The sensor sends data to the smartphone which, after conducting initial processing, sends the filtered data to the cloud. The interesting point of this example is that the smartphone can react in real time if it detects that an unexpected event occurs. In this case, the smartphone could send a notification to the emergency services.

2.3.3 Future challenges of fog computing

2.3.3.1 Resource Management

Nodes, whether in fog computing or edge computing, are resource-constrained devices. This means that to make the paradigms effective, new resource management methods need to be implemented. With new techniques to optimise the use of resources, this will also lead to improved data privacy and better performance. To achieve this task, a way of estimating the resources needed to perform a particular task is important to provide the best optimisation of the computing resource. Conversely, the resources already available in the network of nodes is also something that needs to be computed. This can be represented by the hardware capabilities of each node. The mechanism of estimation of the resource available must be automatic and can not be managed manually [22].

Another goal of the management of resources is the allocation of processes to the nodes. To make the allocation of processes in the different edge devices, previous data is used. In fact, the computational resource of the device at the edge of the network and the resources needed by an application are the two most principal factors needed to optimise the allocation. This part of the allocation of processes must also deal with the reallocation of a process if a node is not available or if the nodes fail to do the task. The main aim of resource management is to optimise the allocation of resources [18]. This optimisation also depends on the metrics that the application wants to optimise. These metrics could be the latency, the bandwidth, the energy, or the cost [21].

Another way to compute a huge task that requires an abundance of computing resources is to partition the task between different nodes. In [28], the authors use edge computing to create their aggregate process. The principal goal of the aggregate process is to divide the computation part across different devices at the edge. However, the main challenge is to synchronise the devices. The authors explain how to create a network of different devices, with the concept of one leader node which sends instructions to its neighbouring node. The neighbour then has the objective to broadcast that instruction to its next neighbour and so on.

One of the most important points in the allocation of tasks in a node is to not overload nodes with computationally intensive workloads. The partitioning of a task can be achieved across geographical space or through time. For instance, to avoid the overloading of the nodes, the tasks can be scheduled according to the peak usage times of the nodes [22].

As seen before, the fog and edge computing paradigms inherit the problems of the cloud. However, there are also problems linked to the IoT devices themselves. One of the main problems is the incredibly vast number of existing protocols and technologies that exist. Though, the IoT field is increasing its efforts to converge on a common framework. Among the most widely used communication protocols is Message Queue Telemetry Transport (MQTT) [29], a lightweight messaging protocol based on the publish/subscribe paradigm [30].

2.3.3.2 Security and privacy

As mentioned before, fog computing provides many advantages when creating a powerful network. To benefit from these advantages, the security and privacy of the users must be guaranteed. In fact, the fog computing paradigms are often used to complete a cloud computing application. Thus, the security problems of the cloud are included within these paradigms. The other problem of the fog computing paradigms is that security breaches of IoT devices and less powerful servers are also present. The challenge of the fog computing paradigms is thus to deal with these security problems, which can be divided into distinct categories: confidentiality, integrity, and availability (a.k.a. CIA) [31].

The first point is confidentiality. Confidentiality means that the user knows who has access to their data and only authorised people have access to it. The best way to ensure that data are safe is to process the data of users locally and avoid sending it to the cloud. If the data must be processed elsewhere, a fog computing application can implement an authentication system coupled with a system of access rights. The complication of the authentication system is that the nodes can join or leave the network without restriction. This also means that the authentication must be speedy to avoid wasting as little time as possible. The additional layer that fog and edge computing add brings a new layer of hardware onto the network. However, with the addition of new hardware comes novel risks to data confidentiality. Users should ask themselves the question:

Can the organisations that own these devices as well as those that will employ these devices be trusted? Another way to ensure the confidentiality of data is to use the nodes layer to remove personal information from the data of the user.

The second point is integrity. Integrity of the data means that the system must ensure that the data are not modified or deteriorated when travelling on the network. To avoid integrity problems, the application must use a secured network or process to ensure that the data are not corrupted in transit.

The third point is availability. This point ensures that the nodes where the data are stored or processed are available when needed. In fog computing and edge computing, the developers have no ways to control the nodes as they do in the cloud paradigm. A system to ensure the persistency of data must be developed.

2.3.3.3 Programmability

One of the advantages of the cloud paradigm is that it is a homogeneous platform and developers only need to develop their application in one programming language. In the case of fog and edge computing, the nodes are heterogeneous platforms and by consequence, the runtime of nodes differs from each other. The developer will face these problems when developing an application in the edge [21].

Another problem in the development of applications in fog computing is the naming of the nodes. In fact, the traditional mechanism of the domain name system (DNS) is not flexible to serve the dynamic edge network. Most of the nodes in edge computing could be highly mobile and resource constrained. Thus, the naming scheme for edge computing needs to consider the mobility of IoT devices, highly dynamic networks, privacy, and security protection [21].

The number of different IoT devices that can send data to a node is also problematic given that each device uses its own format for data. For example, some sensors just return one type of data, while others return data in JSON, XML, etc. Another important aspect regarding the transfer of data is to decide in which level of abstraction the data is filtered. If too much raw data is filtered out, some applications or services could not work as intended. Thus, the challenge for the fog and edge paradigms is to find a good balance when abstracting the data.

When implementing an application in fog computing, other criteria must be considered. The first criterion is the management of the differentiation of the service; a critical service must be treated with priority as compared to less critical service. The second criterion is the mechanism to manage the addition or removal of nodes within the network; the developers must manage how to easily add, remove, or replace a node. The third criterion is to manage how the crash of a node is coped with by the system; the user must be alerted if a device or a node does not work or does not respond. However, the system can manage the third criterion with a series of diagnostic and failure detections. The entire system cannot cease to function if only one node has a problem. The developer can also implement a data placement policy system. The user must choose where their data is stored, and which third party application has access to it.

2.4 Coordination in the fog

There are a multitude of advantages to applying a coordination language like Linda in fog or edge computing. The first advantage is to provide a heterogeneous layer with simple operations. Internet of Things devices can send their data to different tuple spaces. To put data in a tuple space, the device simply has to send an *out* request with the data wrapped in a tuple. To get access to that data, a client must use the *in* or the *read* operation. The language is therefore simple to understand and easy to access.

Clients do not need to know the data structure before performing an operation in the tuple space. They should just perform an *in* or *read* operation on the tuple space with the wildcard symbol (_). The wildcard symbol tells the pattern matching that any value can be found at that point in the tuple. The client will then receive a tuple which contains the data previously entered by a device in the tuple space.

Another advantage to using a coordination language in fog or edge computing is the persistent broadcast. The time, destination, and space uncoupling allows a device to be disconnected from the network and to reconnect to it. It also offers a robustness against connectivity issues and faulty devices. The tuple system allows requests to be coordinated in a very dynamic network. In this section, two pieces of work combining the fog continuum and the Linda coordination language are presented.

In their article "TuSoW: Tuple Spaces for Edge Computing" [11], the authors present a model called TuSoW (Tuple space over the web) to bring tuple-based coordination to the edge and to support interaction among heterogeneous devices without requiring the mediation of the cloud [11]. As seen before, the heterogeneity of the device in the edge is a problem for fog and edge computing. In the cloud computing paradigm, this heterogeneity is hidden by the Cloud Application Programming Interface (API). To bring this homogeneity to fog or edge computing, the Linda coordination language is used. The language is used to manage the interactions among devices and services in the edge. The other aim of the author's project was to allow developers to free themselves from the complexities inherent in low-level networks and focus on how the many participants in a distributed application should interact.

A TuSoW system is composed of several services and clients. Each service wraps a number of named tuple spaces, which are modelled as collection resources. Each service is named with a uniform resource identifier (URI) including the protocol of the service, the name of the service, and the name of the tuple space. The clients are the entities who need the tuple space. Clients may communicate with services and interact through tuple spaces. To communicate with the tuple space, the client has a set of remote API geared toward different settings, including Hypertext Transfer Protocol (HTTP), WebSockets, gRPC Remote Procedure Calls (gRPC), and Message Queue Telemetry Transport (MQTT) [11].

To be considered as a system that can be used in the fog or edge, Tusow still needs to be extended. More specifically, Tusow has no authentication system, no access control system, and no system of data persistency.

In [30], the authors implement a system with the Linda coordination language for fog computing. They present their architecture for the prototype of a fog computing "content island". This prototype interconnects groups of IoT devices to interchange data, processing among themselves and with other content islands. To exchange information between entities, their architecture uses the publish/subscribe paradigm, allowing for the easy use of the communication protocols MQTT [29], a lightweight messaging protocol based on the same paradigm. The authors consider that a MQTT message can represent a tuple, and they implement Linda's primitives accordingly. The authors then compare their solution on different installations and get satisfactory performance results. However, a drawback of the article is that the authors do not mention and discuss communication security or data access control systems.

2.5 Concluding remarks

This chapter first explained what coordination languages are. These come from the generative communication paradigm. The Linda language is one of the best-known coordination languages and has had many extensions. The advantages of Linda such as time, space, and destination uncoupling have been presented. In the rest of this thesis, these advantages will be used in fog computing.

The differences between fog and edge computing has been explained, with the main difference being the location of the device performing the processing. Thereafter, various future challenges of fog computing were explained. In the rest of this thesis, these challenges will be highlighted in an attempt to solve them with coordination languages.

Chapter 3

Coordination language in the fog

3.1 Introduction

This chapter contains the bulk of the work. Here, the different tasks are explained. Then, the different criteria needed to have a coordination language implementation in fog computing are explained. Next, details are provided on the analysis of implementations called pSpace and Rustupolis. The choice of library used to extend these implementations to this work is then explained, and a justification is provided as to the reason why the Rust programming language was used. The characteristics and strengths of Rust are explained. Thereafter, a description of the implementation of the tuple server is given. The criteria and the ways to apply them within the implementation are detailed. The next part focuses on the second implementation, which is a language to communicate with this tuple server. For this language, a Rust interpreter has been developed. Finally, the unimplemented criteria are discussed and ideas are given for their implementation.

3.2 Criteria for a tuple space implementation in the fog continuum

Once the advantages of the tuple-based coordination language have been proved, the aim is to define how to implement that language in fog and edge computing. Actually, an implementation of a coordination language already exists, but we must define whether they fit within the constraints of fog and edge computing. In this section, six criteria are explained to write an implementation that works in fog and edge computing.

3.2.1 Usable on resource constrained devices

Compared to the cloud infrastructure, the nodes in a fog computing application are often less-powerful devices such as a laptop or Raspberry Pi. The implementation must therefore be computationally **'lightweight'**. The definition of lightweight means that the system should use as little ram memory and processor time as possible. To make a lightweight implementation, the choice of programming language is consequently very crucial. The language needs to allow building lightweight software and must therefore allow for the optimisation of the resource allocation.

3.2.2 Scalability to Large Infrastructure

The implementation should allow the construction of a **scalable** system. In fog computing, new devices can be added at any time, whether it's to write or to get access to data. The implementation should allow these two steps to be done with simplicity. That also means that the implementations must be capable of managing **multiple communication protocols**.

The number of clients that make a request to a tuple space is something that the implementation must handle. A mechanism of concurrent management must be implemented. The requests of the different clients that try to access a tuple space must be stored in a queue and processed sequentially.

3.2.3 Data securities properties

The implementation must allow the construction of a system that gives access rights to tuples and ensures their confidentiality. A system composed of fog and edge devices must satisfy the three characteristics of the CIA model (confidentiality, integrity, and availability). In a dynamic IoT network where devices can join or leave the network without any restrictions, a connectivity optimisation mechanism must be implemented. This mechanism must ensure that security is maintained by authenticating a fog or edge node. However, it is difficult to find a mechanism that combines security and performance on resource constrained devices. If the security level is too high and requires strong encryption of the data, the devices may be slowed down or even bottlenecked. On the other hand, a lower security level allows for better performance but also exposes the tuples space to security breaches. Therefore, the right balance must be found between security and performance. This authentication added to the mechanism of access right must manage the **confidentiality** of the users' personal data.

The implementation must be able to manage the movement of data but also to restrict it so that data is, for example, only available in the tuple server and not in the cloud. Personal data is considered protected if the user has the power to decide where data should be processed. As a rule, if the data is processed locally, then the chances of intercepting the data by a malicious attacker are close to zero. This mechanism concerns **integrity**. However, if data is to be transported over networks from one server to another, then a network security mechanism must be put in place to prevent the data from being corrupted. This corruption can be due to a poor-quality network, but also to an attacker who would like to modify it. The third point, **availability** of the data, will be developed further in the data persistency part.

3.2.4 Time performance

The performance time includes the latency between the sensor and the tuple server. Fog computing aims to support the IoT through very **low latency**, very fast data processing, and highly efficient resource utilisation. The implementation must take this into account, even though the performance time depends on the geographical position between the server and the client. It is also the performance in terms of time taken when a device is disconnected and needs to be reconnected. However, operating the network edge requires discovery mechanisms to find nodes and add them to the network. The performance in terms of time also includes the time needed to perform pattern matching to recover the tuple.

3.2.5 Data persistency

The implementation must offer availability and **persistence of data**. In a network where nodes are less powerful devices and the risk of device crashes is higher, ways to keep the user's data available must be put in place. This criterion includes the possibility to manage the disconnection of the server containing the tuples. The implementation might support replication of tuple spaces among several machines.

3.2.6 Data placement policies

The implementation must allow **specifying and enforcement policies** on where tuples can or cannot be stored. The implementation might, for example, have a system to avoid data being stored out of a domain's name or across of a number of selected devices. The owner of the data should be able to decide where the data is stored, and which third party software has access to it.

3.3 Existing implementations

From the list of implementations see section 2.2.3, two languages were selected for analysis according to the above-mentioned criteria. These two language implementations were selected according to the date of the last updates to the repository, the availability of the code and documentation, as well as the ease of use of the code. Accordingly, the goal here is to find an implementation that could be extended to meet the above-mentioned criteria.

3.3.1 pSpace

As already presented in the section on the state of the art 2.1, pSpace is a project between several universities. It aims to provide a simple, yet powerful support for distributed programming [10].

3.3.1.1 pSpace primitive

pSpaces [10] use primitives inspired by the Linda language.

- *put(T)* adds an occurrence of the tuple t to the tuple space.
- *get(T)* takes the tuple t from the tuple space. If the tuple is not present, the execution is stopped until the right tuple is added to the tuple space.
- *getp(T)* is the non-blocking version of the get primitive. If the tuple is non-present, the primitives send a null object.
- *getAll(T)* retrieves all tuples matching a template and removes them from the space.

- *query(T)* checks if the tuple t is in the tuple space. If the tuple is not present, the execution is stopped until the right tuple is added to the tuple space.
- *queryp(T)* is the non-blocking version of the query primitive. If the tuple is not present, the primitives send a null object.
- *queryAll(T)* retrieves all tuples matching a template.

3.3.1.2 pSpace types of tuples

When creating a space, pSpace offers the possibility to choose how the tuple space will manage the behaviour of retrieving a tuple for a query. Here is the list of the several types of tuples:

- *SequentialSpace* returns the oldest tuple that matches with the template.
- *QueueSpace* returns the oldest tuple in the space if the oldest tuple matches with the template, otherwise it returns null.
- *StackSpace* returns the newest tuple in the space if the newest tuple matches with the template, otherwise it returns null.
- *PileSpace* returns the newest tuple that matches with the template.
- *RandomSpace* randomly returns a tuple that matches with the template.

Taking into account the aforementioned criteria and to determine if pSpace is usable in fog computing, the analysis was based on the Java version of pSpace called jSpace.

3.3.1.3 jSpace's usability on resource constrained devices

pSpace, and its Java implementation, called jSpace create Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) servers. Java's implementation uses Java 1.8 and Maven to manage the dependencies. The Java version of the project is therefore more interesting for devices that contain the JVM of the language and may be limited for devices with limited resources.

3.3.1.4 Scalability in jSpace

To add a client that wants to write or to read in a tuple space, it is surprisingly easy. The client just needs to have the URI address of the server and then they could create a connection to it. Once the client is connected, they can call the primitives on the selected tuple space. In this part, security is not mentioned, however, this point will be discussed in later sections.

```
RemoteSpace chat = new RemoteSpace("tcp://chathost:31415/room1?keep")
chat.put("Alice","Hi!")
```

Another point of the scalability of an application is the ease at which a node can be added that will hold one or multiple tuple spaces. It is also remarkably easy to create this node with jSpace. The first step is to create a repository on the device, then the tuple spaces must be created in this repository. Thereafter, to make the tuples available on the network, a gate must be added to the repository. When the gate is created, multiple protocols can be chosen and the desired address and port must also be indicated.

```
SpaceRepository chatRepository = new SpaceRepository();
chatRepository.Add("room1",New SequentialSpace());
chatRepository.Add("room2",New SequentialSpace());
chatRepository.addGate("tcp://localhost:31415/?keep");
```

The main point to improve the scalability of jSpace is the number of protocols implemented. Indeed, now only the TCP and UDP protocols are implemented. Therefore, one way of improving jSpace would be to add new protocols such as Bluetooth or TLS.

3.3.1.5 Security in jSpace

As mentioned previously, the security criteria could be divided in three points: confidentiality, integrity, and availability. For each point, what exists in jSpace will be discussed.

- *Confidentiality*: In the current implementation of jSpace, no system of authentication or rights access is implemented in the code of the master branch of the project's repository. However, in the documentation, the authors explain a system of permission where a client could be permitted or rejected to use specific primitives on specific tuples. This system started to be developed but was never deployed on the main branch of the project's repository. To maintain the confidentiality of the data, the best way with the current implementation is to use the multiple tuple space and the system of gates. Indeed, it is relatively easy to create a gate which only has access to a certain tuple space in the same repository. The Figure 3.1 explains the idea in more detail.

- *Integrity*: At the moment, there is no way to secure communication in jSpace. To ensure data integrity, more secure protocols should be implemented in jSpace.

- *Availability*: The current implementation of jSpace does not implement any mechanism to keep the system available.

