Enhancing Privacy On Smart City Location Sharing

Luís Miguel dos Santos Magalhães
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2021

Orientador
Luís Filipe Coelho Antunes
Professor Catedrático
Faculdade de Ciências da Universidade do Porto

Coorientador
João Miguel Maia Soares de Resende
Professor Assistente Convidado
Faculdade de Ciências da Universidade do Porto

U. PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/________

UNIVERSIDADE DO PORTO

MASTERS THESIS

Enhancing Privacy In Smart City Location Sharing

Author:
LUÍS MAGALHÃES

Supervisor:
LUÍS ANTUNES

Co-supervisor:
JOÃO RESENDE

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Information Security

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

December 30, 2021

Acknowledgements

I want to thank Professor Luís Antunes and Professor João Resende for the opportunity of developing this work on a great environment.

For all the support and ideas I also want to thank Professor Patrícia Sousa.

I want to thank all my friends and family for all the support given in the ups and downs of the work done in this dissertation, keeping me motivated throughout. I want to specially thank my girlfriend and my brother for keeping me motivated, being by my side when I needed the most.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Information Security

Enhancing Privacy In Smart City Location Sharing

by Luís MAGALHÃES

With an estimated 70% of the human population living on cities by 2050[1], there is an increasing pressure for a more efficient use of the city resources, being the notion of a smart city considered a winner in terms of urban strategy enabling the city to make decisions based on the data collected. However the increase on data collection, especially location data, impacts citizen privacy.

The increase on the data collected by the smart cities also increases the pressure over storage on city-premise, requiring constant maintenance and periodic storage increases to cope with all the data being collected. Cloud storage due to its high availability, scalability and low maintenance costs is presented as a solution, however, cloud storage presents new security challenges related to the confidentiality and integrity of the data stored at rest, and data sovereignty issues as well.

Location data, is a special target for attackers since it allows direct access to the personal life of citizens as it allows to know where they where and for how long, enabling an attacker to perceive daily habits and use that information for targeted attacks like stalking and spear-phishing.

In this work we aim to build an smart city infrastructure capable of dealing with an ever increasing number of citizens, allowing cloud based storage persistency and provide confidential, integral and authentic communication channels between clients and client to smart-city communication. We pay special attention to the anonymization of location data due to its high impact on citizen privacy by deploying client side privacy controls.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto

Departamento de Ciência de Computadores

Mestrado em Segurança Informática

Melhorar Privacidade na Partilha de Localização em Smart Cities

por Luís MAGALHÃES

Com um aumento estimado de 70% na quantidade de individuos a viver nas cidades até 2050[1], existe um aumento na necessidade de se ter um uso mais eficiente dos recursos existentes nas cidades. A noção de *smart-city* aumenta a eficiencia do planeamento urbano por permitir que decisões sejam feitas baseadas em informação coletada na cidade. Contudo o aumento da coleta de dados, especialmente dados relacionados com localização, têm um impacto significativo na privacidade dos cidadãos.

O aumento dos dados coletados pela *smart-city* também aumenta a pressão nos mecanismos de armazenamento da cidade, necessitando de manutenção constante e aumentos de capacidade períodicos para ser capaz de armazenar os dados coletados pela cidade. Armazenamento baseado em cloud devido à sua grande capacidade, disponibilidade e baixos custos de manutenção é apresentado como uma solução, contudo, armazenamento em cloud apresenta novos desafios relacionados com confidencialidade e integridade dos dados armazenados e problemas de soberania dos dados.

Dados de localização, são um forte alvo para atacantes, uma vez que permite um acesso direto à vida pessoal dos cidadãos pelo que permite saber onde estes estiveram e por quanto tempo, dando capacidade de perceber hábitos diários e usar essa informação para ataques direcionados como *stalking* e *spear-phishing*.

Neste trabalho nós construímos uma infraestrutura de cidade inteligente, capaz de lidar com um número crescente de cidadãos, com persistência baseada em cloud e capacidade de gerar canais de comunicação confidenciais, integros e autênticos para comunicação entre clientes e cliente para smart-city. Nós neste trabalho também damos especial atenção à anonimização de dados de localização devido ao seu impacto na privacidade, atingindo isso através de controlos de privacidade no lado do cliente.

Contents

List of Figures

List of Tables

Acronyms

ABAC Attribute-Based Access Control. 70, 71

AES Advanced Encryption Standard. 30, 52, 56, 59, 60, 90, 97–100

API Application Programming Interface. 53, 87, 88, 97, 98

AVX Advanced Vector Extensions. 7

AWS Amazon Web Services. 16, 41, 87, 89, 115, 116

BFT Byzantine-Fault Tolerance. 8

CIFS Common Internet File System. 30

CNI Container Network Interface. 95

CPU Central Processing Unit. 34, 116, 119

CRL Certificate Revogation List. x, xi, 64, 74–76, 81, 82, 84, 101–103, 119

CRUSH Controlled Replication Under Scalable Hashing. 35–38, 40, 50

CSR Certificate Signing Request. 44, 55–57, 97

DHT Distributed Hash Table. 6, 31

DNS Domain Name Service. 65, 91, 92, 94, 96, 105, 117

DOS Denial Of Service. 49

ECDSA Elliptic-Curve Digital Signature Algorithm. 44, 54–57, 59, 64, 97

ECIES Elliptic Curve Integrated Encryption Scheme. 54, 55, 57, 59, 97, 127

EMD Earth Mover Distance. 20, 21

FUSE Filesystem in USErspace. 29, 31

GCP Google Cloud Provider. 41, 87, 89

GDPR General Data Protection Regulation. 16

GE Generic Enabler. 14, 15, 69

gRPC google Remote Procedure Call. 93, 101

HDD Hard Drive Disk. 34

HMAC Hash-based Message Authentication Code. 30, 76, 77, 90

HTTP HyperText Transfer Protocol. 68, 88, 103, 107, 108, 118, 119

HTTPS HyperText Transfer Protocol Secure. 116, 117

I/O Input/Output. 40, 50, 82

IoT Internet of Things. x, 2, 3, 9, 11, 14, 16, 44, 53, 61, 64, 66, 71, 73, 74, 76–79, 84, 111, 116–119, 127

IP Internet Protocol. 65, 66, 91–95, 108, 109

IV Initialization Vector. 30, 52, 56, 57, 60, 76

JCE Java Cryptographic Extension. 101

JSON JavaScript Object Notation. 67

JWT JSON Web Tokens. 43

KDF Key Derivation Function. 55–57, 59

NAT Network Address Translation. xiv, 108, 109

NFS Network File System. 30, 34, 74, 112–114

OSD Object Storage Device. xiii, 34–42, 50, 77, 95

OTP One-Time Pad. xiv, xv, 53–58, 61, 97–101, 105–107

PAP Policy Administration Point. 71

PBKDF2 Password-Based Key Derivation Function 2. 97, 98

PDP Policy Decision Point. 71, 73, 116

PEP Policy Enforcement Point. 69, 71, 127, 128

PG Placement Group. xiii, 36–38

PKI Public Key Infrastructure. 97

POSIX Portable Operating System Interface. 35

PVC Persistent Volume Claim. 91, 95, 96

QR Quick-Response. 54, 55, 58, 105–107

RADOS Reliable Autonomic Distributed Object Store. 35–38

RAM Random Access Memory. 116, 119

RBAC Role-Based Access Control. 70, 71

RSA Rivest-Shamir-Adleman. 30, 44, 97

SMARTIE Secure and sMARter ciTIEs data management. ix, xiii, 9–11, 15

SSD Solid-State Drive. 34, 112

SSDP Simple Service Discovery Protocol. 105

SSServProv Smart Secure Service Provisioning. ix, xiii, 12–15

STMS Smart Traffic Management System. 1, 78

TCP Transmission Control Protocol. 37

TLS Transport Layer Security. 44

UPnP Universal Plug n' Play. xiv, 108–110

URL Uniform Resource Locator. 68, 107, 108

VRRP Virtual Router Redundancy Protocol. 93

WML Workload Model Language. 112–114

XACML eXtended Access Control Markup Language. 71–74

YAML YAML Ain't Markup Language. 95

Chapter 1

Introduction

During the last 50 years, city dimensions have been increasing more and more all over the world and it is estimated that by 2050, 70% of the world population will live on cities[1]. The increasing amount of population and dimensions introduce difficulties on the traditional city management techniques that cannot cope with the increasing amount of population, aggravating city traffic, pollution and waste production, leading to reduced quality of life[2]. The Smart City strategy is considered a winner in terms of urban strategies by using technology to increase the quality of life in urban places, improving environment quality and helping to deliver better services to the city citizens[3].

Smart Cities allow the establishment of dynamic networks where citizens can enjoy access to real time information related to the city. Applications like, Smart Traffic Management System (STMS)[4], where citizens can gather real time information about the current traffic situations on the city streets, including information like traffic jams, car crashes and other useful information related to the safe and efficient use of the city traffic network, and Smart Mobility, that aims to use the smart city data to provide a efficient use of city resources to provide better services like public transportation scheduling based on citizens affluence and time of day.

For a smart city to work properly it needs mechanisms to gather data, one of the ways is to place sensor devices at street level capable of sensing, among others, the amount of traffic or people affluence in order to provide the smart city with the information needed. Despite seeming a plausible solution at first place, it becomes unbearable when dealing with the dimensions of today's cities and the maintenance costs that the deployment of these devices impose. To control costs, cities usually apply street level sensors on the most urban areas of the city, meaning that only a small part of the city is capable of providing data to the smart city infrastructure. Another solution is to use devices that the citizens have as sensor devices. Examples of these devices include smartphones, cars, smart-watches and Internet of Things (IoT) devices. This solution provides a low cost and low maintenance sensor network with high geographical coverage, enabling all city regions to provide data to the smart city infrastructure. However privacy and data authenticity problems on current smart city deployments constraint the usage of this type of solution.

Privacy concerns come in the form of the possibility of identifying an individual based on the mobility of that user along the smart city. Studies like those performed by *de Montjoye et. al*[5] could successfully identify 95% of the individuals on a data-set using mobility data of 1.5 million users collected over 15 months. The possibility of identifying a citizen and its traces amidst other citizen traces not only affects the privacy of the citizen but can also affect the security of that citizen, since that data can be used, for example, to stalk or physically threat an individual.

IoT is a broad term and represents a wide development area that attempts to embed computing elements to every day items. Allowing them to connect to the internet, receive commands or provide data about its state. This computing elements are, usually, developed to have low processing power and general low power consumption enabling these devices, when coupled with battery power, to be placed on the environment to collect data about its surroundings. These properties make IoT, and other similar devices, ideal to be placed as smart city sensors, where they gather data about the environment and send them to a centralized service. These properties are also ideal to power user owned sensors, like smart-watches and other wearable technologies, making IoT the perfect type of device to be used to develop a wide range sensing network for the smart city. However, since the majority of IoT devices are battery powered the smart city network should be able to function when a IoT device runs out of battery and be able to securely reintroduce the device when it receives power.

Increasing sensor density increases the data collected and processed in the smart city. A smart city infrastructure usually needs to keep data stored for a long time, in order to do operations, such as mobility analysis and trend detection, on which old data is compared with newer data in order to analyse the differences and reach some conclusion. However the necessity of keeping data stored for a long time increases the stress over the city on-premise storage resources. Cloud storage, due to its low cost, high availability and geo-redundancy has become the perfect candidate to enable the smart-cities to keep older data stored. However problems related with the data confidentiality and data sovereignty on the usage of public cloud storage[6] reduce the usage of this technology to enable long term, persistent storage.

Location Data enables the smart-city to provide a more efficient use of resources and helps to provide better public services, such as transportation, to the citizens, improving quality of life. However location data collection affects severely the privacy of the citizens. Enabling attackers to know where the citizen is located and how and when it travelled to locations on the smart city. Therefore, allowing to understand the citizen behaviour and use it for malicious purposes. Several technologies[7][8][9] attempt to separate the citizen identity from the location data itself, using anonymization techniques to mitigate the privacy impact, however location data properties such as location correlation on continuous updates[10] often reduce the expected privacy gain of the usage of those technologies.

1.1 Motivation

The main motivation behind this dissertation is to protect the privacy of citizens when they use location based services, by using anonymization techniques that mitigate the privacy loss even on continuous updates. We also want to improve current smart city deployments by adding the capacity of provisioning large pools of IoT devices, provide efficient authentication of devices and device access revoking protocols. In addition, we also want to enable smart-cities to use public cloud storage for persistency of old data without compromising the security of the data at rest.

1.2 Proposal

With this work we propose to build a infrastructure centered on location privacy that can be easily extended and applied into a real smart city scenario. This infrastructure will provide the following:

- **Secure device provisioning and decentralized authentication**, enabling to have a large pool of devices capable of communicating with each other without requiring a centralized authentication server;

- **Client-Client communication**, on which client devices are allowed to communicate with each other securely, in a way that the confidentiality between those clients is protected;
- **Access Control policies**, on which the access to resources is allowed or denied according to the policies in place;
- **Client-side location privacy**, on which the client device will be the responsible to anonymize its own location data, improving the client control over its own data.
- **Cloud storage for data persistency**, on which the data generated by the smart city sensors will be stored securely on the cloud with redundancy mechanisms, ensuring data persistency and availability.

1.3 Contributions

During this work, the following paper was published related to the work performed on this dissertation :

- *Provisioning, Authentication and Secure Communications for IoT Devices on FIWARE* published on MDPI's journal Sensors, special issue of Security and Privacy in Cloud Computing Environment, with DOI 10.3390/s21175898 [11], on which I am a co-author.

1.4 Organization

In this document we started by describing some of the background concepts that will help us better understand the related work, then we explore the technologies related to smart cities, location privacy and secure cloud based storage on the related work. After the related work we describe the threat model, attacker model and assumptions on the security requirements chapter. Then with the knowledge obtained from the related work and the notion of the possible attack vectors we design a scalable system architecture that is then implemented in chapter 6. After the implementation is done we test it in chapter 7 and draw some conclusions on chapter 8.

Chapter 2

Background Concepts

In this chapter we aim to present an overview of the concepts used by related work techniques and protocols that will be discussed on chapter 3.

2.1 Anonymization of Personal Identifiable Information

Anonymization often requires that the data of an individual must get scrambled in the middle of other data belonging to other individuals in a way that an attacker cannot tell which data belongs to each individual with more certainty than a random guess. This means that, after the anonymization process, the anonymized data must not contain any information that might help an attacker to identity the individual represented by that data.

To ease the anonymization of a data set, syntactic anonymization require data attributes are assigned with properties that, depending on its risk to identify the individual, should be treated differently. There are 3 main types of attributes: Key, Quasi-identifier and Sensitive.

2.1.1 Key Attributes

Key attributes are those that can identify an individual directly without needing access to any external information. Examples of key attributes are names,emails and social security numbers. Key attributes require removal or obscurity.

2.1.2 Quasi-Identifier Attributes

Quasi-identifiers are attributes that can, with help of external information, identify an individual. Quasi-identifiers, like zip-codes, birthday and gender are the main concern of anonymization techniques, such as K-Anonymity[12], requiring suppression or generalization in a way that satisfies the anonymization technique requirements.

2.1.3 Sensitive Attributes

Sensitive attributes are those that an individual is sensitive about revealing. Sensitive attributes, that include income, type of illness, etc., must be de-linked from the individual in a way that ensures the individual anonymity. Sensitive attributes are usually an important piece of the data-set for analytics and machine learning algorithms, being the de-link of a sensitive attribute to a individual of utmost importance.

2.2 Ensuring data persistence and availability

Nowadays, and more than ever, digital storage is used in the every day needs. Information must be stored in a way that access to that information is always granted even on a hardware failure. Outside of the usual backup methods, other methods attempt to enable redundancy and availability trough replication and distribution of information across different hardware storage components or cloud services. In this section we aim to present the concepts that state of the art storage solutions use to achieve guaranties of data persistence and availability.

2.2.1 Distributed Hash Tables

Distributed Hash Table (DHT) is a technique that uses a set of decentralized and distributed nodes to store $< key, value >$ pairs in a hash table. Each one of the nodes is responsible to store a portion or the totality of the hash table in a way that when a request is performed to obtain a $< key, value >$ pair multiple nodes can answer the request, improving availability. When a $< key, value >$ is stored into a DHT system, multiple nodes store that $< key, value >$ pair, this way increasing redundancy and hardware failure tolerance.

Due to the DHT distributed nature, solutions that use the concept of Distributed Hash Tables can scale with minimal overheads.

2.2.2 Erasure Coding

Erasure Coding is a family of algorithms that take the original data and create a longer version of that data. The result can be used to reconstruct the original even if some parts are missing or corrupted. Algorithms with these properties are important to establish redundancy of data. By dividing the longer version of the data into chunks and storing them across multiple storage devices redundancy is achieved due to the fact that if some storage device is missing or corrupted the original data can still be recovered using the remainder storage devices.

One example of a erasure coding algorithm is the Reed-Solomon coding, presented by I. S. Reed and G. Solomon in 1960 [13]. Reed-Solomon is used in the Linux RAID module, at Microsoft Azure and at Backblaze Vaults [14] as the state of the art redundancy and availability provider.

A Reed-Solomon code is specified as $RS(n,k)$ with s-bit symbols. This means that the encoder takes k data symbols of s length (i.e. k 8-bit characters on a text), then it adds n-k parity symbols with s bits of length making a codeword of length n symbols. With this encoding scheme it is possible to correct at most $t = \frac{n-k}{2}$ s-bit symbols in any part of the codeword. Given t it is possible to say that the more parity symbols a codeword has the more corrections to itself are possible, however less data is added into the codeword meaning that more codewords are needed to encode all the data, this increases the total space consumption of the encoded result.

Reed-Solomon specifies that $n \leq 2^s - 1$, imposing a limit on the length of the keyword dependent on the number of bits per symbols defined.

Current software implementations of Reed-Solomon on a AMD Ryzen 9 3950X CPU with 16 physical cores and Advanced Vector Extensions (AVX) 2 can achieve single threaded encoding and decoding speeds of 9979 MB/s under certain n, s, k combinations.[15]

2.2.3 Byzantine-Fault Tolerance

A Byzantine-Fault is a condition of a computer system where components may fail and there is imperfect information on whether a component has failed. In a Byzantine-Fault, a component such as a storage node can inconsistently appear both as failed and working to other components on the system. This lack of consistency causes the system to not know if a component is truly working or suffered some failure and because of that is not possible to take coordinated actions to fix the problem.

Byzantine-Fault Tolerance (BFT) is a property of a system that is able to continue operating even if some of the nodes fail or act inconsistently/maliciously. BFT can be used to improve resilience in scalable systems that use server replicas to serve contents to a large pool of users simultaneously. The way BFT achieves this is by enabling the system components to reach and agreement on which component is failing, acting inconsistently or acting maliciously to enable all the components to achieve consensus and take coordinated actions.

2.2.4 Cloud-of-Clouds Storage

Cloud-of-clouds corresponds to the usage of multiple cloud backends to fulfill the storage purposes of the system[16]. A cloud-of-clouds system appears to the user abstracted as single storage container, being an installed application or proxy server responsibility to split the information across the clouds on the upload, and unite the information on a download. Cloud-of-clouds together with redundancy mechanisms improve the storage system availability while minimizing the stress on local hardware.

Chapter 3

Related Work

3.1 Smart Cities Architectures

In this section we present state of the art smart city architectures that use data provided by sensing devices, such as IoT devices, to provide services to citizens and improve efficiency of city planning.

3.1.1 SMARTIE

Secure and sMARter ciTIEs data management (SMARTIE)[17] is an European project with the goal to create a distributed framework for IoT-based applications capable of storing, sharing and processing large volumes of heterogeneous information while enabling end-to-end security, allowing the setting of privacy requirements per user and reduce the energy consumption.

SMARTIE envisions an architecture (fig. 3.1), where various sensors provide data into a a data platform and actuators in the city receive commands from the platform. The platform offers interfaces for various kinds of services, such as traffic analysis and energy control management.

SMARTIE introduces security mechanisms to protect the confidentiality, integrity and availability of the system as well as mechanisms to protect the privacy of individual citizens on the network in the form of functional groups.

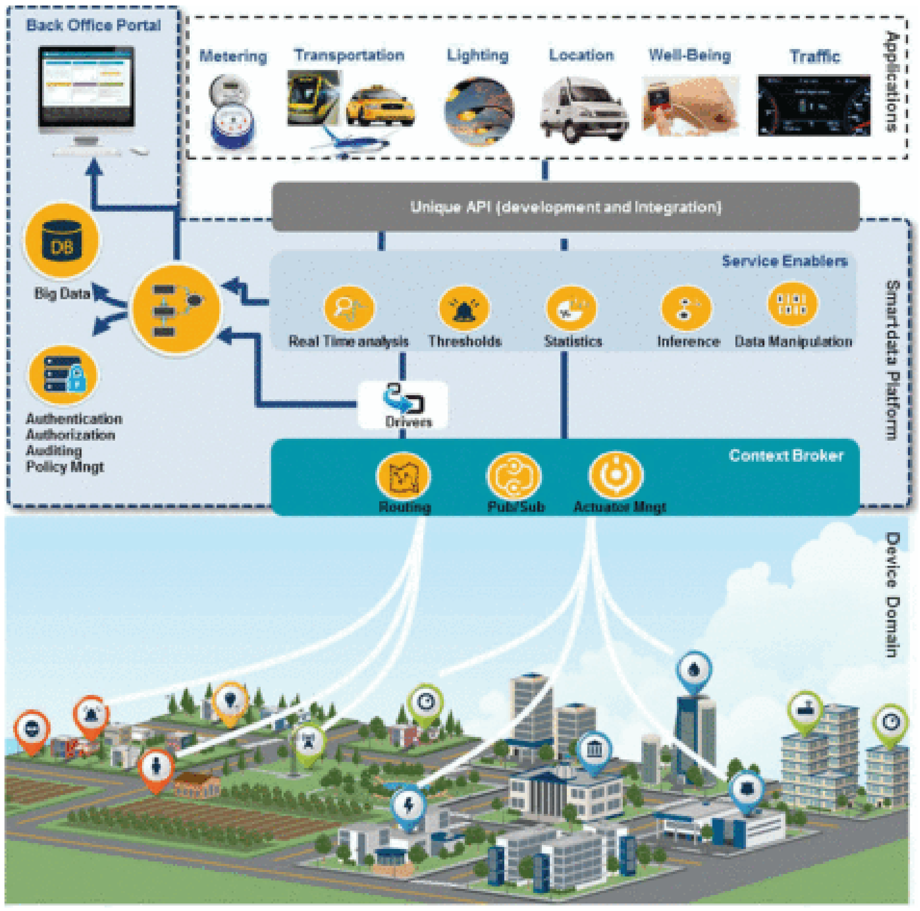


FIGURE 3.1: SMARTIE architecture (extracted from [17])

The security functional group contains authentication, authorization, integrity and confidentiality mechanisms. Authentication and Authorization are provided by *DCapBAC*[18]. *DCapBAC* is authentication and authorization protocol built on top of public-key,elliptic curve, cryptography with optimization to make it deployable on resource constrained devices. When a user is authenticated on *DCapBAC*, it receives a collection of signed resource tokens. Those tokens grant direct access to resources on the smart city. The collection of resource tokens to give a certain user is based on the authorization policies in place. When a user accesses a resource it appends the resource token into the request, and the receiver end is able to verify the validity of the resource token by verifying the signature, allowing to have a distributed access control framework where the verification of the access to a resource is done locally on the receiver without the need to connect to a centralized server. The identity of the user, token validity, and its capabilities are stored on the resource token in clear text, being easily obtainable by every device on the smart city.

In addition to *DCapBAC*, SMARTIE uses *shortECC* library for encryption and digital signature that provide, respectively, confidentiality and authentication to the data sent and received. *shortECC* is designed to have low resource consumption, being more energy efficient, enabling the deployment on resource constrained devices.

Privacy preservation is performed on the IoT service functional group. SMARTIE proposes PrivLoc[19] as a location privacy enabler. PrivLoc ensures that tracking inside defined geo-fences is not possible. To do so it requires a trusted server that is placed between the user and the location based service. The location based service receives the real location from the users and forwards an anonymized result to the location based service.

On PrivLoc, the trusted server starts dividing the geo-fenced region map into a regular grid with fixed-size square tiles - the size being defined on the PrivLoc parameters-, those tiles are then numbered from left to right, top to bottom. Then PrivLoc generates a uniformly random key K with the length specified in the PrivLoc parameters. When the trusted server receives a new location it uses the key K to perform five operations: coordinateOnTile, permuteTiles, rotateTile, flipTile and OPE that translate the real location into an anonymized one. The operations are called in the following order:

1. **coordinateOnTile** - uses as input the grid and the received location and outputs the tile number where the location belongs and the relative location on that tile;

2. **permuteTile** - uses a pseudo-random permutation with K and the tile number from **coordinateOnTile**, outputting a new tile number;

3. **rotateTile** - uses as input the new tile number from **permuteTile** and the relative location obtained from **coordinateOnTile** and rotate the tile associated with that tile number a random amount from the set: $[0,\pi/2,\pi,3\pi/4]$, outputting new relative coordinates;

4. **flipTile** - uses the tile number from **permuteTile** and the relative location obtained from **rotateTile** and flips the tile on its vertical axis originating newer relative coordinates. The authors state that the combination of **rotateTile** and **flipTile** results in a total of $3log_2(nm)$ bits of entropy added on each tile, being nm the number of tiles on the grid;

5. **OPE** - uses Order-Preserving-Encryption to mask the distances of movements executed by devices. It uses the relative location obtained from **flipTile**, the key K and the tile number from **permuteTile**, returning a new relative location. This relative

location marks the end of the anonymization process. After **OPE** the relative location output is then mapped into the absolute location and returned to the location based service providers on the smart city.

These operations enable PrivLoc to share location information without leaking any meaningful information about the location, trajectory and velocity to the service providers but only while the user is kept inside the geo-fenced region(s).

Although it can mitigate tracking inside a geo-fence, PrivLoc does not provide any guarantees about the correlation between real location and the anonymized location, especially when the user enters or leaves the geo-fenced region, causing privacy issues [19, 20].

3.1.2 SSServProv

Smart Secure Service Provisioning (SSServProv)[21] is a security and privacy-aware framework for service provisioning in smart cities, providing end-to-end security and privacy features for trustable data acquisition, transmission, processing and legitimate service provisioning. Figure 3.2 shows the concept architecture of SSServProv.

As it is possible to see from the figure 3.2, the SSServProv architecture is comprised of three distinct sections: Smart city infrastructure and inhabitants, Government and lastly, service providers. The Smart city infrastructure and inhabitants section is responsible for the authentication, access control, confidentiality and data anonymization. The authentication process enables the registration of citizen devices into the smart city, and, uses the citizen identity for the process. The result of a registration is the creation of an access control policy and a certificate or access token that is passed to the device in order for it to communicate with the service providers on the smart city. When a citizen device wants to access a service or publish data it sends a request to the smart city that checks the validity of the request using the access control policies defined on the registration process, it then checks if the access control policies in-place allow the request. If allowed, that request is then sent to the service provider, otherwise it is blocked and a error is returned.

For data confidentiality, integrity and non-repudiation this section uses encryption to achieve the confidentiality, and adds a digital signature field, that uses the certificate received on the registration process to sign the message, providing authenticity, integrity and non-repudiation.

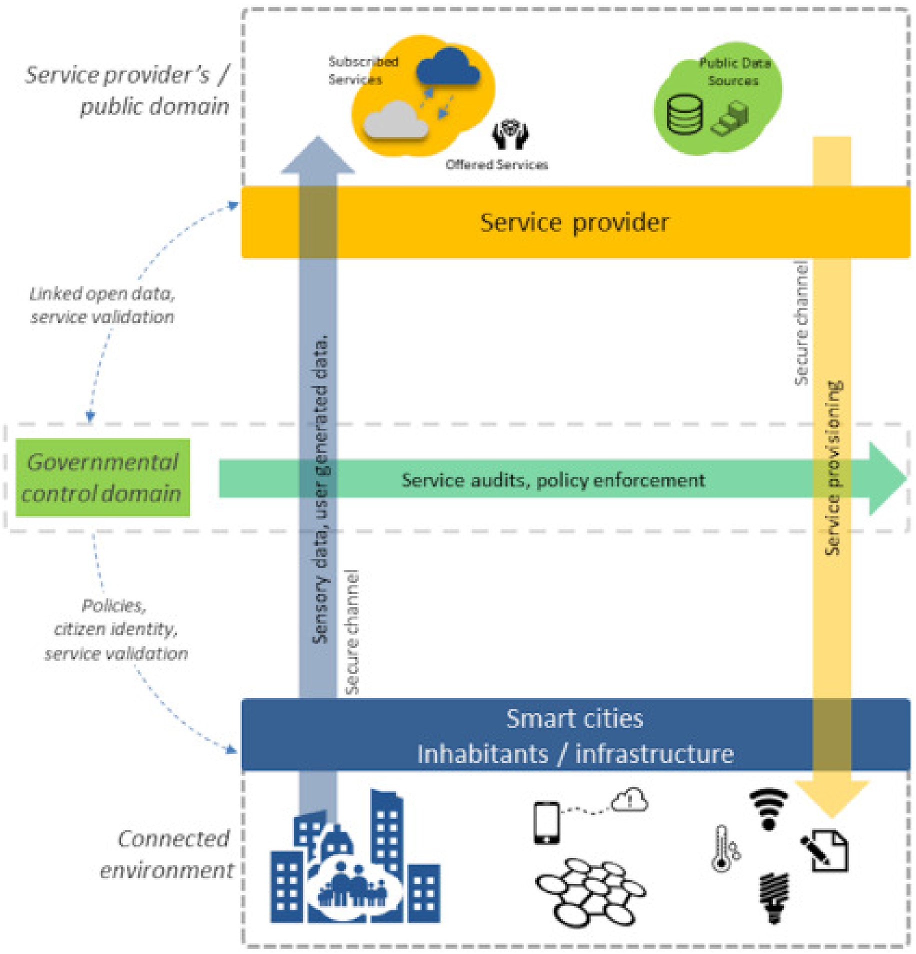

Service provider's / public domain
Subscribed Services
Offered Services
Public Data Sources
Service provider
Linked open data, service validation
Secure channel
Governmental control domain
Service audits, policy enforcement
Sensory data, user generated data.
Service provisioning
Policies, citizen identity, service validation
Secure channel
Smart cities
Inhabitants / infrastructure
Connected environment

FIGURE 3.2: SSServProv framework concept architecture (extracted from [21])

The data anonymization in SSServProv is used to decouple the user identity from the data published and is achieved by replacing the user information with a randomized pseudonym, replacing a specific value with a range of values. This helps to restrain malicious entities ability to link the data to an specific citizen. However, this method is flawed when considering location data, where the utility or privacy can be severely affected by the value range selected for the anonymization. This anonymization method does not provide effective means to mitigate correlation between locations, causing issues to the citizens location privacy.

The Government section is responsible to verify the service providers and deciding which service providers are allowed to operate within the smart city domain, limiting the exposition of citizens to malicious services. The government is also responsible to analyse the data acquisition and service provisioning channels to verify if there is any unauthorized service provider or malicious data being provided by the smart city infrastructure, allowing the government to preemptively check for unexpected behaviour and act accordingly, mitigating security and privacy issues. The government is also responsible to provide citizen identities and for ensuring that when the data is acquired, it is authenticated, and can be trusted. This ensures that any malicious data can be traced back to the sender resulting on a more resilient system.

Lastly, the service provider section, is where data repositories and provisioned services live. Data repositories receive all the data on SSServProv, connecting to the public cloud to offload persistent data to it in a integral and confidential manner using encryption and hashing. Additionally, data repositories also allows data sharing between provisioned services.

3.1.3 FIWARE

FIWARE is an European project which aims to build an open-source platform for smart cities [22] with security and modularity in mind. FIWARE architecture is composed of Generic Enabler (GE)s which are modules responsible to execute one type of task on the city infrastructure. FIWARE provides GEs for context brokers, context management, IoT agents, context processing, analysis, visualization, authorization, identity management and data publication.

The context brokers are similar to the server on a traditional publish/subscribe network as they are only responsible to forward updates from the publishers to the subscribers and to manage subscriptions. Context brokers are the core of FIWARE, enabling the communication between IoT devices, that compose the sensing layer of the smart city, to the smart city itself. Attached to the context broker there can be modules that store the data collected on-premise as the context-broker by default does not have any storage capabilities.

To enable the communication between FIWARE and heterogeneous IoT devices running different communication protocols, FIWARE created IoT agents. A IoT agent is responsible to translate the data received by the IoT device from the IoT device communication protocol (i.e. zigbee) to the communication protocol used by FIWARE, and, when a response arrives, convert it to the communication protocol used by the IoT device.

Context processing GEs enable to build dashboards and trigger events based on the data received by the context broker, allowing to analyze and visualize the smart city behaviour in real-time.

FIWARE also provides more security focused Generic Enablers that usually sit in front of the context broker and mediate the access to it. To mediate the access, FIWARE has identity management, policy administration, policy decision and policy enforcement Generic Enablers. Identity management allows to manage users and their authentication tokens, such as passwords. Identity management is also responsible for logins, logouts and registrations, and producing access tokens (typically OAuth2-based). Policy administration GEs are responsible for creating access control policies and manage them, restricting the access to resources.

Policy enforcement GEs act as a proxy between the context broker and the users, enforcing the authentication and access control, only allowing access to the context broker if the authentication is valid and the policy decision returns a ”Permit” result. Otherwise the request results in a denial of access. The policy decision GE is responsible to receive the policy decision requests from the policy enforcement GE and using the information from the policies on the policy administration GE to produce a ”Permit”, allowing access, or ”Deny”, denying access, result.

FIWARE main advantage is that it is not mandatory to use the GE implementations provided by FIWARE, allowing developers to create their own GE implementations making possible to achieve specific goals that the original implementations did not allow, such as use certificate-based authentication instead of OAuth2 based tokens used on KeyRock[23], which is the default identity management GE implementation.