A certain number of improvements can be added to the jSpace implementation to improve the security of the data in the tuple space. The first thing to add is a system of access rights to the tuple. The basic system described by the authors could be implemented or another more complex system cloud also be designed. To support this system of permission or rights access, a system of tokens could additionally be implemented. The other security problematic is the persistency of the data, but this subject will later be discussed in a dedicated point. To keep the integrity of the data, the use of the TLS protocol could be implemented.

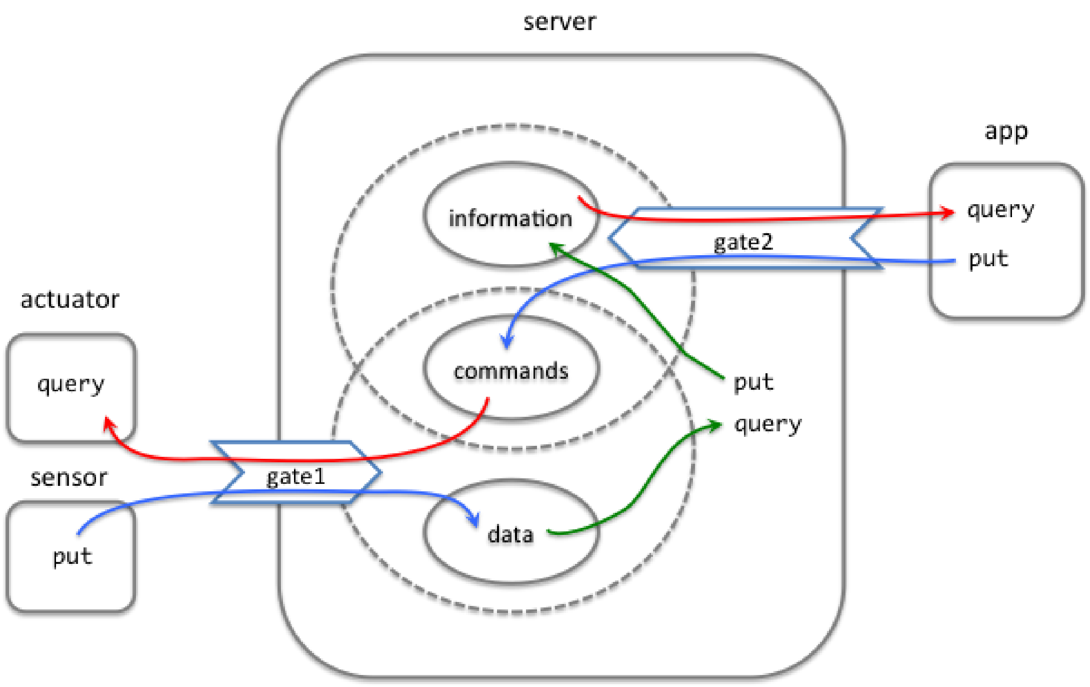


Figure 3.1: The system of gates

3.3.1.6 Performance in jSpace

As seen before, performance relies on two important points: the performance time between the sensors and the tuples, and the performance time to retrieve a tuple in the space. Regarding the first point, jSpace implements the TCP and UDP protocols to transport the data between the various parts of the application. According to the needs of an application, these two protocols must be sufficiently efficient. The distance between the sensors and the tuple server plays a significant role in measuring this performance.

The second point concerns how the implementation manages pattern matching. Below is the code to add a tuple in a tuple space and to retrieve it after.

```java
Tuple tupleA = new Tuple("milk", 1); //Create a new object tuple
space.put(tuple) //Add the tuple in a space

Object[] tuple = fridge.queryp(new ActualField("milk"), new FormalField(
    ↪ Integer.class)); //Retrieve the tuple that has "milk" as first terme

if (tuple != null) {
    int numberOfBottles = (int)tuple[1];
}
```

To find the right tuple, the process that is going to check the template is relatively simple.

Depending on the type of space, the process loops on every tuple in the space until it finds a tuple that corresponds to the template. If the type of space is a sequentialSpace, the process starts at the head of the list of tuples in the space. If the process is a PileSpace, the process starts at the end of the list.

To improve the first point of performance in a fog computing application, more protocols can be added. To improve the process of pattern matching that will be slow if there is a large number of tuples in the space, a new type of space could be added. This new type of space could have, for example, a process to sort the tuples when they are added to the space. This could allow speedier matching with the template.

3.3.1.7 Data persistency in jSpace

There is no mechanism of data persistency implemented in jSpace. If the device that contains the tuple space server is disconnected, the data are lost.

The aim for those criteria will be to add a mechanism to avoid any loss of data. A mechanism of replication of the tuple in different devices could be the solution to mitigate this problem.

3.3.1.8 Data placement policies in jSpace

As seen in the security point, the data can be stored in different tuples and access to these tuples can be managed by gates. Currently, these policies can only be managed on a specific device and not, for example, on several devices in a same domain.

One way to improve the data placement policies is to make it possible to write rules for multiple devices in a same domain. The system of data placement policies must also be more explicit.

3.3.1.9 Summary of the extensions to be added to jSpace

The extensions are classified by importance:

1. Adding a system of access rights to the tuples.
2. Adding a mechanism of data persistency.
3. Adding a mechanism of data placement policies.
4. Improving the matching mechanism with a more efficient one.
5. Adding additional protocols like TLS and/or Bluetooth.

3.3.2 Rustupolis

To validate the first requirement that the implementation work for low-powered devices, tuple-based implementations written in Rust have become a solution. In fact, the Rust programming language allows writing optimised programs for embedded systems. This comes from its memory management system which must be managed by the developer and, unlike languages like Java, is not managed by a garbage collector. On the crates.io website, which contains all the libraries written in Rust, there are two libraries with the 'tuple-space' tag.

The first library called 'rs-object-space' [32] has not been updated for four years. When evaluating the examples given within the implementation, the build was not successful and deprecated function error messages appeared.

The second library called 'Rustupolis' [33] has not been updated for two years, but the tests written in the implementation succeeded. The build of the implementation examples was also successful. For the moment, the Rustupolis library offers three functions:

- Local tuple space for storing tuples and retrieving them via pattern matching.
- Local tuple space with multithreaded and concurrent access.
- Interactive command line interface for creating tuple spaces and pushing or pulling tuples.

One of the objectives of the current project was to have a distributed platform on the network. This has not been developed but the project offers a good basis for the management of primitives and tuples spaces. Below is the list of primitives offered by Rustupolis:

- *out (t)* adds an occurrence of the tuple t to the tuple space.
- *rd(t)* checks if the tuple t is in the tuple space.
- *in (t)* takes the tuple t from the tuple space.

3.3.2.1 Rustupolis' usability on resource constrained devices

One of the important points of the library is that it is written in Rust. Rust is a programming language that seems to be one of the best solutions to write programs for the IoT [34]. Rust ensures compile time memory safety and offers a rich standard library with functional elements. Memory safety and high performance are one of the main reasons why Rust applies semantics at compile time. Rust is a full memory safe language, even across threads. Rust also prevents access to unallocated, uninitialised, freed memory along with pointer addresses beyond data boundaries [34]. Rust is open-source and hosted on GitHub.

3.3.2.2 Scalability in Rustupolis

The current version of Rustupolis only works locally and is not available on a network. This scalability feature is therefore not present in the library.

For Rustupolis to be considered as scalable, the tuple spaces should be available on the network so that clients can easily connect to it. These clients should have the possibility to connect with different communication protocols such as TCP, UDP, and/or Bluetooth. Moreover, installing a tuple space and connecting it to other tuple spaces on the network should be easy. One other element that could be implemented to extend Rustupolis is the addition of a way to manage concurrent requests of a tuple from multiple clients.

3.3.2.3 Security in Rustupolis

Since Rustupolis works only locally, no security mechanisms have been put in place in the current implementation.

To make Rustupolis safe on the network, a security system still needs to be developed. An authentication system must be set up to be able to manage the permission of the different tuple spaces through an access control system. This access control system must also be developed. These two additions will facilitate gains in confidentiality and integrity. As mentioned before, the addition of different secure communication protocols is also a way to gain integrity. The problem with security is the persistency and the availability of the data, but this subject will be discussed later in a dedicated point.

3.3.2.4 Performance in Rustupolis

The performance is divided in two points. The first is the time performance between the sensor and the server containing the tuple space. As Rustupolis only works locally, this point cannot be considered. The other point of the performance requirement is the time to retrieve a tuple when matching with a template. The matching system compares the value of the tuples one by one. This one could be improved because in the of case of an abundance data in a tuple space, it could take a long time.

```rust
pub fn matches(&self, other: &E) -> bool {
    match (self, other) {
        (&E::I(ref a), &E::I(ref b)) => a == b,
        (&E::D(ref a), &E::D(ref b)) => a.to_bits() == b.to_bits(),
        (&E::S(ref a), &E::S(ref b)) => a == b,
        (&E::T(ref a), &E::T(ref b)) => a.matches(b),
        (&E::Any, &E::Any) => false,
        (&E::Any, &E::None) => false,
        (&E::Any, _) => true,
        (&E::None, _) => false,
        (_, &E::Any) => true,
        _ => false,
    }
}
```

The performance between the sensors and the tuple servers must be considered when different communication protocols are developed. The mechanism of matching could be improved to increase the performance of matching. However, this point is less important than the previous ones.

3.3.2.5 Data persistency in Rustupolis

There is no mechanism of data persistency implemented in Rustupolis. If the device that contains the tuple space server is disconnected, the data are lost.

The aim for those criteria will be to add a mechanism to avoid any loss of data. A mechanism of replication of the tuple in across different devices could be the solution to mitigate to this problem.

3.3.2.6 Data policy placement in Rustupolis

There is no way to write rules of data placement in Rustupolis. Since Rustupolis works locally, it seems logical. Currently, the data could only be stored locally.

A mechanism that allows controlling where and in which tuples the data are stored must be developed. To obtain this mechanism, a system of rules could be set up and the storage of these rules could be done in a tuple space.

3.3.2.7 Summary of the extensions to add to Rustupolis

The extensions are classified by importance:

1. Making the tuple servers available on the network and add different communication protocols.
2. Adding a system of access rights to the tuples and a system of authentication.
3. Adding a mechanism of data persistency.
4. Adding a mechanism of data placement policies.

3.4 Concluding remarks

In this chapter, the different criteria for creating a coordination language implementation in fog computing have been presented. These criteria consider all the challenges of fog computing such as security, scalability, resource constrained devices, etc., and express how these challenges should be handled by the coordination language. After defining these criteria, two existing languages were analysed. It has been determined that Rustupolis is an extensible base. Indeed, the Rustupolis library provides the basic primitives of Linda, but does not work in networks and has no security mechanism. In this chapter it was also proven that Rustupolis is a better extensible base than pSpace. The fact that the library is written in Rust was a determining factor in the choice of Rustupolis. In addition, the Rustupolis code is well-organised and easily understandable. In the next chapter, the different extensions to Rustupolis are explained.

Chapter 4

Implementation

4.1 Introduction

In this chapter, the strengths of the Rust programming language and the key mechanisms that made the choice of this language decisive are discussed. Next, the existing Rustupolis code is analysed by dividing the code according to the key concepts of the library. In the following sections, the different additions made to Rustupolis to meet the fog computing criteria are detailed. These additions thus make it possible to build a PoC which meets the criteria previously defined. After this stage, it is possible to create a tuple space server and to store data within it. To send or retrieve data to and from this server, a "client" section has been developed. Next, a section explains how an interpreter written in Rust for the language is built which allows communication with a tuple server. In this same section, the development of the client section of the Rust library is also explained. There are two possible ways to access a tuple server, either via the Rust library or an interpreter for a language. These forms of accessibility are explained in the rest of the chapter.

4.2 Rust

Rust is a system programming language developed by the Mozilla Foundation. Rust is open source and hosted on GitHub. [35] The traditional system programming language like C or C++ leave the security of memory management to the developer. However, according to Microsoft, 70% of software vulnerabilities are due to memory safety issues. [36] [37]

Hence, there was a need for a low-level language to create software that was robust against memory errors. This is the reason behind the development of Rust and why it offers developers an insurance for memory management. In other languages such as Java, a garbage collector takes care of memory management. However, Rust is a systems programming language, where minimising the consumption of resources such as CPU time and memory usage is one of the main concerns. Therefore, giving up this control is not an option. In Rust, the developer still needs to manage the memory, but must respect a number of principles to avoid memory errors. It is at compile time that Rust checks that the principles set up by the language to avoid memory errors are verified. These principles will be detailed further later in this chapter. Rust also offers high performance when processing substantial amounts of data, support for concurrent programming, and an efficient compiler. For all these reasons, Firefox, Dropbox, Cloudflare use Rust in production.

The popularity of Rust has steadily increased in recent years. The Linux kernel developers came up with the idea of adding new functionality written in Rust to the current kernel written in C [38]. Microsoft and Google are also planning to use Rust for Windows and Android respectively [39] [40]. However, it is not only large corporations that are being won over by Rust. According to the StakeOverflow 2021 survey of 80,000 developers, Rust is the most loved language. This is the 6th year in a row that Rust has been the top language in this category [41].

4.2.1 Advantages of Rust

- *High performance and memory security*: Rust offers mechanisms to manage the memory while keeping a high level of performance. These mechanisms are explained in the next section.
- *Support for concurrent programming*: Rust handles concurrency, and the compiler prevents the developer from writing code where two threads are trying to access a resource at the same time. Rust then offers the possibility to manage a lock that will be taken and released by a thread on a resource.
- *The package manager*: Rust's package manager called Cargo allows simple management of a project's dependencies. It also allows managing these dependencies on crates.io [42]. Crates.io is the site where one can find all the libraries written in Rust. Cargo natively builds and tests Rust applications for multiple platforms. Strict dependency versioning and document generation show that Cargo is also powerful enough for cross-platform oriented applications [34].
- *The community*: There are several official and unofficial sites to get help, and Rust has a moderator system and a code of conduct to keep the community healthy.
- *Backward compatibility and stability assured*: Rust promises to be backwards compatible with older versions. A tool called crater [43] has also been developed to assess whether the libraries available on crates.io are still functional with other versions of Rust.

4.2.2 Explanation of the key mechanisms of Rust

4.2.2.1 Ownership

According to [44], possession in Rust can be defined in three rules.

- Each value in Rust has a variable that is called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value is dropped.

The third rule explains a simple principle that can be found in other programming languages, which is that a variable is no longer available outside the scope in which it was initialised. This simple example below explains the principle:

```
{                           // hello is not valid here, it’s not yet
    ↪ declared
    let hello = "hello";    // hello is valid from this declaration

    // usage of the variable hello
}                            // the scope is over, and hello is no longer
    ↪ valid
```

The first and second rules explain that a value can only be linked to exactly one value at a time. The following examples show how this translates into Rust code:

```
let number = 1;
let second_number = number;
```

In this example, the value 1 is assigned to the *number* variable. In the second line, the *second_ number* variable to the variable *number* is initialised. As per the second rule, the two variables cannot point to the same value. However, this will work in Rust, as it is a copy of the value of number that will become the value of the variable *second_ number*. Rust was able to make a copy of the number value because integers are simple values with a known, fixed size, and are pushed onto the stack. At the end of this example, we will have two variables with a value of 1. In the next example, the value does not have a fixed size, so its behaviour will be different:

```
let string = String::from("hello");
let second_string = string;

println!("{},␣world!", string);
```

In this third example, a string object *"Hello"* is assigned the variable *string*. On the second line, the variable *second_ string* is initialised with the first variable. The code then tries to print on the "string" variable. This will not work, and the compiler will display an error. In fact, in the case of an object like a string whose size is not fixed, Rust does not perform a copy pass as in the earlier example. Since a variable can only have one owner, the solution for Rust here is to pass the ownership of the string object from the *string* variable to the *second_ string* variable. The passage of ownership is called ‘moves‘. A variable moves when passed as a parameter to a function. The first variable will then no longer have a value, which will generate an error at print time. If the two variables point to the same value, when the scope ends and the memory of the variables is freed, the same memory will be freed twice. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities [44]. To have the same string in two variables, the solution is to explicitly clone the first variable into the second.

4.2.2.2 Mutable reference

In Rust, variables are immutable by default. However, if a value must be modified, and it must remain in the same variable, the keyword *mut* must be added to the initialisation of the variable. This keyword will then allow a mutable reference to the variable, and thus to avoid a change of ownership. Indeed, when a call to a function occurs, all the values passed in the parameter must not necessarily have their ownership passed to the function. If the program requires these values later, the ownership does not have to be passed. The passing by reference solves this problem. If a variable is not marked as mutable, its reference will be immutable. The following example details this:

```rust
fn main() {
    let mut string = String::from("hello");
    let second_string = String::from(", world");

    change(&mut string, &second_string);

    println!("Variable 1 : {}",&string);        //"Variable 1 : hello, world
    ↪ " is printed
    println!("Variable 2 : {}",&second_string); //"Variable 2 : , world" is
    ↪ printed
}

fn change(string1: &mut String, string2: &String) {
    string1.push_str(string2);
}
```

In the above example, two variables are created. The first one is mutable and the second one by default is not. These two variables are passed by reference to the function which concatenates the second one into the first. The first variable will therefore be modified while the second will not. This is the reason the first variable must have the keyword mut.

However, the risk with a mutable reference is that it can create a memory error called "data race". This error occurs when the following three behaviours occur [44]:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is used to write to the data.
- No mechanism is used to synchronise data access.

To avoid this error, Rust has put conditions on the use of the keyword "mut". The first is that you can have only one mutable reference to a particular piece of data at a time. Moreover, you may not have mutable reference when you have an immutable reference to the same value [44]. The process that uses an immutable reference does not expect the value to change. However, multiple immutable references are allowed because with an immutable reference, the value will not change.

4.2.2.3 Lifetime

When developers use a reference, they borrow ownership of the variable and thus grant temporary access to a data structure. Lifetimes are used to determine how long the reference of a variable will be used. The compiler uses these references to detect memory errors. The objective of lifetimes is to avoid dangling references. Dangling references are those that point to a value that has been dropped and is therefore no longer valid. The following example shows how the borrow checker in the compiler checks the lifetime of references.

```
{
    let first_variable;                         // -------------+-- 'a
    {                                           //              |
        let second_variable = 5;                // -+-- 'b      |
        first_variable = &second_variable;      //  |           |
    }                                           // -+           |
    println!("r: {}", first_variable);          //              |
}                                               // -------------+
```

In this example, the variable *first_ variable* is initialised and the lifetime 'a is linked to it. Then, another scope opens and a *second_ variable* is initialised in it. The lifetime 'b is linked to the reference of this variable. The compiler then checks that the reference assigned to *first_ string* has a lifetime long enough to last at least as long as that of the *first_ string*. The lifetimes of 'a and 'b are compared and since the lifetime of 'b is smaller than that of 'a, an error indicating that the *second_ variable* does not live long enough' occurs. Lifetimes work differently for functions. In most cases, the compiler itself will determine which lifetime is bound to which reference. However, if the function returns a reference and has multiple parameters referenced in its signature, the developer will have to specify the lifetime themselves. This below example, from [44], shows this case and how to define the lifetime:

```
    fn main() {
        let first_string = "abc";
        let second_string = "xyz";

        let result = longest(first_string, second_string);
        println!("The longest string is {}", result);
    }

    fn longest<'a>(first_string: &'a str, second_string: &'a str) -> &'a str
    ↪   {
        if first_string.len() > second_string.len() {
            first_string
        } else {
            second_string
        }
    }
```

In the following example, the "longest" function receives two references to strings as parameters. It will then return the reference of the string that is the longest. However, it cannot determine which strings received as parameters will be returned. The lifetime of the reference returned is therefore unknown. The compiler needs this lifetime to know when the reference will be dropped after the function call. The developer must therefore determine this themselves by adding a lifetime to each reference. These lifetimes are represented by "'a" and they can have any name.

4.2.2.4 Option type

First, the type of variables in Rust are static, meaning that the type of variable can never change. Rust does not have a Null type, but, as in Haskell, it has an optional type that the developer must manage. This avoids Null pointer exception errors. Rust also offers the possibility, as in Haskell, to do pattern matching on enumerations. The Option type, being an enumeration of the None type and the Some(T) type, allows for easy management of the optionals. The following example shows this solution:

```rust
fn manage_optional(string: Option<String>) {
    match string {
        Some(string) => println!("{}", string),
        None => println!("No␣string"),
    }
}
```

4.2.3 Rust in the IoT devices

In this section, the advantages of Rust for developing programs for IoT devices will be discussed. Of course, the above-mentioned advantages also play a key role in the use of Rust for IoT. Indeed, the performance gain coupled with the security offered even across threads when using memory is an especially important advantage of Rust. As the main idea behind using Rust is to program a safe, low-level task with the use of high-level programming concepts, designs with embedded hardware are a typical application for this purpose [34].

One advantage that Rust has over higher-level languages is that it is directly compiled into machine language. An opposing example is Python, which is compiled in C before being compiled in machine language. Rust therefore has a compilation speed equivalent to that of the C language. This makes it possible to reduce the complexity in time and space of compiled programs, which is not negligible for the IoT sector.

Rust has a foreign function interface. With that interface, Rust can make use of services written in other programming languages. This interface can only work because of another feature of Rust, which is the "unsafe" keyword. This allows memory protection rules to be avoided. It can be used when the application takes control of Rust's security mechanisms. Rust has many libraries for mobile devices. For example, one library allows Rust to send messages to the Objective-C runtime, which is the basis of Apple's operating system [45].

4.3 Code of Rustupolis in details

The following section presents how the Rustupolis library works and what its unique features are. The section is divided into four parts covering the important concepts of the library. Thus, the library makes it possible to create tuple spaces and to use Linda's primitives to interact with them. The library offers methods for creating a tuple space and for managing tuples.

4.3.1 Tuple

The first concept to present for the library is that of tuples. A tuple is represented as an object containing a list of values. These values are represented by the enumeration E which allows for the management of the different types of data which can be present in a tuple. This enumeration is presented below:

```rust
pub enum E {
    /// Integer data type.
    /// Implemented as 32-bit integer (i32).
    I(i32),
    /// Floating point data type.
    /// Implemented as double precision (f64).
    D(f64),
    /// String data type.
    /// Implemented as String.
    S(String),
    /// Tuple data type.
    // Implemented as vector of tuple types (Vec<E>).
    T(Tuple),
    /// Any data type.
    /// In context of this tuple, Any stands for the wild card that is used
    ↪ for pattern matching when querying the tuple space for certain tuples,
    ↪  and marks the beginning of a matching range when searching for
    ↪ matching tuples.
    Any,
    /// None data type. In context of this tuple, None represents "no match"
    ↪  when searching, and marks the end of a matching range when searching
    ↪ for matching tuples.
    None,
}
```

A tuple can contain any of the above values. This means that a tuple can contain a type T, which itself contains a tuple and therefore allows for the nesting of tuples. An important method of the library is implemented on this type. Accordingly, the library allows for the comparison of type E's according to their type E. In particular, this method is used at the time of the pattern matching and has already been presented; see section 3.3.2.4. This method is called on a type E represented by the first parameter *self* and will be compared to the second parameter also of type E. The function has a boolean return type depending on whether the types are similar or not.

Different methods are also implemented to create tuples and to access the values within them. The matching method presented here allows for the comparison of two tuples and uses the method previously presented to compare the values present within these tuples. As shown in the example below, Rust offers methods to elegantly achieve this matching:

```rust
pub fn matches(&self, other: &Tuple) -> bool {
    (self.is_empty() == other.is_empty())
        && self
            .0
            .iter()
            .zip(other.0.iter())
            .all(|(ref x, ref y): (&E, &E)| x.matches(y))
}
```

4.3.2 Tuple space - Store

The object called SimpleStore in Rustupolis is what contains the tuples and allows operations to be conducted on them. The operations are the primitives of Linda. They are represented in a feature called *Store*. The traits in Rust are the equivalent of the interfaces in Java. They define the signature of different functions to allow several structures to implement them. The Store feature makes it possible to implement a different tuple space than the one implemented by the library. The trait is defined as follows:

```
pub trait Store {
    /// Read a matching tuple and remove it atomically.
    fn inp(&mut self, tup: &Tuple) -> Result<Option<Tuple>>;
    /// Read a matching tuple.
    fn rdp(&mut self, tup: &Tuple) -> Result<Option<Tuple>>;
    /// Write a tuple.
    fn out(&mut self, tup: Tuple) -> Result<()>;
}
```

Here we find the different primitives of Linda and the signature of the different methods. As mentioned earlier, a tuple space in this library is a SimpleStore object. This object contains a list of tuples and implements the methods of the Store trait. The distinctive feature is that the list of tuples is of type BTreeSet. The BTreeSet type is an ordered set based on a BTree. A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time [46]. The BTree allows a node to have more than two children, which allows for a continually balanced tree.

4.3.3 Match - Space

The concepts of Match and Space allow management of the Future type of pattern matching. This allows for the management of requests in an asynchronous way. This works thanks to the Match enumeration, whose definition is as follows:

```
pub enum Match {
    Done(Result<Option<Tuple>, Error>),
    Pending(Receiver<Tuple>),
}
```

This enumeration is used in the methods of the Space object. A Space object contains a SimpleStore which is a tuple space. The purpose of the Space object is to use redefining primitives to perform operations on the SimpleStore. The purpose of the redefinition is that the various methods must return a Match type and thus implement the Future type.

4.3.4 Rustupolis in the fog computing

In the implementation developed in this thesis, Rustupolis will be used for its primitives and its management of tuple space. The network part, the access rights system to the tuple space and the encryption of requests between servers are added to the Rustupolis library. The list of elements to be added to Rustupolis so that the library can be functional in fog computing is detailed in the section 3.3.2.7.

4.4 Implementation of the server part of the library

In this section, the implementation of the part to create a tuple space server on a device is explained. The way to develop the access control system and the data encryption is also explained. The GitHub repository link for this part of the project is as follows: `https://github.com/Maxbever/LIF_Server`

4.4.1 Make the tuples servers available on the network and adding protocols

The main problem of the Rustupolis library is that the tuple spaces are only locally available. The first step to make this project in fog computing is to have the possibility to manipulate the tuple through the network. To achieve this, communication protocols must be added. A client will then be able to connect to a remote tuple space using a specific IP address and port. For this first implementation, the TCP and UDP protocols are implemented. An object server abstracts the details of each protocol, allowing protocols to be easily added to the project. To implement these means of communication, choices have been made in the implementation. These choices and the functioning of this part are presented in the rest of this section.

The first implementation is the concept of a repository. This concept is inspired by the one seen previously in pSpace, with some notable differences. A repository will contain a list of tuple spaces and a reference to a tuple space in the list, which will contain the permissions and access rights of the users. This particular tuple space is detailed in the section 4.4.2 on implementing the access control system.

Before going into more detail on the implementation, the ways to use this tuple server will be discussed. The code below presents the main method which allows a TCP and UDP server on the desired IP address and port to be launched. The different servers are executed on threads:

```rust
fn main() {
    let ip_address = String::from("127.0.0.1");
    let port_tcp = String::from("9000");
    let port_udp = String::from("9001");

    let repository = Repository::new();

    let server_tcp = Server::new(server::Protocol::TCP, &ip_address, &
    ↪ port_tcp, &repository);
    let server_udp = Server::new(server::Protocol::UDP, &ip_address, &
    ↪ port_udp, &repository);
10
11      let server_list = vec![server_tcp, server_udp];
12
13      crossbeam::scope(|scope| {
14          for server in server_list {
15              scope.spawn(move |_| match server.start_server() {
16                  Ok(_) => {
17                      println!("{}", "OK␣")
18                  }
19                  Err(error) => {
20                      println!("{}", error)
21                  }
22              });
23          }
24      })
25      .unwrap();
26  }
```

In line 6, a new repository is created. In the case of the above code, the repository that contains the tuple spaces is shared between the two TCP and UDP servers. It is also possible to create a different repository for each server, and it is easy to create a new server with a new protocol that would also have access to this repository. Lines 13 to 25 are used to start the servers on different threads. The communication with the server is achieved using different commands. To connect to a server, the client can either use the interpreter with the various syntax commands, see section 4.5.3, or use the client part of the library which allows the same commands in Rust.

4.4.1.1 Management of concurrency

Now that the operations of the server are defined, a way to oversee concurrency can be defined. As mentioned before, Rust prevents the concurrent modification of a resource by a thread. This is of course rightly so, since trying to access the same memory at the same time would create an error. Concurrency must therefore be managed on a tuple space and the tuple space list. Indeed, a server can either modify or read tuples in a tuple space or create or delete the tuple space list at a higher level. To deal with this kind of situation, Rust provides different ways to manage a lock on a resource to ensure that two threads do not access it at the same time. As mentioned above, a different mechanism has been used to oversee the two concurrency cases.

The first mechanism used is the encapsulation of a tuple space in a *Mutex* object, which itself is encapsulated in an *Arc* object. The Atomically Reference Counted (*Arc*) object is a thread-safe reference counting pointer. The type *Arc<T>* provides shared ownership of a value of type T, allocated in the heap. Invoking clone on *Arc* produces a new *Arc* instance, which points to the same allocation on the heap as the source *Arc*, while simultaneously increasing a reference count. When the last *Arc* pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as "inner value") is also dropped [47]. The Arc object thus makes it possible to ensure the sharing of a resource between different threads and ensures that the resource will not be dropped during execution. However, as seen before, Rust prevents the modification of a resource shared between several threads. Thus, the Mutex object has been used.

The mutex object is a mutual exclusion primitive useful for protecting shared data. The mutex will block threads waiting for the lock to become available. Each mutex has a type parameter which represents the data that it is protecting. The data can only be accessed through the guards returned from the method lock, guaranteeing that the data is only ever accessed when the mutex is locked [48]. The combination of a Mutex object in an Arc object therefore ensures that a resource always exists between threads but also that two threads do not access the resource at the same time. This allows total security in terms of memory management. This combination is used to store the tuple space list in a repository.

In reality, it is not a list of tuple space that is stored, but a list of Arc encapsulating a Mutex which itself encapsulates a tuple space. This combination is ideal as it allows different requests to be managed on the same tuple space thanks to the lock. When an operation such as a read or write takes place on a tuple space, the thread takes the lock. It then performs the operation on the tuple space and releases the lock. However, this system is not optimised to solve the second concurrency problem. This problem occurs when two threads want to modify or read the entire tuple space list. Since both threads can modify the tuple space list at the same time by adding or deleting, a lock must also be set on this list. If the Mutex object is chosen to perform this operation, it means that if two threads simply want to read a tuple in different tuple spaces, the first one will take the lock and the second one will be blocked even though it does not want to access the same tuple space. The solution to this problem was to use the Rwlock object instead of the Mutex object.

The RwLock object provides a lock system, but allows, at the time of taking the lock, an indication of whether a read or write operation is going to be done on the data. The object then optimises the access to the data to block the threads only in case of writing. This solution is ideal for the tuple space list. Indeed, the tuple space list will be blocked only when a tuple space is added or removed from it. This is not an operation that will often happen in a normal use case of the project. When a tuple is added to a tuple space, the thread will read the tuple space list. Consequently, the lock will only be taken on a tuple space and not on the whole tuple space list. The following Figure 4.1 better expresses the difference between the Mutex object and the RwLock object.

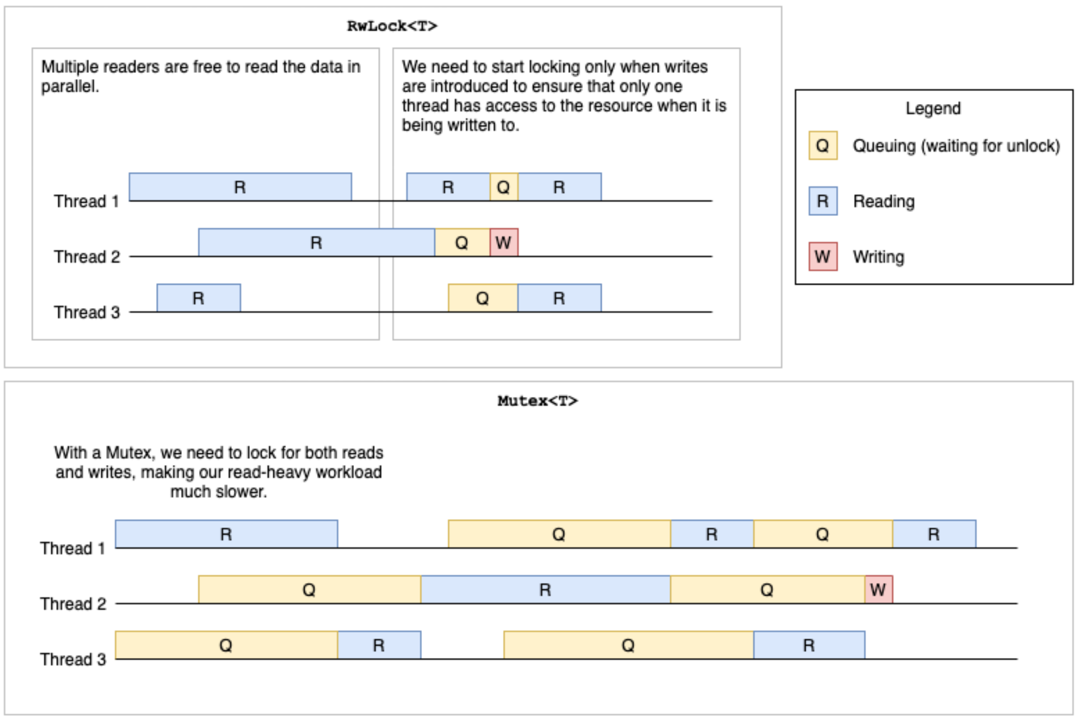

RwLock<T>
Multiple readers are free to read the data in parallel.
We need to start locking only when writes are introduced to ensure that only one thread has access to the resource when it is being written to.
Thread 1
Thread 2
Thread 3
R
Q
W
Legend
Q Queuing (waiting for unlock)
R Reading
W Writing
Mutex<T>
With a Mutex, we need to lock for both reads and writes, making our read-heavy workload much slower.

Figure 4.1: Comparison between Mutex and RwLock [49].

The concurrency will be managed in our server by these different mechanisms. To summarise, the structure of the Repository object is shown below:

```
pub struct Repository {
    tuple_spaces: Arc<RwLock<HashMap<String, Arc<Mutex<Space<SimpleStore
    ↪ >>>>>>,
    permission_tuple_space: Arc<Mutex<Space<SimpleStore>>>,
}
```

4.4.2 System of access right

The first step to find the ideal system of access control for fog computing is to define the requirement of the system in the fog paradigm. According to [50], [51], below are the requirements for a fog computing access control system:

- The access control system must take into consideration that IoT devices have limited resources. The usage of a process that is too powerful can lead to a computational overhead.
- The system must have the capacity to create, delete, and revoke policy.
- The access control system must have the possibility to revoke attributes for a determined user. The user who has an attribute revoked does not have access to the data.

- The time to decide if a user can access the data must be low. One of the advantages of fog computing is the low latency. This should not be spoiled by the access control system.
- The access control system depends largely on the requirements of the application. An application in the health care sector must have a better secured access system than any other application. However, a more secured system is often less efficient.