To our knowledge there is no privacy-preserving GE on the FIWARE platform for location data, requiring that the client that sends the information to preserve its own privacy by controlling the data that it sends.

In *Location Privacy in Smart Cities Era*[20] authors state that both SMARTIE, SSServProv and FIWARE smart city architectures overlook location data properties like correlation between multiple and continuous location releases. Correlation between locations can be used to define paths that a specific citizen takes on the city, affecting citizen privacy. The same authors assert that in order for a smart city architecture to preserve location privacy several issues must be addressed, here we condense those issues into the following:

- **Adaptable Location Accuracy** Citizen needs are dynamic and due to that the location accuracy must be able to be changed to cope with the citizen needs;
- **Controlled Sharing** The location data must be only accessible by a controlled number of services and users;

- **Location Correlation** The privacy loss due to correlation of location data must be mitigated in the smart city architecture

In a effort to search a way to satisfy all the requirements it lead us in the search of related work that would enable us to preserve location privacy, even on correlated data, and to enable the storage of the location data captured by the IoT sensor devices to the cloud.

We decided to use FIWARE as the core of our smart city architecture, due to its modularity, re-using components like authorization providers, context-brokers while adding other components such as cloud storage and anonymization providers that lack on the FIWARE architecture or are insufficient for data privacy and scalability of, both, the IoT sensor network and the smart city infrastructure.

3.2 Anonymization of personal identifiable information

Anonymization is a technique that enterprises and medical institutions can use to increase security of data by preventing access to private, personal identifiable information, such as names and addresses, even on cases of data leaks. This techniques can also be used to reduce the stress on local computational resources, such as storage capacity, by offloading the anonymized data into public cloud storage solutions like Microsoft Azure or Amazon's Amazon Web Services (AWS). Data anonymization can be used as a key piece on organizations data processing pipelines since it prevents the identification of individuals and eases the conformity with data protection laws and regulations (such as General Data Protection Regulation (GDPR)). From our research of the state of the art, we come to the conclusion that anonymization techniques and protocols can be divided into two categories presented bellow.

3.2.1 Syntactic Anonymization

Syntactic anonymization protocols and techniques allows the anonymization of data entries on a data-set. Each data entry is altered in a way that it stays hidden on the middle of other similar data entries, mitigating identity disclosure (linking a individual to a data record) and attribute disclosure (revelation of some information about an individual based on some background knowledge that an attacker has of that individual).

3.2.1.1 K-anonymity

K-anonymity[12], created by L.Sweeney, is a formal model of privacy which goal is to make each record indistinguishable from a defined k number of other records. This way an attacker cannot easily identify the individual represented by the record since there are K-1 other records with similar information. In a k-anonymity setting the quasi-identifiers are altered in a way that enables at least K data entries with the same altered quasi-identifiers.

A data set is considered K-anonymized when, for any data record with a given set of identifiable attributes there are at least K-1 other records with the same attributes. K should be defined in a way that increases the anonymization while keeping the data set as smallest as possible because, in order to make the data set K-anonymized, sometimes fake data records must be added. This is due to the fact that in real-life data-sets there can be fewer records than those needed to perform K-anonymity or because some of the data records on the data-set cannot be easily generalized and grouped with other records, needing additional fake records to obey to the K-anonymization requirement[24]

Name	Zip	Age	Disease
Alfred	4500-343	34	Cancer
Bruce	5400-433	45	Cancer
Gus	4000-100	20	Heart Disease
Peter	5341-110	18	Viral Infection
Vince	5000-100	14	Respiratory Illness
Lara	5000-512	50	Heart Disease
Sherlock	4500-220	30	Viral Infection

Zip	Age	Disease
450*	3*	Cancer
540*	4*	Cancer
400*	2*	Heart Disease
534*	1*	Viral Infection
500*	1*	Respiratory Illness
500*	5*	Heart Disease
450*	3*	Viral Infection
540*	4*	Cancer
400*	2*	Cancer
534*	1*	Viral Infection
500*	1*	Cancer
500*	5*	Heart Disease

TABLE 3.1: Example of data anonymization K-anonymity with K=2, the table at the left is the original data and the table at the right is the k-anonymized data

Table 3.1 shows the anonymization of an example data set using k-anonymization with K=2. The name attribute is considered a key attribute and because of that was removed. Zip and Age are quasi-identifiers and were generalized. To demonstrate the need of adding fake data records we made the Zip keep its first digit. That quasi-identifier generalization forced the addition of fake data records to obey k-anonymity requirement.

With the addition of fake data records, tailored to the data set needs, the data set became 2-anonymized (K-anonymized with K=2).

3.2.1.2 L-diversity

Although K-anonymity improves the anonymity of a data set, by preventing identity disclosure it is insufficient to prevent attribute disclosure being vulnerable to a number of attacks, such as Homogeneity and Background Knowledge attacks. Homogeneity attacks target the lack of diversity on sensitive data upon data records with the same quasi-identifiers, meaning that if all the sensitive data present in all the data records with the same quasi-identifiers is the same, it is possible to know the sensitive data of an individual only by knowing a subset of its quasi-identifiers (i.e. If we know that Bruce is between 40 and 49 years old, we can say that he has cancer).

Background Knowledge attacks can, through usage of outside knowledge, associate sensitive values to individuals with some certainty of success (i.e. If we know that Gus is 20 years old and cases of cancer at this age are very rare we can say, with high probability, that Gus has Heart Disease)

L-diversity is another privacy model, built to work together with K-anonymity that can counter both of these attacks [25], increasing the anonymization of the data set. A data set is considered L-diverse if for each collection of data records with the same suppressed quasi-identifiers there are, at least, L well-defined sensitive values. Machanavajjhala et al. [25] gave a number of interpretations of the term "well-defined" in L-diversity such as **Distinct L-diversity**, **Entropy L-diversity** and **Recursive (c,l)-diversity**. In the remainder of this section we aim to focus on distinct L-diversity since it is the simplest to explain and covers all of the properties of L-Diversity.

A data-set is L-diverse using **Distinct L-Diversity** when every equivalence class (set of data records with the same anonymized quasi-identifiers), has at least L distinct sensitive values represented on them.

L, similar to K, should be defined with a value that increases the anonymization while keeping the data set size on the smallest possible because sometimes the addition of fake data records is needed to make the data set L-diverse.

Table 3.2 shows the result of applying L-Diversity with L=2 on the K-anonymization result of Table 3.1. The L-diverse data set is resistant to the homogeneity and background

Zip	Age	Disease
450*	3*	Cancer
540*	4*	Cancer
400*	2*	Heart Disease
534*	1*	Viral Infection
500*	1*	Respiratory Illness
500*	5*	Heart Disease
450*	3*	Viral Infection
540*	4*	Cancer
400*	2*	Cancer
534*	1*	Viral Infection
500*	1*	Cancer
500*	5*	Heart Disease
400*	2*	Viral Infection
540*	4*	Hear Disease
534*	1*	Heart Disease
500*	5*	Cancer

TABLE 3.2: Example of data anonymization L-diversity with L=2 of the K-anonymized data set from Table 3.1

knowledge attacks since there is no quasi-identifier combination whose data records contain the same sensitive information and the added fake data records were thought to mitigate identification via background knowledge.

3.2.1.3 t-closeness

l-Diversity, although mitigating attribute disclosure is not strong enough, being vulnerable to attacks such as skewness attacks and similarity attacks [26], that attempt to get additional information about the individuals represented on the data-set, possibly linking identities to sensitive attributes.

The skewness attack aims to compare the frequency of the sensitive information within a equivalence class (set of data records with the same k-anonymized quasi-identifiers) and the frequency on the globality of the data-set or real-world sensitive value frequency (i.e. the frequency of lung cancer cases in a elderly population). If the frequencies are largely distinct there might be a loss in privacy, meaning that an attacker can get additional information about the individuals represented on the equivalence class. The skewness attack is possible because l-diversity does not care about the global distribution of sensitive values, only the amount of sensitive values inside a equivalence class.

Similarity attacks tries to evaluate the similarities between the sensitive values inside a equivalence class, extracting private information from them. Let us consider a example equivalence class set of sensitive attributes $\{lungcancer, breastcancer, livercancer\}$, an attacker can say that every individual represented on the equivalence class as cancer. If the attacker previously had knowledge that a specific individual was on the data-set and that is member of the equivalence class it can say that the individual has cancer, breaking privacy.

t-closeness [27] is a novel privacy notion that replaces l-diversity and aims to mitigate skewness and similarity attacks, by requiring that the distribution of a sensitive attribute in any equivalence class is close to the distribution of the attribute in the overall table. In a t-closeness data-set the distance between the distribution of the sensitive attribute in a equivalence class and the distribution of the attribute on the table must be less than a threshold t. The skewness attack is no longer possible because the frequencies of the sensitive values on the equivalence class match the frequencies available on the data-set, minus t, meaning that an attacker can no longer get any additional information about the individuals sensitive attributes. Since the sensitive values frequencies on the equivalence class match the frequencies on the data-set, similarity attacks, that take advantage on the similarities between sensitive attributes are also largely mitigated.

Zip Code	Age	Disease
47677	29	gastric ulcer
47602	22	gastritis
47678	27	stomach cancer
47905	43	gastritis
47909	52	flu
47906	47	bronchitis
47605	30	bronchitis
47673	36	pneumonia
47607	32	stomach cancer

TABLE 3.3: Original data-set

Zip Code	Age	Disease
4767*	<40	gastric ulcer
4767*	<40	stomach cancer
4767*	<40	pneumonia
4790*	>40	gastritis
4790*	>40	flu
4790*	>40	bronchitis
4760*	<40	gastritis
4760*	<40	bronchitis
4760*	<40	stomach cancer

TABLE 3.4: Data-set K-anonymized and t-close when related to the Disease attribute. K=3 and t=0.278

Tables 3.3 and 3.4 aim to demonstrate the process of making a data-set t-close. The distance between the frequencies on the data-set and the equivalence class is calculated using Earth Mover Distance (EMD)[28], with the methodology of chapter 5 of [27]. First a data-set is k-anonymized, in the example K=3 was chosen, with the special attention to generate equivalence classes with the distribution of sensitive values t-close, in the example was t=0.278. As it is possible to see on the table 3.4 similarity attacks and skewness attacks are severely mitigated since the frequencies of the sensitive values on the equivalence classes are in line to the frequencies of the globality of the data-set when using EMD as a measure.

3.2.2 Semantic Anonymization

Semantic anonymization protocols and techniques usually sit as an intermediary on the communication between a client and a data set. The data set content is not altered in any way, only the query result is altered to protect the privacy of the individuals represented on the data-set.

3.2.2.1 ϵ-Differential Privacy

In Differential Privacy, there are three components: A data-set, a curator and a analyst. The data-set contains all the private information in a clear-text form. The curator has access to the complete data-set and answers queries performed by the analyst using the information on the data-set. Finally, the analyst is a party that makes queries to receive some information of the data on the data-set.

ϵ - differential privacy [29] is a concept that measures the amount of change a algorithm K introduces when the data-set is updated with some entry and is proposed in the following way:

$$Pr[K(D_1) \in S] \leq e^{\epsilon} \times Pr[K(D_2) \in S]$$

Where D_1 and D_2 are data-sets that at most differ at one entry, S is a set of all possible results, ϵ is the differential privacy value and K is a randomized function. This expression means that the probability of getting S when using D_1 divided by the probability of getting S when using D_2 is lesser or equal than e^{ϵ}. ϵ is the privacy loss factor and its value indicates the probability of the data entry that is in D_1 but not in D_2 to get identified by an analyst.

One problem with ϵ-differential privacy is that privacy loss(ϵ) is cumulative [30], meaning that the privacy of a data entry decreases with the number of queries made. To fix this, the curators enforce a maximum privacy loss that the query should not exceed before being blocked. This fix limits the utility of a data-set.

3.2.2.2 Diffix

Diffix[31] is a query anonymization protocol based on differential privacy but without the cumulative privacy loss. Diffix acts like a proxy between the analyst and the clear text data-set, adding minimal noise to the query result and without limiting the amount of queries that an analyst can make.

Diffix is resistant to intersection attacks, that aim to make multiple requests and intersect the responses to obtain some private information, and also resistant to averaging attacks, that aim to remove the noise introduced by the curators to anonymize the queries, by making multiple queries and averaging the results. Diffix is resistant to intersection attacks because it adds noise to the query results, that is dependent of the query itself and the query response, making the intersection of multiple queries useless. Diffix is resistant to the averaging attacks because it adds noise that is dependent of the query and query result, meaning that similar queries will receive the same noise value, making averaging attacks useless.

3.2.3 Comparison

In this section we aim to to compare the presented anonymization solutions in terms of vulnerabilities. Table 3.5 aims to show that comparison.

	Background Knowledge Attack Vulnerability	Homogeinity Attack Vulnerability	Skewness Attack Vulnerability	Similarity Attack Vulnerability	Quasi-Identifier Fallacy Vulnerability	Averaging Attack Vulnerability
K-Anonymity	✓	✓	✓	✓	✓	
K-anonymity + L-Diversity			✓	✓	✓	
K-anonymity + t-closeness					✓	
Differential Privacy						✓
Diffix						

TABLE 3.5: Comparison of the anonymization techniques

As is possible to see in table 3.5 K-anonymity by itself is vulnerable to most of the attacks presented on the comparison table, that happens because k-anonymity focuses on protect identity disclosure on the context of the current data-set, k-anonymity has no attribute disclosure protection justifying the background knowledge, homogeneity, skewness and similarity attacks vulnerability. K-anonymity based approaches to the data-set anonymization despite mitigating the majority of the attacks presented are vulnerable to the quasi-identifier fallacy[32] meaning that K-anonymity assumes that an attacker does not know any sensitive attribute about a individual, but in the real world and in a data-set containing multiple sensitive attributes is possible for an attacker to know some sensitive attributes that can be used as a key or quasi-identifier attribute to achieve identity disclosure. Semantic anonymization techniques, like differential privacy and Diffix, treats all of the information that can be learned from a data-set as private information, this way not being vulnerable to the quasi-identifier fallacy. The anonymization is achieved by introducing random noise into the data-set responses. However, the usage of completely random noise can make the responses susceptible to averaging attacks that aim to perform large quantities of similar queries and averaging the results to obtain the de-noised result.

From our analysis, Diffix is the best anonymization solution, being resilient to all of the attacks described, however there is no free or open-source implementation available at the time of writing, making us unable to use it on a architecture and implementation level.

3.2.4 Special Anonymization Case: Location Data

With the growing popularity of location-based systems, allowing unknown/untrusted servers to easily collect huge amounts of information regarding users' location, has recently started to raise serious privacy concerns. One mitigation to the loss of privacy is to implement syntactic anonymization mechanisms such as K-anonymization, that attempt to suppress the location value in order to establish equivalence classes with at least K individuals, or semantic anonymization mechanisms such as differential privacy that attempts to introduce noise into the data in a way that the result of a query to the data-set when it contains or not contain the data should have similar results. Syntactic anonymization techniques are always prone to the quasi-identifier fallacy vulnerability, since it makes assumptions about the attacker knowledge of an individual. However, in the real world and in a data-set containing multiple sensitive attributes it is possible for an attacker to know some sensitive attributes that can be used as a key or quasi-identifier attribute to achieve identity disclosure.

Differential-privacy does not make any assumption about the attacker knowledge, being resistant to the majority of the attacks presented on this related work. The default behaviour of differential-privacy does not provide resistance to data correlation, especially on cases when the same or similar requests are made the randomness of the noise allows an attacker to perform an averaging attack that attempts to get an approximation to the real value by averaging multiple request results, effectively removing the random noise applied. This is due to the fact that there will be high random values and low random values that cancel each other when averaged. However, there are differential-privacy based solutions that mitigate this issue by associating the noise value to the request or result instead of a truly random value, mitigating the averaging attack vulnerability.

Standard differential-privacy techniques can be successfully applied in cases where aggregate information about several users is published, making it poorly suitable on scenarios where a single user sends location data to a location-based service. Standard differential-privacy would require that any update to a single user location data should have negligible effect on the published output, making impossible to communicate any useful information to location-based services. To cope with the challenges of location data anonymization, specialized anonymization techniques were formulated to help to mitigate privacy concerns over location data sharing.

3.2.4.1 Geo-Indistinguishability

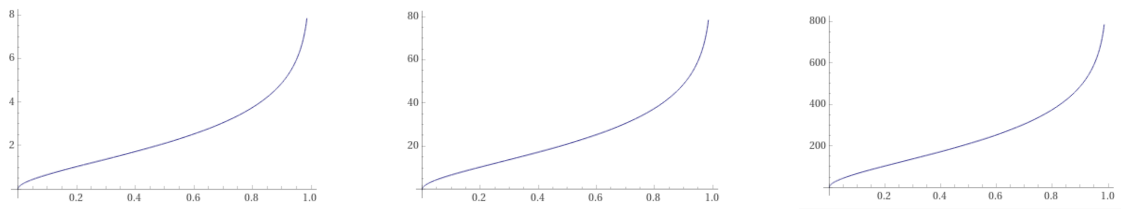
Geo-Indistinguishability[9], is a formal notion of privacy for location-based systems that protects the exact location of a user within a radius r with a privacy level l corresponding to a generalized version of the concept of ϵ-differential privacy. The use of the differential privacy concepts means that geo-indistinguishability privacy holds to any possible side-information that an adversary might possess.

The authors state that a mechanism satisfies ϵ-geo-indistinguishability if and only if for any radius $r > 0$ defined by the user, it enjoys *l-privacy within r*, on which $l = r\epsilon$. Meaning that the privacy level increases with the radius enabling users to provide a more meaningful privacy setting. The privacy level is associated with the ”level of similarity” of the anonymization result of two locations within a radius of r, on which smaller values for the privacy level provide higher privacy. *l-privacy within r* means that given two points x and x' with a distance lower than r, the set of possible user locations X,the possible reported values Z and a mechanism K that converts point from X into a probability distribution over Z system enjoys *l-privacy within r* if and only if $d_p(K(x), K(x')) \leq \frac{l}{r} d(x, x')$.

d_p being the multiplicative distance between two distributions and d the euclidean distance. Simplifying, two distinct points within a radius of r are indistinguishable from one another, if the multiplicative distance of probability distributions of the mechanism that generates the obfuscation points for both points is lower or equal than the real distance between the points more or less the privacy budget. This in turn means that two close locations are indistinguishable by the eyes on an attacker.

The authors of [9] also describe a mechanism based on planar laplace for achieving ϵ-geo-indistinguishability. The mechanism consists of adding 2-dimensional laplacian noise centered at the real user location x. To obtain a point the mechanism generates a vector by selecting two random values $\theta \in [0, 2\pi[$ that will be the angle and $p \in [0, 1[$. p will be placed on the inverse planar cumulative distribution function which is defined as: $C^{-1}(p) = -\frac{1}{\epsilon}(W_{-1}(\frac{p-1}{e}) + 1)$, where W_{-1} is the Lambert W negative branch function and $\epsilon = \frac{l}{r}$. The result of applying p to $C^{-1}(p)$ is the length of the vector. Having the vector length and angle the anonymized location is calculated by translating the real location using the vector defined by the angle and distance from x.

Figure 3.3 shows multiple plots of the $C^{-1}(p)$ with varying ϵ values to show how the possible values of vector length increase with the decrease on the ϵ value.



(A) $\epsilon = 0.8$ (B) $\epsilon = 0.08$ (C) $\epsilon = 0.008$

FIGURE 3.3: Different plots of the $C^{-1}(p)$ function with different ϵ values shows the increase of vector lengths with the decrease of epsilon, smaller the epsilon bigger the privacy

As it is possible to see the values of $C^{-1}(p)$ increase exponentially when the p is close to 1, causing the generation of obfuscated locations that are too far away from an expected length close to the radius. This can be easily shown on figure 3.3a, where the set of p values $[0.0, 0.9]$ comprise 90% of the possible values of p but only comprise output values from 0 to 6. The remainder 10% of the possible values of p go from 6 up to values such as

20 (p =0.999999) which produce obfuscated locations further away than expected, reducing the utility of the anonymization result. In order to control such behaviour one might say that for, sake of utility, the possible outputs of $C^{-1}(p)$ could be truncated to max out at the radius value, however [33] states that by doing so the privacy guarantees of Geo-Indistinguishability no longer hold if some truncation is built to ensure a minimum utility for the users.

The Geo-Indistinguishability mechanism suffers from a vulnerability, which is also present on differential-privacy, that is the application of the averaging attacks on correlated data. Cases where the same real location is used on multiple consecutive location obfuscations exemplify the best this type of vulnerability.

By using the obfuscated location releases an attacker will try to guess the real location point by averaging the noise that originated the obfuscated points.

3.2.4.2 Clustered Geo-Indistinguishability

Mendes et. al in *Impact of Frequency of Location Reports on the Privacy Level of Geo-Indistinguishability* [34] concludes, through the analysis of real data-sets, that the frequency of location reports directly impacts the correlation between reports which in turn can be used by an adversary to average the location reports to find the real location and cause a privacy loss bigger than what was expected from Geo-Indistinguishability. The correlation is bigger on cases where the the same location is being captured by the device and to each report a new noise is applied making the system prone to the same attacks that ϵ-differential privacy is prone to.

In a effort to mitigate the privacy loss due to correlation between updates Cunha et. al [10] presents Clustered Geo-Indistinguishability as a correlation mitigation. Clustered Geo-Indistinguishability works by sending the same noisy location report when the user is within a radius s from the first location report. But, when the user leaves that radius a new noisy location report is calculated and the center of the circle with radius s is shifted to the real location that originated the location report. 3.1 presents the pseudo-code algorithm for clustered Geo-Indistinguishability. In this algorithm x is the real location, e is the epsilon value of geo-indistinguishability, s is the cluster radius and z is the anonymization result. The algorithm returns the previousZ when the real location is still within the radius s, when it abandons it a new location report is generated and returned.

```
function CLUSTERING(x,e,s)
    if first_report then
```

```
        c = x
        z = planarLaplace(x,e)
        previousZ = z
    else
        distance = euclideanDistance(c,x)
        if distance < s then
            z = previousZ
        else
            c = x
            z =  planarLaplace(x,e)
            previousZ = z
    return z
```

LISTING 3.1: Clustered Geo-Indistinguishability algorithm

The authors compared the Clustered Geo-Indistinguishability with the Adaptative Geo-Indistinguishability[35] and with the traditional Geo-Indistinguishability presented described before using a state-of-the-art Map-Matching technique [36] and real mobility data-set with the objective of finding a path based on the location reports. If a path is found using the anonymized reports that corresponds to the real path on the data-set the location privacy was lost. The authors, to attribute a quantitative value to the Map-Matching based attack results, used the F_1 anonymization score[37]. F_1 score is calculated in the following way: $F_1 = 2 * \frac{precision*recall}{precison+recall}$ where $precision = \frac{Lcorrect}{Lmatched}$ and $recall = \frac{Lcorrect}{Ltruth}$. *Lmatched* is the length of the path that an attacker, using map-matching over the reported location, found, *Ltruth* is the length of the real path and *Lcorrect* is the size of the portions of the real path that match with the path found by the attacker. F1 score values varies between 0 (worst path match) and 1 (best path match), where the lower the score the better was the privacy protected.

The authors concluded that techniques like Clustered Geo-Indistinguishability and Adaptative Geo-Indistinguishability indeed provide mitigations to the linear privacy loss of the simple Planar Laplace method [9] on which n similar location reports cause a privacy loss of $n\epsilon$ instead of just ϵ. Based on the authors results, Clustered Geo-Indistinguishability provides a better privacy-utility trade-off than Adaptative Geo-Indistinguishability while also removing the dependency between privacy and the location report frequency since the real location report frequency is masked by the reporting the same location while within a circle with radius s, centered on a previous real location. Figure 3.4 aims to provide visual representation of Clustered Geo-Indistinguishability process.

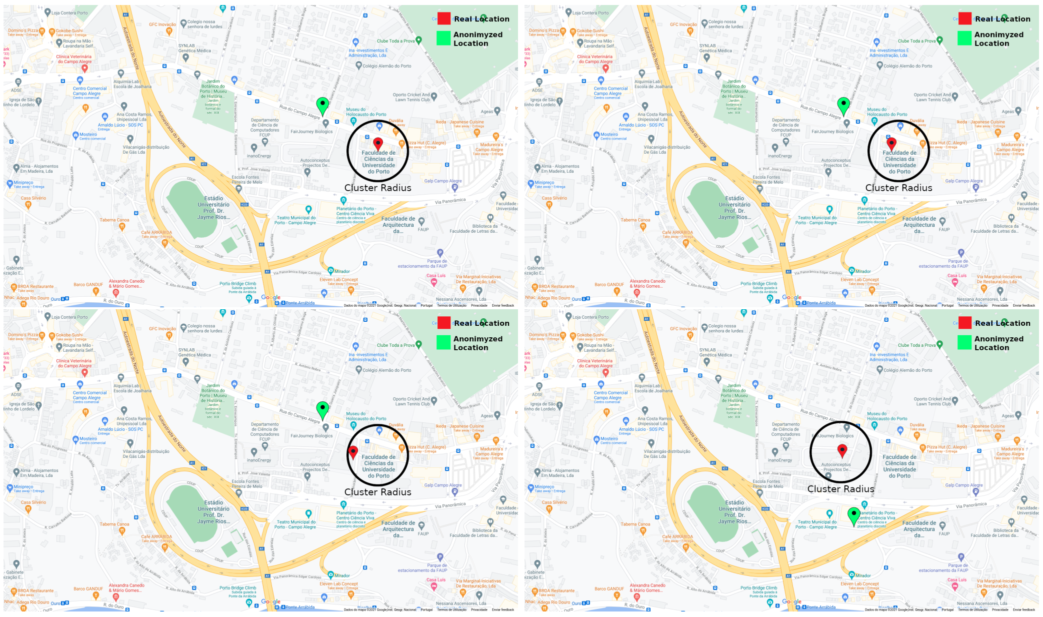


FIGURE 3.4: Example of how the anonymized location report, cluster radius and real location interact to produce Clustered Geo-Indistinguishability. From left to right, top to bottom: the real location is moving but while remains inside the cluster radius the reported location does not change. However when the real location leaves the cluster radius a new reported location is generated and new cluster radius, centered at the real location is formed

3.3 Data Availability and Persistence

Data persistence and availability techniques are one of the core components of a storage system since without these techniques storage would become unreliable and could compromise critical information. Nowadays, there are 3 main ways that systems provide storage and in this section we will discuss them.

3.3.1 Standalone

In standalone systems there is no controller server or proxy relay, the communication is done directly from the client device into the cloud storage. This type of system puts the responsibility of performing all the security and privacy measures on the client. Systems like these are vulnerable to loss of information and usually lack capabilities such as secure file sharing. Standalone systems can also use cloud-of-clouds to increase availability of the stored data by having multiple copies of the same data or erasure coded data stored on different cloud backends.

3.3.1.1 rClone

rClone[38] is a command line application that enables the management of files on cloud storage. rClone enables the configuration of multiple remote storage backends, allowing a user to easily upload, download, list and delete any kind of file or folder. rClone also allows local encryption of files before sending them to the remote cloud storage.

rClone does not provide automatic erasure coding between multiple cloud storage backends, creating a risk to the availability of the data stored on the remote servers.

3.3.1.2 CHARON

CHARON[39] is a FUSE virtual hard drive system with three distinct features: Does not require trust on a single entity (like a proxy server), Does not require any server beyond the cloud storage servers and lastly, has efficient large file support by splitting the files in smaller chunks before upload. CHARON allows single cloud or multiple cloud storage increasing redundancy and availability. CHARON stores file metadata in the same storage as the file itself, removing the need for coordination services such as DepSpace[40] while also providing features such as controlled sharing like in SCFS[41].

3.3.2 Hybrid/Proxy-based

Hybrid or Proxy-based systems use a central server that receives the information from the clients and stores them on the cloud systems indirectly. By having a centralized service all the responsibility of performing all the security and privacy measures can be put on the server, instead than in the client like in standalone systems. This type of systems can allow secure file sharing between clients that send information to the same server and can implement caching mechanisms to improve file download speeds. One major problem with this kind of systems is that they can introduce the central server as a single point of failure, placing the availability of a hybrid storage solution at risk.

3.3.2.1 ARGUS

ARGUS[6] is a broker that acts as a proxy to the existing public cloud infrastructure, having at least three cloud storage services connected in a cloud-of-clouds configuration. In ARGUS the broker is responsible to manage the redundancy and the cloud providers storage, where the data sits in an encrypted form. The encryption itself can be performed on the client-side or on the server-side, meaning that if the user does not want to send clear-text information to the broker server it can perform the encryption locally and the broker only creates the redundancy and stores of the information on the cloud. The broker also implements verifications to check if the data uploaded to a cloud provider was changed during the download using Hash-based Message Authentication Code (HMAC)'s. Implementations of the broker server also provides identity management via Keycloak[42], which is a open source identity and access management platform, providing authentication trough the usage of OAuth2 tokens. This identity management platform allows to register users into the platform. For each user a Rivest-Shamir-Adleman (RSA) public/private key pair is generated on the server or on client side, depending on the client-side or server-side encryption chosen. This public and private key pairs are used to encrypt 256 bits Advanced Encryption Standard (AES) keys,used for file encryption, that are stored on the broker premises, encryption using public key only allows users with the private key associated with the public key to decipher the information, this way blocking the access to encryption keys that do not belong to the user. A new AES key and Initialization Vector (IV) are generated for each file encrypted, meaning that if one file encryption key is compromised the remaining files are still secure. Sharing is also implemented in ARGUS through the usage of a combination of symmetric and asymmetric cryptography, where the file is deciphered and re-encrypted using a new symmetric key, that symmetric key is encrypted using the public keys of the users that the file is going to be shared with. The encrypted key is then sent to each one of the users that the file will be shared with.

3.3.2.2 BlueSky

BlueSky[43] is a network file system with integration with Network File System (NFS) and Common Internet File System (CIFS) protocols using a cloud storage backend. The Bluesky server is deployed on-site and has caching mechanisms to improve performance by reducing the amount of communication to the cloud storage. BlueSky only supports one cloud backend to be configured at a given time, meaning that there is no possibility of cloud-of-clouds redundancy making the system more susceptible loss of data availability, compromising the usage of this storage system.

3.3.3 Distributed

Distributed systems improve the single point of failure problem of the hybrid approach by making use of multiple server instances, storage nodes, or fault tolerant coordination services, providing all the necessary infrastructure to allow secure file storage with high availability and redundancy. Although powerful, distributed systems often require coordination between the servers or nodes becoming a problem that multiple implementations try to solve differently.

3.3.3.1 SCFS

SCFS[41] is a cloud storage system that provides a FUSE like virtual hard drive being able to work in single cloud environments or in cloud-of-clouds environments. It uses erasure coding to enable data redundancy, and encryption of enable data confidentiality. SCFS uses a Bizantyne-Fault Tolerant distributed coordination service, such as DepSpace[40], enabling high system availability, being the coordination service job to store metadata that enables storage and retrieval of information from the cloud-of-clouds storage backend.

SCFS through the usage of the coordination services allows controlled file sharing without relying in a proxy server like in ARGUS[6], reducing the single point of failure and improving the availability of the system.

3.3.3.2 Kademlia

Kademlia [44] is a Distributed Hash Table (DHT) system that uses XOR metric to enable distribution of content across multiple storage banks, allowing redundancy and high availability of the stored data. Kademlia implementations include software suites such as Emule/Kad, Overnet and Azureus, currently used by millions of users everyday[45]. Kademlia was firstly designed to be implemented on decentralized peer to peer systems but can also be used as a storage manager for servers and data centers since it allows storage systems to scale with minimal configuration, down times and overheads. Kademlia works by storing $< key, value >$ pairs on nodes with closer node IDs. Both node ID and $< key, value >$ pair key are 160-bit values and the closeness between the key and the node ID is calculated using the XOR metric.

The XOR metric calculates the distance between two 160-bit values by doing XOR of both of them, the result shows the distance between the two values and is used to find the node closest to the key of the $< key, value >$ pair. With the notion of distance and to ensure that the system is dynamic, allowing addition and removal of nodes on demand, each node builds a routing table for itself. The routing table sorts the node ID's of the nodes that the current node is aware of, based on XOR metric. Allowing that the routing of data storage and retrieval queries can take the shortest path based on XOR metric.

A Kademlia routing table is constructed based on the node ID of the node that is creating it. First the routing table columns are created, each column represents a zone on the 160-bit space. On each of these zones the node must have at least one contact in order to be capable to reach all the 160-bit key space, therefore being capable to get and store $< key, value >$ pairs for every possible 160-bit key.

Table 3.6 shows one example of constructing a routing table for a node with id = 0011 inside a smaller 4-bit id space. A 4-bit id space was used to make the example smaller, yet understandable. 4 bit identifiers created on this step represent the zones where the node must, at least, know one node of that zone to have coverage of the entire key space. Figure 3.5 attempts to explain how the identifiers are built.

0011 Routing Table	k0	k1	...	kn
1000				
0100				
0000				
0010				

TABLE 3.6: Example of constructing Kademlia routing table on a node with id 0011 on a 4-bit id space

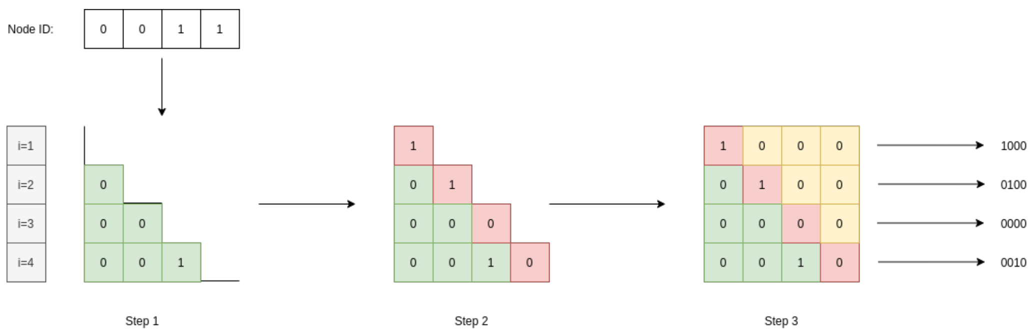


FIGURE 3.5: Example of constructing the routing table column identifiers for the routing table of a node with id 0011 on a 4-bit key space

As is possible to see in figure 3.5, first we translate the first i-1 bits into the columns, then the i'th bit is switched to the opposite (0 becomes 1 and vice versa). Lastly the remainder of the rows are filled with zeros in order to have 4-bit identifiers. Having these identifiers we can now start to populate the routing table inserting received node information on a row that has a identifier with the lowest XOR distance to the received node id when compared with the XOR distance of the remainder identifiers. The maximum number of entries on a routing table column is defined by a global value K, being a routing table column also called K-bucket. K is usually defined as 20.