To reach the perfect system that responds to the requirements, several types of access control systems have been analysed:

- Attribute based access control (ABAC): This model is based on the attribute. An attribute is a feature that is given to a user, a resource, or the environment depending on its needs and its role in the application. The policies are defined by the owner of the data. The users are assigned to some attributes. The user can access data only when their attributes satisfy the specified access policy.
- Discretionary access control model (DAC): This model is based on the identity of the user who request access. An authorised entity can thus grant the permission for the user. The entity that owns the data can set the access permissions based on users' identities in some group. A DAC model is more flexible and less secure, and therefore it is generally used in environments that emphasise convenience and does not need a high level of security, such as the UNIX operating system.
- Role-based access control model (RBAC): This model is based on the role. The different permissions like write, read, create, and update are bound to a role and not to a user. When they are created, every user receives a role. When the user requests to use some data, their role is compared to the policy rules. Depending on their role, the access is permitted or rejected. This type of access control is very scalable. Indeed, if you add a rule to a role, all the users who already assigned the role also, by definition, are assigned the new rule. The drawback of this system is that if you want to add a special permission to a specific person, a new role must be created. This can become difficult to manage if there are many permissions possible.
- Rule-based access control model (RBAC): This model allows the association of a list of rules in the data. When a user tries to access to some data, the system checks if they respect all the rules.
- Mandatory access control model (MAC): Mandatory access control is a method of limiting access to resources based on the sensitivity of the information that the resource contains. It is also based on the authorisation of the user to access information with that level of sensitivity. You define the sensitivity of the resource by means of a security label. The security label is composed of a security level and zero or more security categories. The security category defines the category or group to which the information belongs (such as Project A or Project B). Users can only access the information in a resource to which their security labels are entitled [52].

To choose the best access control system for the implementation, the type of application that is going to use the software needs to be defined. As mentioned earlier, the level of security needs to be determined to choose the best solution. An application that is going to work in the health care sector or in the self-driving cars sector needs to have a more secure access control system than other applications in different sectors. In the case of this work, the goal is to find the most optimised solution that has a reliable level of security without compromising performance.

According to the authors of [50], [51], the best type of access control for fog computing is the ABAC model. Indeed, this model highly scalable because an attribute simply needs to be assigned to a user and thereafter the user has complete access to data with that attribute. Different permissions would then be assigned to an attribute, and the user with that attribute could then have the necessary permissions.

To add this access control system to the implementation, certain choices were made. Firstly, as no authentication system has been developed in the project, the client must send the attributes to the server itself. This can be achieved by means of the various commands in the above section 4.5.3. The permissions linked to an attribute are stored in a dedicated tuple space which is present in the repository tuple space list. To maximise performance, as the permissions check is done for each operation on a tuple space, an Arc object having the pointer to the tuple space containing the permissions is retained in the Repository object see section 4.4.1.1. This avoids having to browse the list of tuple spaces when permissions need to be checked. Permissions are stored in a tuple in the following format:

```
({tuple_space_name}, {action}, ({attribute}, {attribute}, ...))
```

The first parameter of this permission type is the name of the tuple space. Next, the action that is affected by this permission is found. The actions can be the out, in, read, and/or delete operations. The last argument is the list of attributes that allow the permission. The format of this tuple has an exception; the tuple that allows the creation of a new tuple space on a server and therefore has no name:

```
(create , ({attribute}, {attribute}, ...))
```

To check that a user has permission to perform an operation, simply call the read primitive with the tuple name and the requested operation. The primitive should return the tuple with the list of attributes, allowing the operation on the tuple space as the third element. This list is then compared with the list the user entered when using the attach command. If an attribute is present in both lists, the user is allowed to perform the operation and the operation continues to run. If there is no match, the operation stops, and the user receives an error message telling them that they do not have permission. The permission rules are created with the creation of the tuple. The client who creates the tuple space can then choose whether the same attributes are assigned for each operation, or whether they choose different attributes.

When the repository is created, the very first tuple space created is the one containing the permissions. By default, the tuple space creation rule and the *out*, *in*, *read* rules on the permission tuple space are set with the administrator attribute. The only permission that does not exist is the delete operation to prevent the permission space tuple from being deleted. This allows an administrator or any user with the necessary attributes to perform operations on the permissions in the tuple space just like they would in any other tuple space in the repository.

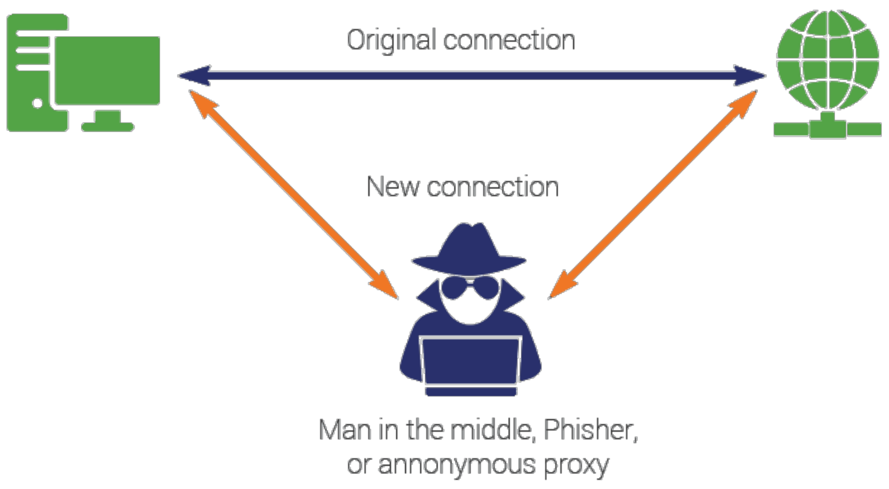


Figure 4.2: Man-in-the-middle attack schematic diagram [54].

4.4.3 Encryption of communication

Now that the access control is secured by means of an attribute, the messages must be protected against attackers trying to retrieve it by sniffing the network. In fact, the transmitted data must be secured against man-in-the-middle attacks. This attack is a cyberattack where the attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other, as the attacker has inserted themselves between the two parties [53]. The scheme of this attack is explained in the Figure 4.2.

To prevent an attacker from reading the transmitted data, one solution is to encrypt it. An encryption method has therefore been defined to achieve this. As our solution is to operate in fog computing, the criteria defined above for our solution should also be applied for the choice of the encryption method. The following criteria must be considered when choosing an encryption technique:

- The encryption solution must work on resource-limited devices and avoid using all the resources of the device.
- Encryption and decryption should be performed in a short time.
- The solution must nevertheless be secure and provide a high enough level of encryption.

First, it was necessary to define whether a symmetric or asymmetric encryption method would be used.

- *symmetric encryption*: Symmetric encryption is also called secret key encryption, and it uses just one key, called a shared secret, for both encrypting and decrypting. This is a simple, easy-to-use method of encryption, but there is one drawback: the key must be shared between the sender and the recipient of the data, so a secure method of key exchange must be devised. Otherwise, if a third party intercepts the key during the exchange, an unauthorised person can easily decrypt the data [55].

- *asymmetric encryption*: To address the problem of key exchange, another type of encryption was developed. Asymmetric encryption is also called public key encryption, but it in fact, it relies on a key pair. Two mathematically related keys, one called the public key and the other called the private key, are generated to be used together. The private key is never shared; it is kept secret and is used only by its owner. The public key is made available to anyone who wants it. Due to the time and amount of computer processing power required, it is considered "mathematically unfeasible" for anyone to be able to use the public key to re-create the private key. Consequently, this form of encryption is considered extremely secure [55].

The encryption system chosen is a symmetrical encryption system. This is because it offers better performance in terms of encryption/decryption time and resource time. Asymmetric encryption would take longer to connect and require more performance, which would not meet our criteria, see section 3.2. However, the transmission of the encryption key as well as the access control must be conducted by a third-party system. This solution is not more optimal in terms of security and distribution, given the passage through this third party. Nevertheless, it is the solution that appeared to be the most relevant for the encryption of requests. Indeed, asymmetric key solutions are difficult to implement in the fog.

The symmetric encryption method used is the Advanced Encryption Standard Galois/Counter Mode (AES GCM) method. The architecture of the AES GCM encryption algorithm is explained in the Figure 4.3. AES GCM is known for a design-based principle which has substitution and permutations and is said to be fast in both software and hardware [56]. It has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. In the case of our PoC, a 128-bit key is used. AES considers each block as a 16-byte (4-byte x 4-byte = 128 bits) grid arranged in a column major order. Advanced Encryption Standard Galois/Counter Mode will be performing many rounds of transformation to convert the plaintext to cipher [56]. The number of encryption rounds depends on the number of bits in the chosen key.

- 10 cycles of repetition for 128-bit key.
- 12 cycles of repetition for 192-bit key.
- 14 cycles of repetition for 256-bit key.

Each round contains four steps which are described below:

- *SubBytes*: In this permutation step, each byte is substituted by another byte. It is performed using a lookup table also called the S-box. This substitution is done in such a way that a byte is never substituted by itself and is also not substituted by another byte which is a compliment of the current byte. The result of this step is a 16-byte (4 x 4) matrix as mentioned before [57].

- *ShiftRows*: Each row is shifted a particular number of times.
  - The first row is not shifted.
  - The second row is shifted once to the left.
  - The third row is shifted twice to the left.
  - The fourth row is shifted thrice to the left.
- *MixColumns*: This step is a matrix multiplication. Each column is multiplied with a specific matrix and thus the position of each byte in the column is changed as a result. This step is skipped in the last round [57].
- *AddRoundKey*: Now the resulting output of the previous step is submitted to an XOR operation with the corresponding round key. Here, the 16 bytes are not considered as a grid but simply as 128 bits of data [57].

The encrypted text decryption stage performs these steps in opposite using the supplied key to recover the plain text.

For the PoC, the encryption was performed in Rust using the RustCrypto [58] library. This library was audited by an external company called NCC Group, which did not find any security flaws.

4.5 Implementation of an interpreter for a language client

In this section, the creation of a language using the commands presented in the server is detailed in the section 4.5.3. The language called LiF (Linda In the Fog) has been implemented through an interpreter written in Rust. In addition to using the server commands, the languages allow certain operations on tuples and some basic calculations. The choices of language construction and interpreter implementation will be presented in the next chapter. A use case of the language and the server is also presented in the last section of this chapter. Before going into the details of the implementation, an example of what is done with the language is presented. The GitHub repository link for this part of the project is as follows: `https://github.com/Maxbever/LIF_Interpreter`

4.5.1 LiF language

To present the language, an example code showing the different functionalities of the language is presented below. The following code connects to two servers, retrieves and processes data from the first one before sending the processed data back to the second server.

```
var server_udp_name = "UDP_server"
var admin_attribute = "admin"
var tuple_space_name = "data"
var tuple_space_name_mean = "tuple_space_mean"
var attribute = "attribute"
var key = "secret_encrypt_key"

connect server_tcp_name tcp:127.0.0.1:9000 "secret_encrypt_key"
```

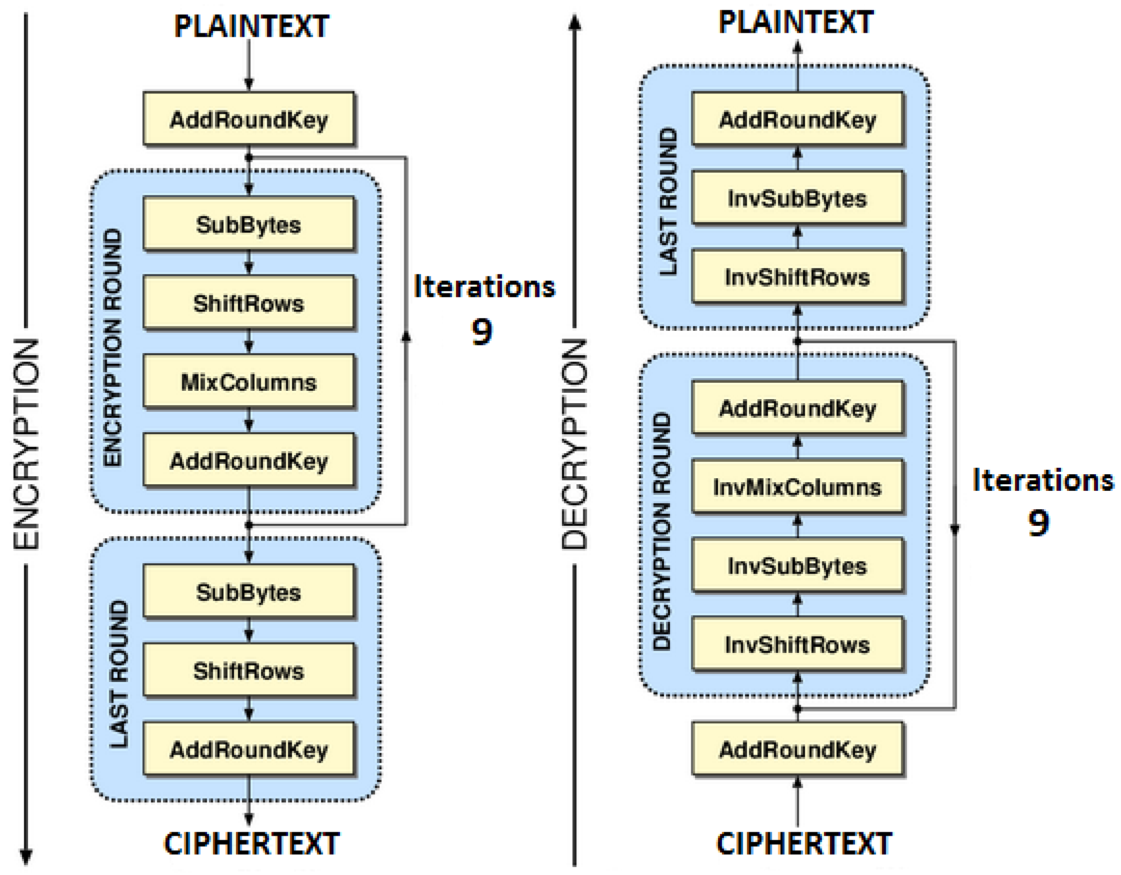


Figure 4.3: Architecture of AES Algorithm [56]

```
 9 connect server_udp_name udp:127.0.0.1:9001 key
10
11 create admin_attribute server_tcp_name:tuple_space_name attribute
12 create admin_attribute server_udp_name:tuple_space_name_mean attribute
13
14 attach server_tcp_name:tuple_space_name attribute {
15     out ("temp", 21),("temp", 23),("temp", 29),("temp", 25),("temp", 20)
16
17     var data = in ("temp", _),("temp", _), ("temp", _), ("temp", _), ("temp"
       ↪ , _)
18     var sum = 0
19
20     for iterator = 0 to (data.len()) {
21         var tuple = data.get(iterator)
22         var sum = sum + tuple.get(1)
23     }
24
25     var mean = sum / data.len()
26     var test = mean
27 }
28
29 attach server_udp_name:tuple_space_name_mean attribute {
30     out (mean)
31 }
```

The first seven lines of code define the variables that will be used in the rest of the code. Lines 8 and 9 connect to a TCP and UDP server, respectively. A name must be associated with each server. The name of the server is used in different operations. The server's name is used for the create and delete operations. This determines on which server the tuple space should be added or deleted. The key used to encrypt the data to the server is also passed as a parameter to this operation. An example of tuple space creation is shown in lines 11 and 12. In these two creation examples, only one attribute is indicated, meaning that this attribute will be used to carry out the various access permissions on the tuple spaces. If the client had entered four attributes, these would have been respectively linked to the permissions: read, in, out, and delete. In lines 14 and 28, the attach command indicates on which tuple space of which server the following operations must be executed. The scope of the attach operation makes it easy to see on which tuple space the next operations are taking place. In addition to this, the client must also indicate the attributes it has. The client can therefore specify between one and four attributes. The operations from lines 16 to 26 will therefore take place on the server and the tuple space defined on line 14, whereas the operation on line 30 takes place on the server and the tuple space defined on line 28.

On line 16, the out operation writes to the tuple space. The out operation can take one or more tuples separated by a comma. In this example, the data is fake and added manually, but one could imagine that a sensor has added these different data to the tuple space.

On line 18, the in operation allows getting the tuples that correspond to the pattern list passed as arguments. The operation can have one or more patterns as arguments. These will be managed by the server sequentially, and a list of tuples respectively corresponding to each pattern will be returned to the server. This list will be a tuple which contains each response tuple. The read operation, which is not present in this example, can also be used to indicate a list of patterns and receives a list of tuples in response. However, as it does not remove tuples from the tuple space, this must be considered when writing patterns. Indeed, in this example, if the *in* operation is replaced by a *read* operation, the "data" variable would receive:

```
(("temp", 21),("temp", 21),("temp", 21),("temp", 21),("temp", 21))
```

The rest of the program uses a series of operations available on tuples and number data. Indeed, the language does not allow operations on strings. A *for* loop allows browsing the list of tuples received in response. To use this for loop, a range must be defined. It is each number of this range which will be passed to the iterator, and which will allow recovering the value of a tuple. To retrieve the value of a tuple, the *get(number)* function takes a number as an argument which indexes it to a value of the tuple. The operation then returns the value corresponding to the index. For the loop to stop at the last element of a tuple, the size of the tuple must be known. For this, the *len()* function has been implemented and returns the size of the tuple. Addition, subtraction, multiplication, and division operations are available for any numerical value.

Instead of the for loop, a *while* loop is also available in the language. The while loop allows for incrementally steps to be performed and, as seen in the following example, for the retrieval of data until an empty tuple is received. The empty tuple is represented as follows: *()*. The following code performs the same operation as lines 23 to 29 in the above example code.

```
var data = in ("temp", _)
var sum = 0

while ( data != () ){
    var data = data.get(1)
    var sum = sum + data
    var data = in ("temp", _)
}
```

The while loop offers many possibilities for additional loops and operations. The *while* loop works with a boolean operation, meaning that different symbols for doing operations need to be added to the interpreter's grammar. These different symbols are:

- ==: allows testing the equality between two values.
- !=: allows testing the difference between two values.
- <=: allows testing whether value A is smaller than value B.
- >=: allows testing whether value A is greater than value B.
- ||: allows testing whether either of the two operations are true.
- &&: allows testing whether both operations are true.

To continue with the explanation of the example code, the code allows the sum of the numbers in the second place of each tuple to be returned by the server. This sum is then divided by the number of tuples received to obtain the average of the numbers received. Finally, this average is sent to another tuple space in another server.

The language offers the possibility to easily connect to a server and then to a tuple space with different security attributes. Operations can be performed on this tuple space but the data in it can also be processed by the language. The language also offers the possibility to connect several servers and thus to move data from one server to another. In the following sections, the grammar of the language and its implementation are presented.

4.5.2 Grammar of the language

First, before detailing the grammar and implementation, the meaning of an interpreter will be defined. To understand what an interpreter is, one must first understand what a programming language is. A programming language L includes:

1. a pair of sets (P_L, D_L), where P_L represents the set of all programs written in L and D_L the set of all objects that a P_L program can manipulate
2. a function

$$\boxed{[[.]]^L : P_L \rightarrow (D_L \rightarrow D_L).}$$

   which represents the semantics of the language [59].

In other words, a programming language L allows writing programs P such that these programs receive objects D from the language L and manipulates them into other objects D. Now that a programming language is formally defined, an interpreter can be defined. To make a programming language useful, one must at least be able to write an interpreter for the language. The interpreter is a program which is either written in another programming language or, in some cases, the same language. Formally, the notion of an interpreter is defined as follows:

Let $L = (P_L, D_L)$ and $M = (P_M, D_M)$ be two programming languages. In assuming that L allows coupling and that $(P_M \cup D_M) \subseteq D_L$.

- We call a partial function interpreting M by L $i : D_L \rightarrow D_L$ such that

$$\boxed{\forall P \in P_M : \forall t \in D_M : [[P]]^M t = i(P, t).}$$

- We call an interpreter of M by L a program $int \in P_L$ such that $[[int]]^L$ is a interpretation function of M by L [59].

In the above definition, M represents the language interpreted, while L represents the language in which the interpreter is written. In our case, the interpreted language is LiF and the interpreter language is Rust. Now that the formal definition of an interpreter is defined, the grammar of the programming language must be defined.

To create a programming language, the developer must create a lexer and a parser. The definition of these concepts will first be explained before discussing the grammar of the language.

When code from the created language is submitted to an interpreter, the interpreter will first use a lexer to recognise the various symbols or tokens in the code. A token can be a sign or a list of signs. Once the entire code has been passed through the lexing function, the code is now a list of tokens in the eyes of the interpreter. This list of tokens will be passed to the parser which will oversee finding the lists of tokens that correspond to a certain expression defined by the programmer. In the following example in Figure 4.4, the parser recognises the expression *sum* which is itself an *expr* expression.

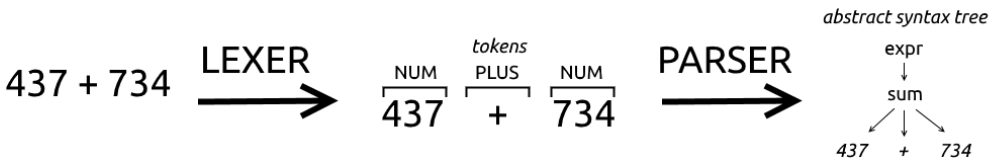


Figure 4.4: Process of parsing and lexing[60].

In the context of the implementation of this interpreter, the antlr [61] parser generator was used. Antlr facilitates, from a grammar, the generation of the parser and lexer of a language. The grammar given to Antlr to generate this language can be found in the appendices of this work see appendix 7.2. With this grammar, we obtain a parser that contains the syntax tree of the code entered by the interpreter is obtained.

The syntax tree of our language will therefore contain a series of instructions. The developer will have to implement the behaviour of each of these instructions. For the interpreter, the syntax tree of an assignation followed by a *connect* instruction looks like Figure 4.5.

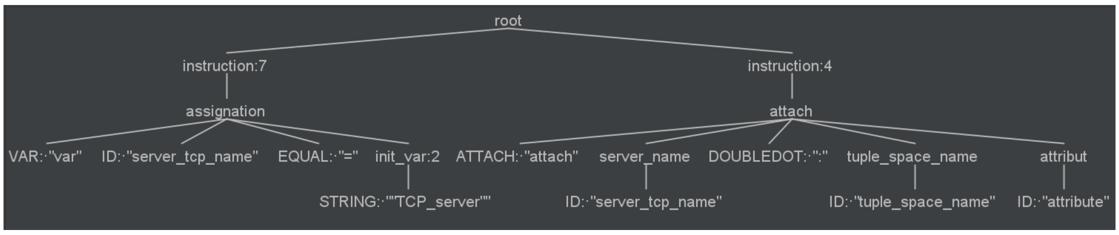


Figure 4.5: Parse tree.

4.5.3 Syntax of the language

The interpreter offers the possibility to use different commands to connect to a server and manage space tuples. Below is a list of the different possible commands:

```
create {creation_attribute} {tuple_space_name} {permission_attribute} {
    ↪ encryption_key}
create {creation_attribute} {tuple_space_name} {read_permission_attribute} {
    ↪ in_permission_attribute} {out_permission_attribute} {
    ↪ delete_permission_attribute} {encryption_key}
delete {delete_permission_attribute} {tuple_space_name}
attach {tuple_space_name} {permission_attribute}*
out {tuple}
out {tuple}(,{tuple})*
read {tuple}
read {tuple} (,{tuple})*
in {tuple}
in {tuple} (,{tuple})*
```

The *create* command allows for the addition of a new tuple space to the server repository where the client has connected. This command requires the creation of an attribute, which is the permission to create a tuple space. Next, a name must be given for the newly created tuple space. After choosing a name, the client has two choices. In the first choice, the client indicates only one attribute which will then be the same for the different actions on the tuple space. These actions are as follows:

- read: Reads tuples in the tuple space.
- in: Removes tuples from the tuple space.
- out: Adds tuples to the tuple space.
- delete: Deletes a tuple space.

In the second choice, the client decides to associate a different attribute for each action. This can be useful in case where they want to give access to only one of the actions on the tuple space.

The delete command is used to delete a space tuple. The delete attribute of the tuple space must be specified with the command.

The attach command allows the client to define which tuple space, in the repository the client is connected to, the commands out, read, and in should be executed in. The command also takes a list of attributes depending on what the client can provide as a parameter.

The commands out, in, and read are used to perform operations on the tuple defined within the attach command. The commands can take one or more tuples as parameters. The command adds the tuple list to the tuple space. The in and read commands return a list of tuples corresponding to the matching pattern of each tuple in the list passed as parameter. The wildcard is represented by the _. However, these commands, which correspond to Linda's primitives, do not respect the blocking aspect of Linda.

In addition to these operations, the interpreter offers the possibility to perform loops. The syntax of the latter is as follows:

```
    for {variable_iterator} = {starting_number} to ({max_number}) {
        {body_of_the_loop}
    }

    while ( {boolean_condition} ){
        {body_of_the_loop}
    }
```

The language also offers the possibility to instantiate variables and to perform arithmetic operations such as addition, subtraction, multiplication and division.

```
    var {variable_name} = {variable_name or value}
    var {variable_name} = {variable_name or value} + {variable_name or value
    ↪ }
    var {variable_name} = {variable_name or value} - {variable_name or value
    ↪ }
    var {variable_name} = {variable_name or value} * {variable_name or value
    ↪ }
    var {variable_name} = {variable_name or value} / {variable_name or value
    ↪ }
```

To manage tuples, two operations are available. The len() operation allows receiving the maximum number of elements present in a tuple. This allows you to loop through the different elements. The get (index) operation is used to retrieve the value found in the index passed in the parameter.

```
    {tuple}.len()
    {tuple}.get({index})
```

The details of the implementation's different instructions are explained in the next section.

4.5.4 Implementation of the language

To understand how the different instructions were implemented, one must first understand what the code generated by Antlr offers. Antlr provides an interface which, for each node of the syntax tree, offers a method. This method receives, as a parameter, a context object which contains the instruction. To express this, the enter_connect() method is defined below:

```
fn enter_connect(&mut self, _ctx: &ConnectContext<'_>) {
    if let Some(server_name) = _ctx.server_name() {
        if let Some(protocol) = _ctx.protocol() {
            if let Some(ip_address) = _ctx.ip_address() {
                if let Some(port) = _ctx.port() {
                    let server_name = self.validate_server_name(server_name)
    ↪ ;
                    let server = Server::new(
                        ip_address.get_text(), port.get_text(),
                        protocol.get_text(),server_name.clone(),
                    );
                    self.server_list.insert(server_name, server);
12                 }
13             }
14         }
15     }
16 }
```

In this method, the reference of a ConnectContext object is collected as a parameter. Due to this context, the different parts of the instruction can be retrieved. This is the purpose of the code on lines 2 to 5 which ensures that there is no None type in the different parts of the instruction. Once these parts are retrieved, the connect method behaviour can be implemented. A server with the retrieved data is then launched.

The operation of the clients that will connect to the tuple space server is slightly unconventional. In fact, to avoid having to reconnect for each instruction, a client is launched on a thread. For each instruction connected to a server, a new thread with a new client will be started. Therefore, we set up a way to communicate with the different clients the requests that were parsed. Indeed, it is necessary to communicate to the thread containing the client the request which must be sent to the server. For that, the mpsc::channel() method of Rust was used.

This method creates a new asynchronous channel, returning the sender or receiver halves. All data sent to the sender will become available to the receiver in the same order as it was sent, and no *send* method will block the calling thread [62].

When a thread with a client is created, two mpsc channels are created. The first channel allows to send requests that the client sends to the tuple server. The second channel is used to send the response from the tuple server to the main thread so that it can continue parsing. In case of no response from the tuple server, a time-out system prevents the interpreter from getting stuck. This thread and channel system is explained in the following sequence diagram in Figures 4.6 and 4.7.

In Figure 4.6, the main thread detects that there is a connect expression by parsing the code. This is then executed by creating the different channels as explained before. The main thread receives at the end the sender of the request channel to make a request to the server and the receiver of the response channel to wait for the response from the server. The client thread will wait for a request to be sent in the request channel.

In Figure 4.7, the server parses a request for an out on the previously created server. It then sends the request through the request channel transmitter. Thereafter, it starts waiting for a response to its request to be sent in the response channel. The client thread listening to the request channel has received the request for an out and so passes it on to the tuple server on the network. The tuple server waits for the response from the server and then passes it on to the response channel where it will be received by the main thread. The client thread then listens to the request channel again.

The other point that the implementation must manage is the assignment of variables. Indeed, variables must be stored throughout the syntax tree of the code. The creation of the listener object, which implements the Antlr method and will use the syntax tree received by the parser a symbol table, is created. The concept of a symbol table allows retaining all variables, constants, and functions in an interpreter or compiler. Different information about the data held is also associated with it, for example, for a variable, its value, or type can be held.

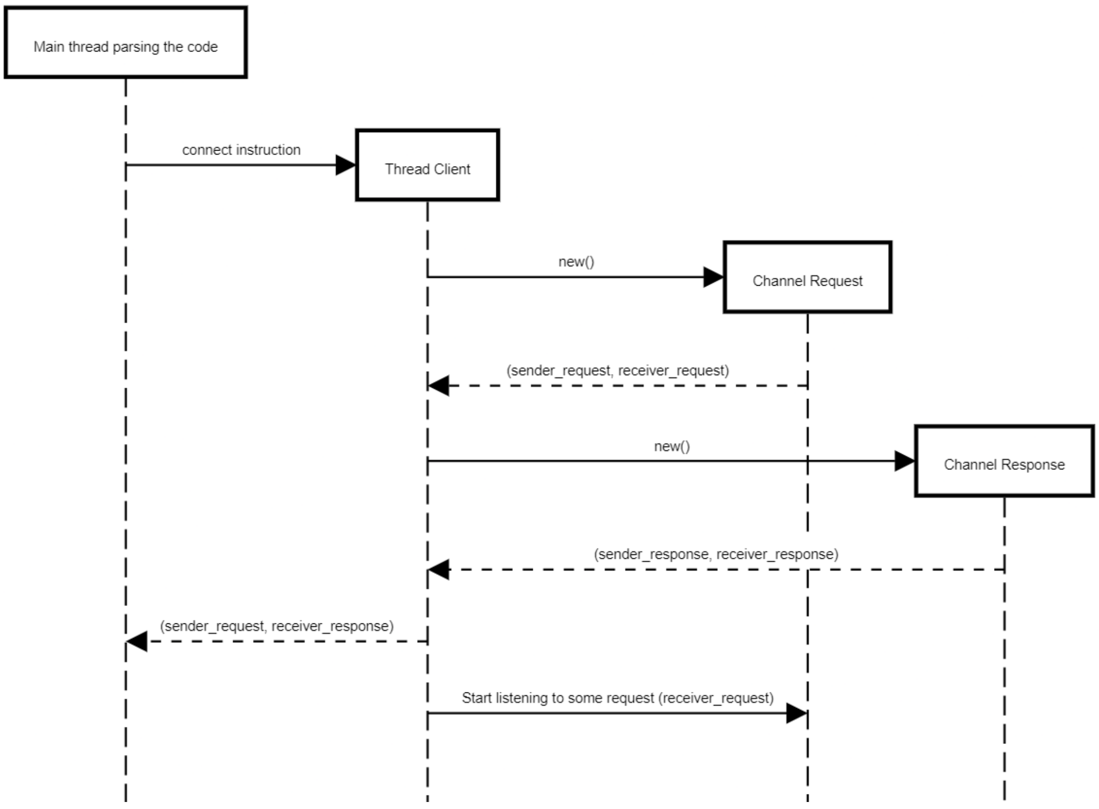
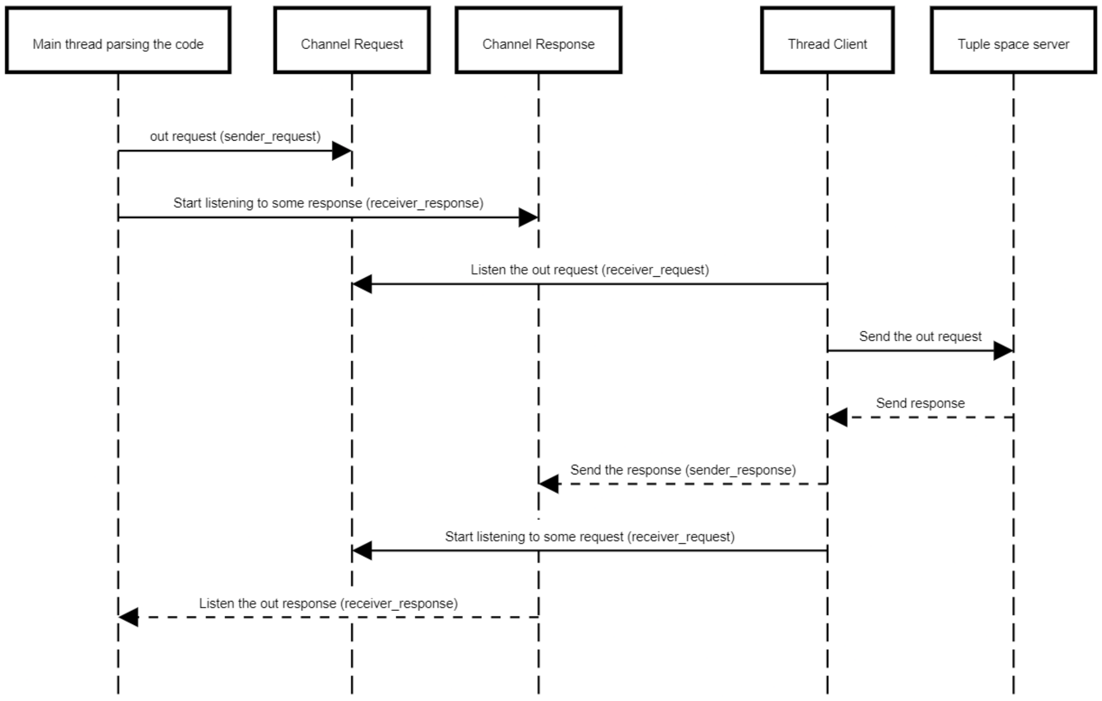

Main thread parsing the code
connect instruction
Thread Client
new()
Channel Request
(sender_request, receiver_request)
new()
Channel Response
(sender_response, receiver_response)
(sender_request, receiver_response)
Start listening to some request (receiver_request)

Figure 4.6: Connect instruction diagram



Main thread parsing the code
Channel Request
Channel Response
Thread Client
Tuple space server
out request (sender_request)
Start listening to some response (receiver_response)
Listen the out request (receiver_request)
Send the out request
Send response
Send the response (sender_response)
Start listening to some request (receiver_request)
Listen the out response (receiver_response)

Figure 4.7: Out instruction diagram

In the case of this implementation, the symbol table is a hashmap containing a String which will be the key and name of the variable as well as an object of the Value enumeration. This enumeration will be discussed later. Using a hashmap as the symbol table ensures that each key is unique and it is therefore impossible to have two variables with the same name.

The Value enumeration is used to hold the value of a variable according to its type. Below is the definition of this enumeration:

```
enum Value {
    String(String),
    Number(f32),
    Tuple(Vec<Value>),
    Char(char),
    ID(Box<Value>),
}
```

The different types of the created language are in this enumeration. A String type remains the normal Rust String type. The Number type is a 64-bit float in Rust. The Tuple type will contain a Vec which is a list type in Rust which itself contains a Value of this enumeration. The Char type will contain a classic Rust Char type. Finally, the ID type will contain a Box to another value. This type allows a variable to contain another variable which will itself have a value.

4.5.5 Implementation of the client part of the library

In addition to the interpreter written in Rust, a section allowing to communicate with a tuple space server is also available in the Rust library. This section allows the same operations as the interpreter. However, by being directly integrated into the Rust language, it allows everything that can actually be in Rust. When installing the tuple server, it is therefore possible to either communicate with it via the Rust library, or to use the interpreter. In case the Rust compiler is not available on the device, just installing the interpreter can be a solution. Moreover, the client part of the library allows for having the tools to create a server as well as those to connect to a server and get data in the same library. This makes it possible to be both a producer and a consumer.