In addition to routing tables and XOR metric, kademlia also provides 4 algorithms: Ping, Store, Find Node and Find Value.

Ping is used to check if some node is up and ready to receive requests. Ping receives a node ID and checks the routing table to verify if the node is known to the current node. If the node ID exists on the routing table a ping query is issued. The node might not answer to the query request, this signals that the node is not running or not ready to receive requests, if the node answers then it signals that the node is running and ready to receive requests.

Store is used to store a $< key, value >$ pair of data into the storage system in a way that enables redundancy. Using the key, the node checks its routing table and gets the node information of all the nodes on the routing that belong to the same K-Bucket as the key of the $< key, value >$ pair to be stored. This means that store will search the routing table for the nodes closest to the key of the $< key, value >$ pair, XOR metric wise. If the K-Bucket is not full, store will get the remainder node information from other K-buckets until it gathers K nodes or has searched all the routing table information. Having the nodes from the routing table it sends a store request to them, the nodes that receive the request will store the $< key, value >$ pair locally and return a confirmation.

Find Node it is used to find nodes on the network and populate the routing table. It receives a node ID (Q), then using that node id it checks the local routing table and gets α closest nodes that are in the routing table (in a similar way to what happens in **Store**). Having those α nodes it checks if they are available using a **Ping** command, and for all the α nodes that are available a find node request is issued. The return of a find node request is a set of the K-closest nodes to the Q present on the routing table of the node that received the request. When the node receives all the responses it compiles them into one set, storing it as B, then select α nodes from B that were not previously used and repeat the process. Find node ends when the B from the requests using the current α nodes equals the B from the requests made with the previous α nodes. It is important to say that find node should be run regularly in order to have the routing table updated with all the changes that might occur on the nodes of the network. α is called a system-wide concurrency parameter in [44] and it is a value, usually 3, is used to bound the number of selected nodes in each round of find node. In Find Node, to increase performance, the procedure spawns α threads, each one responsible to query one of the α nodes chosen.

Find Value it is used to get the value associated to a key on the Kademlia network. It works similarly as the find node, but if a node that receives the find value request has the value associated to the key, it returns the value instead of returning K-closest nodes to key on the request. When the node receives, as a response, the value associated to the key find value is stopped and the value is returned.

One important thing to mention is when a node receives any query or response, it updates the routing table, adding the node id of the node that sent the response/query on the respective K-Bucket. If the K-Bucket is full, first it pings all the nodes on the K-Bucket, and if some nodes does not answer that node is removed and the new node entry is added. If all the nodes respond to the ping query the routing table update is discarded.

Kademlia creates replicas of the $< key, value >$ pair by default, since every $< key, value >$ pair is stored on at least K nodes, and only one of those K nodes must be up and running for the $< key, value >$ retrieval to occur. Since the original node that uploaded the pair keeps re-uploading it from time to time, even if all the initial K nodes disconnect, there will be other nodes where the $< key, value >$ is ready to be retrieved.

Kademlia, although providing a fast and reliable method to store information in a high availability manner, it does not have any mechanism of ensuring deletion or updates on the data across all the replicas stored on the nodes because nodes that stored the information can become offline when the deletion or update query is issued, causing inconsistency on the network.

3.3.3.3 Ceph

Ceph[46] is a distributed file-system which provides high performance, reliability and scalability. Ceph differs from other remote storage solutions, such as Network File System (NFS), since it replaces the traditional block-level storage interface with Object Storage Device (OSD)s. OSDs are intelligent devices, composed of a Central Processing Unit (CPU), memory and a persistent storage device (such as HDD or a SSD) attached to it. OSDs replace the traditional block-level storage interface with one in which clients can read or write larger, variable sized, named objects. The conversion between named objects and the block-level storage is performed by the OSD itself, this way reducing the need to set global and storage device specific instructions or limits, improving scalability and manageability.

Ceph differs from other distributed storage solutions, such as Kademlia[44], by ensuring consistency on data replicas stored on the distributed storage. It is achieved by using a deterministic placement algorithm called Controlled Replication Under Scalable Hashing (CRUSH)[47].

CRUSH is a pseudo-random, deterministic, algorithm that maps a input value, typically an object, to a list of devices on which to store/retrieve object replicas. In order to work properly and return the same results when executed on multiple machines with the same input CRUSH needs a hierarchical description of the OSDs comprising the storage cluster and knowledge of the replica placement policies defined for the OSD cluster.

The hierarchical description, also called cluster map, is composed of devices and buckets, both of which have numerical identifiers and weight values associated with them. Buckets can contain devices or other buckets, forming a type of tree, where the devices are tree leaves and the buckets are remaining nodes of the tree. To storage devices a weight is applied, the weight is used to control de relative amount of data that the device is responsible to store. Weight levels are usually related to the relative capacity of the device when compared to other devices on the cluster. The CRUSH algorithm takes into consideration the weight values on its output, reducing the usage of low capacity devices, increasing the usage of high capacity devices with the purpose that at the end all the devices on the cluster have the same relative load. The weights of the buckets are the sum of the weights child devices and buckets. Each bucket and device have a name and a type which are used for forward requests to them and to construct placement rules, that information is also stored on the cluster map.

Ceph architecture has three main components: The client, which exposes a near-Portable Operating System Interface (POSIX) file system to a host or process. A cluster of OSDs which collectively stores all data and metadata. And, lastly, a metadata server cluster which manages directory structure and other POSIX metadata, such as permissions, users, file locks, etc.

The cluster of OSDs runs on top of the Reliable Autonomic Distributed Object Store (RADOS)[48] protocol. RADOS seeks to leverage device intelligence to distribute the complexity surrounding consistent data access, redundant storage, failure detection and failure recovery in OSD clusters consisting of thousands of heterogeneous, highly-dynamic, devices while also providing client applications with the illusion of a single logical object store with strong consistency guarantees. It is the job of RADOS to build and update the cluster map, that is then replicated to all the storage devices and clients.

By providing storage nodes with complete knowledge of the distribution of data in the system, thanks to CRUSH and the cluster map, devices can act semi-autonomously using peer-to-peer protocols to self manage data replication, consistently and safely process updates, participate in failure detection and use the resulting changes to the cluster map to re-replicate or migrate data objects. This semi-autonomy that RADOS gives to the storage devices, reduces the stress on monitor devices, which have a unique function: listen for changes on the devices, update the cluster map accordingly and distribute the new version of the cluster map t across the cluster. In RADOS each map version is represented by an increasing epoch value, the bigger the value the most recent the cluster map is. This epoch value is then used by the clients and storage devices to check if the cluster map circulating is newer than the cluster map that they are currently using and updating the cluster map accordingly.

Given the dynamic nature of distributed storage solutions, OSDs can fail or be created at a regular basis, which results in lots of cluster maps. To reduce the amount of information to be sent to the remainder devices, the monitor nodes send incremental maps which state the change that current epoch cluster map caused on the previous cluster map, enabling each device to the incremental data to generate the new map without the need for it to be sent.

RADOS defines a two level hierarchical description of the cluster: Placement Group (PG) and failure domains. A PG is a logical collection of objects that are replicated by the same set of devices, on which, each object's PG is determined by a hash of the object name o, the desired level of replication r and a bit mask m that controls the number of PGs in the system. The PG that will be responsible for the object has an $pgid = (r, hash(o)\&m)$, where & is a bit-wise AND operation. Each PG contains a set of devices in different failure domain that will be responsible for the storage and replication of the PG objects. Figure 3.6 shows the interaction between the storage devices and the PGs.

The cluster map in RADOS also provides information about the liveness of each OSD, this is used to provide information about temporary downtime that some OSD might have suffered, redirecting all the traffic that was bound to that OSD to another available Object Storage Device.

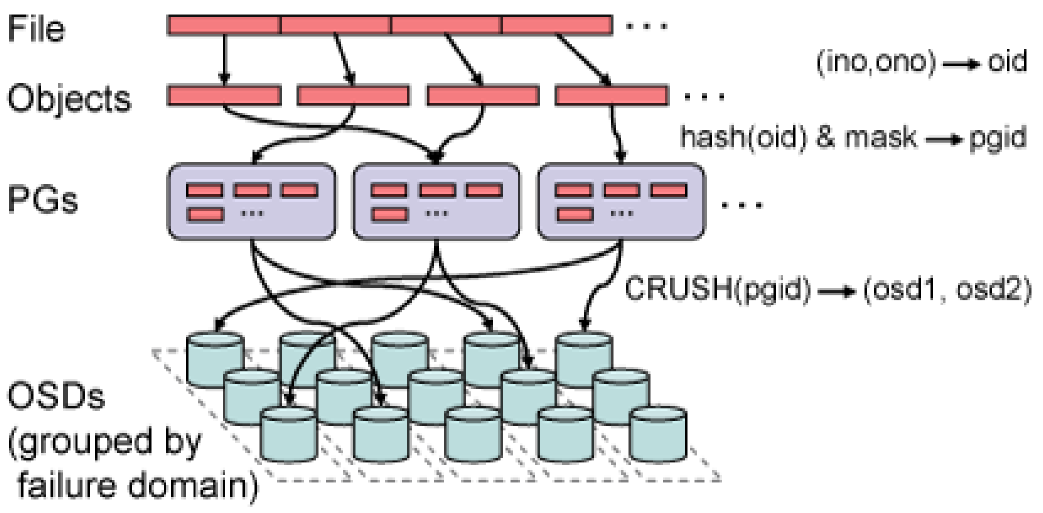


FIGURE 3.6: Interaction between Ceph OSDs and the Placement Groups (Extracted from [46])

The knowledge of the data distribution encapsulated in the cluster map allows RADOS to use the OSD processing capabilities to distribute the management of data replication, failure detection and failure recovery. RADOS implements n-way replication on which a client device send only a single write request to the first OSD result on the CRUSH algorithm. This first OSD, also called primary, will then replicate the received data across $n - 1$ OSD devices, according to the cluster map. In order to ensure consistency on the operations, every communicating device must have the same cluster map, otherwise the system will not work properly. Each message on RADOS contains the cluster map epoch of the sender. The receiver then checks the epoch to see if it matches the current epoch that the receiver knows of, if it does the message is processed otherwise two things might occur: If using the most recent cluster map and the object name the CRUSH algorithm result does not contain the receiver, it will send back to the sender the newer version of the cluster map, allowing it to do the request to the right OSD. Otherwise if using the most recent version of the cluster map and the object name on the CRUSH algorithm still marks the receiver as a valid OSD it will do the request but still appends to the response the new cluster map. If the sender has a epoch more recent than the epoch of the receiver the receiver will ask the monitor node for the most recent epoch, being then able to respond to the request.

RADOS employs an asynchronous, ordered, point-to-point, messaging library for communications using Transmission Control Protocol (TCP) protocol. A failure on a TCP socket leads to limited number of reconnect attempts before a failure being reported to the monitor node cluster. Storage nodes use peer-to-peer communications to send heartbeats to each node on the same PG to ensure that devices failures are detected, if some OSD is mistakenly marked down by their peers it simply syncs all the data to the disk and kills itself for the sake of consistency.

In RADOS, OSD addition, removal or failure, cause changes to the cluster map. A change to the cluster map, usually invokes data migration and failure recovery methods in order to keep the data replicated and to ensure consistency with the CRUSH algorithm. RADOS failure recovery and data migration mechanism can be subdivided into two parts: Peering and Recovery.

Peering is driven by the first OSD on the PG, also called primary OSD. For each PG the OSD stores locally information about its relation with each PG (ie. primary, replica or stray). Then devices send a notify message to the primary. The notify message includes the most recent PG updates, information about the changes that occur to the data stored between the last update and the current update, and the most recent epoch that the OSD knows of. Upon receiving the multiple notify messages, the primary generates a *prior set* which includes all the OSDs that may have participated on the PG since the last update. With all the information on the *prior set* the primary can build an updated PG log that will be forwarded to the active replicas. With the PG log updated all the devices on the PG are able to check what are the objects that they are missing from their storage and start the recovery process.

In RADOS the recovery process is coordinated by the primary. Since the primary has already knowledge of the objects that each replica is missing it preemptively gets the missing objects and sends them to the replicas. Simplifying the recovery logic, by reducing the number of reads that the recovery process needs to do. This method prevents issues related to not getting the same object from the same OSD at multiple read requests, since the data might by changing from one request to another. Since the primary is the one that read the missing data, and it only does it once, all the replicas on the PG will have the same version of the data, improving consistency.

Lastly, in order to store a master copy of the cluster map and to be able to make periodic changes to it in response to cluster configuration changes and OSD state changes RADOS uses monitor nodes.

In order to ensure availability multiple monitor nodes are generated, and all of them belong to a single monitor cluster. The monitor cluster is designed to favor consistency and durability by basing the communication between the multiple monitor nodes on the Paxos part-time parliament algorithm[49]. The cluster starts by electing a leader to serialize map updates and manage consistency. Once elected, the leader begins by requesting the map epochs stored by each monitor. The monitor nodes have a fixed amount of time to answer the request and join the *quorum*. After the *quorum* is built, a epoch synchronization process begins, ensuring that all the monitor nodes on the quorum have the same recent epoch. After the synchronization process the leader begins distributing short-term leases to active monitors.

Each lease grants active monitors permission to distribute copies of the cluster map to OSDs or clients that who request it. If the lease expires without any renewal the leader is assumed dead and a new election occurs. If a lease is not acknowledge by some active monitor the leader assumes it dead and rebuilds the *quorum*. This behaviour allows to have all the members on the *quorum* active and able to process requests.

When an active monitor receives an update request caused by some OSD state change or configuration update, the active monitor checks if the update is new and it is not a replay of a previous update by some other device on the storage cluster. If it is recent it notifies the leader, that interrupts all the leases, generates a new cluster map, increments the epoch and sends the update proposal to all the members of the *quorum*. If the majority accepts the update, the cluster map is sent to all the active monitors and the new leases are created, allowing for the update to be propagated to the rest of the storage cluster. This methodology ensures that no monitor node is sending previous versions of the cluster map, since it will not be leased to do it, and that all the map changes result on a consistent progression of map versions allowing the remainder devices on the cluster to know which cluster map is the most recent.

Metadata operations often make up as half of file system workloads [50], making the metadata server cluster critical to the overall system performance and availability. The metadata information itself is stored on the OSD cluster, like the object data. To split the load across the metadata server clusters Ceph makes use of Dynamic Subtree Partitioning[51] which partitions the file tree into smaller chunks and attributes the responsibility of that chunk to a metadata server. Since there are partitions that are more used than others by the clients the Dynamic Subtree Partitioning uses the notion of busy/hot directories and delegates management of that directory to a set of metadata server, in addition to the one responsible by the partition were the directory is placed on the file tree.

The metadata includes information about the file and directories, including name, owner, size, file locks and so on, making it very important to protect I/O operations when multiple concurrent users attempt to write or read the same file/directory.

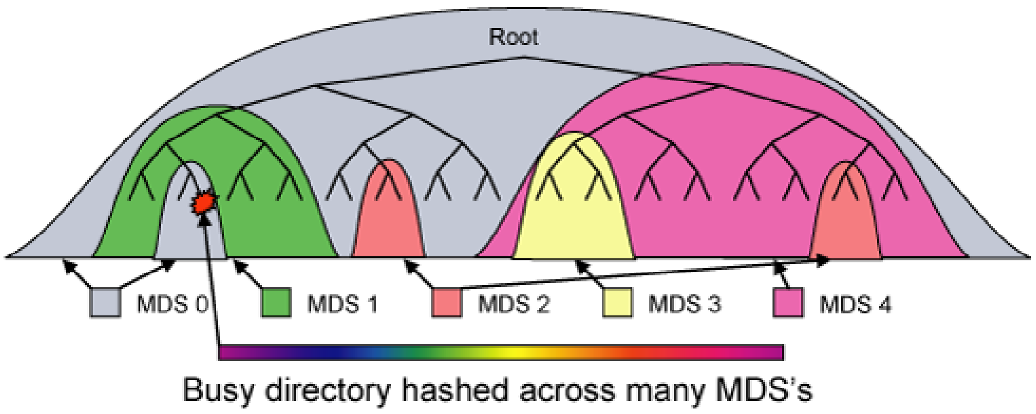


FIGURE 3.7: Dynamic distribution of the file tree across several metadata servers based on the current workload (Extracted from [46])

One of the advantages of Ceph is the capability of performing synchronous Input/Output (I/O) operations by making writes atomic, meaning that only one write to a specific file is allowed at a time, blocking any other operation (reading or writing) thus mitigating write-write conflicts. Reading on synchronous I/O also blocks any writing until the reading operation succeeds mitigating read-write conflicts. The atomicity of the operations is achieved through the usage of file locks, which are stored on the metadata servers. The clients that want to perform a request on a object that is locked must wait for the lock to open for it to do the operation.

The capability of self-healing from OSD failure or removal together with the ability to distribute the load on each OSD based on their capacity, OSD overload protection, the lack of the need to perform lookups to find the path where to obtain the data and the possibility of synchronous I/O allows Ceph to provide great availability,reliability and data consistency even on commodity hardware, reducing data center costs.

Unlike Kademlia, the removal and rectification of data are propagated directly to the respective OSDs on the cluster, due to the way that Controlled Replication Under Scalable Hashing (CRUSH) works the client devices can query data removal/updates directly on the OSDs that have that data, improving the system consistency and mitigating old data propagation.

3.3.4 Comparison

In this section we aim to compare the solutions described above in order to easily point the pros and cons of each one of the solutions when compared to other solutions. Table 3.7 illustrates that comparison. The upload and download speeds were achieved by averaging the results of uploading/downloading files of 100MB,500MB and 1GB 10 times for each file size without any compression or additional processing. rClone is configured to use a Microsoft Azure blob store as the remote. CHARON is configured to use 4 AWS blob storage containers without pre-fetching, without compression and with 12 processing threads, this removal of features was done to test the raw throughput of CHARON and to make it more comparable with other cloud storage solutions. ARGUS is configured to use 3 cloud containers (one AWS blob storage container , one Google's Google Cloud Provider (GCP) blob storage container and one Microsoft Azure blob storage container), using 10 upload and download threads. SCFS was configured to use 4 Amazon AWS blob storage containers, with sync-to-cloud and printer flags. Kademlia was configured based on our implementation with 3 AWS ac2 virtual machines as nodes with K=20 and α=3, this way making it more comparable with the remainder multi-cloud backend storage solutions which have 3/4 storage backends. Ceph was configured with 3 AWS ac2 virtual machines, each one running one OSD and one monitor node running in containers, the storage is done using erasure coding as the replication provider. Having 3 OSDs makes the system more comparable to the multi-clouds storage solutions which have 3/4 storage backends.

We tried to compile and execute the available implementation of BlueSky [52] but we faced several compile time and execution issues that made impossible to upload and download data.

As shown on table 3.7 rClone, achieve the best upload speed and one of the best download speeds, since rClone is a standalone system using a single cloud backend it has neither scalability nor redundancy built on the system, data availability is dependant on possible redundancies that the cloud provider might build into its storage. Since it is a serverless system it is always available to the user.

CHARON, like rClone is a standalone serverless system, but contrarily to rClone it performs data redundancy by splitting data across a set of cloud providers, this way ensuring data availability. CHARON being a serverless system is always available.

ARGUS is one of the systems with the best performance of all the systems we tested, it has no server scalability available but performs data redundancy and splits the redundancy result across a set of cloud providers ensuring data availability. Since ARGUS requires a server in order to upload and access the stored information and no server scalability is implemented, the availability of the system is compromised.

SCFS is a distributed storage solutions on which file metadata is stored on a scalable, redundant and fault tolerant coordination service, ensuring system availability. Redundancy mechanisms are implemented that aim to protect data against corruption or cloud storage fault, improving data availability. Although providing all the data and system availability measures needed to ensure a storage offload system availability and data protection, its performance renders the system unusable on any real world application.

Kademlia provides one of the best speeds of all the systems tested, the Kademlia distributed nature enables data availability, since when a file is stored on Kademlia, multiple nodes receive a copy of that file, and each one of them is capable of sending the file to other nodes if needed. Since there is always multiple nodes, the storage needs are balanced across a large pool of nodes introducing system availability.

Ceph is a system that is highly scalable and deploys data replication techniques and ensure data availability even on OSD malfunction. Ceph allows to increase the storage without high overheads due to its placement algorithm. Ceph is a highly used solution, being connection to Ceph storage servers supported on the Linux Kernel without any additional software installation.

	Upload Speed	Download Speed	Server Availability	Data Availability
rClone	16 MB/s	7.09 MB/s	Not Applicable	No
CHARON	8.20 MB/s	8.32 MB/s	Not Applicable	Yes
ARGUS	10.21 MB/s	12.22 MB/s	No	Yes
SCFS	0.72 MB/s	2.88 MB/s	Yes	Yes
Kademlia	7.78 MB/s	10.41 MB/s	Yes	Yes
Ceph	7.91 MB/s	8.54 MB/s	Yes	Yes

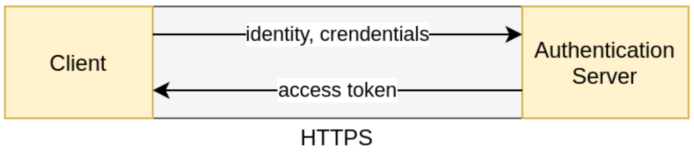
TABLE 3.7: Comparison table of the features of each one of the working solutions presented on the Data Availability and Persistence of the State of the Art

3.4 Authentication

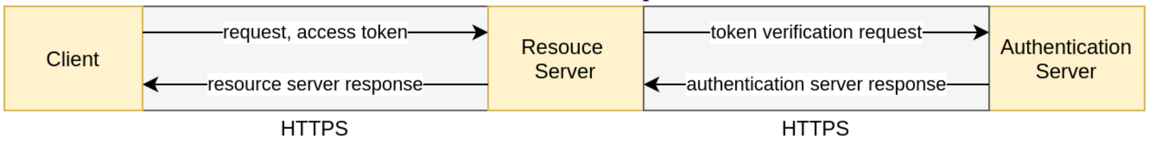
Authentication is the process on which the identity of a user or device is assessed. Authentication is the core of security on every smart city infrastructure, powering access control protocols that restrict the access to resources, such as location data from another citizens. There are multiple ways to achieve authentication, here we state the most popular.

3.4.1 Bearer Token Based

Bearer token based authentication power authentication and authorization protocols such as OAuth2[53]. When a entity wants to get access to a certain service, it issues a token granting request using the entity credentials to an authentication server. The authentication server checks the credentials and returns an access token. The access token can be a simple hexadecimal string or a structured token like JSON Web Tokens (JWT). Once issued the access token must be confidential on both transit and storage, the only parties that should ever see the access token are the client itself, the authorization server and the resource server[54]. When the user has the access token it appends it to every communication to the resource server - server that contains the services that the entity wants to use -, the resource server will use the access token to check the identity of the requester and using that identity verify if it is authorized to access the resource on the request. Those checks are done on the authentication server. Figure 3.8 provides a visual representation of the usage of token based authentication on client-server communications.



(A) authentication phase



(B) authorization phase

FIGURE 3.8: Authentication and authorization phases on Bearer Token based authentication

Token based authentication imposes several issues, related to mutual authentication and confidential channel creation. Imagine a use case where two users want to share some information between each other without using a centralized server as a relay. They need a way to prove their identity to each another, if the first requests an access token from an authorization server and sends it to the other party, it enables the other party to impersonate the first. This makes token based authentication hard to be implemented on a system where peer-to-peer communications might occur, which is the case of several IoT sensor networks.

The most common secure channel protocols, such as Transport Layer Security, use public key cryptography and public key certificates, to prove the server authenticity and to generate a ephemeral session key that is then used to provide confidentiality. This means that certificate based authenticity is already necessary to establish confidential communication channels, and to perform secure peer-to-peer IoT device communication public key certificates would be required in addition to access tokens.

Due to those facts technologies based on OAuth2, or any other token-based authentication, such as FIWARE's KeyRock [55] are not the most suitable to provide authentication for a city wide IoT sensor network and their inherent communications.

3.4.2 Certificate Based

Certificate based authentication methods use public key cryptography to achieve authentication. In order to register a user it first needs to generate a public/private key pair using a public key algorithm such as Rivest-Shamir-Adleman (RSA) or Elliptic-Curve Digital Signature Algorithm (ECDSA). Having that public/private key pair it requests the signature of the public key by an entity that is trusted by every other user, usually a centralized server, using a Certificate Signing Request (CSR) that contains user information such as name and domain as well as information about the public key, including algorithm used, key size, etc. The signature process requires the usage of a private key, that is stored safely and it is never sent on communications. The signer, which can be a centralized server, signs the CSR producing a public key certificate. The public key certificate binds the public key to an identity, being that binding signed by an trusted entity. One particularity of public key certificate is that, contrarily to access tokens, the public key certificates do not need to be stored securely and neither require confidential channels to be sent to other devices. The only thing that must be confidential is the private key associated with the public key on the certificate and the private key that signed the certificate. Those private keys must never leave the device that generated it.

Public key certificates can be used on mutual authentication protocols [56][57] as well to build confidential, integral and authentic communication channels[58][59], being the perfect building block to an authentication protocol on a city wide IoT sensor network.

Chapter 4

Security Requirements

Identification of assets, actors and attack vectors are crucial to the development of secure system designs. In this chapter, we describe the threat model of our system and describe possible security attacks. With the information gathered on this step we deployed countermeasures at architecture and implementation level enabling us to have a more robust system.

4.1 Threat Model

Threat modelling is a process that enables to expose the potential threats to the assets, allowing us to enumerate and prioritize threats based on their risk to affect a asset value. We first define our assets, then the actors on the system, trust model and end with the threat agent identification

4.1.1 Assets

An asset is something on which a stakeholder deposits value into. The main asset in our system is privacy, any attack to the privacy of the stakeholder reduces its value and the trust on the system. The second is trust in a sense that the stakeholders must know to whom their are communicating with. The third is confidentiality, meaning that the data produced is not accessible to any actor that is not considered by the stakeholder. The last one is availability. A smart city must be always accessible, and a disruption to the smart city availability can have large consequences on the lives of its citizens.

4.1.2 Actors

Actors are the people that interact with the system. In our design we consider the following actors:

- **City Administrators** - A set of people that are able to manage the smart city, they value the confidentiality, trust and availability of the system they manage.
- **Citizens** - Englobes all the citizens that use the smart city services. They can publish or subscribe to data. Depositing value on the privacy, confidentiality and availability.

4.1.3 Trust Model

Trust model enables us to set assumptions on which the security of the system will rest upon. Any vulnerability on a component of the trust model compromises the security of the entire system.

We assume that all the cryptographic operations are secure and the private cryptographic keys, when they are stored on the device are not retrievable by any attacker even if it has physical access to the device.

4.1.4 Threat Agents

Threat agents are those who attempt to leverage any vulnerability on the system to affect the system assets. Here we describe the ones considered in our design:

- **Honest-curious user**

  This is a typical user who will explore the application and its features, e.g., exploring the user interface, and will not look for any other kind of information. Their behavior will be what is considered normal and will not change their movement pattern in any way.

- **Tech-savvy user**

  This agent has some technical knowledge and will try to understand the operation modes and mechanisms used. It has access to the system via the client and can set up WiFi and mobile antennas or routers to eavesdrop traffic from and to the endpoint. It can also use tools to decompile and modify the client in order to create a misbehave/malicious client.

- **Backend administrator**

  This agent can control the smart-city backend, access databases and perform system maintenance and change system configuration. This user can also cause service downtime on purpose or by mistake.

- **Eavesdropper**

  This type of user will be able to observe traffic for all clients on the network as well as analyze traffic patterns, inspect requests, analyze communications' sources and destinations. This includes Internet Search Providers, Internet sniffers and exit nodes of mix networks such as Tor.

4.2 Attack Vectors

Having defined the threat model we will describe the possible attack vectors that the threat agents might utilize to affect the value of an asset. Here we describe the ones that we considered on the design of the system.

- **Eavesdropping**, in this vector an attacker attempts to gather confidential information by listening to the communications between two parties. Eavesdropping requires encryption or other confidentiality preserving operations to mitigate eavesdropping;

- **Impersonation**, in this vector an attacker impersonates another user by intercepting the communication and pretend to be the other user. Impersonation mitigation requires authentication mechanisms to be built in the communication protocol;

- **Stealing**, in this vector an attacker steals a smart-city connected device and attempts to use it to send malicious payloads to the smart-city. To mitigate the security issues related to this attack vector, mechanisms of input sanitization and device revoking are necessary;

- **Man In The Middle**, in this vector an attacker is capable of intercepting, altering and removing communication packets while they are being transmitted. To mitigate security issues related to this vector authentication, confidentiality and integrity mechanisms are necessary to mitigate it;

- **Denial of Service**, in this vector an attacker uses a set of devices to flood the city servers with as much traffic as possible, denying the access of citizen to the city services. To mitigate this issue architectures that are scalable and capable of managing large spikes on incoming traffic are required;

Having defined the threat model, trust model and attack vectors we have a more precise idea of the security challenges that our system might face and we will incorporate attack vector mitigation's in both architecture design and implementation to enable us to build a more robust system.

Chapter 5

Architecture

In this chapter we aim to present the underlining architecture of your proposed solution. We will split the architecture into 3 distinct layers that interact with each other to provide the desired functionality. These layers are: Storage, Access and Client Device Layer.

5.1 Storage Layer

The storage layer of the architecture is responsible to store and retrieve data from on-premise (local storage) and/or off-premise (cloud storage) sources in a way that ensure the following properties:

- **Storage Redundancy**, this property makes the system resilient to failure of some storage units without any data loss.
- **Integrity**, integrity checks allow for the client to notice that the file was altered in the process of storing or retrieving data. Integrity is therefore a important feature of any secure storage system.
- **Scalability**, this property enables the system to increase storage capacity and processing power to deal with an spontaneous or continuous increase of storage usage.
- **Availability**, by definition means that system must be present or ready for immediate use, meaning that for a system to provide availability it must have small, close to zero, down-times, caused by maintenance, hardware and software failures or by Denial Of Service (DOS) attacks[60]. A storage system with high availability means that it is ready to accept requests and execute them with success whenever a client requests it.

- **High upload and download speeds**, with the ever increasing file sizes of today's digital content and the diffusion of high speed networking, storage systems that can provide high, and stable data throughput are mandatory to make the system usable.

5.1.1 On-Premise Storage

Large organizations, such as companies or cities, have a large number of connected devices that are constantly trying to store and retrieve data. After a storage infrastructure deployment, there is a limited storage capacity that will eventually be increased by adding new storage devices, or by upgrading existing ones. On storage servers, hardware failures often occur, sometimes preventing access to the data stored on that hardware completely. To cope with all the problems that occur from a highly dynamic, high usage storage service, solutions that are built with mitigations that improve data availability, reliability and persistence even on hardware failures are needed.

Ceph[46] is a scalable, high-performance distributed object file system that aims to provide mitigations to all the problems discussed above. Ceph ensures data consistency with high performances. The use of a cluster map and a deterministic placement function[47] do not require any lookup server to determine where is the data stored in the distributed storage. When the case of a storage device addition or removal the cluster monitors agree with each other on the state of the cluster and then issue a new cluster map. The OSDs that constitute the storage component of the cluster, are constantly checking the state of the cluster on a peer-to-peer fashion. If some OSD is reported failing, the remainder OSDs use the information from the new cluster map and the data stored on the active OSDs to move the necessary data to other nodes reconstituting the necessary data replication. Ceph, unlike Kademlia[44], does not require the client device to re-upload data to ensure that the data stored on the distributed storage does not lose the redundancy properties, which can cause reliability issues. Ceph also provides consistency on data updates, due to the deterministic placement algorithm CRUSH[47], something that standard Kademlia implementations do not provide.

Ceph, additionally, provides synchronous I/O support, meaning that even on multiple concurrent accesses to the same file the consistency of that file is preserved. This property becomes very useful when Ceph is used as storage to multiple replicas of a same service, which could cause read-write or write-write conflicts.

Ceph is used in our architecture as a scalable, high-performance on-premise storage enabler, being used to store required services state and the uploaded data coming from the client device layer for the necessary amount of time before a upload to the off-premise storage is completed.

5.1.2 Public-Cloud (Off-Premise) Storage

Sometimes the local resources are not enough to deal with the storage and performance needs, and, with the increase in the number of users the bigger is the amount of hardware and networking equipment needed to cope with the resource demand, increasing costs and difficulting management.

Storage solutions often require off-site backups to deal with possible problems that affect the local storage infrastructure, such as fires, floods, etc. Those accidents result in a unrecoverable data loss, imposing problems to the system availability and data persistency.

Public cloud storage provides a easily manageable, cheap and reliable way to store information that is accessible at high speeds, with high availability, and with multi-regional replication. The usage of public cloud as a storage solution brings issues related to the data sovereignty caused by the lack of trust in storing private and sensitive information outside of the premise controls. Resende et al. in *Enforcing Privacy and Security in Public Cloud Storage*[6] proposes ARGUS as a solution that reduces sovereignty and trust problem by splitting the information across multiple cloud storage solutions, reducing the amount of control that each cloud provider has over the data. The data is splitted using Reed-Solomon erasure coding, that enables the reconstruction of the entire original data even if some parts get corrupted or destroyed. The trust issues are resolved through a combination of encryption and HMAC that ensure the confidentiality, integrity and authenticity of the data at rest.