4.6 Performance test

The purpose of the performance tests is to determine whether the tuple space server is functional on power-constrained devices. For this, the server will be deployed on two different devices: the first is a Raspberry Pi and the second is an Android phone. To assess the performance of the tuple space server, the CPU and Ram consumption were monitored during different operations on the server. Here is the list of operations that were monitored:

- Creation of 100 tuple spaces on the server from the server.
- Creation of 100 tuple spaces on the server from a client.
- Out operation on 100 tuples on the server from a client.
- In operation on 100 tuples on the server from a client.
- Read operation on 100 tuples on the server from a client.
- Delete 100 tuple spaces on the server from a client.

The performance measured in the tests contains the performance of the Rustupolis library combined with the various extensions added to it. There is thus the performance of the servers which in the case of the test is a TCP server, the performance of the encryption system and the performance of the access right system.

4.6.1 Performance on the Raspberry Pi

A Raspberry Pi is a credit card sized computer that can do things one could do with a regular computer. However, due to its small size, its power is limited. As a result of its low power consumption, a Raspberry Pi is often used to automate small tasks that do not require a lot of computing power or to store small servers. The operating system (OS) installed on the Raspberry Pi is Raspberry Pi OS Lite which does not contain a graphical interface and is therefore more powerful. The model used in this performance test is the "Raspberry Pi 4 Model B". Its specifications are as follows [63]:

- *Processor :* Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz

Type of test	CPU load usage	Memory consumption
Creation of 100 tuple space from the server	2.973 for 10 seconds	1 mb
Creation of 100 tuple space from the client	0.431	1 mb
Out of 100 tuples	0.097	1 mb
In of 100 tuples	0.112	7 mb
Read of 100 tuples	0.142	1 mb
Deletion of 100 tuples spaces	0.266	2 mb

Table 4.1: Results of the tests on Raspberry Pi

- *Memory :* 2GB LPDDR4-3200 SDRAM
- *Wifi :*2.4 GHz and 5.0 GHz IEEE 802.11ac wireless

The results of the tests of the different performance criteria are presented in the table 4.1.

The CPU consumption in the case of the Raspberry is expressed in CPU Load. This means that if the CPU Load is 4, 100% of the processor cores were used. The number 4 comes from the fact that the Raspberry Pi processor has 4 cores. The memory consumption has been calculated based on the memory available during a certain task. This means that the memory available is recorded before performing an operation and when the operation is performed. Then the two numbers are subtracted to give the memory used during the operation. To obtain these different data, the RPI-Monitor [64] package was used, which presents graphs of CPU and RAM usage. Figures 4.8 and 4.9 show examples of the consumption graphs.

The data collected shows that the programme uses very little RAM. The CPU usage is also very low except for the tuple space creation operation from the server itself. This operation could therefore be optimised. Indeed, when it is performed at the client level, the CPU consumption is lower. One hypothesis of this difference in CPU usage is that the TCP server queue allows for better management of the tuple space addition requests.

4.6.2 Performance on Android

The tuple server was tested on a Samsung A52S 5G. The specifications of the phone are as follows [65]:

- *Processor:* Qualcomm SM7325 Snapdragon 778G 5G (6 nm), Octa-core (4x2.4 GHz Kryo 670 & 4x1.9 GHz Kryo 670)
- *Memory:* 6 GB RAM
- *Wifi :* 802.11 a/b/g/n/ac/ax 2.4G+5GHz, HE80, SISO, 1024-QAM

To install the tuple server on the Android device, a Java application must be created. For the Java application to use the Rust library, it must be compiled in C. This can be done with

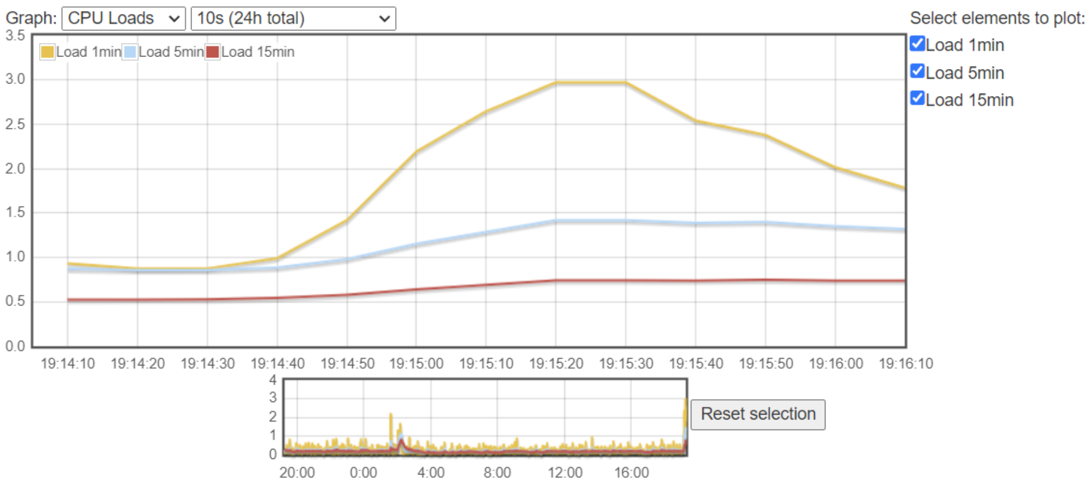


Figure 4.8: CPU Load

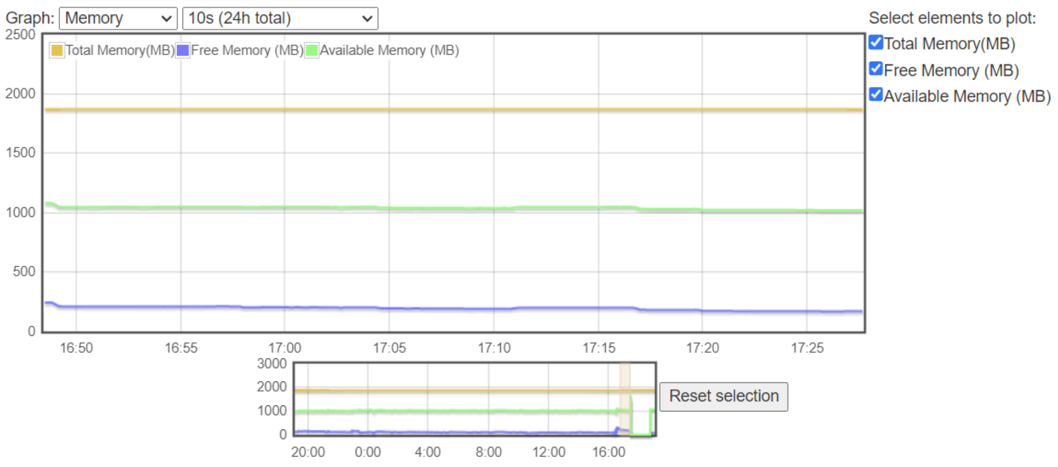
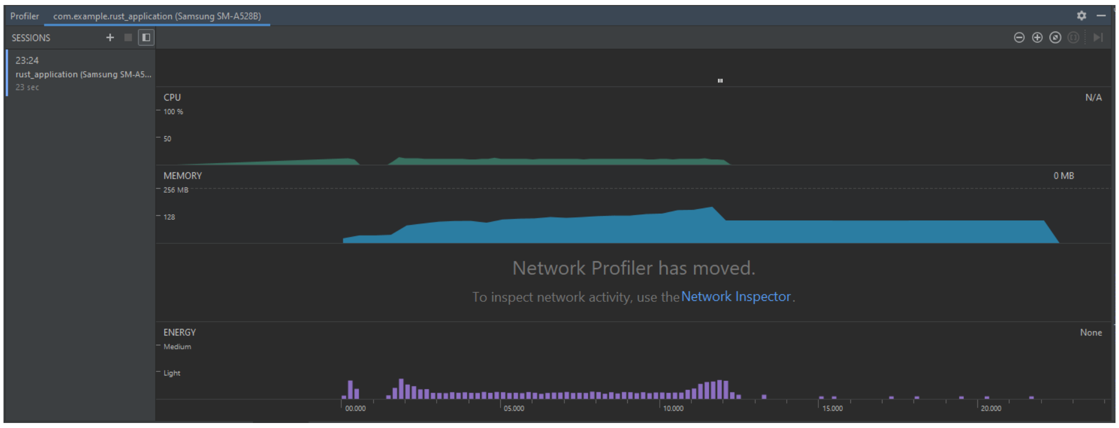


Figure 4.9: Memory consumption

Type of test	CPU usage	Memory consumption
Creation of 100 tuple space from the server	5-7 % When adding tuple space for 1 seconds	142 mb
Creation of 100 tuple space from the client	16 % On start	141 mb
Out of 100 tuples	5 % When adding tuple	141 mb
In of 100 tuples	6-7 % When adding and removing tuple for 2 seconds	147 mb
Read of 100 tuples	6-7 % When adding and removing tuple for 3 seconds	147 mb
Deletion of 100 tuples spaces	4-5 % When removing tuple for 1 seconds	141 mb

Table 4.2: Results of the tests on Android



Network Profiler has moved.

To inspect network activity, use the Network Inspector.

Figure 4.10: Android profiler example

the JNI library that allows for the binding of Java calls within the Rust function compiled in C [66].

On the Android platform, performance data was retrieved using the Android profiler tool from Android studio. For each test, the peak power consumption was recorded and noted in the above table. An example of the CPU consumption and RAM consumption is displayed in Figure 4.10 from Android Studio.

The analysis of the data collected on the Android, see table 4.2, application must consider that the Rust library runs in a Java application. Therefore, the resources consumed by this Java application are also considered in the monitoring of the application. The graphical interface, for example, consumes part of the resources. By analysing the RAM consumption, this corresponds to what the data recovered from the Raspberry Pi had shown. Indeed, very little RAM is used by the Rust library. The RAM used here is used by the application. The processor consumption is also higher than on the Raspberry Pi and this can also be explained by the fact that the Rust library is wrapped in a Java application. Another cause of this CPU consumption is the use of the JNI library which allows Rust functions to be called in Java. This adds an extra step and therefore extra resource consumption.

Type of operation	Execution time in ms
Connect	1.69ms
Attach	4.37ms
Create	3.27ms
Out	4.41ms
Read	7.18ms
In	6.84ms
Delete	4.65ms

Table 4.3: Results of the tests on Android

4.6.3 Performance in time

Another performance to be assessed in our PoC is the response time of a request. This assessment considers the encryption of the request on the client side, the decryption of the request on the server side, the execution of the request, the encryption of the response, the sending of the response, and the decryption of the response. To assess this performance, each of the possible operations is tested and the time of the request is monitored. The tuple space server is located on a Raspberry Pi and the client on a computer. Both devices are on the same Wi-Fi network. The timer starts as soon as the request has been sent and stops when the response has been decrypted. Here is the list of operations where the time has been recorded:

- Connect operation to connect to the server.
- Attach operation to define the tuple space where the operations will take place.
- Create operation to create a new tuple space.
- Out operation to add a tuple to a tuple space.
- Read operation to read a tuple from a tuple space.
- In operation to remove a tuple from a tuple space.
- Delete operation to delete a tuple space.

The results of the performance tests, see table 4.3, show that it takes very little time for the operations to finish. This also depends on the quality of the network, but in the case of a local network this has little influence on the result. The fastest command is the command to connect to the server. This makes sense, as this is not encrypted and is just a request to connect to the server. The operations that take the longest time are the *read* and *in* operations, which is logical since these operations use pattern matching that may explain why these operations take the longest.

4.7 Concluding remarks

In this chapter, the ways to implement the Rust library to create a tuple space server and build the interpreter and client part of the library have been explained. The choice of the Rust language and the different advantages of it in the project have also been presented. The implementation of the tuple server was directed by the previously defined criteria. The first four criteria have been applied in this implementation. The criteria of data placement font and persistence have not been developed. The interpreter and the client part of the library offer a workable solution to communicate with a tuple server. As proven by performance tests, the solution consumes few resources and is optimised. The next chapter of the work detail examples built with this language and the tuple server.

Chapter 5

Applications

5.1 Android Use case

A use case must prove that the performances previously shown work well in real applications. The developed application displays, on a map of computer science faculty of Namur, phones according to their GPS position. In addition to the GPS position, a sensor on the phone is used to retrieve the light exposure of the phone's environment. This is expressed in lux. The lux (lx) is the SI derived unit of illuminance, measuring luminous flux per unit area. It is equal to one lumen per square metre. In photometry, this is used as a measure of the light intensity, as perceived by the human eye, that hits or passes through a surface [67]. The light exposure from each phone connected to our system will be displayed. The use case is explained in the Figure 5.1.

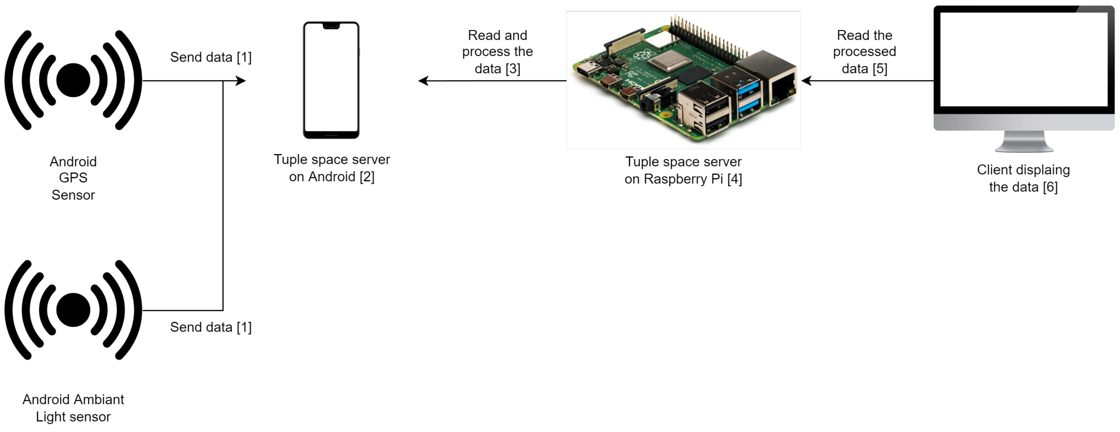

Send data [1]
Read and process the data [3]
Read the processed data [5]
Android GPS Sensor
Tuple space server on Android [2]
Tuple space server on Raspberry Pi [4]
Client displaing the data [6]
Send data [1]
Android Ambiant Light sensor

Figure 5.1: Use Case

To implement this system, two tuple space servers have been installed. The first server is in the Android system and the second server is in the Raspberry Pi. [1] The first server in the

Android system retrieves data from the GPS and luminescence sensors. [2] The data is then sent to a tuple space on the server installed on the phone. This allows the data to be stored and is available when the client on the Raspberry needs it. [3] For each phone, the Raspberry Pi contains a client part that is responsible for retrieving the data. This data follows the following format:

$(latitude, longitude, altitude, lux)$

[4] The Raspberry Pi will then assign to this tuple an ID and store this data with its ID in a general tuple space for all smartphones in the use case. The tuple format will be as follows:

$(id, latitude, longitude, altitude, lux)$

The tuple space server on the Raspberry Pi allows one to retrieve data from each phone and put it into a single tuple space. In the case of this example, the aim is to display the data from the different phones on a web interface. However, a website cannot connect to a TCP socket like the server on the Raspberry Pi. [5] An intermediate step has been added to solve this problem. It acts as a bridge between the TCP socket and a web socket and is used to send data to the web interface. This step has also been coded in Rust and uses the client part of the library to retrieve tuples from the server and send them to the web socket. To overcome this problem, the web socket system could be added to the protocols used by the application. This would allow calls to be made directly from the web. The HTTP protocol could also be a solution for making calls from the web.

[6] Once the interface receives the data from the web socket, it must then be displayed (Figure 5.2). To do this, each phone appears as a circle of light on the map of the faculty according to their GPS position. The size of the circle of light depends on the intensity of the light.

The update of the light intensity and the GPS position is done in real time. The reliability of the phone's GPS sensor when it is in the centre of the building and surrounded by walls can be questioned. The reliability of the Wi-Fi network can also be a slowing factor. Finally, the reliability of the phone's light sensor can also be questioned. The light sensor detects the ambient brightness of the phone, but this also depends on how the phone is positioned. In this experiment, the phones were left on their backs, with the front camera and therefore the light sensor pointed at the ceiling.

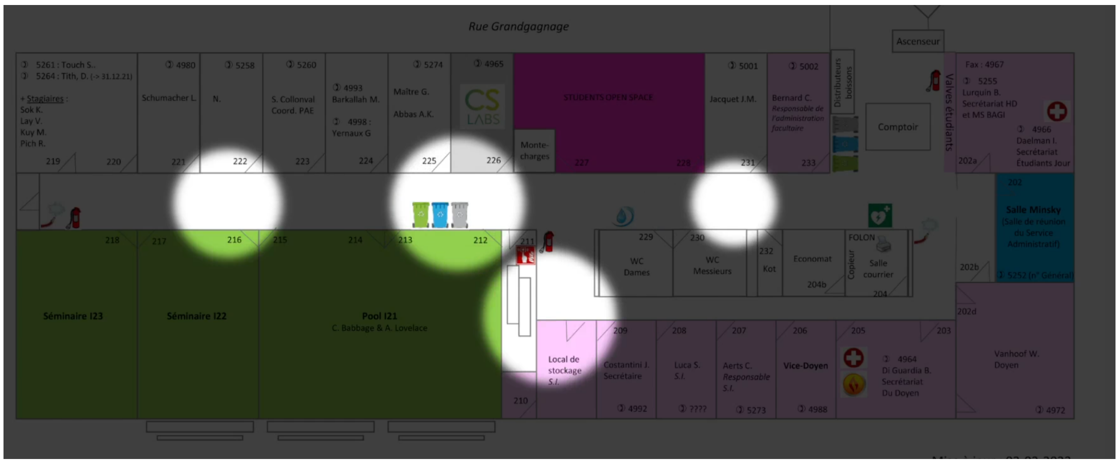


Figure 5.2: Illustration of the use case

To create the Android application, the same system used for the performance tests was employed. First, the code in Rust which calls the library allowing for the creation of the tuple space was compiled in C. Owing to the JNI library, the Java code of the mobile application can call these Rust functions. However, the JNI library does not offer a way to manage the brightness sensor directly from Rust. To compensate for this, the sensor data is retrieved from the Java part of the application and sent to the tuple space in the Rust code. To retrieve data from each phone, a new thread is created per mobile device. This allows requests to be sent independently of the other devices and the response to them. It also allows managing the case when a device does not respond and to avoid that the sending of requests to other devices is disturbed by this. The following code is used to create the different threads. The client part of the library is used to perform operations on the tuple spaces. The interpreter cannot be used here as it does not allow the creation of threads.

```
let clients = vec![(1, "192.168.0.4", 9000)];

crossbeam::scope(|scope| {
    scope.spawn(|_| {
        server_launcher.launch_server();
    });

    for mobile in clients {
        scope.spawn(move |_| {
            let tuple_space_name = String::from("GPS_DATA");
            let attribute = String::from("admin");
            let mut client = Client::new();
            let (id, ip_adr, port) = mobile;
            client.connect(String::from(ip_adr), port.to_string(), String::
↪ from("tcp"), &id.to_string(), key);
            client.attach(&id.to_string(), vec![attribute.clone()], &
↪ tuple_space_name);
            /*Operation on the client*/
        })
18      }
19  })
```

The first line of the code contains the *client* variable which will contain an array of tuples. Each tuple represents the identifier, IP address and port of a tuple space server on a mobile phone.

On line 3, the crossbeam library is used to manage threads. This library allows access to resources on the stack even in the scope of a new thread, which is not possible with basic Rust threads.

Line 5 starts the tuple space server which will be on the Raspberry.

Lines 8 to 17 are used to create a thread for each mobile phone entered in the *client* variable. The thread will then connect to the tuple server on the phone and retrieve the data from it.

The GitHub repository link for this android part of the project is as follows: `https://github.com/Maxbever/LIF_android`

To create the application on the Raspberry Pi, it was much easier. Indeed, it was necessary to first write the Rust code to retrieve the data from the IP address of the server on the Android phones. Then, a new server tuple space that contains the retrieval with a different ID per phone was created. To retrieve the tuples, the client part of the library was used. Here is the link of the GitHub repository: `https://github.com/Maxbever/LIF_raspberry`

To realise the web interface, as presented previously, a link between the TCP socket and a web socket was implemented in Rust. This interface just transmits messages from the TCP server to the web socket and vice versa. The interface itself was developed in HTML, CSS, and JavaScript for the logic and display of the data from the web socket. Here is the link of the repository of the front-end and of the web socket : `https://github.com/Maxbever/LIF_front`

5.2 Traffic lights use case

This second use case is inspired by Figure 2.4 which provides a practical example of fog computing. In the example, a first level of fog computing takes care of managing the different traffic lights at a given intersection. The traffic lights can therefore be synchronised from the same device. This device will also send data about the number of cars in the intersection to the second level of fog computing. The second level of fog computing will receive the data from the different intersections (i.e., the first fog computing layers) to anticipate the arrival of cars at another intersection. This makes it possible to inform the first level of fog computing about the number of cars arriving in each direction. The traffic light turns green according to the number of cars arriving and prevents them from waiting at the traffic light, creating traffic jams. The second layer of fog computing deals with different intersections in a city district. This makes it possible to synchronise the different intersections efficiently, in turn avoiding having to manage too many intersections and thus reducing the efficiency of this second fog layer. This layer also sends information about the different junctions to a third fog computing layer. This third fog computing layer is responsible for synchronising the different districts in the same city. It also sends information for traffic statistics to the cloud.

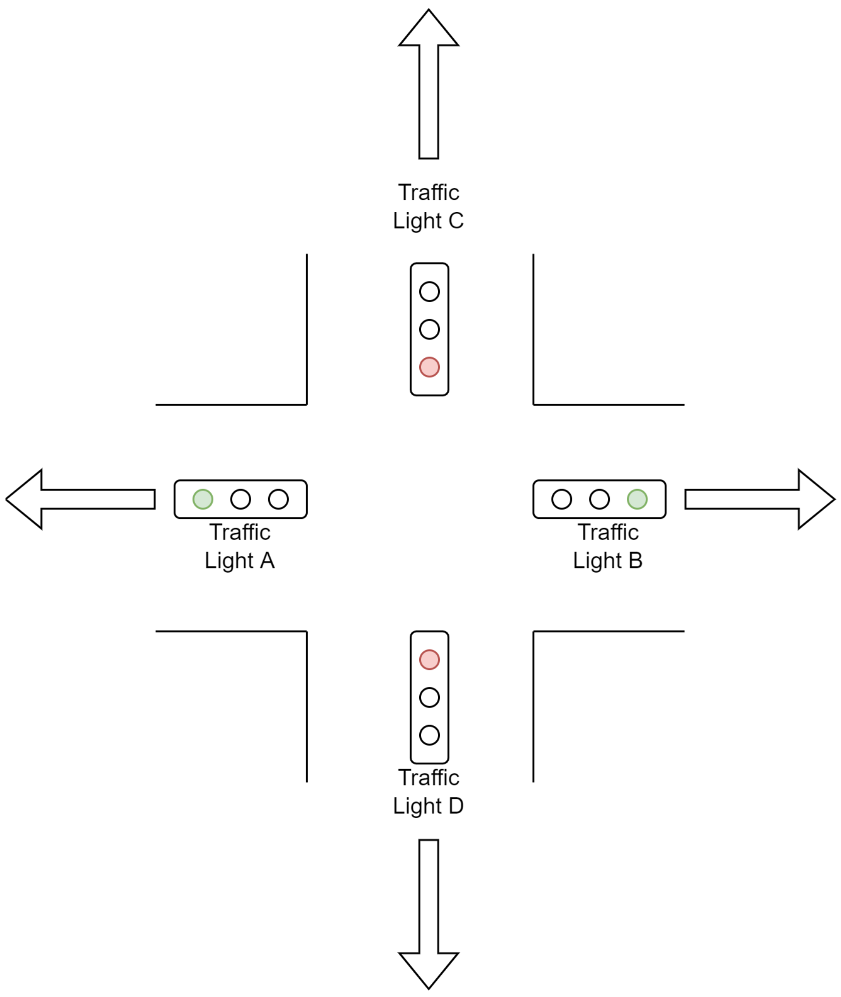


Figure 5.3: Traffic light example.

To create the test case, the different tuple servers will be launched on the same machine. However, this would require too many devices to make a real use case. Thus, the first step is to determine how the traffic lights will work and what data they will send back to the first level of fog computing. A traffic light will contain a tuple space where it will store its state and its identifier. The state of a traffic light can be green, orange, or red. When the light is green, the device will count the number of cars crossing the intersection and passing under it. When the light is red, on the other hand, the light will count the number of cars waiting to pass. Every 30 seconds, it will send the number of cars heading for the next intersection to the first layer of fog computing, which will then send it to the second fog layer. When the traffic light is red, it will send the data on the number of stopped cars to the first fog computing layer. Figure 5.3 explains this. Traffic light A counts the number of cars that are heading in the direction of the arrow to the left of the traffic light. Traffic light C also counts the number of cars going towards the arrow above, but it additionally counts the number of stationary cars.

The first level of fog computing will therefore receive two types of data from the traffic lights.

Either the number of cars waiting at the traffic lights or the number of cars that have crossed the intersection. Depending on the number of cars waiting or the time since the first car was waiting, the device decides to change the red traffic lights to green and the green traffic lights to red. The change to orange is overseen by the traffic lights when they receive a request to change to red. For this example, the number of cars waiting must be at least eight for the traffic lights to change state. The number of minutes the first car must wait for the traffic light to change state is three minutes. In the implementation, the number of cars passing in one direction will randomly be chosen between five and fifteen cars. The same applies to the number of cars waiting at the traffic light.

The second level of fog computing receives data from the first level on the number of cars that have left an intersection and informs the other intersections of the number of cars that will arrive. The first fog layer, which manages an intersection, will therefore also receive data from the second level and will have to manage this data so that as few arriving cars are waiting at the traffic lights as possible.

In the implementation of this example, the third level of fog computing is not implemented as it is akin to the second level, but with an additional step. Three intersections are implemented in the use case. This corresponds to simulating twelve traffic lights that will send data to three tuple spaces in the first level of fog computing, which in turn will be coordinated by one tuple space in the second level of fog computing. Below is the link of the GitHub repository of the traffic light use case: `https://github.com/Maxbever/LIF_traffic_light`. The three different traffic lights in this use case have been represented as follows 5.4.

The particularity of this use case is that each element of it is a producer and consumer of data. Indeed, the traffic light will send data to the first fog layer and in parallel will wait for a response from it. The principle is the same for the first fog computational layer which sends information to the traffic lights and to the second fog computing layer while waiting for information from them. This use case shows that from the PoC, it is easily possible to create servers that will be producers and consumers. In the code of each of the elements of the use case, one thread manages the production of data while another thread is dedicated to its consumption.

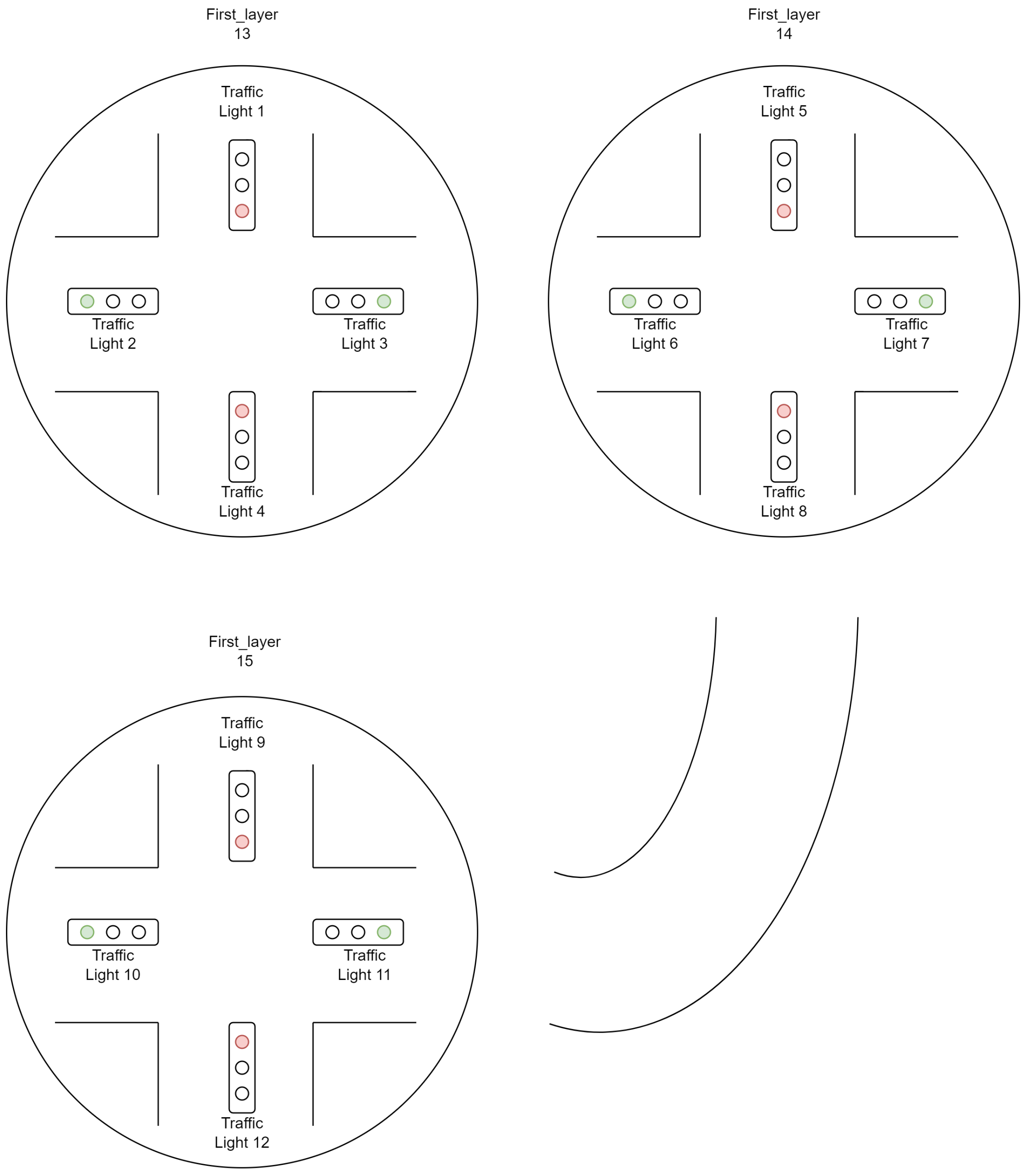


Figure 5.4: Traffic Light Use case.

This diagram shows what each layer handles. The second layer deals with the management of cars moving from one intersection to another. In our use case, for example, the cars leaving the first intersection at traffic light 4 will arrive at the third intersection at traffic light 9.

Chapter 6

Conclusions

6.1 Summary of contributions

The objective of this work was to find a solution for a coordination language to be efficiently implemented in the fog computing paradigm. This implementation should address some problems of fog computing. The first result of this work is the establishment of criteria for creating an implementation of a coordination language in fog computing. These criteria allowed us to select the way of designing the implementation.

Indeed, the first criterion and contribution concerns the importance of an implementation running on devices with limited computing capacity. This had an impact on the choice of programming language for the implementation, but also on the way the implementation was developed. The Rust programming language was chosen for its ease of creating high-performance applications, but also for avoiding memory errors. The former also had an impact on the choice of how to manage data security; the data encryption system had to be secure yet but powerful enough to work on low-powered devices. Performance tests also showed that the PoC worked very well on devices with limited resources (see Figure 4.6).

Another advantage of Rust leads to the second criterion concerning scalability. Rust can be compiled to a vast number of platforms, which facilitates scalability of the applications written in the language. Another point which deals with the scalability criterion is the creation of an interpreter which directly writes code allowing to access a tuple server. This interpreter can therefore operate independently of the Rust compiler and thus offers other possibilities. Furthermore, the interpreter can run on a device that does not have the Rust compiler installed.

The third criterion and the second contribution concern the security constraints that an application in fog computing must respect. That has also been validated in the implementation by means of two different mechanisms. The first mechanism is the installation of an access control to the data through attributes that are used as permissions to access and perform actions on the data. This system is distinct because permissions are stored as tuples in a dedicated tuple space. This can be used in the future for the data persistence criterion. The second mechanism allows the encryption of data and transmissions between a client and a server. This prevents someone in the middle of the transmission from reading or modifying the data.

The fourth criterion concerns the performance of the implementation. This criterion is divided into two parts: the first is performance at the level of the network and the second is the performance at the level of the recovery of the tuples through pattern matching. Regarding the first performance point, it is difficult to measure latency, as it depends strongly on the geographical distance between the devices. However, regarding the second performance point, performance tests have shown that it takes very little time to decrypt a client's request and to retrieve a tuple from a tuple space (see section 4.6.3).

The last two criteria have not been implemented and will be discussed in the future work part of this conclusion.

6.2 Assessment of contributions

The first point of the evaluation of this work is that two criteria were not developed due to a lack of time. However, a solution for these criteria is presented in the next section. Regarding the different criteria assessed, they correspond well to what an implementation of a coordination language in fog computing should respect. Nevertheless, the criteria are quite difficult to implement and require a lot of development time. For this reason, some points of the proof-of-concept could be improved.

First, security could be improved. Currently, the attribute giving access rights to the data must be sent to the client through a third party. To solve this problem, a system allowing the identification of a client and therefore the binding of attributes to it could be put in place. However, this was rather complex, as clients are devices that can be disconnected and must be able to connect automatically.

The other point of security improvement is that just like the data access attribute, the encryption key of the data must also be sent by a third party. This could be solved by an asymmetric encryption system, but at the current stage, it is complicated to implement this in a fog computing context.

The criteria where the implementation has succeeded are scalability and running on devices with less computing power. In fact, developing the PoC in Rust allowed us to obtain satisfactory performance results in these criteria.

The PoC also showed that coordination languages and especially their persistent broadcast can work very well in a fog computing context. Indeed, message coordination brings real added value in managing the constraints inherent in a fog computing application.

6.3 Future work

The last two criteria have not been developed in the implementation. However, some suggestions for additions are made.

The first missing criterion concerns a data persistence mechanism. This allows the definition of a mechanism that is put in place when a fog node is no longer available on the network.

One envisaged solution is to set up a system for duplicating tuple spaces on another device. The system would copy the entire tuple space of a device to another device, which would also copy the tuple space containing the access rights and thus preserve them. However, the client trying to connect to the unavailable server would have to be able to find the copy easily and this would have to be done automatically. For this, a tuple space containing the list of devices on the network and the list of their copies could be made available. This tuple space could be placed in the cloud to ensure that it is always available. The client would only need to contact this tuple space to make a request for a copy of the unavailable tuple space. To keep the data constant, the master tuple would have to notify each of its copies whenever it is changed. When the unavailable master tuple is put back on the network, it should start from the updated version of its copies.

Another solution to improve data persistence is to save the data in the device where the server is located. The data could be stored in files or in a database. When the device is unavailable, it can be rebooted and the data stored in its memory can be recovered. However, this solution depends on the ability of the devices containing the tuple server to handle file writing and reading or to handle a database system.

The second missing criterion is that of a data placement policy mechanism. A user with data in a tuple space should be able to choose in which devices their data is stored. This mechanism can be implemented through the tuple space containing the different devices available on the network. With the tuple space providing a map of the network, a user could indicate in which device, and at which address, they would like their data to be stored. The user's placement choices concerning their data are stored in a tuple like access rights.