Although the available implementation of ARGUS broker server is not designed to provide scalability the inherent process of data encryption, HMAC, erasure coding and multiple cloud uploads can be used. We remove the need of the broker server at the storage layer, being one of the access layer responsibilities to assume a similar role as the broker server, that is, encrypting,doing integrity checks on the data and sending it to the public cloud infrastructure. We used ARGUS has the security base of our public cloud storage component since in our Related Work tests it had one of the best performances while providing strong security to the uploaded data.

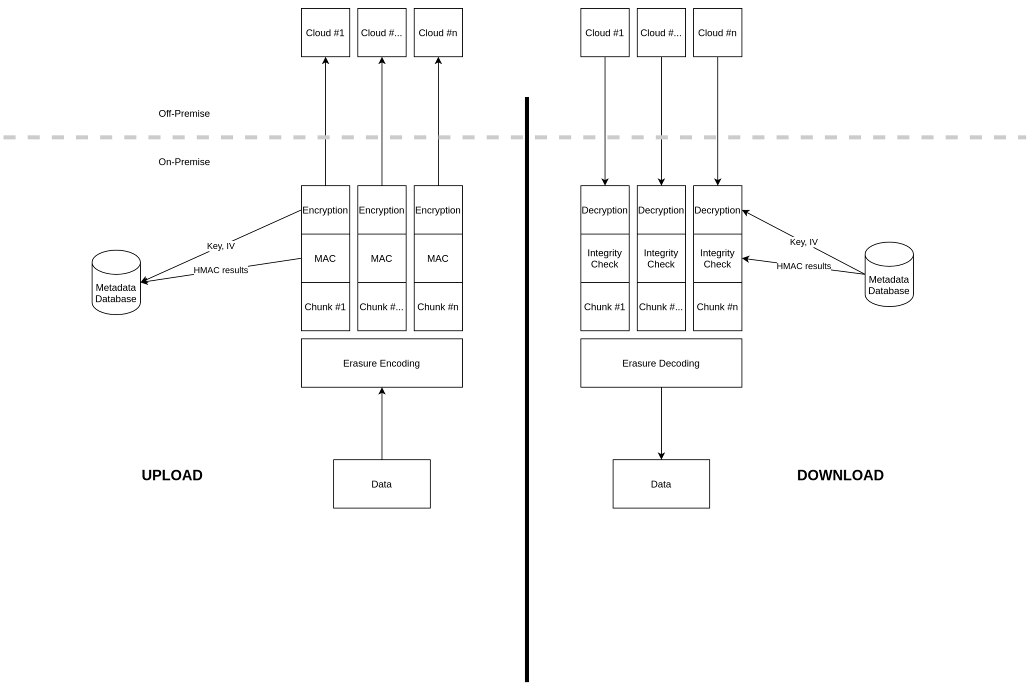


FIGURE 5.1: Public cloud upload and Download Architecture

Figure 5.1 shows the architecture of the public cloud storage component of the storage layer, the entire on-premise process is done on the access-layer by the off-premise storage controller, where the data is splitted into chunks using erasure encoding, then the HMAC hash is calculated and stored. After that, the chunk is encrypted by using Advanced Encryption Standard (AES) CBC with a key that is randomly generated for the entire data that will be uploaded, and a Initialization Vector (IV) generated at random for each chunk to be uploaded (left). The download procedure (right) does the inverse of the upload, on which the data is downloaded, deciphered, checked for integrity issues and then all the chunks are united using erasure decoding, reconstructing the originally uploaded data.

The function of the public cloud storage is to reduce the need to have a ever increasing on-premise storage by uploading older data to the cloud-storage, removing it from the on-premise storage which helps to keep the on-premise storage needs in check while also providing off-site, multi-regional, backups.

5.2 Access Layer

The access layer, as the name states, acts as a component that grants the access to the remainder of the system, abstracting the complexity of the integrated components by introducing Application Programming Interface (API) that the client devices use to access system resources.

As it is possible to verify the storage layer does not provide client authentication nor client authorization, which in turn makes the system insecure since any client can download the data stored, and if the encryption is done on server side, receive the plain-text data. The job of providing client authentication and client authorization is in the responsibility of the access layer.

The access layer also allows the construction of confidential, integral and authentic communication channels between the client devices and the system resources, mitigating attacks such as eavesdropping and man-in-the-middle.

Since our architecture targets the usage of IoT devices and their interaction with a smart city infrastructure, the access layer also contains a IoT message brokering server to help the publishing and retrieval of information between the IoT client devices.

5.2.1 Provisioning

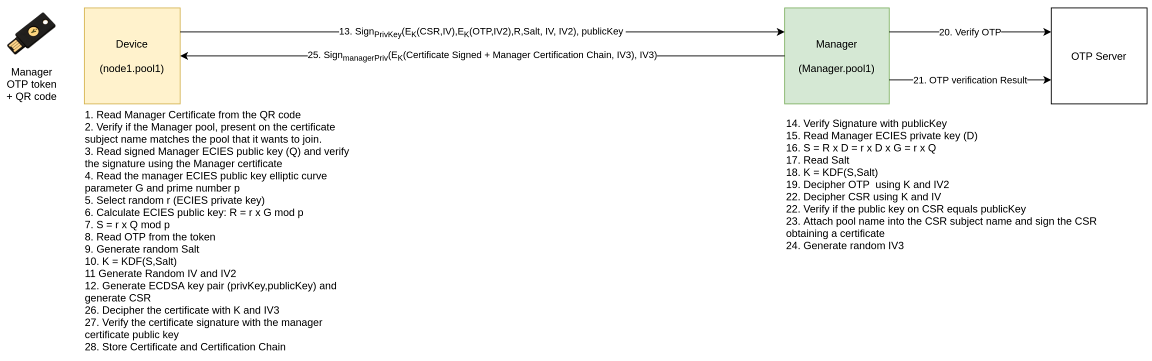
Due to fact that the architecture targets the interaction of IoT devices and the smart city infrastructure, authentication and device registration protocols must be tailored to provide secure device provisioning (the processing of registering a device into the network), and a way that the authentication can be done automatically when a device restarts after some device failure or power outage. We chose to use the general idea of authentication and provisioning behind Sousa et al. *Secure Provisioning for Achieving End-to-End Secure Communications* [61] paper, which uses digital certificates and hardware One-Time Pad (OTP) tokens to securely authenticate and provision devices into the network. We took that idea and expanded it into a smart city scenario.

YubiAuthIoT, the name that we gave to the device provisioning, authentication and communication protocol requires three components: A device that wants to be provisioned, a pool manager and a otp server attached to the pool manager.

YubiAuthIoT sub-divides IoT devices into pools, each one containing a unique set of IoT devices, being each pool managed by a unique pool manager. The pool manager can provision and remove devices from the pool. Sub-dividing into pools, enables the provisioning process to scale without the need of a central service intervention. When the pool manager is first created it generates a Elliptic-Curve Digital Signature Algorithm (ECDSA) key pair, then it generates a self signed certificate by signing the ECDSA public key with the ECDSA private key. The self signed certificate subject name is in the form of "manager.poolToManage" (ie. Manager.pool1) where poolToManage is the name of the pool that the manager will manage. The pool manager will also generate a Elliptic Curve Integrated Encryption Scheme (ECIES) key pair that will be used to establish a confidential communication channel on the authentication phase. The manager certificate and the ECIES public key are stored on a Quick-Response (QR) code that must be read in combination with the OTP token, by the device to be able to authenticate with the manager and be provisioned into the pool. The ECIES key inside the QR code is digitally signed with the pool manager ECDSA private key, providing authenticity and integrity to the ECIES public key.

Attached to the pool manager there is a OTP server, which the responsibility is to verify if the OTP received by the manager on the provisioning process comes from the manager OTP hardware token.

When a device wants to be provisioned into a pool, the pool administrator, a person with the QR code and the OTP token, must insert the token into the device that wants to be provisioned and make the device read the QR code for the provisioning process to start. Figure 5.2 illustrates the provisioning process.


Manager OTP token + QR code
Device
(node1.pool1)
13. Sign$_{PrivKey}$(E$_K$(CSR,IV),E$_K$(OTP,IV2),R,Salt, IV, IV2), publicKey
25. Sign$_{managerPriv}$(E$_K$(Certificate Signed + Manager Certification Chain, IV3), IV3)
Manager
(Manager.pool1)
20. Verify OTP
21. OTP verification Result
OTP Server

1. Read Manager Certificate from the QR code
2. Verify if the Manager pool, present on the certificate subject name matches the pool that it wants to join.
3. Read signed Manager ECIES public key (Q) and verify the signature using the Manager certificate
4. Read the manager ECIES public key elliptic curve parameter G and prime number p
5. Select random r (ECIES private key)
6. Calculate ECIES public key: R = r x G mod p
7. S = r x Q mod p
8. Read OTP from the token
9. Generate random Salt
10. K = KDF(S,Salt)
11 Generate Random IV and IV2
12. Generate ECDSA key pair (privKey,publicKey) and generate CSR
26. Decipher the certificate with K and IV3
27. Verify the certificate signature with the manager certificate public key
28. Store Certificate and Certification Chain

14. Verify Signature with publicKey
15. Read Manager ECIES private key (D)
16. S = R x D = r x D x G = r x Q
17. Read Salt
18. K = KDF(S,Salt)
19. Decipher OTP using K and IV2
22. Decipher CSR using K and IV
22. Verify if the public key on CSR equals publicKey
23. Attach pool name into the CSR subject name and sign the CSR obtaining a certificate
24. Generate random IV3

FIGURE 5.2: YubiAuthIoT provisioning procedure

In the provisioning process, the device first reads the manager certificate from the QR code. The manager certificate contains information about the pool that the manager manages, the device verifies if the pool information contained in the certificate matches the one on that the device wishes to be provisioned into. If a match occurs the protocol is allowed to proceed, otherwise a error is thrown. The next step is to read the signed manager ECIES public key, the signature is needed to prove that the ECIES public key indeed belongs to the manager represented on the received certificate since if it does not match the QR code might have suffered some Man-In-The-Middle attack. Currently one might say that ECIES digital signature seems to not be very helpful mitigating Man-In-The-Middle attacks since the received manager certificate is a self-signed certificate being possible for an attacker to forge the certificate and the signature of the ECIES public key. However when extended to the smart city scenario, there will be a root manager, that signs all the pool manager certificates and is trusted by every device. With that in mind, the process of signing the ECIES public key provides proof that the certificate and ECIES public key were originated from the same pool manager.

After the verification of the ECIES public key signature the device obtains the elliptic curve parameter G and the prime number p from the ECIES public key. It is done to obtain the exact same point on the curve that was used to generate the public key and the modulus that generated the elliptic curve finite field. The parameter G and the prime number p can be of public knowledge without compromising the security of the protocol. Having G, the device generates a random number r, this number will be the ECIES private key of the device and it is only used on the authentication process and then discarded. Next the device uses the ECIES private key to generate a public key R by multiplying the elliptic curve point G with r obtaining a new point on the curve that will be the public key. The result of the ECIES protocol is a shared point S that both communicating parties can calculate but no eavesdropper is capable of calculating. S is calculated the following way on the device $S = r \text{ x } Q \bmod p$, being Q the manager ECIES public key. The manager calculates S by doing $S = R \text{ x } D \bmod p = r \text{ x } D \text{ x } G \bmod p = r \text{ x } Q \bmod p$, being D the manager ECIES private key.

Having the access to the OTP hardware token, the next step is to request a OTP code from the token, that will be sent to the manager for it to assess if the device is being provisioned by the administrator and if so grant access to the network. The granting is given by the signature of the device certificate containing the device ECDSA public key.

The next step is to generate a AES symmetric key K using the point S. We achieve that by using a Key Derivation Function (KDF) that takes the point S and a randomly generated salt, producing a 256 bit key that will be used to improve the confidentiality of the communication and the security of the authentication protocol.

The device then generates a ECDSA key pair and builds a Certificate Signing Request

(CSR) requesting to sign a certificate containing the generated ECDSA public key. The generated ECDSA private key is considered a long-term key and must be stored safely on the device. We assume that an attacker cannot gather the stored private key even if it has access to the device hardware, this can be achieved using trusted computing environment technologies, such as ARM TrustZone [62], however we do not do that in this dissertation. The generated CSR contains the device name on the subject name (ie. node1) and it will be used by the manager to append it to the the pool name of the pool that he manages, generating a certificate with the subject name in the format: nodeName.poolName (ie. node1.pool2).

Finishing the CSR generation the device has all the necessary information to perform a provisioning request near the pool manager. The message constructed by the digital signature of the encryption of the CSR and OTP using the symmetric key K and the AES CBC algorithm with randomly generated IVs. R, the KDF salt and the IVs are also digitally signed. The digital signature is performed with the ECDSA private key and the ECDSA public key is also sent outside of the signature.

From inside out, by encrypting the OTP with the key k we mitigate attacks were an attacker interrupts the communication of the request, fetches the OTP, crafts a new provisioning request using the captured OTP and the attacker ECDSA key pair, forcing the manager to generate a certificate for the attacker, provisioning itself into the pool, breaking the provisioning protocol in the process. The encryption hides the OTP value, only being readable by the manager, therefore mitigating this kind of attack. The CSR is ciphered with the same key as the OTP, that being done on purpose. When the manager receives the request and successfully deciphers and verifies the OTP means that K was successfully calculated, and that the encryption was performed by the device and not an attacker, since the attacker cannot have access to K to decipher the OTP and re-encrypt it with another K. If the same K is used in the CSR encryption, means that the same entity that ciphered the OTP also ciphered the CSR, mitigating attacks were the CSR is altered via a man-in-the-middle attack.

Since R, Salt and IVs are safe parameters, meaning that by themselves do not transmit any private or confidential information, they can be sent in clear text. An attacker can try to change these values, but the only possible outcome is that they might achieve is to provoke failed provisionings to honest devices that would be provisioned if the attacker was not there, however they cannot break the protocol and get provisioned.

The device that received manager OTP might attempt to sign a certificate to a public key that the device has no matching private key. To help prevent this, the digital signature of the entire authentication request is necessary, introducing the following properties: private key knowledge, integrity, authenticity and non-repudiation. To be able to sign the request the device must have knowledge of the private key, and if the public key used to verify the signature differs from the one on the CSR the authentication process fails. Integrity means that when some part of the request changes the signature becomes invalid, so changes to the request are noticeable by the manager. Since the private key is needed to perform a signature and that private key is only on the possession of the device it provides authenticity (only the device with the private key could have signed the request) and non-repudiation (the device cannot say that it did not sign that request since no one else has that private key).

The pool manager when receives the authentication request it verifies the request signature, and then using its ECIES private key it generates the shared point $S = R \text{ x } D \bmod p = r \text{ x } D \text{ x } G \bmod p = r \text{ x } Q \bmod p$, being D the manager ECIES private key. Then it uses S and the Salt provided on the request and computes symmetric key K using the same KDF algorithm as the device that wants to be provisioned. Having K calculated and the IVs on the request the manager deciphers the OTP, checking its validity on the attached OTP server. If the otp verification is successful the OTP was generated by a OTP hardware token belonging to the manager administrator. Next the manager deciphers the CSR and verifies if the public key on the CSR matches the public key used to verify the request signature. If it does match, the manager generates a signed certificate with the device name and the manager pool on the subject name, ie. node1.pool1. Then the manager signs the certificate with its ECDSA private key, authenticating the device and allowing it to be inserted on the pool.

The pool manager sends the previously signed device certificate and its certification chain back to the device. The authentication response is ciphered with K and a randomly generated IV. The encryption result is then digitally signed with the pool manager ECDSA private key. This ensures that the response comes from the pool manager, and the certificate information is protected against eavesdropping, where an attacker attempts to gather information about the data sent. The usage of digital signatures also provides integrity, removing the possibility of an attacker to change the response mid communication. The encryption also unables other devices to read the sent certificate since the encryption key was calculated between the device to be authenticated and the pool manager.

The device when receives the authentication response, it checks the response signature using the public key received from the QR code, and using K deciphers the response obtaining the signed certificate and the manager certification chain.

It is important to note that a OTP code is only usable once. If other request uses the same OTP code it is be deemed invalid by the OTP server, this way mitigating replay attacks that attempt to replay some captured communications with the purpose to defeat the protocol and get provisioned wrongfully.

After the provisioning procedure the device receives a public key certificate signed by the pool manager. The manager certificate is added to the device trust anchors allowing the device to communicate with all the devices on the same pool. When the device wants to communicate with another device it no longer needs the manager presence, reducing single point of failure problems. The pool manager is only required to be active when a device wants to be provisioned into the pool as it can be turned off when no provisioning is needed. The pool-manager stores a copy of every signed certificate and uses that information to know which devices are provisioned and to revoke the access to the pool when needed.

5.2.2 Secure communication

Figure 5.3 details the *YubiAuthIoT* communication procedure. This procedure requires that both devices were previously authenticated and obtained their signed certificates. The secure communication procedure can be divided into 3 parts: device-device authentication, symmetric key agreement and communication.

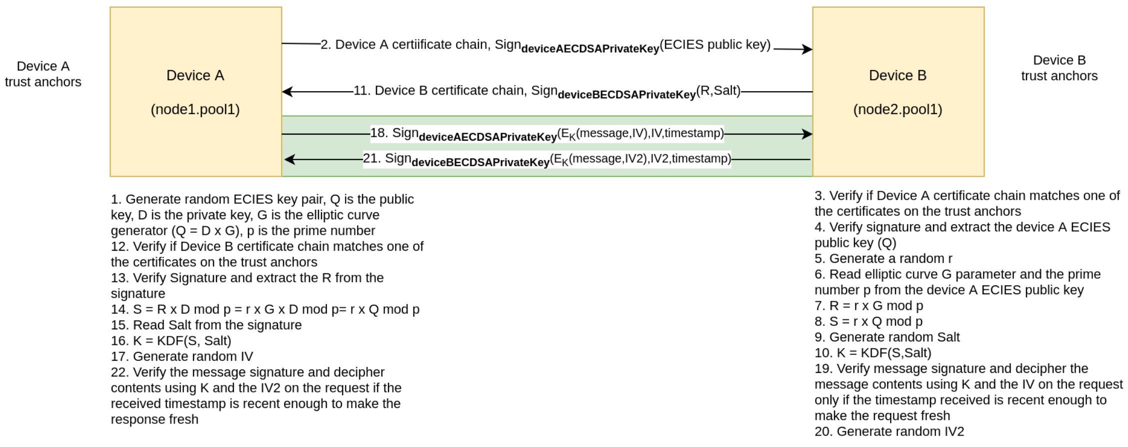

Device A trust anchors
Device A
(node1.pool1)
2. Device A certiificate chain, Sign$_{\mathbf{deviceAECDSAPrivateKey}}$(ECIES public key)
11. Device B certificate chain, Sign$_{\mathbf{deviceBECDSAPrivateKey}}$(R,Salt)
18. Sign$_{\mathbf{deviceAECDSAPrivateKey}}$(E$_K$(message,IV),IV,timestamp)
21. Sign$_{\mathbf{deviceBECDSAPrivateKey}}$(E$_K$(message,IV2),IV2,timestamp)
Device B
(node2.pool1)
Device B trust anchors
1. Generate random ECIES key pair, Q is the public key, D is the private key, G is the elliptic curve generator (Q = D x G), p is the prime number
12. Verify if Device B certificate chain matches one of the certificates on the trust anchors
13. Verify Signature and extract the R from the signature
14. S = R x D mod p = r x G x D mod p= r x Q mod p
15. Read Salt from the signature
16. K = KDF(S, Salt)
17. Generate random IV
22. Verify the message signature and decipher contents using K and the IV2 on the request if the received timestamp is recent enough to make the response fresh
3. Verify if Device A certificate chain matches one of the certificates on the trust anchors
4. Verify signature and extract the device A ECIES public key (Q)
5. Generate a random r
6. Read elliptic curve G parameter and the prime number p from the device A ECIES public key
7. R = r x G mod p
8. S = r x Q mod p
9. Generate random Salt
10. K = KDF(S,Salt)
19. Verify message signature and decipher the message contents using K and the IV on the request only if the timestamp received is recent enough to make the request fresh
20. Generate random IV2

FIGURE 5.3: YubiAuthIoT secure communication procedure

When a device wants to communicate with another device it generates a random ECIES key pair. The random ECIES key pair is a ephemeral session key and is only used on the current communication, being immediately discarded afterwards. By generating random ephemeral keys for each communication, even if a key is compromised only the communication that used that key is compromised, all the previous communications remain confidential. Having the ephemeral ECIES key pair generated the node that wants to initialize the communication, A, sends its ephemeral ECIES public key digitally signed with the device A ECDSA private key to the receiver device, B. The public key certificate obtained from the authentication is also sent to B. Device B upon receiving the request checks if A's certificate certification chain matches one of B's trusted anchors. If it matches B accepts communication from A.

The next step is to verify the signature of the received, device A, ephemeral ECIES public key, using the device A certificate. The signature here serves two purposes: ensure the sender authenticity and ensure content integrity, meaning that an attacker is not able to change the contents and it is not able to impersonate device A since it does not have the device A ECDSA private key that is used to perform digital signatures. After a successful verification of A's ECIES public key signature, B generates a random number r. Then gets the elliptic curve generator point G and the prime number p from A's ECIES public key and generates its own ECIES public key ($R = r$ x G mod p). Then using the received device A ECIES public key (Q) and r the device B generates the shared elliptic curve point $S = r$ x Q mod p, that will be used to generate the shared AES symmetric key K using a KDF and a random salt. The device B sends its own ECDSA certificate and R plus the Salt digitally signed by B's ECDSA private key.

Upon receiving the response, device A checks if the received device B certificate chain matches one of A's trust anchors. If it does the communication between A and B is allowed. The next step is to verify the signature of the received R and Salt against the received B ECDSA certificate to check if B was the one to produce R and the Salt that will be used to generate the symmetric encryption key K. This allows to prevent impersonation attacks were and malicious device pretends to be the device B, forcing the communication between A and the malicious device without A's knowledge.

Having R and the Salt the device A calculates $S = R$ x D mod $p = r$ x G x D mod $p = r$ x Q mod p, D being A's ECIES private key. Using S and the Salt A generates the symmetric key $K = KDF(S, Salt)$ obtaining the same key as B.

The symmetric key K together with an encryption algorithm, such as Advanced Encryption Standard (AES), allows to achieve communication confidentiality with improved performance and arbitrary message dimensions, properties that public key encryption alone does not provide.

After setting K devices A and B can start communicating with each other. To do so each message is encrypted using K and a randomly generated IV, thus providing confidentiality. However confidentiality is not enough to mitigate attacks such as integrity and replay attacks that, respectively attempt to change parts of the ciphertext to cause unexpected behaviour and attempt to replay captured communications to cause unexpected behaviour on the receiving device, allowing it to query information, do logins and logouts, etc.

To mitigate those attacks we add digital signatures as our identity and integrity provider, additionally digital signatures also provide non-repudiation properties to our communication, mitigating integrity and impersonation attacks. To mitigate replay attacks we add timestamps to each request, that are also digitally signed to avoid an attacker to change the timestamp to a current one. The timestamp is used as nonce - number only used once - and the receiver device only allows communication that contains a timestamp newer than the previous one received minus some locally defined skew. This way mitigating replay attacks.

5.2.3 Expanding to the Smart City Scenario

YubiAuthIoT provides secure authentication and communication channel establishment between devices inside the same pool. However to cope with the dimension and management challenges that come bundled with large deployments, such as smart cities, we chose to use the pools provided by *YubiAuthIoT*, where each pool contains a unique set of devices and each pool with one dedicated sub-manager. This means that the management of devices is shared across a set of sub-managers, each one capable to manage a sub-set of the smart city connected devices. Figure 5.4 aims to present a visual representation of the sub-management solution.

This solution also allows for companies or services to work together with the smart city to provide the city with data, but the management of the company devices done by the companies themselves when they become a smart city sub-manager. One important thing to mention is that a domain is set for each sub-manager, and a sub-manager is not

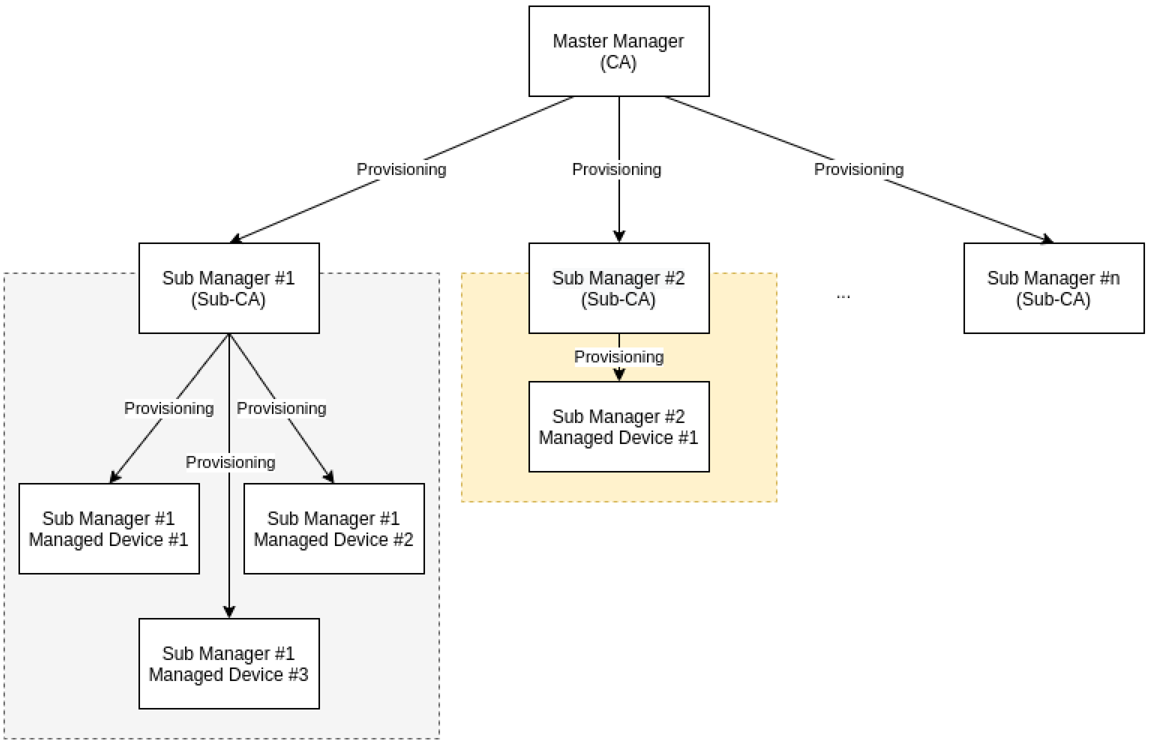


FIGURE 5.4: YubiAuthIoT application into the Smart City Scenario

allowed to manage devices that do not belong to the same domain as the sub-manager itself. This is direct application of pools described on the *YubiAuthIoT* protocol.

The job of the master manager is to provision the sub-managers, allowing their devices to use smart-city services, such as IoT message brokers, in an secure and authenticated way. The authentication comes in the form of certificates that are signed by a sub-manager, which in turn also has a certificate signed by the master manager. Establishing, this way, a certification chain that has its root on the master manager certificate.

When a pool p that has a pool manager m wants to use the smart city infrastructure it must request the city administrator for it to go in person to the pool manager m and plug the city OTP hardware token. After that, the authentication procedure occurs as stated on 5.2.1 resulting in a pool manager certificate signed by the city manager device and the provisioning of the pool into the smart city.

At this moment all the devices on p still do not know that p was appended to the smart city infrastructure and cannot communicate to the smart city infrastructure since their trust anchors do not contemplate the smart city certificate, resulting on communication protocol failure. To be aware of pool appending updates, the pool devices, periodically request the manager for appending updates, and if the manager is running, it returns the new certification chain and the certificates that the device will now assume as trust anchors. The communication between all the devices and managers on the smart city follows the *YubiAuthIoT* secure communication protocol described in 5.2.2.

Figure 5.5 aims to provide a representation of the trust anchors and certification chains of each device.

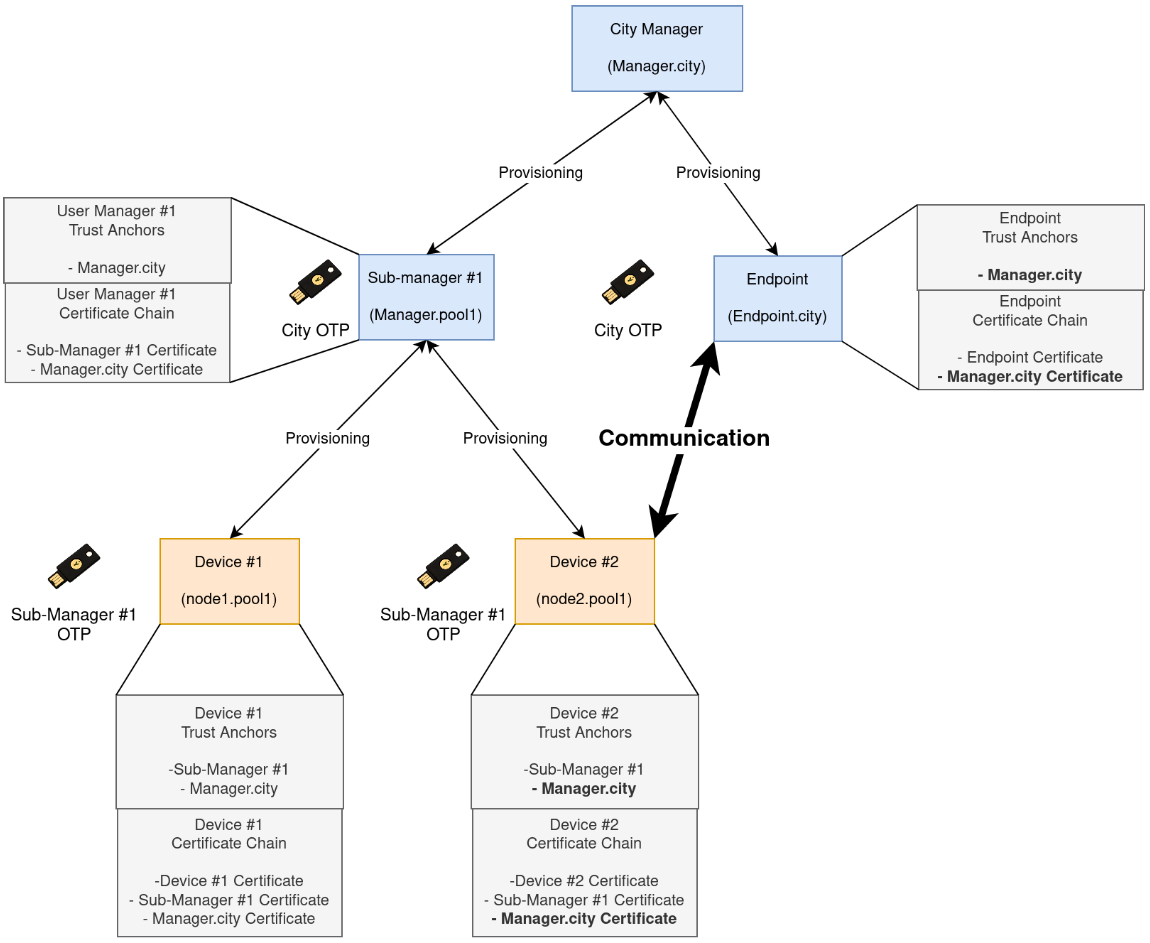


FIGURE 5.5: Provisioning and communication on a Smart City Scenario

After a pool is appended to the smart city, the pool-manager certificate is updated to one that is signed by the city manager, instead of a self-signed one, this means that the certification root of that pool has moved from the pool-manager to the city manager. The city manager certificate is a self signed certificate, representing the city certification root. Since the pool-manager certificate was changed, the certification chain of the pool devices will be updated to contain the device, pool-manager and city manager certificates. The trust anchors of the devices will be also updated to contain the city certificate as a trust anchor. This will allow the communication of a device to any other device, belonging or not to the same pool. However, depending on the way that pools want to be isolated from each other, this might impose some security concerns.

To give the pool-managers more control over which devices are allowed to connect to the managed devices, we introduce communication policies. The communication policies are generated by the pool-manager and given to the device on the provisioning process, so there can be general policies and device specific policies allowing for an increased control on the communications that the devices are allowed to receive and execute. For the communication policies to work, we assume that a provisioned device will always respect the imposed policies.

When the communication policies need an update the pool-manager generates a new policy and sends it in the device periodic update response or on a immediate communication request directed to the device that will receive the new policies, enabling policy updates.