Once all these criteria are implemented, the implementation can be applied to different use cases. As discovered earlier, many applications would benefit from fog computing which would have its shortcomings improved by a coordination language.

Bibliography

[1] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985, ISSN: 0164-0925. DOI: `10.1145/2363.2433`. [Online]. Available: `https://doi.org/10.1145/2363.2433`.

[2] D. Gelernter and A. J. Bernstein, "Distributed communication via global buffer," in *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '82, Ottawa, Canada: Association for Computing Machinery, 1982, pp. 10–18, ISBN: 0897910818. DOI: `10.1145/800220.806676`. [Online]. Available: `https://doi.org/10.1145/800220.806676`.

[3] A. K. Atkinson, "Tupleware: A distributed tuple space for the development and execution of array-based applications in a cluster computing environment," Ph.D. dissertation, University of Tasmania, 2010.

[4] L. Bettini, R. D. Nicola, and M. Loreti, "Implementing mobile and distributed applications in x-klaim," *Scalable Comput. Pract. Exp.*, vol. 7, 2006.

[5] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces principles, patterns, and practice.* Addison-Wesley Professional, 1999.

[6] V. Buravlev, R. D. Nicola, and C. A. Mezzina, "Tuple spaces implementations and their efficiency," in *International Conference on Coordination Languages and Models*, Springer, 2016, pp. 51–66.

[7] *In-Memory Digital Integration Hub (DIH) | GigaSpaces*, [Online; accessed 13. May 2022], Apr. 2022. [Online]. Available: `https://www.gigaspaces.com`.

[8] S. Craß, E. Kühn, and G. Salzer, "Algebraic foundation of a data model for an extensible space-based collaboration protocol," in *Proceedings of the 2009 International Database Engineering Applications Symposium*, ser. IDEAS '09, Cetraro - Calabria, Italy: Association for Computing Machinery, 2009, pp. 301–306, ISBN: 9781605584027. DOI: `10.1145/1620432.1620466`. [Online]. Available: `https://doi.org/10.1145/1620432.1620466`.

[9] A. Omicini and F. Zambonelli, "Coordination of mobile agents for information systems: The TuCSoN model," in *6th Convention of the Italian Association for Artificial Intelligence (AI*IA'98)*, S. Badaloni and C. Minnaja, Eds., AI*IA'98 Workshop on Knowledge Integration, Padova, Italy: Edizioni Progetto Padova, 1998, pp. 94–98.

[10] pSpaces, *Programming-with-Spaces*, [Online; accessed 13. May 2022], May 2022. [Online]. Available: `https://github.com/pSpaces/Programming-with-Spaces`.

[11] G. Ciatto, L. Rizzato, A. Omicini, and S. Mariani, "Tusow: Tuple spaces for edge computing," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, 2019, pp. 1–6. DOI: `10.1109/ICCCN.2019.8846916`.

[12] D. Balzarotti, P. Costa, and G. P. Picco, "The lights tuple space framework and its customization for context-aware applications," in *International Journal on Web Intelligence and Agent Systems (WIAS)*, 2007, pp. 215–231. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/lights-tuple-space-framework-customization-context-aware-applications/`.

[13] L. Bettini, V. Bono, R. D. Nicola, *et al.*, "The klaim project: Theory and practice," in *International Workshop on Global Computing*, Springer, 2003, pp. 88–150.

[14] Micutio, *rustupolis*, [Online; accessed 13. May 2022], May 2022. [Online]. Available: `https://github.com/Micutio/rustupolis`.

[15] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[16] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, "Vision and challenges for realising the internet of things," *Cluster of European research projects on the internet of things, European Commision*, vol. 3, no. 3, pp. 34–36, 2010.

[17] *Cisco Annual Internet Report (2018–2023) White Paper*, [Online; accessed 10. May 2022], Jan. 2022. [Online]. Available: `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`.

[18] S. Dustdar, C. Avasalcai, and I. Murturi, "Invited paper: Edge and fog computing: Vision and research challenges," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 96–9609. DOI: `10.1109/SOSE.2019.00023`.

[19] *Difference between edge computing and fog computing - geeksforgeeks*, `https://www.geeksforgeeks.org/difference-between-edge-computing-and-fog-computing/`, (Accessed on 05/02/2022).

[20] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 73–78. DOI: `10.1109/HotWeb.2015.22`.

[21] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. DOI: `10.1109/JIOT.2016.2579198`.

[22] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. Nikolopoulos, "Challenges and opportunities in edge computing," Nov. 2016. DOI: `10.1109/SmartCloud.2016.18`.

[23] *Aerospace Manufacturing and Design*, [Online; accessed 10. May 2022], May 2022. [Online]. Available: `https://www.aerospacemanufacturinganddesign.com/article/millions-of-data-points-flying-part2-121914`.

[24] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14, Bretton Woods, New Hampshire, USA: Association for Computing Machinery, 2014, pp. 68–81, ISBN: 9781450327930. DOI: `10.1145/2594368.2594383`. [Online]. Available: `https://doi.org/10.1145/2594368.2594383`.

[25] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 73–78. DOI: `10.1109/HotWeb.2015.22`.

[26] *Edgeos_ug.pdf*, `https://dl.ubnt.com/guides/edgemax/EdgeOS_UG.pdf`, (Accessed on 05/10/2022).

[27] M. Gusev and S. Dustdar, "Going back to the roots—the evolution of edge computing, an iot perspective," *IEEE Internet Computing*, vol. 22, no. 2, pp. 5–15, 2018. DOI: `10.1109/MIC.2018.022021657`.

[28] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Engineering collective intelligence at the edge with aggregate processes," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104 081, 2021, ISSN: 0952-1976. DOI: `https://doi.org/10.1016/j.engappai.2020.104081`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0952197620303389`.

[29] *MQTT Version 5.0*, [Online; accessed 7. May 2022], Oct. 2019. [Online]. Available: `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`.

[30] P. Manzoni, E. Hernández-Orallo, C. T. Calafate, and J.-C. Cano, "A proposal for a publish/subscribe, disruption tolerant content island for fog computing," ser. SMARTOBJECTS '17, Snowbird, Utah, USA: Association for Computing Machinery, 2017, pp. 47–52, ISBN: 9781450351416. DOI: `10.1145/3127502.3127511`. [Online]. Available: `https://doi.org/10.1145/3127502.3127511`.

[31] M. Farooq, M. Waseem, A. Khairi, and P. Mazhar, "A critical analysis on the security concerns of internet of things (iot)," *International Journal of Computer Applications*, vol. 111, pp. 1–6, Feb. 2015. DOI: `10.5120/19547-1280`.

[32] *Rs-object-space*, `https://crates.io/crates/object-space`, Accessed: 2022-03-28.

[33] *Rustupolis*, `https://crates.io/crates/rustupolis`, Accessed: 2022-03-28.

[34] T. Uzlu and E. Şaykol, "On utilizing rust programming language for internet of things," in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2017, pp. 93–96. DOI: `10.1109/CICN.2017.8319363`.

[35] rust-lang, *rust*, [Online; accessed 14. May 2022], May 2022. [Online]. Available: `https://github.com/rust-lang/rust`.

[36] *A proactive approach to more secure code – Microsoft Security Response Center*, [Online; accessed 14. May 2022], May 2022. [Online]. Available: `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code`.

[37] R. Jung, "Understanding and evolving the rust programming language," 2020. DOI: `http://dx.doi.org/10.22028/D291-31946`.

[38] *Linus Torvalds weighs in on Rust language in the Linux kernel*, [Online; accessed 14. May 2022], May 2022. [Online]. Available: `https://arstechnica.com/gadgets/2021/03/linus-torvalds-weighs-in-on-rust-language-in-the-linux-kernel`.

[39] L. Tung, "Google backs effort to bring Rust to the Linux kernel," *ZDNet*, Apr. 2021. [Online]. Available: `https://www.zdnet.com/article/google-backs-effort-to-bring-rust-to-the-linux-kernel`.

[40] ——, "Programming languages: Rust for Windows just got another update," *ZDNet*, May 2021. [Online]. Available: `https://www.zdnet.com/article/programming-languages-rust-for-windows-just-got-another-update`.

[41] *Stack Overflow Developer Survey 2021*, [Online; accessed 14. May 2022], May 2022. [Online]. Available: `https://insights.stackoverflow.com/survey/2021`.

[42] *crates.io: Rust Package Registry*, [Online; accessed 14. May 2022], May 2022. [Online]. Available: `https://crates.io`.

[43] rust-lang, *crater*, [Online; accessed 17. May 2022], May 2022. [Online]. Available: https://github.com/rust-lang/crater.

[44] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

[45] *About objective-c*, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, (Accessed on 08/14/2022).

[46] Contributors to Wikimedia projects, *B-tree - Wikipedia*, [Online; accessed 15. May 2022], May 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=B-tree&oldid=1086163784.

[47] *Arc in std::sync - Rust*, [Online; accessed 15. May 2022], Apr. 2022. [Online]. Available: https://doc.rust-lang.org/std/sync/struct.Arc.html.

[48] *Mutex in std::sync - Rust*, [Online; accessed 15. May 2022], Apr. 2022. [Online]. Available: https://doc.rust-lang.org/std/sync/struct.Mutex.html#method.try_lock.

[49] L. Mara, "Understanding Rust Thread Safety," *Customer Engagement Blog*, Jan. 2022. [Online]. Available: https://onesignal.com/blog/thread-safety-rust.

[50] M. A. Aleisa, A. Abuhussein, and F. T. Sheldon, "Access control in fog computing: Challenges and research agenda," *IEEE Access*, vol. 8, pp. 83 986–83 999, 2020.

[51] P. Zhang, J. K. Liu, F. R. Yu, M. Sookhak, M. H. Au, and X. Luo, "A survey on access control in fog computing," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 144–149, 2018.

[52] *IBM Docs*, [Online; accessed 15. May 2022], Mar. 2021. [Online]. Available: https://www.ibm.com/docs/en/zos/2.2.0?topic=environment-mandatory-access-control-mac.

[53] *Man-in-the-middle attack - wikipedia*, https://en.wikipedia.org/wiki/Man-in-the-middle_attack, (Accessed on 06/25/2022).

[54] *Mobile critters: Part 1. man in the middle attack. by kaymera - secure calls & encrypted messages for you and your team*, https://kaymera.com/how-does-the-man-in-the-middle-attack-work/, (Accessed on 06/25/2022).

[55] L. Shinder and M. Cross, "Chapter 12 - understanding cybercrime prevention," in *Scene of the Cybercrime (Second Edition)*, L. Shinder and M. Cross, Eds., Second Edition, Burlington: Syngress, 2008, pp. 505–554, ISBN: 978-1-59749-276-8. DOI: https://doi.org/10.1016/B978-1-59749-276-8.00012-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597492768000121.

[56] A. Vishwanath, R. Peruri, and J. (He, *Security in fog computing through encryption.* DigitalCommons@ Kennesaw State University Kennesaw, Georgia, USA, 2016.

[57] *Advanced encryption standard (aes) - geeksforgeeks*, https://www.geeksforgeeks.org/advanced-encryption-standard-aes/, (Accessed on 07/12/2022).

[58] *Aeads/aes-gcm at master · rustcrypto/aeads*, https://github.com/RustCrypto/AEADs/tree/master/aes-gcm, (Accessed on 07/12/2022).

[59] W. Vanhoof, *Calculabilité et complexité : Syllabus*, [accessed 15. May 2022], Mar. 2019.

[60] *A Guide To Parsing: Algorithms And Terminology*, [Online; accessed 16. May 2022], Mar. 2022. [Online]. Available: https://tomassetti.me/guide-parsing-algorithms-terminology.

[61] *About The ANTLR Parser Generator*, [Online; accessed 16. May 2022], May 2022. [Online]. Available: `https://www.antlr.org/about.html`.

[62] *channel in std::sync::mpsc - Rust*, [Online; accessed 16. May 2022], Apr. 2022. [Online]. Available: `https://doc.rust-lang.org/std/sync/mpsc/fn.channel.html`.

[63] *Raspberry pi 4 model b specifications – raspberry pi*, `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/`, (Accessed on 08/14/2022).

[64] *Xavierberger/rpi-monitor: Real time monitoring for embedded devices*, `https://github.com/XavierBerger/RPi-Monitor`, (Accessed on 08/14/2022).

[65] *Samsung galaxy a52s specs - phonearena*, `https://www.phonearena.com/phones/Samsung-Galaxy-A52s_id11801`, (Accessed on 08/14/2022).

[66] *Jni - crates.io: Rust package registry*, `https://crates.io/crates/jni`, (Accessed on 06/22/2022).

[67] *Lux - wikipedia*, `https://en.wikipedia.org/wiki/Lux`, (Accessed on 06/22/2022).

Chapter 7

Appendices

7.1 Installation of the server and interpreter

To test the tuple server, you first need to :

- Clone the repository at : `https://github.com/Maxbever/LIF_Server`
- Install Rust following the instructions for your OS: `https://www.rust-lang.org/tools/install`
- Run the command *cargo run –example multiple_ server*

To test the client part of the library, you need to :

- Run the command *cargo run –example client*

To test the interpreter, the steps to follow are as follows:

- Download the interpreter from this link: `https://github.com/Maxbever/LIF_Interpreter/tree/master/interpreter`
- Open a terminal and launch the interpreter with a *.lif* file as argument

7.2 Grammar of LiF

7.2.1 Parser Rules

```
grammar lif;

import lifWords;

/*
 * Parser Rules
 */
root:instruction*;

instruction:        connect
                |   create
                |   delete
                |   attach
                |   out
                |   for_instr
                |   assignation
                |   while_instr;

connect : CONNECT server_name protocol DOUBLEDOT ip_address DOUBLEDOT port encryption_key;

create: CREATE attribut server_name DOUBLEDOT tuple_space_name (attribut)*;

delete:  DELETE attribut server_name DOUBLEDOT tuple_space_name;

attach: ATTACH server_name DOUBLEDOT tuple_space_name (attribut)* LEFT_BRACE  (instruction)* RIGHT_BRACE;

out: OUT tuple(COMMA tuple)*;

for_instr : FOR ID EQUAL operation TO LPAR operation RPAR LEFT_BRACE instruction+ RIGHT_BRACE;

while_instr: WHILE LPAR boolean_operation RPAR LEFT_BRACE instruction+ RIGHT_BRACE;

boolean_operation:      basic_boolean_operation
                    |   boolean_operation AND basic_boolean_operation
                    |   boolean_operation OR basic_boolean_operation;

basic_boolean_operation:    right_expr EQUAL EQUAL (right_expr || empty_tuple)
                        |   right_expr RCHEVRON EQUAL right_expr
                        |   right_expr LCHEVRON EQUAL right_expr
                        |   right_expr EXCLAMATION EQUAL (right_expr || empty_tuple);

operation :   get_function
            | len_function
```

```
             | right_expr
             | operation PLUS operation
             | operation MINUS operation
             | operation KLEENE operation
             | operation SLASH operation;

get_function: tuple DOT GET LPAR right_expr RPAR;

len_function: tuple DOT LEN LPAR RPAR;

right_expr : ID | NUMBER;

assignation : VAR ID EQUAL ( init_var  |   read   |   in_instr | operation);

read: READ tuple(COMMA tuple)*;

in_instr: IN tuple(COMMA tuple)*;

attribut: STRING | ID ;

encryption_key: STRING | ID ;

tuple : LPAR (tuple_content (COMMA tuple_content)*) RPAR | ID;

tuple_content : init_var | WILDCARD ;

tuple_space_name: STRING | ID ;

server_name: STRING | ID ;

init_var:   NUMBER
            | STRING
            | CHARACTER
            | ID
            | tuple;

protocol : UDP | TCP;

ip_address: NUMBER DOT NUMBER DOT NUMBER DOT NUMBER;

port: NUMBER;

empty_tuple : LPAR RPAR ;
```

7.2.2 Lexer Rules

```
lexer grammar lifWords;

```

```
/*
 * Lexer Rules
 */

// Words
CONNECT : 'connect';
ATTACH : 'attach';
CREATE : 'create';
DELETE : 'delete';
OUT : 'out';
READ : 'read';
IN : 'in';
TCP : 'tcp';
UDP : 'udp';
VAR : 'var';
GET : 'get';
LEN : 'len';
FOR : 'for';
TO : 'to';
WHILE : 'while';
AND : 'and';
OR : 'OR';

// Char
LPAR: '(';
RPAR: ')';
COMMA: ',';
QUOTE:'\'';
SLASH : '/';
BACKSLASH:'\\';
LBRACKET:'[';
RBRACKET:']';
DOT:'.';
DOUBLEDOT : ':';
SEMICOLON : ';';
KLEENE : '*';
WILDCARD: '_';
EQUAL : '=';
PLUS: '+';
MINUS: '-';
RIGHT_BRACE : '}';
LEFT_BRACE : '{';
LCHEVRON : '<';
RCHEVRON : '>';
AMPERSAND : '&';
EXCLAMATION : '!';

ID: LETTER (LETTER | DIGIT | WILDCARD)* ;
fragment LETTER: 'A'..'Z' | 'a'..'z' ;
fragment DIGIT: '0'..'9' ;

//Types
NUMBER: (DIGIT)+;
STRING : DOUBLEQUOTE(LBRACKET|BACKSLASH|COMMA|NEWLINE|RBRACKET|ID)+
    ↪ DOUBLEQUOTE;
CHARACTER : QUOTE (NUMBER|LETTER|DOUBLEDOT|DOT|SLASH|BACKSLASH|SEMICOLON)
    ↪ QUOTE;

// Comments -> ignored

LINECOMMENT : (SLASH SLASH .*? (NEWLINE|EOF)) ->skip;
COMMENT: (SLASH KLEENE .*? KLEENE SLASH) -> skip;

// Whitespaces -> ignored

NEWLINE: '\r'? '\n'  -> skip ;
WS: [ \t]+ -> skip ;
```