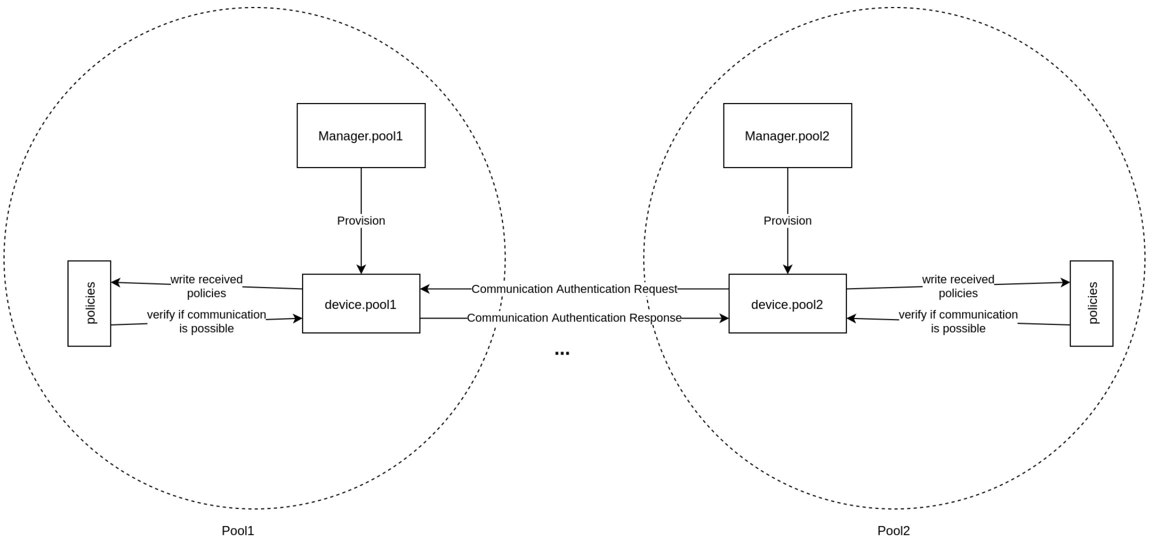

Manager.pool1
Provision
device.pool1
policies
write received policies
verify if communication is possible
Manager.pool2
Provision
device.pool2
policies
write received policies
verify if communication is possible
Communication Authentication Request
Communication Authentication Response
...
Pool1
Pool2

FIGURE 5.6: Usage of policies on pool-pool communication on a Smart City environment

Figure 5.6 shows how the policies are used to allow or block communication requests. The policy verifications are done on the authentication phase of the *YubiAuthIoT* communication protocol. In this case the policy rules are based on the name of the device that wants to communicate and the pool where it belongs, enabling to allow/block communication from a singular device or to an entire pool. The policies in place on the communication phase of the *YubiAuthIoT* communication protocol are based on the resource that is about to be accessed and the identity of the device accessing it, enabling the pool-manager to have fine control of which resources can be accessed by which devices. The default policy is to block, meaning that devices and resources not contemplated on the policies will be blocked. The communications to/from the pool-manager are always allowed since they are critical to preserve the state of the pool.

It is important to state that policies can be designed for intra-pool communications as well, limiting communications from devices inside the same pool.

At this point we have processes to authenticate, communicate and authorize communications using digital certificates and communication policies. As one might have noticed the removal of devices is not yet contemplated. The device removal is essential for the smart city management since there will be the need to discontinue devices that were previously provisioned or were attacked and now are malicious. The job of revoking devices is of the responsibility of the pool-manager.

When a device needs to be revoked, the pool-manager sends the certificate of the device to be revoked digitally signed, together with a hash version of the certificate that is signed by the pool-manager ECDSA private key, to the smart city Certificate Revogation List (CRL) service. The CRL provides a publicly visible list of revoked certificates, enabling to every node to check if the device behind some communication was removed from the city network. The device is only considered revoked if the manager of the pool where the device belongs signs the hash digest of the device certificate, this way preventing other pool-managers to revoke certificates not belonging to them.

5.2.3.1 Endpoint Device

The Endpoint device, depicted on figure 5.5 is a special device directly appended to the city manager that allows provisioned devices from pools appended to the city to access smart city resources, such as IoT message brokers, in a secure and authenticated way. A endpoint is treated as the bridge between the smart city intranet, where services are deployed, and the city internet, where client devices and sub-managers live, connecting devices to the city services.

In order to have a scalable system capable to cope with large quantities of requests the endpoint device is made to be easily replicated, allowing horizontal scalability. We achieve this by reducing the amount of state that a endpoint saves locally and making sure that all the critical information is stored on databases external to the endpoint device itself. This way reducing the need for data replication across endpoint device replicas, easing the replication process.

Having a set of replicated endpoints, we need a way to forward requests to them in a way that utilizes the resources from the replicated endpoint devices equally.

To do that, we deploy a load balancer that sits between the client devices and the replicated endpoint devices.

The job of the load balancer in the architecture is to forward the incoming communications to the endpoint device replicas, that will process the request, in a way that the resources of each replica are equally consumed.

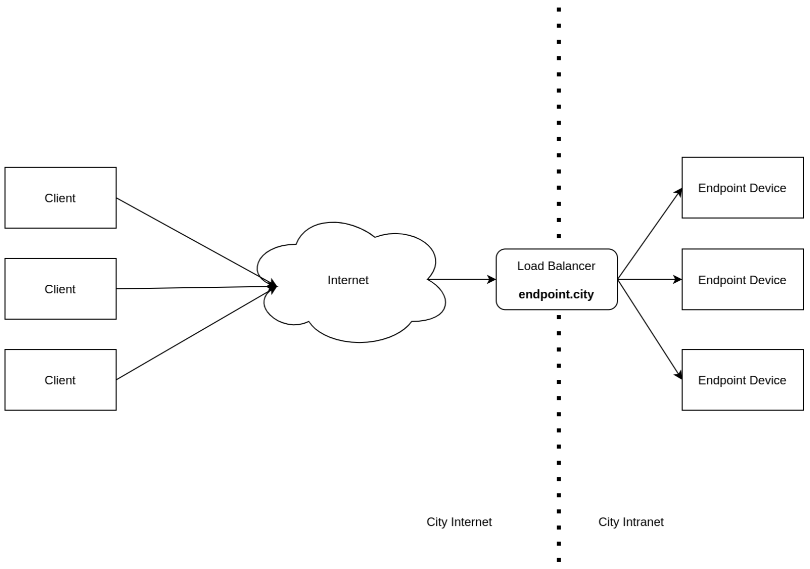


FIGURE 5.7: Endpoint Device Load Balancing to Achieve Availability

Figure 5.7 show the interaction of the load balancer on the communication between the client device of a pool appended to the city and the city endpoint device replicas. As said previously, when a device on the city internet wants to access some city-provided service, it needs to know how to reach the endpoint device, more specifically, know the IP address of the endpoint device. When using a single instance of the endpoint device a DNS record (i.e. endpoint.city A record) pointing to the endpoint device public IP address is enough to enable a device to reach the endpoint device. However when we use multiple endpoint device replicas capable of responding to requests the DNS record must point to the load balancer instead of a specific endpoint device replica. The load balancer will then forward the connection to the endpoint device replica with the lowest resource consumption, achieving a more efficient resource consumption and higher scalability.

The usage of a single load balancer alone does not provide high availability since the load balancer can be seen as a single point of failure. To mitigate the single point of failure we deploy backup load balancer servers in a failover configuration. In a failover configuration when the primary load balancer stops working due to some issue, a backup load balancer is selected to replace the primary load balancer, receiving the communications that were directed to the primary load balancer, improving system availability. Failover processes usually use the concept of virtual IPs, that are addresses that depending on the state of the deployment the same IP can be assigned to the primary load balancer or to a backup load balancer if the primary fails. This means that for the client device the ip address is always the same, but internally can be directed to a primary or backup load balancer.

5.2.4 IoT message brokering

IoT devices due to their low processing power, low cost, embeddable, and internet connected nature are perfect candidates to deploy a city wide sensor network.

IoT devices typically publish data about the environment and use the data published by others to perform some other operations. This behaviour can be easily mapped to the publish/subscribe pattern.

The publish/subscribe enables the exchanging of messages between two types of devices: publishers and subscribers. The publishers are devices that write data and subscribers are devices that read data. The communication between publishers and subscribers relies on the existence of a message broker that relays the messages from the publisher to the subscribers.

The publish/subscribe pattern is widely considered a fundamental way to enable scalable and flexible communications in highly distributed systems [63], where the most significant advantage is the decoupling of communication parties in space and time[64], meaning that the communicating parties do not need to know each other to communicate, and depending on the message broker, do not need to be connected at the same time to have the messages forwarded from the publisher to the subscriber.

In many publish/subscribe patterns, subscribers typically receive only a subset of the total messages published. The process of selecting messages to be forwarded to a subscriber is called filtering. There are usually two types of filtering:

- **topic-based**, in this system, messages are grouped into topics, and the subscriber receives all the messages posted on the topics that he chooses to subscribe;

- **content-based**, in this system, messages are only delivered to the subscriber if the content matches the constraints defined in the subscription, giving the subscriber more control over what kind of messages it wants to receive.

We choose to use the FIWARE Orion as our message broker server due to its adoption on real smart city scenarios, such as in the city of Porto[65]. The message broker, also called context broker, is the core component of the FIWARE component architecture (Fig.5.8) since it allows the communication between the city sensors (publishers) and citizens/organizations (subscribers)

On FIWARE Orion all the published information is in the form of entities using the JavaScript Object Notation (JSON) format, containing the following structure:

```
{
  "id": "entityID",
  "type": "entityType",
  "attr_1": <val_1>,
  "attr_2": <val_2>,
  ...
  "attr_N": <val_N>
}
```

LISTING 5.1: FIWARE Orion entity publishing format

Where the id and type are related with the entity identification, and the attributes contain the values of the data captured by the publisher sensors.

Orion subscription follows the content-based subscription model, where the subscriber selects the id and the type of entities that wants to receive data from, using the complete id and type or by using regular expressions. The subscriber can also choose to be notified only when some condition on the entity attributes is matched, giving more control to the amount of information that a subscriber wants to receive. The subscription request also uses the JSON format but with the following structure:

```
{
    "subject":{
        "entities": [
            {
                "id"/"idPattern": <id>
                "type"/"typePattern": <type>
            }
        ],
        "condition": [
            {
                "attrs": [<attributes that must be present>],
                "expression": [<condition expressions related to the attributes>]
            }
        ]
    },
    "notification":{
        "attrs":[<attributes to be included on the notification>]
        "url":{
            "http":<url to listen for notifications>
        }
    }
}
```

LISTING 5.2: FIWARE Orion subscription format essential components

Where the subscriber selects from all the entity subjects a specific sub-set of those subjects by filtering by id and type. From that subset of subjects the notification is only triggered when at least one of the attributes in condition.attrs is updated and the condition.expression for that attribute is valid. The subscriber, in order to receive the notifications, opens a http server on its side and places the HyperText Transfer Protocol (HTTP) Uniform Resource Locator (URL) on the subscription request. When a notification is available, the context-broker will send the notification information on a POST request to the HTTP URL on the subscription.

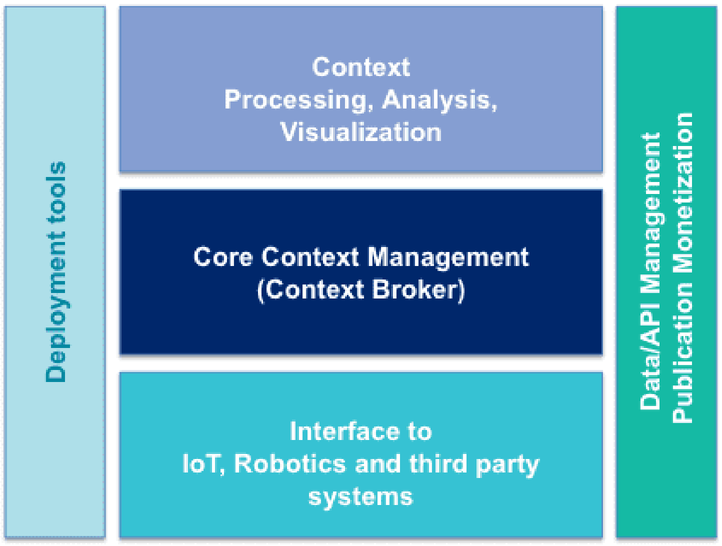

Deployment tools
Context Processing, Analysis, Visualization
Core Context Management (Context Broker)
Interface to IoT, Robotics and third party systems
Data/API Management Publication Monetization

FIGURE 5.8: Fiware Components with the context broker at its core ([66])

FIWARE Orion, due to its close interconnection with other FIWARE components, does not provide by itself any authentication or authorization built in. The context broker in our architecture is only accessible via the endpoint device, meaning that every communication with FIWARE Orion is first authenticated on the endpoint device, introducing device identity guaranties.

The endpoint device can be seen on the FIWARE component architecture as a Policy Enforcement Point (PEP) Generic Enabler (GE) implementation, meaning that it is a device placed in front of the context broker and only forwards communications to the context broker if the authentication can be achieved and the authorization policies based on the content that is about to be accessed and the identity obtained from the authentication result, have a granting result otherwise blocking the communication to the context broker. The authorization service will be discussed on the next subsection.

The usage of a singular message broker server causes single point of failure problems which in turn reduce the availability of the system. FIWARE Orion fixes this by enabling the message broker server to be easily replicated to enable horizontal scalability. Fiware Orion achieves this by storing the subscriptions and other related metadata to a MongoDB database cluster [67]. The MongoDB database cluster is responsible to provide high availability and redundancy to the data stored by the message broker.

Similarly to the endpoint device replication, having a set of message broker replicas requires a way to forward the requests to them in a way that utilizes the resources from the replicated message brokers equally. To achieve that Orion requires the installation of a load balancer that receives the requests and forwards them to the replica that has less resource consumption.

As said previously the usage of a singular load balancer can cause availability issues since it becomes a single point of failure. To mitigate this we deploy a series of backup load balancers that are ready to receive requests if the primary load balancer becomes unavailable. Figure 5.9 provides a visual representation of the FIWARE Orion high availability architecture discussed above.

5.2.5 Authorization

The client authorization, also provided by the access layer, uses the identity obtained from the authentication process to grant access to the requested files/resources according to the defined authorization policies.

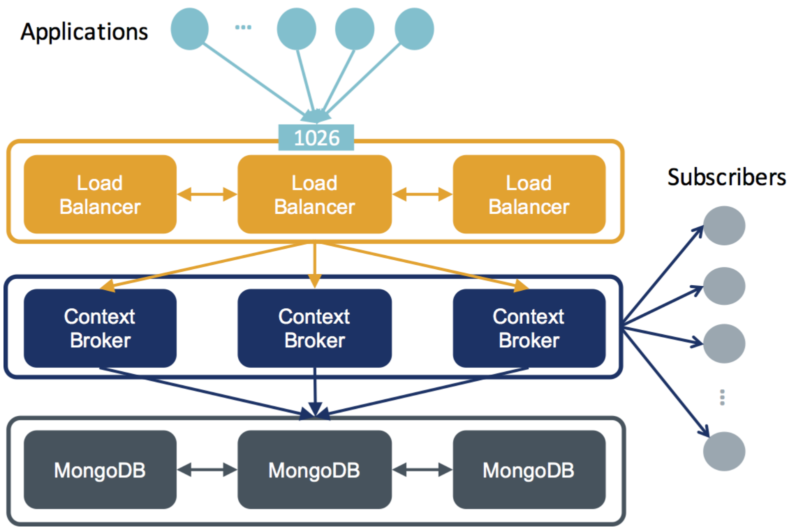


FIGURE 5.9: FIWARE Orion high availability architecture (Extracted from [67])

Authentication is not enough to prevent unauthorized use of resources. Authentication is only used to assess the identity of the device/individual accessing the system, nothing more. Authorization, through the usage of policies, introduces rules that allow or deny access to certain resources/services based on the entity identity and the resource itself, providing a fine grain of detail of which resources a certain entity is allowed to use.

Currently, there are several authorization policy schemes that allow more or less detail on the resources that certain entity is allowed to access. We focus on the more popular ones: RBAC and ABAC.

- **Role-Based Access Control (RBAC)** is used to restrict the access to resources based on the roles that a user has on an organisation. This means that the policies in place are always related to the role where the user belongs, not the user itself. If a restriction needs to target a certain user, a new role must be created with only that user, making the management difficult. Usually each role as a set of resources that it can access using standard actions like read, write and execute. RBAC can be used divide the user pool into hierarchies, setting specific resources that each hierarchy is allowed to use. The division into hierarquies is very useful on companies and public organizations.

- **Attribute-Based Access Control (ABAC)** is used to restrict the access to resources using information about the user which is accessing the resource, the resource itself, the action to be performed and the environment on which the action is being performed, like time of day and location. ABAC works by setting policies about the access to resources where there can be several policies in place related to a certain access to a resource, and the result of the policy evaluation can require that all the policies in place allow the access or at least one to allow the access. The actions on a ABAC policy can differ from the standard read and write to accommodate actions specific to the attributes of the request, such "transfer" when dealing with resources like money. ABAC therefore enables a more precise control over the access to resources being more adaptable than RBAC via the enabling of custom policies.

Analysing the pros and cons of each of the authorization policy schemes, ABAC is the clear winner since it allows the establishment of rules according to specific request attributes, the identity of the user, and other information about the user, such as groups, therefore providing a great level of detail on the policies.

FIWARE Authzforce[68], is a implementation of ABAC authorization policy scheme using the eXtended Access Control Markup Language (XACML)[69] standard. FIWARE Authzforce places itself on the FIWARE architecture as a Policy Decision Point (PDP), where the policies are evaluated, returning a permit or deny result, and as a Policy Administration Point (PAP), where policies are created and maintained.

Since every publication made by client devices is sent to the IoT message broker, the authorization server, in this case, FIWARE Authzforce, allows to set restrictions on the access to the data sent to the message broker, by, for example, blocking subscriptions that want to access to data that is not marked as public and was not produced a member of the same pool. The usage of an authorization service to block the access to certain publications by certain subscribers enables the smart city infrastructure to be more privacy concerned and increase the general utility of the infrastructure. When a request to the IoT message broker reaches the endpoint device, it uses the request attributes and the identity of the device that made the request to provide the FIWARE Authzforce PEP with the necessary information for the authorization. Authzforce uses the information received and the policies in place returns "Permit" or "Deny" back to the endpoint device. If the endpoint device receives a deny then a error is issued to the requester blocking the access to the IoT message broker.

The FIWARE Authzforce PAP is what enables the creation of policies using the XACML standard specification. XACML divides the policy creation into 4 hierarchies: domains, policy-sets, policies and rules.

Rules are the core component of a policy. Essentially each rule as a target and an effect. The target defines a set of requests to which the rule is intended to be applied in the form of a logical expression on attributes of the request. A target is defined using a set of logical tags, such as <AnyOf>...</AnyOf> and <AllOf>...</AllOf>, and a set of <Match>...</Match> tags. There can be several logical tags and several match tags inside of those logical tags. A match tag contains a attribute value, a comparison function and a attribute designator. The attribute value is a value set when the rule is created and it will be used together with the value from the request attributes in the comparison function. The comparison function uses the rule attribute value and the attribute value of the request and performs an operation that always returns a boolean value. The operations provided by XACML include: string-starts-with, string-equals, and many others. The attribute designator is used to specify which attribute of the request is required and used to match the rule.

Using the boolean result from each match and the boolean operations that surround those match tags a single boolean result is returned stating if the rule is or is not applicable to the request. If the rule is applicable, then the rule returns the result specified on the effect, otherwise returns the opposite of the effect. The effect of each rule can only be Permit or Deny.

Policies are sets of rules that together return a Permit or Deny. The result of a policy is based on the result of each rule and the rule combination algorithm selected for the policy. Policies also have a target, similarly to the rules, that is used to restrict the policy usage to a specific purpose. If the target matches, then the rules are called and the combination of the rules result is returned using the rule combination algorithm, otherwise a error is returned. XACML provides an extensive list of combination algorithms, but here we focus on deny-unless-permit and permit-unless-deny. Deny-unless-permit by default returns deny, even if a rule returns some error. The only way the deny-unless-permit returns ″Permit″ is if some rule returns ″Permit″. Permit-unless-deny is the opposite of deny-unless-permit meaning that by default the result is to ″Permit″ unless some rule returns ″Deny″.

Policy-Sets are sets of policies that determine the policies in place for an organization **domain**. Each **domain** as a specific **policy-set** in place. Like policies and rules, a target is also defined on a policy-set with the purpose of restricting the policy-set usage to a specific set of attribute conditions. If a policy-set target matches, then the return of the policy-set is the combined result of all the policies inside the policy-set using a policy-combining-algorithm defined for the policy-set, in a similar way to what happens with policies and rules. The return of a policy decision request is the return of the policy-set.

The FIWARE Authzforce PDP is the service that allows the policy decision based on the attributes on the request and the identity of the requester. The policy request places a set of attributes on the request. Each attribute as an category and id, that will be used by the PDP to match the attributes received with the attributes required on the rules attribute designators. XACML provides a large characterization of attributes from subject-related ones, like subject-id, to resource and action-related ones. To each attribute a value is appended, the value appended will be used by the rules together with the rule attribute value and the comparison function to return a result that will grant or deny access to the resource specified.

In the appendix A there is an example of the format of rules, policies, policy-sets and policy decision requests using the XACML markup language.

Since the objective of the authorization service is to prevent the access of data from other pools while still granting access to the data marked as public we decided to create a domain for each pool appended to the smart city. This domain is created when the sub-manager is provisioned into the smart city infrastructure and the name of the domain equals the name of the pool. Then having the domain generated, the city-manager will build a policy set with policies that require that the client device creating some IoT message broker entity needs to be in the same pool as the entity. To provide a easier way to establish the connection between the pool where the data was originated and the entity stored on the broker we force that the entity id is in the format: pool-deviceName[::public]. The pool where the data came from must be placed on the start of the entity id and before the ”-” character, and must be the same pool as the client device pool. By forcing this with a rule destined to the resource that the client device is attempting to create/access we make sure that every entity on the IoT message broker is in that format and that the pool on the entity id matches the pool of the client device that created it. The ”::public” component when appended to the entity id signals the authorization server that the client is trying to access a public entity granting access to it. The deviceName is just the name of the device.

Subscriptions, as stated before allows client devices to be notified of entity updates. The policy rules on our architecture only allow the subscriber to subscribe to entity ids that start with the same pool as the subscriber pool name, or entity ids marked as public. This way mitigating the access to information of other pools that is not marked as public.

Subscriptions also allow the set of a idPattern, which uses regular expressions to allow the subscriber to subscribe to updates from more than one entity at the same time. To still be able to protect the access to data belonging to other pools, we restrict the idPattern values to be ”pool-.*”,”.*::public” or ”pool-.*::public” being *pool* the name of the pool of the subscriber. This way we allow, respectivelly, the subscriber to access information about every entity belonging to the subscriber pool, to all the public entities and to all the public entities in the subscriber pool.

Since we use data coming from the sub-managers and client-devices, such as pool name and device name, to produce policies and execute queries to those policies, and data coming from these devices might have malicious XACML code on that input that defeats the policies and allows unrestricted access to all the entities on the IoT message broker, we need to deploy countermeasures to mitigate malicious inputs. We mitigate that by blocking every request that contains characters outside of the a-zA-Z0-9?.:*?-_ set, blocking the introduction of XACML tags that could make the policy creation and policy decision query vulnerable.

In order to improve the availability of FIWARE Authzforce, its authors provide a way to have multiple instances of FIWARE Authzforce running, the synchronization between instances is done using a external storage file-system, such as Network File System (NFS) or Ceph, that allow to mount virtual drives on the FIWARE Authzforce instances on which every file change on one instance is immediately made available for all the other instances. Similarly to what happens to the endpoint device replicas, we need a way to forward the requests to the FIWARE Authzforce replicas in a way that consumes the resources equally. To do that, we deploy a load balancer.

5.2.6 CRL Server

The Certificate Revogation List (CRL) server, stores information of previously provisioned devices that currently have their certificate revoked, meaning that they are no longer trusted and the communication of these devices to the smart city infrastructure must be denied. The CRL server uses the on-premise database persistency and consistency properties to safely store information about certificate revoking.

Only the sub-manager that provisioned the device should be able to revoke devices from the pool managed by it, this way mitigating problems were another sub-manager from another pool attempts to revoke devices from other pool, resulting on unauthorized access and blocking the access to the revoked devices to the smart city services. We achieve this result by a combination of digital signatures and the authentication provided by the *YubiAuthIoT* communication protocol (section 5.2.2).

When the sub-manager is first provisioned, the city manager creates a table on the CRL server containing the pool-name and the manager certificate information. When the sub-manager revokes a certificate on its pool it sends a digital signature of the hash digest of the certificate to be revoked together with the certificate itself. The CRL verifies if the pools of both sub-manager that issued the revoke and the device to be revoked are the same, meaning that the sub-manager is revoking a device that it manages. If the pool verification succeeds then the CRL server uses the stored sub-manager certificate to verify the hash digest digital signature to check if the sub-manager truly issued the request and to give non-repudiation guarantees, meaning that the sub-manager cannot dispute the authorship of the revoke request. After the verification, the CRL server stores the revoked certificate entry into the sub-manager pool revogation table.

When a device wants to verify if the certificate received from a peer is valid to establish communication, it first checks the expiration of the certificate and if the certificate is not expired. It then sends the certificate received to the CRL server. The CRL server when receives the request, it accesses the table belonging to the pool where the certificate belongs and checks if the hash digest of the certificate is stored on that table. If the hash digest is stored then the certificate is revoked and the communication to that device is interrupted, otherwise the device can continue the communication securely.

As said previously, a unique instance of a CRL server running can cause availability issues since if that unique instance becomes unavailable the access to the CRL server services is denied, which in turn blocks all the communications from devices on the smart city platform. To help improve the availability of the CRL server we resort to the same mechanism used to improve availability of the endpoint device. Instead of having a single instance running, there will be multiple replicas of the CRL server running at the same time, distributing the load across all of them and reducing availability issues if some replica becomes unavailable, since all the remainder replicas will be able to respond. To control the distribution of the load on the replicas and to provide a single address to access the CRL server we also deploy a load balancer that connects to the CRL server replicas and distributes the load equally among the replicas.

5.2.7 MongoDB Database

In the process of sending a file to the off-premise storage, HMAC digests, symmetric keys and IVs are generated and need to be stored. That is the job of the metadata database. In order to reduce the amount of components that the smart city infrastructure needs, we decided to use the same mongoDB cluster that is used to store information related to the IoT message broker operation, but using a different database than the database used by the message broker. That database can only be accessed by a specific user and password combination, protecting the metadata database from attacks that compromises the IoT message broker database and then attempts to reach all the other remaining databases.

The MongoDB Database is an essential component to the smart city infrastructure since it is a dependency of the IoT message broker and it is used by the metadata database to store the metadata related to the upload of data off-premises. However, by running a single instance of a mongoDB database introduces a single point of failure that directly affects the quality of service of the entire smart city infrastructure.

To improve the availability of MongoDB, we use a replica-set cluster. A replica-set in MongoDB refers to the usage of multiple MongoDB instances that together manage the same data set [70]. In a replica-set, there are a set of data bearing nodes and one optional arbiter node. The nodes agree between themselves to elect a data bearing node as primary. The primary node is the node that receives read and write requests, performing the operations locally and then replicating them on each one of the remainder data bearing nodes, called secondaries. The replica-set nodes send heartbeats between each other, the heartbeats are used to ensure that the primary and secondary devices are active. If the primary does not respond to the heartbeats for more than a specified time, one of the secondary devices calls for an election to nominate itself as a new primary. When the election succeeds, a new primary is elected with majority and the cluster resumes normal operation. This process is called automatic failover. However, depending on the number of secondary devices, an election might never result in a majority, making that two or more secondaries think that they are the primary because they had the same number of votes and with the higher number of votes individually. To prevent this the arbiter is called to elect a primary from the highest voted devices, enabling the choice of a primary with majority.

Since Ceph, our scalable and highly available on-premise storage, can be mounted as a virtual hard drive, we use it as the storage for each mongoDB instance on the replica set, reducing the need to set specific storage to each mongoDB instance, reducing costs and increasing manageability.

5.2.8 Off-Premise Storage Controller

The off-premise storage controller is responsible to upload older data to the off-premise storage and perform all the necessary erasure coding, encryption and HMAC validation/-generation (Fig. 5.1) to increase confidentiality, integrity and reliability of the data stored.

Real time services, such as real-time traffic monitoring, constitute the largest majority of the application of IoT sensors on smart city scenarios. Real-time services by their nature require the most recent data in order to operate correctly. Other services, like artificial intelligence model training, require both old and recent data to perform their operations correctly.

Having services that can require older data, it must not be deleted since it is still required by some services, however, the access to that data does not need to be as fast as the recent data. If we store all the old and recent data on the on-premise storage, the storage needs on the on-premise storage will be ever increasing, increasing costs since there will be need to set more OSD to cope with the storage needs. To keep older data stored but to not increase the costs with the on-premise storage the off-premise storage controller sends data older than 1 day to the public clouds storage, deleting it from the on-premise storage.

To reduce the amount of requests to the public cloud storage we do the following: after the end of the day n of collecting data, upload to the off-premise storage the data of the day $n-1$ and delete it from the on-premise storage. This way it is always sent the data of an entire day of collecting data, reducing the amount of storage requests from multiple a day to just one. The data sent is always older than one day preserving fresh data on-premise.

In an effort to reduce the amount of identifiable information present on the uploaded data but without decreasing the utility of that data, attributes like entity id and pool name are removed from the uploaded data. Other attributes, like location and collection timestamps are stored unchanged.

5.3 Client Device Layer

The client device layer contains all the IoT devices that are connected to the smart city infrastructure. The devices can be owned by the city itself, companies or by the individual citizens. The client device layer responsibilities include sensing the environment, publishing the data obtained from the sensing process, and request data published by other IoT devices for their own needs. A perfect example that illustrates the two responsibilities of a IoT client device is the following:

Imagine a Smart Traffic Management System (STMS) deployment, where every vehicle has a IoT client device connected to the smart city infrastructure sending periodic location updates to the smart city. The location updates provides information about the location where the vehicle is and the amount of time that a vehicle spends on a location, enabling to verify if a traffic jam is occurring at a certain location, by checking if for a large quantity of vehicles near one another are taking a large amount of time to move some distance. When a citizen wants to know if there is a traffic jam forming on a certain street on real-time it uses a client device (i.e. smartphone) to query the smart city for location updates on the vehicles that are on the street, processing the differences in location between updates to calculate if a traffic jam is forming or not, notifying the citizen accordingly.

In this example the devices that are publishing data are the ones on the vehicles, and the devices using the data are the citizen client devices. Additionally, it is is possible in our architecture to have client devices that both publish and access data, performing the two responsibilities on the same client device.

The communications between the client devices and the smart city infrastructure are done using the *YubiAuthIoT* secure communication protocol, and for the client devices to be able to communicate with the smart city they need to be provisioned first using the *YubiAuthIoT* provisioning protocol. This means that every IoT device belongs to a specific sub-manager or belongs directly to the smart city. On a client-client, or client-server, communication certificate revocation verifications and policy verification are made between authentication phase of the communication phase of the *YubiAuthIoT* communication protocol, stopping the communication if the certificate is revoked. Figure 5.10 provides a visual representation of the client communication used in this dissertation.

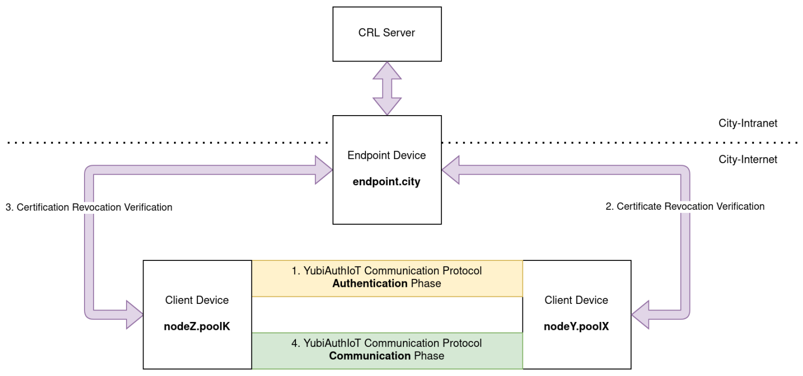

CRL Server
City-Intranet
City-Internet
Endpoint Device
endpoint.city
3. Certification Revocation Verification
2. Certificate Revocation Verification
Client Device
nodeZ.poolK
1. YubiAuthIoT Communication Protocol
Authentication Phase
4. YubiAuthIoT Communication Protocol
Communication Phase
Client Device
nodeY.poolX

FIGURE 5.10: Client Communication

5.3.1 Location Sharing

Our architecture enables to IoT devices to send and receive data about their surroundings, enabling for a more efficient planning and usage of city resources. Despite enabling the transmission of data other than location data, in this dissertation we focus on the perks and downsides of location sharing on both utility of the system and impact on the privacy of the individual citizens.

In our architecture we provide three types of location sharing: Public, Pool-Private and Anonymized.

5.3.1.1 Public Location Sharing

City services like public transportation are used every day by a large amount of the city population, since it provides a cheap way to travel large distances, increasing the mobility of people and helping the city economy. Current public transportation deployments often suffer scheduling problems, where public transports, such as buses, arrive late at their stops, increasing citizen dissatisfaction and reducing quality of life. Since there is no easy solution to the delays on public transportation, one of the mitigations of citizen dissatisfaction is to provide real-time schedules that use the real-time location of public transportation vehicles to predict the time that they will arrive at their respective stops. This way the citizens can have a real-time view of the public transportation vehicles, improving their quality of life since they know the exact time when the public transport will arrive.

In our architecture, this can be achieved using public location sharing, meaning location data accessible by every authenticated device without any additional access policies. To do this the client device that is publishing public location data marks the data as public data by appending "::public" to the entity id on the publishing. A device that has subscribed to public data updates receives the location update immediately after publishing, enabling that device to have a real-time updated location of the public transports. Locally, having the location of the device, it is possible to calculate the distance between the public transports and the citizen location to estimate the time of arrival of the public transport to the stop where the citizen is located.

The public location data is stored on the on-premise storage as it arrives and it is then uploaded to the off-premise persistent storage when it gets old. The data stored can then be used to feed analysis tools, such as traffic analysis tools, that help to understand traffic flows within the urban environment[71] providing information for better city planning and efficient usage of existing resources improving quality of life of all citizens.

5.3.1.2 Pool-Private Location Sharing

Sometimes there is a need to share location data within the pool itself without making that data available to every device on the smart city infrastructure, to do so the client device that publishes data only needs to not append "::public" to the end of the published entity id. In our architecture, the endpoint device will block by default all the subscriptions that attempt to access data that does not belong to the subscriber pool or are not marked as public.

There are several examples of the applicability of the pool-private location sharing, such as company vehicle tracking, in which a company sets client devices on the company vehicles to check in real-time where the vehicles are without making the information about the vehicle location publicly available to everyone.

5.3.1.3 Anonymized Location Sharing

When a client device needs to share its location data to provide the city infrastructure with data that then will be used for multiple purposes, from traffic monitoring to track the movement of individual vehicles or people, the sending of the real location data can cause privacy issues, especially if the client device is a citizen owned one.

To mitigate the privacy loss provided by data sharing we introduce client side location data anonymization using the location anonymization techniques described on the Related Work (3.2.4). The client chooses which technique and the parameters to apply according to the privacy and utility considerations. None of the services on the access layer apply any additional processing to the location data, meaning that the client device has full control over its data.

Anonymized Location Data can be applicable on every service that does not require a precise location, like locating restaurants based on a radius from the point where the user is or finding public transportation near the user location.

By doing the anonymization on the client side, no private information reaches the servers, only the anonymized result. Other advantage is that the client device owner or sub-manager can set the anonymization parameters that better suit the privacy and utility needs of the client device stakeholder.

The anonymized location sharing can be coupled with the pool-private or public location sharing to control scope of the data sharing allowing the sharing of the anonymized location data with public or only within the pool.

5.4 Complete Architecture

Having discussed the several layers of our architecture, we verified that the access layer possesses a large amount of small interconnected services that together grant the authenticity, integrity, confidentiality, availability, authorization and privacy capabilities to our architecture. In this section, we aim to provide a complete view of the component integration while also attempting to reduce the number of possible duplicated components, such as load balancers, that are required by more than one service.

5.4.1 Service Clusters

In an effort to reduce the amount of components needed to build the smart city platform, such as the number of load balancers for each service, we decide to group components together in service clusters. The service clusters take advantage of the architecture service replicability and provide each cluster with the necessary service replicas to perform a specific set of tasks.

For example, services like the authorization service and the CRL server need the authentication provided by the endpoint device in order to function, it will make sense to fuse all those components into one cluster since it will reduce the amount of load balancers needed for each service, reducing costs and improving manageability. Using a service cluster also improves horizontal scalability since to scale the deployment, an administrator only needs to launch another service cluster instead of replicating each component individually.

It is important to state that the synchronization of services inside of a cluster is done by using storage and databases external to the service cluster itself. However conflicts like write-write, on which multiple services simultaneously attempt to write data on the same file, or read-write, on which some services are reading on something that is simultaneously behind written can still cause synchronization problems between replicas or services.

Modern databases, such as MongoDB, introduce concurrent write-write conflict protection by deploying techniques such as locking [72], that ensure that a write operation is done fully without any other write operation being allowed while a previous write is not yet terminated. The read-write conflicts are resolved in a similar manner, the access to the new data is locked until the write is successful, reducing the risk of reading corrupted data.

Ceph[46], the technology that powers our on-premise storage, provides synchronous Input/Output (I/O) by making writes atomic, meaning that only one write to a specific file is allowed at a time, blocking any other operation (reading or writing) thus mitigating write-write conflicts. Reading on synchronous I/O also blocks any writing until the reading operation succeeds mitigating read-write conflicts.

5.4.1.1 Endpoint Service Cluster

The endpoint service cluster comprises a endpoint device, CRL server, FIWARE Orion and FIWARE Authzforce replica. Figure 5.11 depicts the endpoint service cluster and its communication with services external to the cluster by internal to the city intranet, such as mongoDB and Ceph.

When a request originated from an client device reaches the endpoint service cluster, the first component to interact with that request is the endpoint device. As said previously, the endpoint device is the bridge between the city internet, where client devices are, and the city intranet where the city services are placed. The endpoint device then checks if the request needs to be forwarded to the CRL server or if it needs to be forwarded to the FIWARE Orion. In case of the later, the endpoint extracts information about the requester identity and the resource that it is trying to access to construct a policy decision request

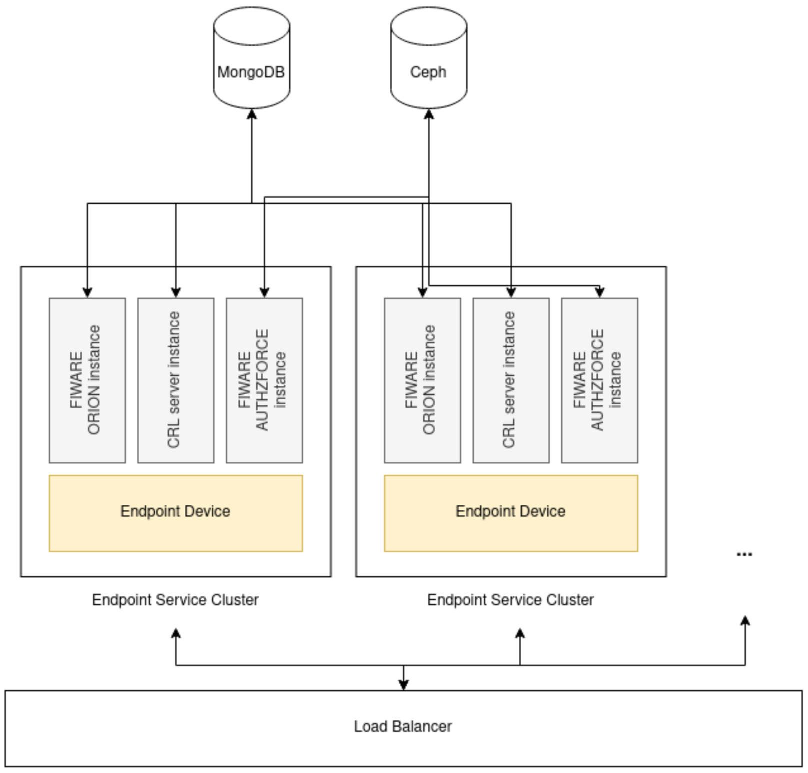


FIGURE 5.11: Endpoint Service Cluster

that forwards to the FIWARE Authzforce, which returns "Permit" or "Deny" based on the policies in place. On a "Permit" response, the endpoint device forwards the request to FIWARE Orion, otherwise returns a error. The response from FIWARE Orion is forwarded to the device that made the request via the endpoint device. The client device to endpoint device communication protocol is *YubiAuthIoT*, discussed previously.

The access to components other than the endpoint device is only allowed from the cluster itself, external access is blocked. This is done to enforce that the request is first authenticated by the endpoint device and then if a request is destined to the FIWARE Orion the endpoint device verifies if the request is authorized near the FIWARE Authzforce instance. If the request is authorized the request is then forwarded to the FIWARE Orion which does the necessary processing, returning the result to the endpoint device that forwards the response to the client device.

By grouping all of these components into a service cluster, the amount of load balancers needed on the architecture decreases and the manageability of the system increases, since instead of launching a new replica of each individual service each time that the availability requirements demand for it, a new service cluster is launched, containing all the services within it.

5.4.1.2 City Manager Service Cluster

The city manager service cluster comprises the components needed for the city manager to perform the tasks discussed previously: Provision sub-manager devices, create policies, and initialize the sub-manager pool Certificate Revogation List (CRL). To do that, the city manager connects to the FIWARE Authzforce and to the CRL server to perform those operations. Figure 5.12 provides a visual representation of the city manager service cluster. Similarly the access to the FIWARE Authzforce and the CRL server instances are not accessible from outside of the cluster, the only component that is accessible from outside of the cluster of the city manager, this mitigates vulnerabilities where an attacker attempts to directly connect to the FIWARE Authzforce instance and alter the authorization policies to allow it to access all the IoT message broker entities regardless of the pool they belong.

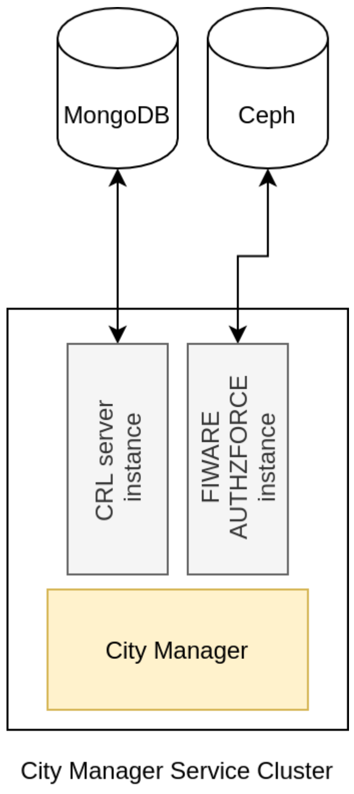


FIGURE 5.12: City Manager Service Cluster

5.5 Putting all together

With all of the components that integrate our architecture described, the figure 5.13 aims to provide the visual presentation of the integration of all the components, including service clusters.

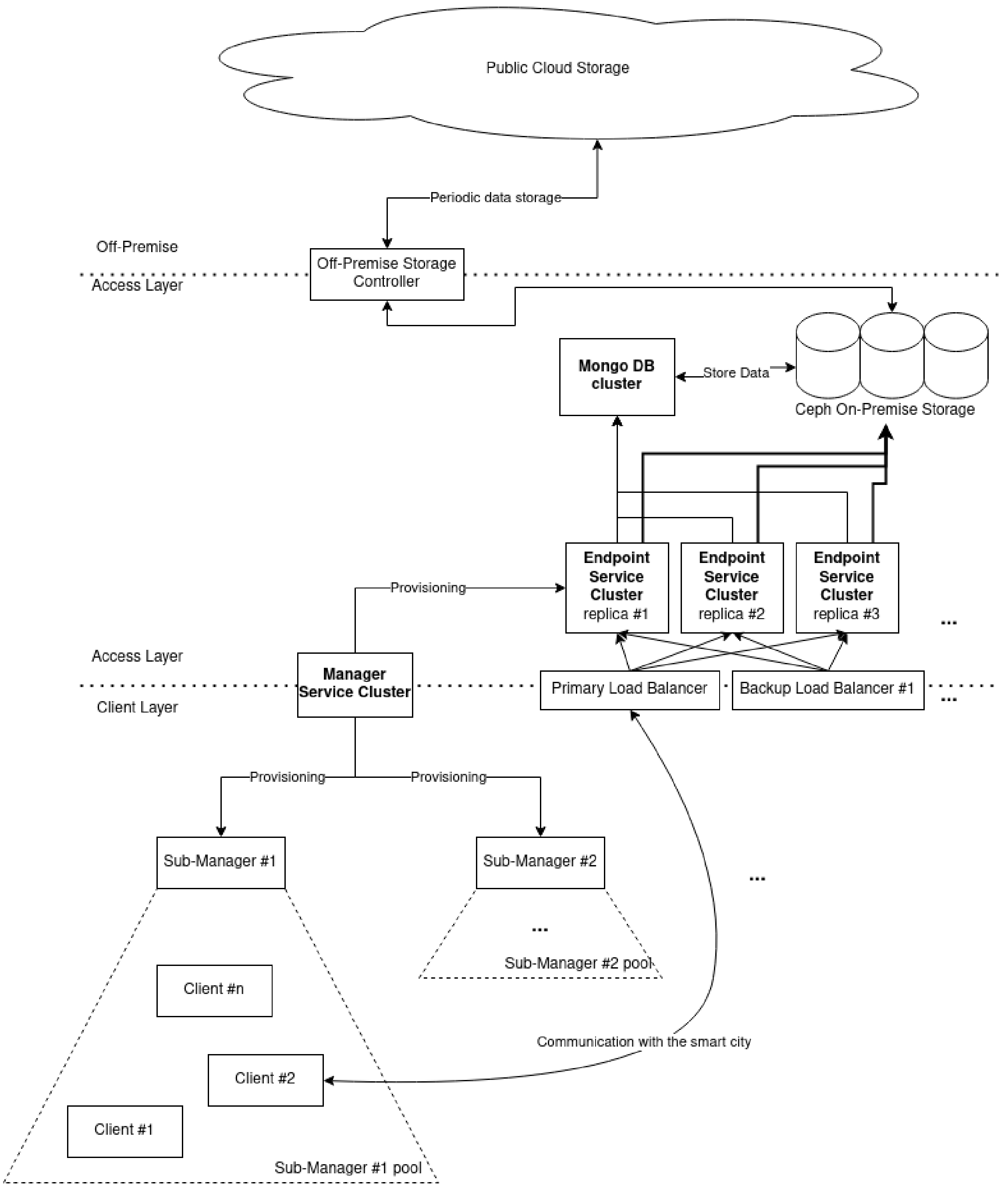


FIGURE 5.13: Complete Architecture

Chapter 6

Implementation

In this chapter we aim to provide details about the technologies we used to build a prototype of the architecture described in chapter 5. We divide the implementation into 3 sections, presented on this chapter on ascending order of execution, meaning that the first section was built first and the last section was the last to be built. Each section will discuss how a set of components of the architecture were built, discussing the technologies in use and the justifying their usage by pointing the advantages of each technology in the context of our architecture and proposal.

6.1 Off-Premise Storage

We started by verifying technologies that enable us to use a cloud-of-clouds configuration for the off-premise storage, mitigating availability issues and reducing the sovereignty data each cloud provider has over the published data. We base our off-premise storage offloading on ARGUS [6] that enables us to offload data to a set of cloud providers, implementing erasure coding to enable the access to the data even if some cloud providers gets inaccessible. Erasure coding improves traditional data replication, since it allows to access data even when a cloud provider is inaccessible while storing less data, in comparison to complete data replication, to achieve that guaranty.

The off-premise storage controller is the component on our architecture responsible for the offloading of data to the cloud storage. The off-premise controller uses the JClouds[73] API to provide connectivity to the most popular cloud providers, such as Google Cloud Provider (GCP), Amazon Web Services (AWS) and Microsoft Azure, abstracting the individual cloud behaviour into a single consistent API that enables us to easily connect to cloud providers, and to easily swap cloud providers, if needed. JClouds only offers its API to the Java language, for that reason , we choose to use Java as the language to build our off-premise storage controller. To easily manage the dependencies that the off-premise storage might need and to automate compilations we use Apache Maven[74] as our dependency management and compilation tool.

When we first started building the off-premise storage, it was going to connect at the end of each day to the mongoDB cluster and gather the Orion entity updates related to the previous day, merging all the updates into a file, and then upload it to the cloud providers. However, upon analysis of the inner workings of FIWARE Orion we noticed that Orion does not store information about the received entity updates, the updates when arrive are forwarded to the active subscriptions that wish to receive notification about those updates. The information after being forwarded to the subscriptions is discarded and not stored. Upon inspection of FIWARE Orion configuration, we came to the conclusion that FIWARE Orion does not provide any configuration option to store the entity updates locally or in the mongoDB database [75].

Since we need the ability of storing the entity updates to provide services based on data analysis, that require both recent and older data to provide efficient and more accurate predictions, we created an on-premise subscription that receives updates on all the entity updates by using *idPattern=".*"* on the subscription. The subscription to be accepted must be directly sent to a FIWARE Orion instance running on-premise, otherwise the request will be rejected by the access control policies that protect the access of city-internet devices to the city-intranet resources. This behaviour is similar to how the FIWARE Cygnus persistency enabler [76] connects to FIWARE Orion to receive the updates [77].

It is important to say that in our architecture all the components of the city-intranet are assumed to be trusted, and due to that, the subscription created by the off-premise storage controller - that is a city-intranet component - with the *idPattern=".*"* does not impact our security model for the architecture.

In order to be able to receive the notification, the off-premise storage controller deploys a small HTTP server that listens on the city-intranet for the notifications generated by FIWARE Orion on a entity update. The HTTP server is built using the Spring Boot framework in Java. We choose to use Spring Boot since it is full-featured web server, with documentation available and a large community support.

When the off-premise storage controller receives an notification from FIWARE Orion it looks at the timestamp of the notification and appends it into a file that is used to store all the updates from the same day as the day marked on the timestamp. This will generate large files where each one of the files only contains data related to a single day. When the time comes to offload the data of a day to the off-premise storage, the file containing the data of that day is gathered and uploaded. It is important to state that the off-premise storage controller will uses the on-premise storage provided by Ceph to store those files before uploading to the off-premise storage.

The upload process consists of splitting the file into fixed size chunks, we choose to use chunks of size 20MB since, from our performance testing and cloud storage price analysis it provided a good trade-off between performance and cost. Figure 6.1 shows a graph of our findings.

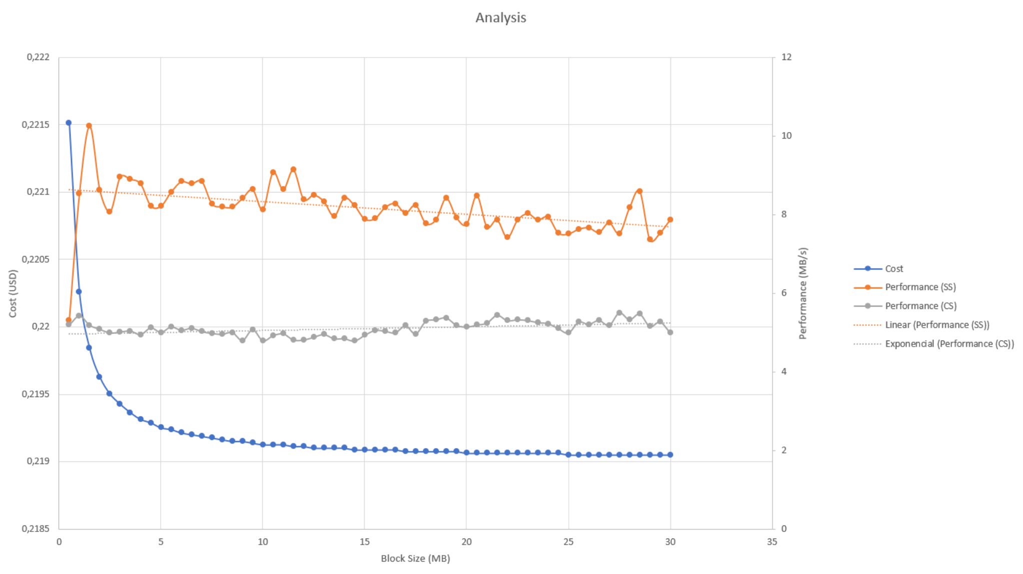


FIGURE 6.1: Performance and cloud storage cost analysis of the available implementation of ARGUS in client-side(cs) encryption and server-side(ss) encryption modes, using multiple chunk sizes. The file used to upload and download is a 1024MB file

The cloud storage costs on the most popular cloud storage solutions is based on two main metrics: the amount of requests made to the cloud storage and the amount of data stored. For our costs calculations we use the available storage cost description for standard storage buckets on Google's GCP[78], Amazon's AWS S3[79] and Microsoft's Azure [80] websites. From our analysis, the amount of requests is directly dependant on the chunk size selected, the smaller the chunk bigger the costs related to the amount of requests. As we increase the chunk size, the costs related to the number of requests reduces significantly, being almost unnoticeable. The cost associated with the amount of data stored is constant, regardless of the chunk size. In terms of performance impact, the bigger the chunk size, the heavier is the hit on the erasure coding - that on ARGUS is always processed on the server side, reducing the throughput of the data offload. The mode of operation of ARGUS that will resemble the operation of the off-premise storage controller is the server-side encryption, where encryption, integrity and authenticity calculations and erasure coding are done on a single machine. In the client-side encryption the encryption and integrity and authenticity verifications are calculated on another device.

To perform encryption we use AES in the CBC mode with a 256 bit key. We choose to use this because it is a well supported algorithm, being available in almost all the cryptographic suites. To ensure authenticity and integrity to the data uploaded, we use Hash-based Message Authentication Code (HMAC) since they are a efficient and well supported mechanism to ensure those guaranties.

6.2 On-premise

Having the off-premise storage dealt with, the next step is to build the on-premise infrastructure. As it is possible to see on the complete architecture (fig. 5.13) the on-premise of our architecture comprises the large majority of components. The manual management of each one of the components can be complex when notions of service replicas are introduced into the system, to mitigate the complexity we decided to use Kubernetes as our orchestration tool.

6.2.1 Kubernetes

Kubernetes is a container orchestration tool built for highly dynamic application needs [81], as it helps with the deployment of services across a cluster of machines, ensuring the availability and scalability of the services deployed. Kubernetes, to achieve this, developed an architecture comprised of 5 essential components: control plane, node, pod, deployment and service.

The control plane is a set of machines that together manage a Kubernetes cluster. The control plane is also responsible for deploying containers into the Kubernetes cluster by allocating them to the Kubernetes node that has less consumed resources in a process called scheduling.

A Kubernetes node is a machine where the deployed containers are scheduled and run. Each Kubernetes node on the cluster has two essential pieces of software running: a container runtime - such as docker - and kubelet. The container runtime is the environment on which the pods are deployed and kubelet is the software that receives the pod scheduling configuration from the control plane and ensures that the pods scheduled for that node are deployed and are running healthy, notifying the changes on the state of the deployed pods to the Kubernetes control plane, that if needed reschedules the pods to another node.

Pod is the smallest unit on the Kubernetes architecture and comprises a container and a abstraction layer that enables to control the container independently of the container runtime on each of the Kubernetes nodes. Pods are units that are dynamic, meaning that they can be deleted, re-scheduled and re-created at any given time due to some misbehaviour of the container or due to the lack of resources on the node hosting the pod. Pods given their dynamic nature, are ephemeral and no data that is required to be persistent should be stored on them. The lack of internal data enables pods to be easy replicable to achieve service availability to an arbitrarily large number of clients.

Deployments in Kubernetes are a set of instructions that specify how a pod is created. Deployments allows to set the container base image, the environment variables, mount points of external volumes or volumes requested via a Persistent Volume Claim (PVC), number of replicas of that pods, etc. To each deployment an identifying tag can be added to make it easy to manage and to attach Kubernetes services to all the pods created via the deployment.

Since pods can be created, scaled and destroyed at any time, it is very difficult to provide a service inside a pod that other pods or external clients can access. The principal problem is that each pod receives a IP address that can be inserted into the pods that want to access the service and that grants access to the service on the pod, however, due to dynamics of pods the IP address can be changed any time to cope with a pod deletion and re-scheduling, preventing the access to the service inside the pods since the other pods did not know that the IP changed.

A Kubernetes services allows to define a static way to access services on pods by providing a static DNS name, that returns the IP of the pod replica that is the most available to answer the request, using load balancing techniques. When the IP of a pod changes the service is notified and it alters the DNS name accordingly. This way all the pods that want to access the service on the pod only need to know the DNS name and the port where the service is listening in, to be able to always access the service without worrying about the dynamic nature of the underlying pods.

Depending on the access of the service (public, internal), Kubernetes defined 3 essential types of services: ClusterIP, NodePort and LoadBalancer, that will be discussed below:

- **ClusteIP** Exposes the Service on a cluster-internal DNS name, making the service only reachable to pods internal to the cluster.
- **NodePort** Opens the selected port on every kubernetes node in the cluster, enabling the service to be accessible from the outside of the cluster, however there is no load balancing between the nodes on the access to the service.
- **LoadBalancer** Exposes the Service externally using a cloud provider's load balancer, enabling to have a external load balancer that manages the load on the kubernetes cluster nodes.

LoadBalancer is considered the best type of service when there is a need to expose the service to outside of the cluster. However, it only works when the cluster is on a supported cloud provider, what is a problem when there is a need to create an on-premise, bare-bones, cluster that is typical of smart city scenarios.

To use the same functionality of the LoadBalancer service type without setting the cluster on a supported cloud provider, we use a combination of the NodePort service type and a set of external HAProxy[82] load balancers in a fail-over configuration using a virtual IP and keepalived[83]. The NodePort gives us open ports on every one of the Kubernetes nodes on the cluster, that directly communicate with the internal service load-balancing and gives access to the services provided by the pods.

We could place the public IP addresses of each one of the Kubernetes nodes, on multiple A DNS records pointing to the same DNS name, however, this makes the clients responsible for selecting IP associated to an Kubernetes node in a way that the resources on each node are equally consumed. This solution can cause availability issues, since malicious clients can decide to always connect to the same Kubernetes node, consuming all of the node resources that when overloaded can shutdown forcing the Kubernetes cluster to rebuild the lost pods, resulting in possible down-time.

One fix to this problem is to deploy an load balancer that is external to the Kubernetes cluster but is in the same internal network that joins all the Kubernetes nodes. The load balancer will be responsible to connect clients to the Kubernetes node in a way that the connections are equally distributed across all the Kubernetes nodes. We choose to use HAProxy as our load balancer since it is a well-documented load balancer with an wide adoption and capability of working with newer communication protocols such as google Remote Procedure Call (gRPC), that will be used on the communications of *YubiAuthIoT*. The HAProxy is configured to use the least-connection load balancing technique that forwards the request to the Kubernetes node with the least number of active connections, thus equally consuming the resources of the Kubernetes nodes.

A single load balancer can be seen as a single point of failure to the system. In case of hardware failure or overloading, the load balancer is shut down and the access to the services on the cluster is blocked, reducing the availability of the system. To mitigate that we also deploy a backup load balancer with the same configuration as the primary load balancer, that is ready to receive requests if the primary load balancer fails. To provide a service where the clients do not need to select which load balancer to use we deploy a failover mechanism using keepalived and a virtual IP. Keepalived uses the Virtual Router Redundancy Protocol (VRRP) to attach the virtual IP to the load balancer that is currently running and accepting requests. If both primary and secondary load balancers are active the virtual IP attaches to the primary load balancer. The virtual IP is a public IP that is allocated to the smart city but does not belong to any network interface in specific. The attachment process is to place on the network interface that communicates with the city-internet a secondary IP that is the virtual IP. The result of this is that the router will forward the requests with the destination IP equal to the virtual IP to the machine that has the virtual IP as a secondary IP.

To generate the kubernetes cluster and to attach the nodes into the cluster we used kubeadm. Kubeadm[84] enables to connect multiple machines to one cluster and defines which are control plane machines or node machines, the only requirement is that they need to be accessible from one another through any network.

In our implementation prototype we used 8 virtual machines and one public IP address: 2 machines for the primary and secondary load balancers, 1 load balancer for the control plane - required by kubeadm -, 2 machines for the control plane nodes and 3 machines for the kubernetes nodes. Figure 6.2 depicts the general organization of the machines on the implementation prototype.

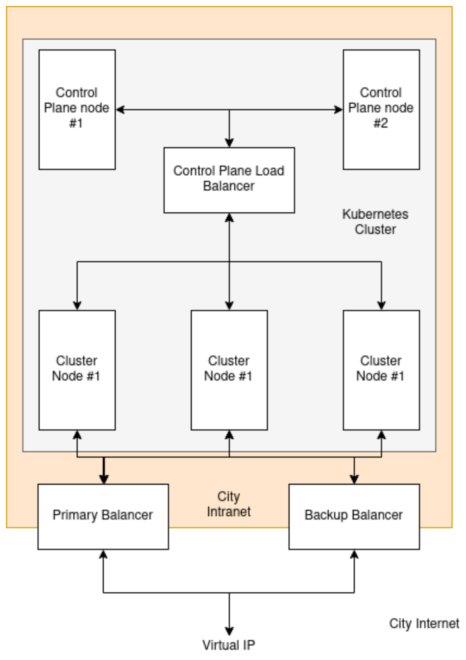


FIGURE 6.2: Organization of on-premise machines in our implementation prototype

It is important to state that this organization is the bare minimum to have highly available on-premise services. The prototype can be scaled to incorporate more machines into the Kubernetes cluster and more machines to be load balancers.

To be able to deploy the kubernetes cluster using kubeadm we first need to create a singular DNS name that points to the control plane load balancer, this DNS name will be used by the Kubernetes nodes to connect to the cluster and to receive and provide updates on the cluster state and scheduling. The control plane load balancer in our prototype runs HAProxy in a round-robin configuration, and it is only accessible from the city intranet. The IPs that the control plane load balancer balances are the internal ips of the two control plane nodes.

Having the hardware configurated on one of the control plane nodes, we start the initialization of the Kubernetes cluster by doing *kubeadm init* with the control plane load balancer DNS name and the option to upload the generated certificates to all the Kubernetes control nodes, to be sure that it reaches all the control nodes from the control plane load balancer DNS it is important for the load balancer configuration to be round-robin. At the end of this process all the control plane nodes have a copy of the certificates, the only thing left to do is to explicitly join them using the *kubectl join* commands that the *kubeadm init* outputs. The init command also outputs the join commands to add worker nodes - nodes that will have pods running on them - into the cluster.

When the *kubeadm join* command is executed on all the control plane and worker nodes the cluster is now ready, only needing to set the Container Network Interface (CNI) that will be used by Kubernetes to route the cluster-internal traffic. We choose to use Calico[85] as our CNI because it was one of the CNI implementation that worked more consistently over the many cluster delete and re-creation, that we made during the implementation process.

Having the Kubernetes cluster created and the load balancers in place, we will start to populate the cluster with on-premise services, starting from the on-premise storage.

6.2.2 Ceph

We chose to deploy Ceph on-premise storage first since it the component on the architecture that directly, or indirectly, is a dependency of all the other components on the access layer. Ceph is a complex storage system requiring a large quantity of different components working together to achieve Ceph's guaranties of scalability, availability, consistency and persistency. To ease the deployment of the Ceph components we used Rook's [86] Ceph open source storage deployment for Kubernetes. Rook provides a set of YAML files that contain configurations and deployment definitions that when applied into a Kubernetes cluster enables the cluster to have at least one OSD built per node, to have a controllable number of monitor nodes generated and to have a metadata server deployment capable of providing a shared file-system to all the pods on the cluster that require it via a PVC. In order for Rook to deploy OSDs on a node, the node must have a RAW - not formatted to any type like ext4 or NTFS -, physical hard drive attached to the node. The RAW drive drive will be formatted and used by the OSD pods to store the data created by the clients when using the Ceph storage.

6.2.3 MongoDB

Having Ceph deployed on the Kubernetes cluster, the next step is to build a Mongo DB replica-set on the Kubernetes cluster to provide a highly available database system. In MongoDB, all the members of the replica set must always accessible by the same IP and port configuration, something that is not possible due to the highly dynamic behaviour of the Kubernetes pods. To provide a static way to access a mongo DB pod, each pod will be created by a distinct deployment and each deployment will have a ClusterIP service, enabling each pod to have a static DNS entry and port combo that will be used by the mongo db replica set to elect a primary mongo db pod and to replicate data from the primary to the secondary mongo db pods. In our prototype we created three mongo db deployments, each one with a ClusterIP service attached to it. The DNS names of the ClusterIP service is the name of the mongo db deployment tag, in our case mongo0,mongo1 and mongo2. We then use mongo0 to execute the replica set initalization command (Listing 6.1). This command will start the replica-set initialization process, running and primary pod election and creating a new replica set with the id *rs0* that contains the three Mongo DB Kubernetes deployments.

```
rs.initiate( {
  _id : "rs0",
  members: [
      { _id: 0, host: "mongo0:27017" },
      { _id: 1, host: "mongo1:27017" },
      { _id: 2, host: "mongo2:27017" }
  ]
})
```

LISTING 6.1: Mongo DB replica set initialization command

At this stage, each one of the mongoDB deployment pods needs persistent storage, since due to the dynamic nature of the pods, data could be lost if the data is only saved on the pod. In order to have storage persistence on the mongo db pods, each pod connects to the Ceph storage using a Kubernetes Ceph Persistent Volume Claim (PVC). A PVC allows pods to have access a persistent storage volume, each PVC request creates an isolated volume from all the other PVCs.

Ceph then acepts the PVC and appends a new Ceph shared file-system to the mount point defined on the mongo deployment of each of the mongo db instances on the replica-set. It is important to note that every mongo db instance must use different PVC request, since, due to the nature of the mongo db replica-set replication, the primary pod replicates the writes to each one of the secondaries, and if all of the mongo db pods shared the same PVC the secondaries would be re-writing the primary data causing conflicts.

Having MongoDB and Ceph deployed on the Kubernetes cluster and accessible via ClusterIP services, the last components to create are the endpoint and city manager service clusters that connect to the mongoDB and Ceph deployments to be able to have persistence and synchronization.

6.2.4 Service Clusters

Endpoint and City Manager Service Clusters, as stated on section 5.4.1, places a replica of every service that endpoint device and manager device, respectively, need to work properly inside the same deployment. Since both service clusters use similar technologies, we describe first the technologies in use by each one of them and then specify the behaviour of each one.

6.2.4.1 YubiAuthIoT

YubiAuthIoT is one of the core components of our architecture, being responsible for providing device provisioning, authentication and secure communication. The functionality provided by *YubiAuthIoT* is based on public key cryptography and PKI management, OTP token management and symmetric encryption. *YubiAuthIoT* implementation prototype is built in Java since this way is possible to build simultaneously an API that can be used on regular desktop computers as well as Android smartphones with minimal configuration. In the *YubiAuthIoT* implementation we used Apache Maven as our package manager and automatic compilation tool.

In our implementation prototype we choose to use elliptic curves in our public key cryptography operations, since it provides the same security as RSA while using smaller key sizes[87], this way being more efficient on the cryptographic operations. To help us with the PKI management, which consists on the creation of Certificate Signing Request (CSR), the CSR signature and certification chain verification, we used the BouncyCastle library, since it provides easier to use, cryptographic operations that helped us construct the CSR creation, and CSR signature - producing a signed certificate - . We used ECDSA with 384 bit keys in the process of key pair generation and certificate signing.

It is possible to see on the architecture we use, in addition to the ECDSA certificate, the ECIES certificate. ECIES helps us to generate a shared encryption key without ever sharing that key through the network. In our implementation prototype we based the ECIES protocol on the implementation available on [88], that helped us to understand the core of the ECIES protocol and, using that information, to protect the ECIES protocol messages using digital signatures.

Once a symmetric key is established, the encryption algorithm used is AES with and 256 bit key generated using the ECIES protocol resulting key and the Password-Based Key

Derivation Function 2 (PBKDF2) algorithm using the methodology described on section 5.2.1.

Another important part of *YubiAuthIot* is the integration with OTPs, which allows the manager to perceive if a provisioning request should be processed. We use Yubico's Yubikey 5 NFC in our implementation prototype, since it was the hardware OTP token that we had available during the implementation. Using Yubikeys helps us on the OTP integration since they are a popular hardware token and have APIs built for several languages and platforms, such as Java and Android, that enables us to easily interact with the hardware token. The OTP token used in our implementation prototype is the YubicoOTP token that is provided by default on all Yubikeys.

The YubicoOTP usually requires the communication with the YubiCloud OTP server for the token verification. Yubico also provides a way to create a local OTP server, requiring to register each one of the Yubikey that the server will accept[89]. Any Yubikey is accepted on the OTP verification if YubiCloud is used, that property makes YubiCloud unusable on our implementation. Concluding, our only choice is to run a manager-local OTP server.

Being forced to deploy a local OTP server has several consequences, but the most important that there is no Java implementation of the Yubico OTP server, meaning that having a sub-manager running on an Android smartphone is not possible. In an effort to circumvent this issue we analysed the sources of existing open-source implementations on other languages, such as PHP[90] and the inherent validation protocol documentation[91].

To first understand how the validation protocol works we need to understand the structure of a Yubico OTP token. The Yubico OTP token is a 44 characters long, hexadecimal string, comprised of two sections: The yubikeyID and the encrypted OTP. The yubikeyID is a unique, 16 character long, identifier of the Yubikey, it is used by the OTP server to know which AES key to select from the database to decipher the encrypted OTP.

The encrypted OTP is a 32-character, hex-encoded string that when decrypted with the 128 bit AES key associated to the yubikey stored on the OTP server gives us access to the OTP information. The deciphered OTP is comprised of 5 fields positioned on the deciphered OTP according to the table 6.1 [92].

The private id is a copy of the private id field configured on the yubikey - the yubikey configuration process generates the yubikey public id, yubikey private id and the AES key where the AES is shared with the OTP server - and can be used as an identity for

Byte offset	Size	Description
0	6	Private (secret) id
6	2	Usage counter
8	3	timestamp
11	1	Session counter
12	2	Random number
14	2	CRC

TABLE 6.1: Yubico OTP deciphered OTP part field list

internal services. It is private because only the entities with the access to the AES key are able to get the private id field.

The usage counter is a non-volatile counter that is stored on the yubikey and gets incremented on every first use after a power-up or reset on the yubikey. The session counter is a volatile counter that initializes at zero each time the yubikey is powered up, and each time a new OTP is generated this counter increases by one, when the yubikey is powered down the counter value is lost. The combination of the the usage and session counter is what gives a unique counter to each generated OTP.

The timestamp is initialized at a random value when the yubikey is powered up, being incremented at a rate of 8Hz as long as the yubikey is powered, when a OTP is to be issued it is appended to the OTP the most recent timestamp so far. The timestamp can be used by the OTP server to detect the time elapsed between two subsequent OTPs received during a session, allowing to have custom behaviour depending on the time elapsed.

The random number is picked from the yubikey internal number generator as is used to add some additional entropy to the OTP generated.

Lastly, the CRC field is a 16-bit ISO13239 first complement checksum added to the end of the deciphered OTP part. The checksum is verified by calculating the checksum of all the 16 bytes of the deciphered OTP - including the CRC field -. The result of the checksum should be always the hexadecimal 0xF0B8 if the checksum is valid, meaning that the deciphered OTP is integral, otherwise the checksum is invalid and the Yubico OTP should be rejected.

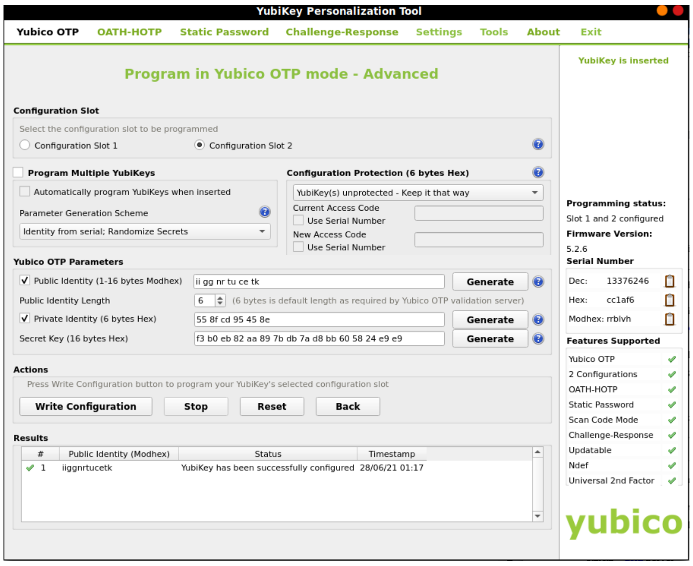
Having the knowledge of the Yubico OTP token is can more easily understand the OTP validation protocol. From our analysis of the OTP validation protocol and the available implementations we came to the conclusion that the OTP validation can be executed by doing the following steps:

1. First on a registration phase store the yubikey public id and AES key generated on the yubikey configuration process on a persistent storage. Create two fields that will store the received usage and session counters, initialized both at -1;
2. When a token verification is requested, get the first 16 characters from the token to get the yubikey public id. Knowing the public id, get the stored AES key associated with that public id;
3. Convert the remaining 32-character hexadecimal string into a byte array and decipher it using the AES key. From our analysis the AES algorithm in use is AES CBC;
4. Calculate the checksum using the CRC16 algorithm and check if the resulting checksum equals 0xF0B8;
5. If the checksum is valid, check if the usage counter equals the usage counter stored and the session counter is bigger than the one stored, or, if the usage counter is bigger than the stored usage counter;
6. If one of the conditions hold true, the OTP is valid and the stored session and usage counter are updated to match the ones on the received OTP

This can be easily mapped into *YubiAuthIoT*, on the token verification phase of the provisioning protocol, without requiring to setup an additional OTP server, simplifying the deployment of *YubiAuthIoT* on Android smartphones. Only requiring that the public id and AES key of all the yubikeys that will be accepted on the manager being configured. This process is only done once, since that data will be stored persistently on the configured device.

So far we talked about the Yubikey after the configuration but we did not mention how the Yubikey configuration is done. In our implementation prototype we use the Yubikey Personalization Tool[93] graphical environment to configure the Yubikeys. Figure 6.3 depicts the configuration of a Yubikey.

As it is possible to see, this tool allow us to easily set the public id, private id and secret AES key that will be used in the process discussed above to enable the verification of tokens produced by the configured Yubikey.



Program in Yubico OTP mode - Advanced

Configuration Slot

Select the configuration slot to be programmed

Configuration Slot 1 [✓ Configuration Slot 2]

Program Multiple YubiKeys

Automatically program YubiKeys when inserted

Parameter Generation Scheme: Identity from serial; Randomize Secrets

Configuration Protection (6 bytes Hex): YubiKey(s) unprotected - Keep it that way

Current Access Code
Use Serial Number

New Access Code
Use Serial Number

Yubico OTP Parameters

✓ Public Identity (1-16 bytes Modhex)	ii gg nr tu ce tk
Public Identity Length	6 (6 bytes is default length as required by Yubico OTP validation server)
✓ Private Identity (6 bytes Hex)	55 8f cd 95 45 8e
Secret Key (16 bytes Hex)	f3 b0 eb 82 aa 89 7b db 7a d8 bb 60 58 24 e9 e9

Actions

Press Write Configuration button to program your YubiKey's selected configuration slot

Results

#	Public Identity (Modhex)	Status	Timestamp
✓ 1	iiggnrtucetk	YubiKey has been successfully configured	28/06/21 01:17

YubiKey is inserted

Programming status:
Slot 1 and 2 configured

Firmware Version:
5.2.6

Serial Number

Dec:	13376246
Hex:	cc1af6
Modhex:	rrblvh

Features Supported

Yubico OTP	✓
2 Configurations	✓
OATH-HOTP	✓
Static Password	✓
Scan Code Mode	✓
Challenge-Response	✓
Updatable	✓
Ndef	✓
Universal 2nd Factor	✓

FIGURE 6.3: Configuration of the Yubico OTP

6.2.4.2 CRL Server

The CRL server is a simple server that is only responsible to create CRL tables for the pools that connect to the city-infrastructure, add CRL entries to the tables and to check if a certificate is revoked. Details about the security of creating, adding and verifying is discussed on the CRL server architecture on section 5.2.6. For the signature verification and certificate manipulations we use Java's built-in Java Cryptographic Extension (JCE) and the server is built in Java. In order to keep in use the same technology stack whenever it is possible we made the CRL server a google Remote Procedure Call (gRPC) server. We use a mongoDB database, password protected and isolated from the databases of FIWARE Orion, to store the information about the revoking of certificates. Each pool addition to the CRL server creates a document collection on the mongoDB database with the name equal to the name of the pool about to be added. On that collection a document is created containing the manager certificate information, the _id of this document is "managercrt", easing the search of manager information on the collection. When a certificate is revoked it is created a document on the pool collection containing the certificate subject and hash version of the revoked certificate signed by the pool-manager, ensuring authenticity and non-repudiation of the revoking. In a verification request the CRL server searches for documents on the pool collection that have the same subject name as the certificate to be verified, to all the documents found, the hash digest of the received certificate is compared with the hash digest stored on each one of the documents found, if a match is found the certificate is revoked and the communication should be denied. Otherwise, the communication should be allowed since the certificate is not revoked.

6.2.4.3 FIWARE Orion and FIWARE Authzforce

We decided to deploy these technologies using the high availability configurations available on the technologies respective sites. These technologies are only accessible directly from the City-Manager and the Endpoint Device - the off-premise storage controller does not require access to FIWARE Orion, the FIWARE Orion will send the notifications to it without the off-premise storage controller having access to FIWARE Orion -, any other access is required to go through the endpoint or city-manager device, this way ensuring that the proper execution of authentication and authorization flows.

6.2.4.4 City Manager Device

The City-Manager Device is a simple application which only has 3 functions: append sub-manager pools to the city, create CRL for the sub-manager pool and revoke sub-manager pools. The sub-manager appending process requires the creation of an new domain and policy set on Authzforce following the considerations discussed on the architecture (section 5.2.5), then generate a new domain on the CRL server that will house the certificates revoked on the sub-manager pool. If no appending is required the City Manager device can be turned off without breaking the functionality of the smart city infrastructure, it is possible to do this due to the choice of certificate based authentication as our authentication method, the authentication is provided by the *YubiAuthIoT* authentication protocol (section 5.2.1).

6.2.4.5 Endpoint Device

The Endpoint device, as described on the architecture (section 5.2.3.1), is the component that provides access from an external device to city-intranet services, such as FIWARE Orion, in an authentic, integral and confidential way with authorization checks built in. The Endpoint Device, similar to the City-Manager device has it core running on top *YubiAuthIoT*, with additional code to support forwarding communication with other components and perform access control verifications, verify CRLs and other necessary actions. To be able to do this we decide to create a modular approach where the developer creates modules that receive the input coming via *YubiAuthIoT* channel after deciphering and authentication process and outputs data that will be sent via the *YubiAuthIoT* in an authenticated, integral, and confidential manner.

In our current implementation prototype we have two modules: *FiwareOrionProcessor* and *CrlProcessor*. The job of the *FiwareOrionProcessor* is to gather the information about the request, create a policy decision request, forward the policy decision request to the Authzforce instance running on the service cluster via HTTP, process the response of Authzforce policy decision, create a HTTP request to to the FIWARE Orion instance running on the service cluster if the policy decision grants the access and lastly, read the result from Orion, and output the result that will be forwarded back to the requester using the same *YubiAuthIoT* communication channel used to perform the request.

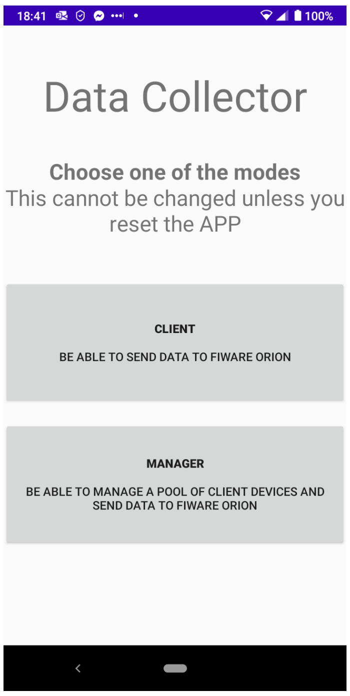
The *CrlProcessor* is a simple module that enables to communicate with the CRL server instance on the service cluster to perform CRL verification and certificate addition to the CRL and output the results that will be sent back to the requester using *YubiAuthIoT*.

6.3 Client device layer

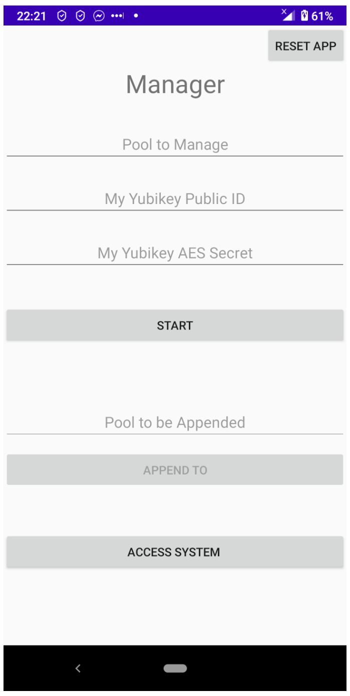
Having created the on-premise services, the next task is to create the clients that will use the on-premise service to send and received location updates. To have a prototype that incorporates the mobility properties of client devices that send and receive location updates, we choose to build the client using the Android Platform.

The Client Device Android App, is a simple app that uses *YubiAuthIoT* to power the sub-managers and client devices of our architecture. It is composed of 5 distinct android activities: Selector, Client, Manager, Location Sharing and Location Gather, and a background service to continue the location sharing when the application is closed. Figure 6.4 shows the 5 distinct screens.

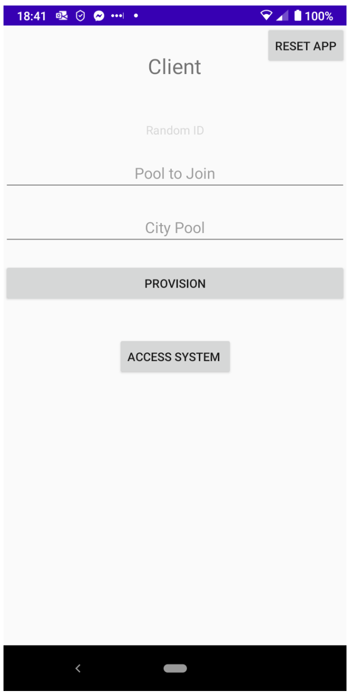
When the client app is first initialized the Selector Activity is displayed (fig. 6.4a), this activity allow to select which role the device will perform - be a sub-manager or a client device -. In our implementation, both sub-managers and client devices can share location data, but only the sub-manager can request to be appended to the smart city infrastructure and add client devices into the pool managed by it. In our implementation prototype, the pool policies only allow direct, client-client, communication between devices inside the same pool, direct inter-pool communication is blocked, this way isolating the pools.


18:41 100%
Data Collector
Choose one of the modes
This cannot be changed unless you reset the APP
CLIENT
BE ABLE TO SEND DATA TO FIWARE ORION
MANAGER
BE ABLE TO MANAGE A POOL OF CLIENT DEVICES AND SEND DATA TO FIWARE ORION

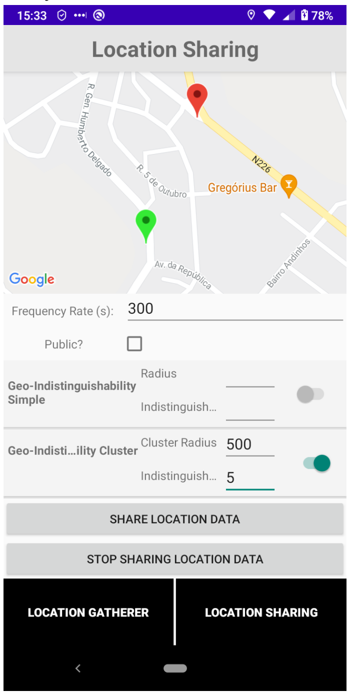
(A) Selector Activity



(B) Sub-Manager Activity



(C) Client Activity



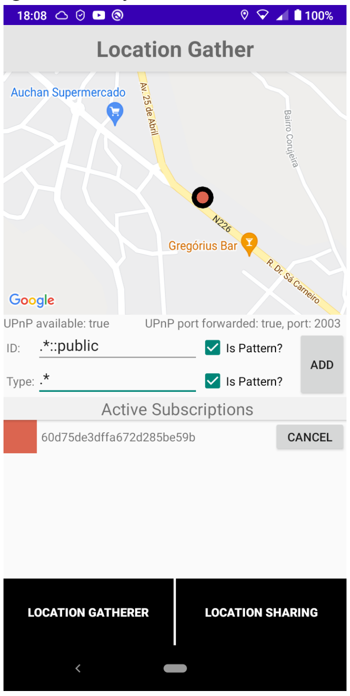
Frequency Rate (s): 300

Public? ☐

Geo-Indistinguishability Simple — Radius: ; Indistinguish...:

Geo-Indisti...ility Cluster — Cluster Radius: 500; Indistinguish...: 5

(D) Location Sharing Activity



UPnP available: true UPnP port forwarded: true, port: 2003

ID: .*::public ✓ Is Pattern?

Type: .* ✓ Is Pattern?

60d75de3dffa672d285be59b

(E) Location Gather Activity

FIGURE 6.4: Android client application activities

To build our client device, we used several technologies to be able to have an application that enables us to read and generate QR codes (ZXing[94]) and read OTP tokens from the Yubikey (Android YubiKit [95]).

6.3.1 Sub-Manager

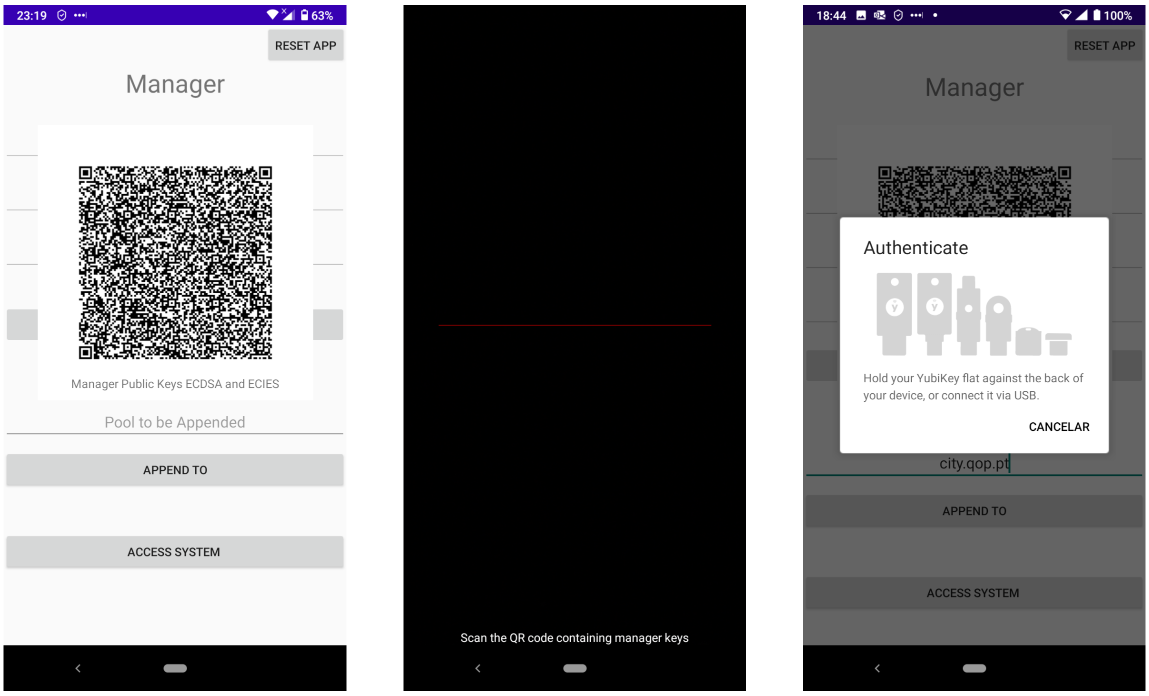
If the sub-manager is selected, the option is saved on the app and the Sub-Manager activity is shown (fig. 6.4b). The Sub-Manager Activity allows to create a new pool, set the Yubikey OTP public id and Yubikey OTP aes secret to define which Yubikey tokens are trusted by the new Sub-Manager. When a new pool is created in this prototype a QR code is showed (fig. 6.5a), containing the Sub-Manager public keys and the public key certificate (as detailed on section 5.2.1 and illustrated on figure 5.2). The QR code allows the android clients to read the Sub-Manager public keys and together with the Sub-Manager Yubikey token get provisioned into the pool.

The Sub-Manager activity allows, after the creation of the pool, to append the pool into the smart city infrastructure. To do this, the app must read the city manager QR (fig. 6.5b) code and the city manager Yubikey token (fig. 6.5c) in order to be able to have the necessary information to perform the *YubiAuthIoT* provisioning protocol detailed on section 5.2.1. At the end of protocol the pool is provisioned into the smart city infrastructure.

Figure 6.5 shows the Sub-Manager Activity state on the several interactions that result in the pool generated being provisioned into the smart-city.

6.3.2 Client

If the client mode is selected, the selection is saved and the Client Activity is displayed (fig. 6.4c). In the Client Activity, two inputs are required: The pool to join and the client name. When the provisioning process is started, the Android device requests to read a QR code (fig. 6.6b) containing the pool to join sub-manager public keys and public key certificate, then after reading the QR code the device prompts for the sub-manager Yubikey token to be inserted into the device (6.6c). In our implementation prototype, after reading the OTP, the client device tries to connect with a device with DNS name equal to *"manager.poolToJoin"* to start the *YubiAuthIoT* authentication and provisioning protocol. This requires that the sub-manager publishes its address into a DNS record that is accessible from the client devices. On a future implementation we would like to switch to a service discovery protocol like Simple Service Discovery Protocol (SSDP).



Hold your YubiKey flat against the back of your device, or connect it via USB.

(A) Sub-Manager Activity with the pool generated

(B) Sub-Manager Activity reading the City Manager QR

(C) Sub-Manager Activity read the City Manager OTP

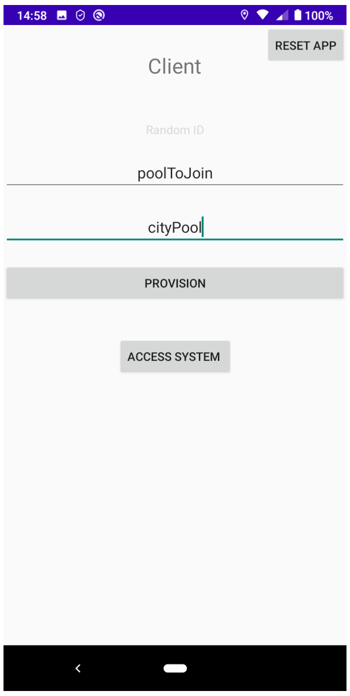
FIGURE 6.5: Android client Sub-Manager Activity in detail

Figure 6.6 shows the Client Activity state on the several interactions that result in it being provisioned.

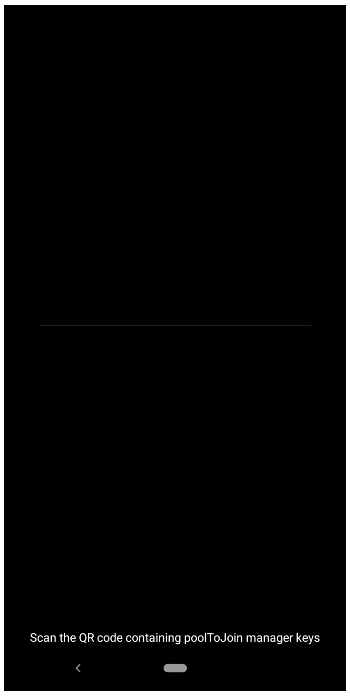
6.3.3 Location Sharing

After initializing the Sub-Manager or the Client on the Android app, the user can navigate the city-services. In our implementation prototype the city contains two services: location sharing and location gathering. Location sharing enables the device to publish real-time location data and location gathering allows to obtain location data from the city via FIWARE Orion subscriptions.

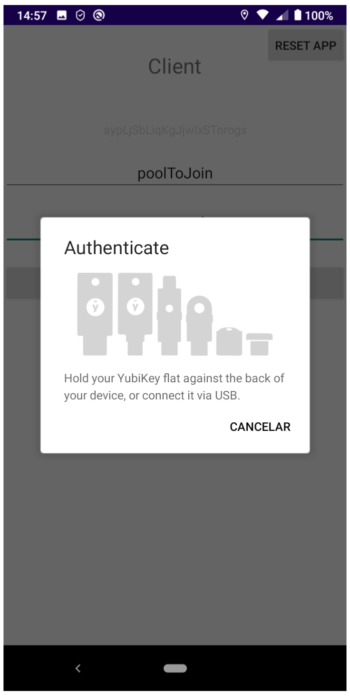
The Location Sharing Activity (fig. 6.4d), contains a GoogleMaps MapView[96] with two pins: the red pin points to the real location and the green pin points to the anonymized location. The anonymized location is calculated by employing one of the location anonymization algorithms described in the related work (section 3.2.4). The choice of the location algorithm and their related parameters is done by the app user, the app shows a preview of the anonymized location on the MapView and the anonymized location is only shared when the user presses on the ”SHARE LOCATION DATA” button.



(A) Client Activity inserting the pool and name information


Scan the QR code containing poolToJoin manager keys

(B) Client Activity reading the City Manager QR



Hold your YubiKey flat against the back of your device, or connect it via USB.

(C) Client Activity reading the City Manager OTP

FIGURE 6.6: Android client Client Activity in detail

The "SHARE LOCATION DATA" button starts a background service that uses the algorithm selected and its parameters to share the anonymized location to the smart city using a *YubiAuthIoT* secure communication channel.

In our implementation prototype, to build the Geo-Indistinguishability algorithm implementation, we analyzed the code of the Location Guard browser plugin [97] developed by the authors of the original Geo-Indistinguishability paper [9], transcribing the JavaScript code into Java, to be used on the Android Application. To build the Clustered Geo-Indistinguishability algorithm implementation we used the implementation of the Geo-Indistinguishability transcribed from the Location Guard plugin and the pseudo code from the Clustered Geo-Indistinguishability paper [10].

6.3.4 Location Gather

The Location Gather Activity (fig. 6.4e), in a nutshell, allows to create subscriptions to receive Location Updates and display the updates on a GoogleMaps MapView, where each circle color on the MapView represents a different subscription.

FIWARE Orion subscriptions require an HTTP notify URL to be provided, that will be used by Orion to send the updates back to the subscriber (check subscription format in the listing 5.2). To be able to provide an URL the Android App must have an HTTP server capable of receiving requests coming from the internet, however that can be difficult since it is very common to have Android devices inside a Network Address Translation (NAT) network, that depending on the type can change the IP and port to random values outside of the control of the Android app [98, 99], making it impossible to have a server inside a NAT network. Over the years multiple techniques appeared that enabled to add exceptions to the NAT network behaviour and enable to set custom port translations that are in the control of the user or network administrator, bypassing the NAT. These techniques include Port-Forwarding, Universal Plug n' Play (UPnP) and hole punching.

Port-Forwarding consists of creating rules on the network router, that statically maps internal ports and internal devices to external ports, enabling to have always the same external port for a given internal service running on an network-internal machine. Port-Forwarding demands that the user needs to have administrator access to the router management, which is not always possible.

In UPnP, UPnP clients can obtain the external IP address of for your network and add new port forwarding mappings as part of its setup process, by communicating directly with the router if it supports UPnP configurations. However, not all routers support UPnP and those that support it, require the administrator to activate it.

When a UPnP port-forwarding setup is not possible to achieve, hole-punching provides a way to solve the issue. Hole-punching techniques base themselves on the knowledge that active connections usually have the same *internal_ip:internal_port* to *external_port* map for the entirety of the connection, in a way, a port-forwarding rule is created for the entirety of the connection. Figure 6.7 aims to depict the typical hole-punching on a port restricted cone NAT, which is one of the most common NAT types for home networks.

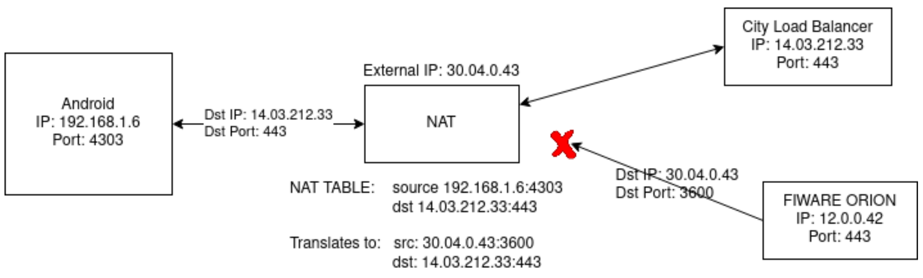


FIGURE 6.7: Hole-punching on a port restricted cone NAT

On a port restricted the NAT translation table records the internal IP, internal port and external IP and port. The only external machines that are capable to respond back and use the punched hole are the external devices with the same source (src) IP and source port equal to the destination (dst) IP and destination port, any other port and IP combination will be rejected by the NAT and do not arrive to the android device.

Hole-punching, however, does not work well with our architecture where the component that sends the notifications is not the same as the one that is used to create the hole-punch. In our architecture the hole is punched by making requests directed to the virtual IP that in turn is associated with one of the load balancer machines, the component that makes the notifications is FIWARE Orion, and when it attempts to use the port punched on the NAT, the NAT will compare the request source IP with the destination IP used on the hole punch process, rejecting the request.

Given this issue with hole-punching, we decided to use the method that demands the fewer router configurations, choosing the UPnP technique. To open ports on the router using the Android App we used the weUPnP library[100]. weUPnP enables us to check if the network router allows UPnP and to those that allow it, get router information, such as external IP, and most importantly ask to open a specific port on the router. When the router opens the port, external access to the service listening to that port is allowed.

One might say that this solution is not scalable to the context of a smart city infrastructure, in fact the requirement to have UPnP in order to receive updates is harder to obtain outside of residential networks. Usually client devices when they are travelling on the smart city they are connected to cellular networks, that, to our knowledge, do not support UPnP. However, we can use the intra-pool communication enabled by our architecture to enable all the devices on the pool to receive updates, only requiring that a small set of client devices of that pool, or the pool manager itself, to be stationary on a network that supports UPnP. The figure 6.8 shows an example deployment of the usage of intra-pool communication to provide update notifications to all the devices in the pool, without any restriction to which network they are connected.

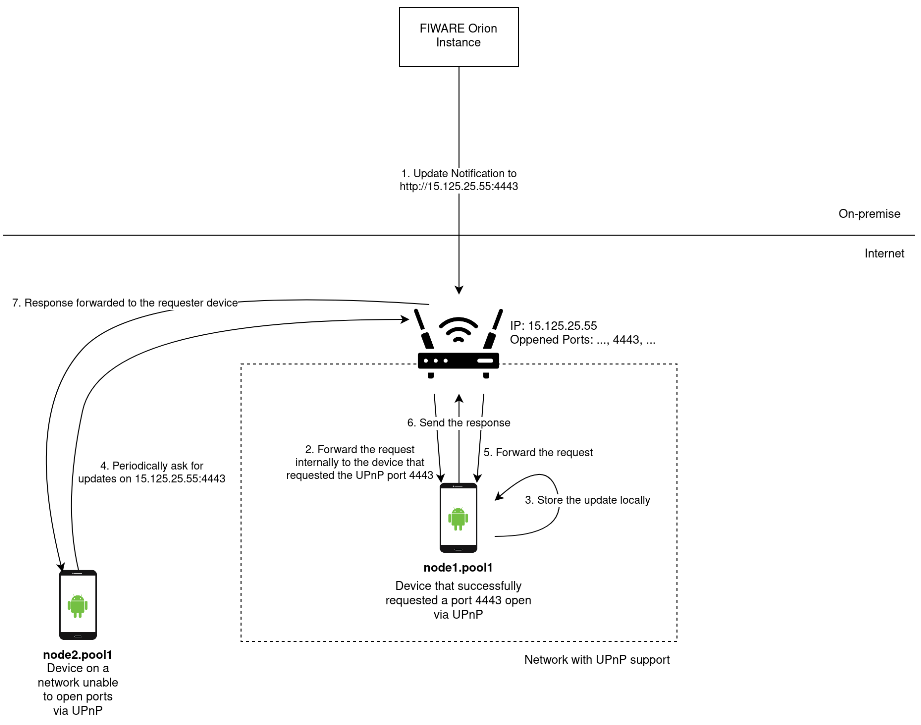

FIWARE Orion Instance
1. Update Notification to http://15.125.25.55:4443
On-premise
Internet
7. Response forwarded to the requester device
IP: 15.125.25.55
Oppened Ports: ..., 4443, ...
6. Send the response
2. Forward the request internally to the device that requested the UPnP port 4443
5. Forward the request
4. Periodically ask for updates on 15.125.25.55:4443
3. Store the update locally
node1.pool1
Device that successfully requested a port 4443 open via UPnP
Network with UPnP support
node2.pool1
Device on a network unable to open ports via UPnP

FIGURE 6.8: Intra-pool communication to get updates without having access to UPnP

Chapter 7

Testing the Proposed Solution

This chapter is responsible to test our architecture and implementation prototype by analysing performance of our implementation prototype in terms of on-premise and off-premise storage performance using testing mechanisms adapted to the role they assume in our architecture and the overhead that *YubiAuthIoT* adds to the communication when compared with the normal behaviour of a FIWARE deployment in terms of time and amount of data transmitted.

We also test the ability of our solution to scale by testing our solution with an increasing number of concurrent client devices and test the quality of the location data anonymization of our solution.

Due to the fact that in our architecture the process of uploading data to the cloud and the process of sending data to smart city are not directly connected, caused by having a on-premise 24 hour caching before sending data to the cloud, it does not make sense to make a performance testing of the communication client to the cloud, because they are not directly dependent of one another. Testing the storage and the IoT message brokering parts separately allows us to test the solutions on their respective use case.

7.1 Performance

This section will present the results of the performance analysis of our implementation, presenting first performance measurements of the on-premise and off-premise storage using tests adapted to their respective role on our architecture. Then measuring the overheads that our communication architecture and implementation prototype cause when compared with a normal FIWARE deployment by measuring the amount of data transmitted on each communication and comparing them.

7.1.1 Storage Performance

Storage performance allows us to perceive the capacity and throughput that the storage solutions used for on-premise and off-premise storage are capable to provide.

7.1.1.1 On-Premise Storage

Ceph provides a shared file-system across all the service replicas allowing them to operate consistently. Since it is a file-system the access to files and folders is not only dependent on the raw throughput of Ceph but with the performance associated with metadata operations that comprise as much as half of the file system workloads [50]. To be able to test metadata operations we make use of the Filebench microbenchmark suite [101]. Filebench uses the Workload Model Language (WML) to enable the definition of custom benchmarks by setting the file operation that will be performed, file sizes, number concurrent threads, etc. Filebench repository also provides a set of prebuilt benchmarks using Workload Model Language (WML), that enable the benchmarking of metadata operations such as create, create directory , stat, delete, list contents of a directory and delete. Since Ceph is used as a virtual hard drive it is more useful to compare it with other solutions that use the same virtual hard drive user interface. We will compare Ceph metadata performance with Network File System (NFS)[102] and ext4. In this test all the backends to each solution will be local on the device that will make the benchmarking, this way the performance tests will not incorporate noise, such as network latency, into the measures that reduce the benchmark precision. Ceph infrastructure will be running on docker containers locally on the device, NFS will have a server running locally and the ext4 testing is regarding a local Solid-State Drive (SSD) formatted with the ext4 file-system.

For our testing, we use the pre-built WML files provided by the filebench installation. From the files available, we selected the ones that represent more our system metadata operations: create files and directories, delete files and directories, and list the contents of a directory. The files selected and their operations are defined below.

- **filemicro_createfiles.f** - Attempts to create 20000 files with 1KB size on a single thread, recording the number of operations per second (ops/s) the file-system allows. This WML file allows us to test the performance of file creation metadata operations, which is one of the typical operations that a file-system is able to provide.

- **makedirs.f** - Creates 10000 directories that can contain child directories, recording the number of ops/s that the file-system is allowing. **makedirs.f** helps us to understand the performance of directory creation, which is a typical metadata operation on a file-system.
- **listdirs.f** - Creates a directory tree with 50000 children nodes, with a mean root to leaf width of 5 and attempts to list the contents of several of the created directories, recording the number of operations per second of this kind that the file-system allows to be done. Listing directories is a common operation on file-systems and this test allows us to better perceive the performance of that operation.
- **removedirs.f** - Creates 10000 empty directories and attempts to remove all of them, recording the number of delete operations per second. This WML file will help us to understand how the file-system performs on rmdir operations.
- **filemicro_delete.f** - Creates 50000 files with 16KB in size and then attempts to delete them, recording the number of sucessfull delete operations per second. This test checks the performance of delete operations on the file-system.

Table 7.1 shows the average and the standard deviation of the operations per second (ops/s) reported by filebench. We run each WML file 10 times to have a better measurement of the performance of Ceph when compared to other technologies such as NFS.

Metadata Performance	Ceph	NFS	ext4
Create File	2750.233 ± 187.646	499.716 ± 22.295	11251.036 ± 0.141
Create Directory	9998.182 ± 0.137	9996.866 ± 4.298	9998.339 ± 0.345
List Directory Content	31116.188 ± 10999.502	14172.211 ± 96.266	49311.380 ± 230.366
Delete Directory	1203.545 ± 69.431	386.482 ± 11.633	9997.991 ± 0.684
Delete File	515.912 ± 0.020	257.961 ± 0.005	504.027 ± 2.848

TABLE 7.1: Comparison of the on-premise storage metadata performance

Contrarily to what happens with off-premise storage, the typical operations on on on-premise involve appending and reading several small-sized files multiple times corresponding to writes in a database or updates to Authzforce policies. To analyse the performance of this kind of data manipulation, we used filebench, the same benchmarking tool as before, but this time using WML files that allowed us to test the read and writes of data. From the pre-built WML files provided by the filebench installation we selected the following:

- **filemicro_createrand.f** - Creates an empty file and then appends random bytes of size between 1B and 1MB until the file-size reaches 1GB. This test is useful to understand the behaviour of the file-system on the appending of data to an existing file, which is the behaviour of collecting the data relative to a day into the file that represents that day on which for every entity update the change is appended to the file representing that day.
- **filemicro_seqread.f** - Attempts to sequentially read a 1GB file one 1MB block at a time, recording the number of read operations per second the file-system allows. This helps us understand how the system behaves when it is needed to read an entire file.
- **filemicro_rread.f** - Reads 2KB chunks of an 1GB file at random positions, stops when 128MB where read, storing the the number of random reads that the file-system allowed per second. This is useful to verify the performance of the usage of the file-system as a database storage where random access to small portions of a file is frequent.

Table 7.2 shows the average and standard deviation of the operations per second (ops/s) reported by filebench. We run each WML file 10 times to have a better measurement of the performance of Ceph when compared to other technologies such as NFS and ext4.

Data Performance	**Ceph**	**NFS**	**ext4**
Appending	127.638 ± 2.306	203.762 ± 2.421	238.993 ± 6.158
Sequential Reading	4379.399 ± 106.140	4333.268 ± 347.361	4462.406 ± 202.560
Random Reading	3001.283 ± 2586.820	2868.203 ± 1594.067	15912.350 ± 1237.606

TABLE 7.2: Comparison of the on-premise storage data manipulation performance

From the data available on table 7.1 it is possible to see that Ceph metadata performance sits between ext4 and NFS, being greater than NFS - one of the most widely used remote file-system - on all the metadata operations tested. On operations over data (table 7.2), it is possible to see that Ceph's performance is closer to NFS, being lower that NFS on appending operations. This is a compromise between the scalability of Ceph and the performance overheads that distributing content over a distributed network causes.

With this evaluation, it is possible to conclude that Ceph can be used as our on-premise storage enabler, providing a good trade-off between performance and scalability, being also capable of performing all the operations needed for the on-premise storage, in a similar way of having a real hard drive installed on the machines, easing the configuration and implementation.

7.1.1.2 Off-Premise Storage

Unlike to what happens with on-premise storage, the most important property of the off-premise storage is the throughput, that is, the amount of data that is transferred in unit of time. Since the off-premise storage controller will deal typically with large files, each one with the data collected in one day. We tested the upload and download performance with files with sizes: 100MB, 500MB and 1GB in order to check our system adapts to an increasing file size. To verify the throughput stability, we run the upload and download of each file 10 times.

In order for the tests to resemble the use case on which the off-premise is deployed in our architecture, we will use four Amazon Web Services (AWS) S3 standard access blob buckets as the storage backends to each one of the solutions tested. AWS S3 was chosen to be the storage backend because it was the only off-premise storage solution that fully worked with the available CHARON implementation. Table 7.3 shows the average of the results and the standard deviation with all the results in MB/s.

	UPLOAD		
	100MB	500MB	1GB
Off-premise controller	9.23 ± 0.33	10.90 ± 0.43	10.49 ± 0.14
Charon	6.64 ± 0.77	9.02 ± 2.13	8.96 ± 1.56

	DOWNLOAD		
	100MB	500MB	1GB
Off-premise controller	10.10 ± 1.05	12.45 ± 2.48	14.12 ± 1.85
Charon	8.20 ± 0.42	8.27 ± 0.45	8.50 ± 0.33

Table 7.3: Upload and download performance comparison of the Off-premise controller and Charon

The average is useful to determine the average speed of the download/upload process and the standard deviation helps us understand how much the individual tests deviate from the average, being a strong measure of throughput stability.

We choose to compare our solution, based on ARGUS, with Charon because it uses a cloud-of-clouds storage methodology which is used as well by our off-premise storage controller, being a more fair comparison.

From the results available on table 7.3 it is possible to conclude that our off-premise controller has higher throughput speeds than Charon but the individual test results deviate more from the average, providing less stable throughput speeds. However, even when comparing with the lowest throughput result that our solutions offered in the tests, it is still higher that the highest throughput that Charon offered in the same tests, under the same environment.

7.1.2 Client Communication Performance

This section provides a performance analysis of the communication between the client and the server. The performance analysis in this section is only related to the communication from the client until it reaches the FIWARE Orion and a response is issued back. This means that this section does not test the storage and off-premise upload performances, the performance related to the storage and off-premise upload is analysed on section 7.1.1.

As stated in the client device section of our architecture (section 5.3) the client communication is based on the *YubiAuthIoT* protocol and includes authentication, certificate revocation,policy checks and lastly secure communication. We analyse the communication overheads between the communication used in this thesis and the communication using standard a FIWARE deployment, on which the authenticity is provided by KeyRock[23], access control is provided by AuthzForce [68], PDP is provided by Wilma[103], IoT message broker is provided by Orion[104] and the communication protocol is HTTPS.

For all the testing in this section we deployed our solutions using the deployment described on the implementation using AC2 virtual machines on Amazon AWS, this was done to have a deployment closer to a real use case, where the machines running the services are not in the local network. The standard FIWARE deployment will be also deployed on AC2 virtual machine with the same Central Processing Unit (CPU), Random Access Memory (RAM) and storage resources to keep consistency. Figures 7.1 and 7.2 provide a simplified visual representation of the components of both our solution and the standard FIWARE deployment.

Our deployment is described on the architecture and implementation chapters of this document. In our test, we generated one sub-manager that will be also our client device,

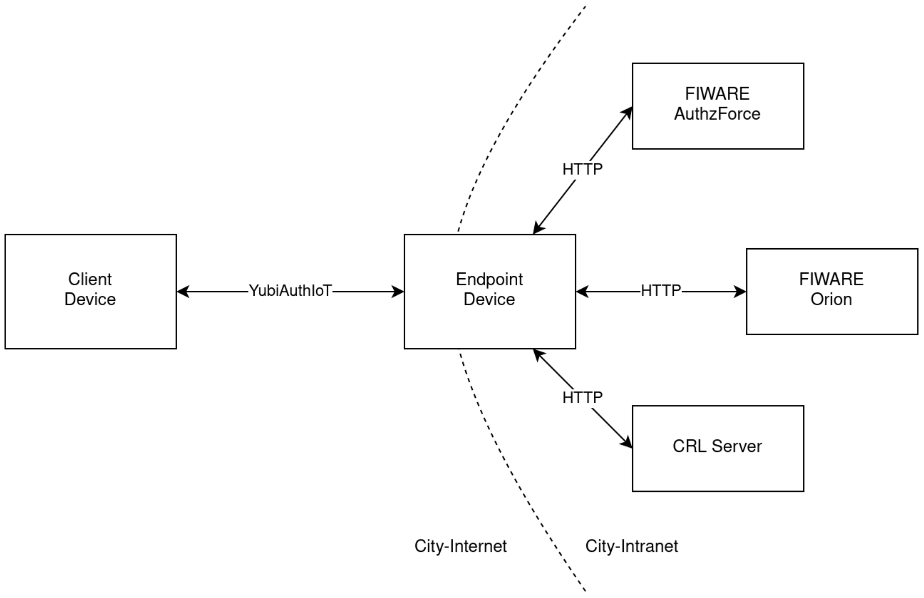


FIGURE 7.1: Simplified representation of our architecture

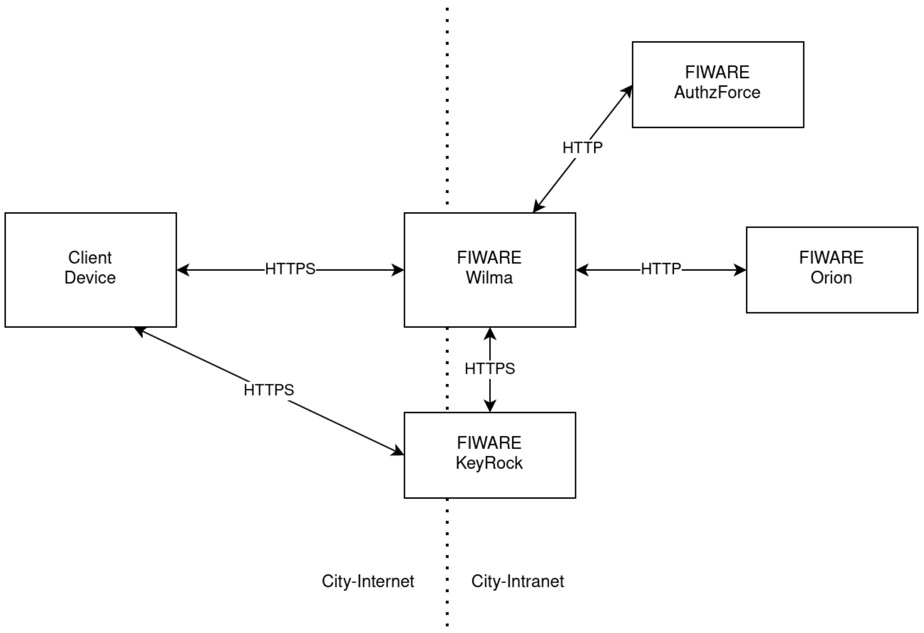


FIGURE 7.2: Simplified representation of a standard FIWARE deployment

the sub-manager will be provisioned into the smart city and communicate with the endpoint device to access the smart city IoT message broker service, powered by FIWARE Orion.

To deploy the Standard FIWARE, we first started by deploying FIWARE Keyrock, configured to listen HTTPS requests and with a DNS domain name ”keyrock.city”. Then inside the Keyrock IdM management web interface we create an application called Orion that will be responsible to process the authentication of the user that want to access the

IoT message broker. We then created organizations inside the KeyRock IdM to simulate the pools we have in our architecture. We then forced that every user must only belong to a single organization, simulating the pool isolation of our architecture. After configuring KeyRock we configured AuthzForce with one policy-set per organization, limiting the access to entities that where created by members of the same organization as the user trying to access the entities. Lastly we configured FIWARE Wilma to verify the authentication by communicating with the KeyRock IdM and to verify if the user is allowed by using the information of present on the access token, which includes username name and organization where it belongs, together with the request information, including HTTP method, entity id, etc. to communicate with Authzforce instance and get a policy decision. Then according to the policy decision, forward the request to orion or refuse the forwarding, throwing the error back to the user.

Upon analysis of our deployment (fig. 7.1) and standard FIWARE deployment (fig. 7.2) components we can see that they are very similar intranet wise, where there is a FIWARE Authzforce and FIWARE Orion instance that are only accessible via a proxy device - Wilma on standard FIWARE and Endpoint device on our architecture -, the main difference is in the authentication and registration process. We power our authentication process using *YubiAuthIoT* on the same communication channel that is used to send the data. Standard FIWARE uses KeyRock for authentication and user registration, requiring additional communication to KeyRock to receive the authentication token, before communicating with Wilma to reach the city-intranet services. Wilma also needs to communicate with KeyRock to verify the authentication token before doing any policy verification and forward the request to FIWARE Orion.

Comparing the overhead of the *YubiAuthIoT* protocol described on this thesis to a standard FIWARE deployment, enables us to verify how much overhead do we add to the communication in terms of data transferred. To compare the overheads we ran a network capture using Wireshark[105] and measured the amount of bytes transferred and received related to the requests made by our overhead testing using the Follow TCP Stream tool. We present here two tables: table 7.4, where we show the amount of data sent and received on the provisioning process on our architecture and the registration process on the standard FIWARE deployment, and table 7.5, where we show the amount of data sent and received on the process of updating an entity with 3 attributes: latitude, longitude,timestamp. Totalling 152 bytes of request body. The response to this request is an

HTTP 204 with empty body.

	Our Solution	Standard FIWARE
Sent (B)	801	1524
Received (B)	1647	3796
Total (B)	2448	5320

TABLE 7.4: Overhead comparison on the provisioning/registration process

	Our Solution	Standard FIWARE
Sent (B)	5721	3013
Received (B)	4001	2698
Total (B)	9722	5711

TABLE 7.5: Overhead comparison on entity update communication process

From the results of our testing we can see that our solution provides smaller overheads on the provisioning process, meaning that our solution provides a more efficient provisioning/registration process. For the secure communication our current solution implementation has higher overheads, due to the requirement to always check the CRL server for the certificate revocation which adds an additional 3237 bytes sent and 1993 bytes received, meaning that our solution without CRL verification has lower overheads that standard FIWARE (2848 bytes sent and 2008 bytes received). By reducing the amount of CRL verification requests is possible to reduce the communication overheads and improve the protocol performance.

7.1.3 Scalability

The ability to be capable to adapt to an increasing number of concurrent connections without causing denial of service to the clients is a very important property on a resilient smart city architecture. We deployed our architecture to be easily scalable by using load balancing technologies, such as HAProxy and container orchestration technologies, such as Kubernetes. In this section, we aim to test the scalability of our implementation with an increasing number of concurrent requests directed to the IoT message broker generated by a set of client devices.

In our testing we generated 1200 virtual client devices - client devices running on the same machine - in one Amazon AC2 virtual machine with 8 CPU cores, 32GB of RAM and 5Gb network throughput. We choose to use virtual machines hosted on the cloud as clients to have a greater network bandwidth than a standard residential internet connection. Each one of the client devices will constantly send requests to the smart city infrastructure. We started by testing with only one virtual client, increasing the number of virtual clients in steps of 1, until it reached 1200 simultaneous virtual clients connected and performing entity update requests. All the tests were run with 3 endpoint service cluster (section 5.4.1.1) replicas running on the Kubernetes cluster deployed on the city-intranet. Figure 7.3 shows our testing results.

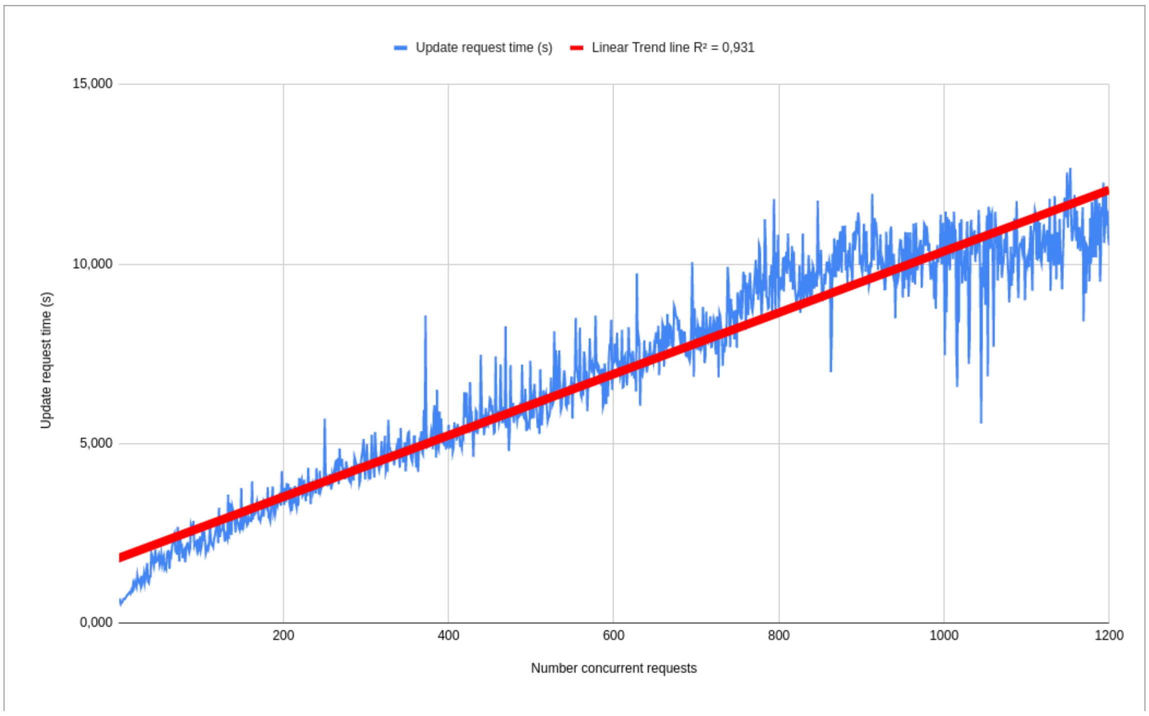


FIGURE 7.3: Average of time per request versus the number of concurrent users when using 3 endpoint service cluster replicas

From our testing we saw that there is a trend of linear increase in the amount of time per request with the increase of the number of concurrent users but not reaching a point where the increase is exponential, which can occur if we only use 3 endpoint service replicas. However, Kubernetes allows to configure auto-scaling to the endpoint service cluster which periodically checks the load on the replicas that are active and increase the number of replicas if the current replicas are getting overloaded, achieving protection against denial of service attacks.

7.1.4 Location Data Anonymization

The principal problem related with continuous data sharing is that it is possible to correlate location updates and with that information define a path on the map using noisy locations with a method called Map Matching. Location Anonymization techniques such as Geo-Indistinguishability[9] appear as a solution for correlation of location updates on a sporadic scenario were the number of updates is small and the updates themselves are far between in terms of time but the privacy level in reduced on continuous scenarios [34]. Techniques, such as Clustered Geo-Indistinguishability[10], improve the privacy of the Geo-Indistinguishability on a continuous scenario by limiting the output of the anonymization to be the same when the real location is within a radius centered on a previous location. (Fig. 3.4). In this section, we validate the findings on [10] by using map matching over anonymized and real location data obtained and related to the location of electric scooters captured in real-time by the Porto digital urban platform from April 2021, days 1 to 6, using different privacy levels.

The information about the real time location of the electric scooters was made publicly available, even when they were being driven, which imposes a risk to the privacy of the citizens that uses this type of transportation. This behaviour was reported by us and currently that information is no longer accessible to the public.

The electric scooters send location updates every 2 minutes, independently of their state (i.e. parked, inUse, etc) The first thing we did was clean the captured location updates by removing all the information that is not important to us, only preserving the reported location, id and timestamp of the location updates provided by the electric scooters. This helped us to reduce the amount of data to be processed from 584MB to 47MB.

Having reduced the amount of data to process, we created individual files for each electric scooter id, placing on the respective file all the location updates of that scooter. This generated 900 different files containing the location updated of each electric scooter that sent data to the Porto digital urban platform.

With the location updates divided by file, we prepared the tools for the map-matching process. For this we used two tools: Valhalla[106] and QGIS[107]. Valhalla is an open source routing engine which includes tools for time+distance matrix computation, isochrones, elevation sampling, map matching and tour optimization. We use the Valhalla map matching tool to enable us to use the location updates to draw a path on a map by snapping the location to the street that is closer to that point according to the Viterby[108] algorithm with costing automatically generated by Valhalla. QGIS allows us to have a visual representation of the outputs of the Valhalla map-matching. Figure 7.4 shows the result of applying the map-matching to electric scooter location update side by side with the raw location data.

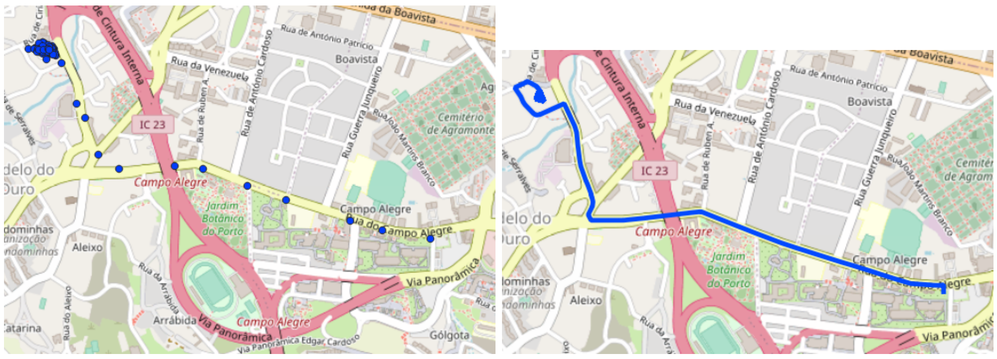


FIGURE 7.4: Example of Valhalla map-matching using a scooter location updates displayed on QGIS

As it is possible to see from the figure 7.4 map-matching can easily provide a path based on the periodic, possibly noisy, updates from an electric scooter, which can compromise the privacy of the citizens that use that electric scooter. For example, if some attacker knows the electric scooter id that a citizen used, it can know the path it took on the city a compromise the citizen privacy by revealing that information to the public.

In a way, if we can mask the path that a citizen took in the city, we severely mitigate the privacy loss, since the attacker will no longer be capable to trace the real path taken by the citizen. To test this, we anonymized the location updates of the electric scooters on the smart city using simple Geo-Indistinguishability and Clustered Geo-Indistinguishability with distinct radius values as parameters. Using the anonymized location updates we call the map-matching algorithm to attempt to trace a path using the anonymized location updates, and then compare the matching of the anonymized map-matching result with the original map-matching result using the F_1 score[37] as described in section 3.2.4.2. The lower F_1 score the lower was the match between map-matchings and higher the privacy. Figure 7.5 shows a visual representation of the F_1 score on a path that was anonymized using simple Geo-Indistinguishability using $l = log(4)$ and $r = 50$.

To reduce the number of parameters in our testing we decided to keep l constant and equal to $log(4)$ changing radius r to obtain different privacy levels ($\epsilon = lr$). To reduce the number of tests, we tested our solution with 5 distinct r values: 500, 200, 100, 50, and

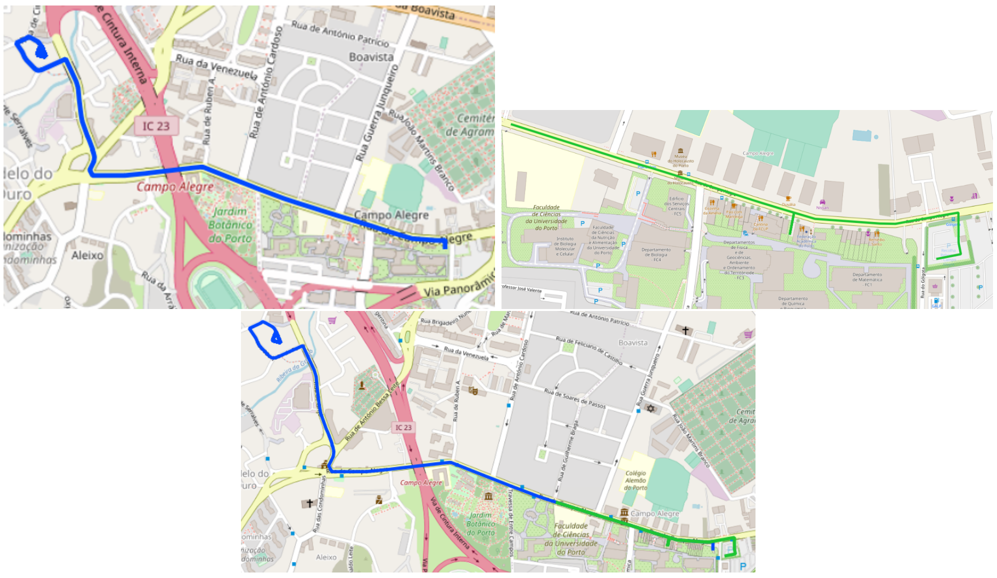


FIGURE 7.5: Visual representation of the F_1 score calculation, the blue line is the map-matching with the original data and the green one is the map-matching of the anonymized data. The resulting F_1 score is $F_1 = 0.22$

20 meters, allowing us to have an general perception of how the privacy changes with a decrease in the anonymization radius.

Table 7.6 shows the average F_1 score of applying the different privacy parameters on the simple Geo-Indistinguishability and Clustered Geo-Indisitinguishability over the location updates of the city electric scooters on days 1 to 6 of April 2021.

	Simple Geo-Indistinguishability	Clustered Geo-Indistinguishability
$r = 500$ ($\epsilon = 0.001$)	0.1621	0.1473
$r = 200$ ($\epsilon = 0.003$)	0.2692	0.2297
$r = 100$ ($\epsilon = 0.006$)	0.3376	0.2876
$r = 50$ ($\epsilon = 0.013$)	0.4650	0.4370
$r = 20$ ($\epsilon = 0.030$)	0.6720	0.6514

TABLE 7.6: Average of F_1 scores with a decreasing anonymization radius

From our testing the Clustered Geo-Indistinguishability offers better privacy preserving by providing lower F_1 scores. Reducing the radius r and keeping the indistinguishability level the same $l = log(4)$ increases the privacy loss but improves the utility of the collected data, since it is more accurate. The choice of a anonymization radius is a trade-off between anonymization and data utility that we place at the hands of the client devices, allowing them to adapt the anonymization levels according to the needs of the client. The results measured here are similar to the results measured in [10] when

$\Delta_t = 120s$, attesting that our implementation of the anonymization algorithms is working properly.

Chapter 8

Conclusions

Having discussed the architecture and implementation, and performed tests of the implementation, we end this document by doing an overview of the work done, stating some of the limitations of the system and draw some final conclusions

8.1 Overview

On this work we started by researching some of the current smart-city infrastructures while paying special attention to the way they address privacy of location updates. With that analysis we verified that the smart-city infrastructures analysed provide poor guaranties of location privacy, especially when related to the correlation between location updates which enables attackers to draw paths that citizens take in the city by using algorithms such as map-matching. We also verified that the use of cloud based storage offloading technologies in smart-cities is not widely used due to the lack of control over data processing and sovereignty issues related to storing data on a external entity that is not controlled by the smart-city.

Having those limitations in mind we analyse the state of the art of data anonymization techniques, finding that semantic anonymization techniques, such as differential privacy, provide better guaranties than syntactic anonymization techniques, such as K-Anonymization, due to the fact that syntactic anonymization techniques requires assumptions about the attacker knowledge to derive which attributes are identifying, quasi-identifying or sensitive, not providing any quantitative privacy value. Due to the fact that traditional differential privacy-based anonymization techniques are better suited to use in databases with the private information in clear text, and that the application of the standard differential privacy noise reduces the utility of the anonymized location data the search of anonymization algorithms that provide the same guaranties as differential privacy but tailored to improve the utility of the anonymized location data was needed. In our research, we found that Geo-Indistinguishability allowed us to add similar properties as differential privacy in an algorithm designed for location data anonymization on the client device, improving the control that citizens have over their data.

Since Geo-Indistinguishability, similarly to differential-privacy, is susceptive to averaging attacks where similar data is anonymized time and time again, allowing an attacker to average the anonymized results and gather the real location leads to the analysis of an upgrade to the Geo-Indistinguishability that limits the amount of similar locations to be anonymized with different noises by creating a radius along a location where all the updates inside that radius will have the same anonymized output, mitigating averaging attacks.

After analysing anonymization techniques, we analysed state of the art storage solutions in terms of upload and download performance, service availability, and data availability. From the storage solutions analysed, two of them contrast when compared with the others. The first is ARGUS, that is a proxy server that uses cloud-of-clouds to store and retrieve data in a way that the data is erasure coded and split accross the colud-of-clouds cloud storages. ARGUS mitigates the problems related to cloud storage since it uses encryption and integrity checks to ensure that the data stored is not accessible to any attacker with access to the cloud storage. ARGUS, by splitting the information across several cloud providers, mitigates sovereignty issues since it reduces the amount of data that a single cloud possesses. The second solution is Ceph. Ceph is used to generate scalable local storage, enabling to increase the storage capabilities when needed without any additional configuration, since Ceph is able to detect changes in the storage and do the proper setup of the storage devices.

After the state of the art storage solution analysis, we compared the traditional token based authentication with certificate-based authentication and from our analysis, certificate based authentication is better for smart-city deployments since it allows to set mutual authentication protocols and provide lower overheads on the registration/provisioning of devices into the smart city when compared to OAuth2 based registration.

We then used the information gathered from the related work to build an smart-city architecture with an authentication and provisioning protocol based on certificates called

YubiAuthIoT. We built *YubiAuthIoT* to be able to exchange information to generate a secret private key using ECIES on the authentication phase, enabling confidential communications. To mitigate impersonation, man-in-the-middle and replay attacks, we used digital signatures and timestamps to introduce, respectively, authenticity and integrity verification and message freshness.

In our architecture we attempt to integrate as much as possible with the FIWARE, since due to its modularity we could change the components required to enable all the security properties we wanted and keep other components unchanged. In short, we changed two components and discarded one completely. We changed the PEP proxy with one created by us named endpoint device. This was done to be able to communicate to the clients using *YubiAuthIoT*. We also changed the data persistency enabler with the one built by us that connects to the public cloud and stores the data collected on the public cloud storage. Our data persistency enabler is based on ARGUS.

Having an architecture designed we implemented all the components, using Kubernetes to orchestrate all the smart-city centralized services, such as PEP proxy, message broker and data persistency enabler, in a way that they can scale according to the load that the smart-city is received mitigating denial of service. We them developed an Android application to work as our client device on which there are location controls and a preview map where the real location and the location that is being reported are shown, placing the control on the user hands.

We then test the performance of our implementation and the quality of the anonymization algorithms implemented by using map-matching algorithm as our attacker behaviour, in which it tries to trace the path that a citizen takes on the city to affect the privacy of that citizen. Concluding that our solution provides location privacy on an architecture that is scalable and provides high storage offloading performance while keeping the costs lower when compared to on-premise persistency.

8.2 Limitations

This system requires the deployment of a client capable of using the *YubiAuthIoT* protocol on the devices that which to publish or subscribe data to the Smart City, which can be difficult to be achieved on close hardware or close software IoT systems, such as Phillips Hue, requiring modification of these devices in order for them to be able to communicate with our infrastructure.

8.3 Conclusion

With this work we where able to create an smart city architecture capable of having secure device provisioning, decentralized authentication and secure client to client or client to smart-city communication, powered by *YubiAuthIoT*. Our architecture also has access control policies to limit the access to data on the smart city and deployed a client-side location privacy layer where the client has full control over the data shared. We made our architecture similar to FIWARE, where there the access to the message broker is conditioned by an PEP proxy that seats between the citizens that the message broker. The job of the PEP proxy is to verify the authentication and access control policies to grant or deny the access to the message broker. The persistency layer on our system is based on the public cloud storage, connecting to the message broker and caching the updates for 24hrs before storing them on the public cloud storage.

Appendix A

XACML Policies Format

A.1 Policy Creation

A.1.1 Rule

```
<Rule RuleId="allow" Effect="Permit">
    <Description>Rule to allow subscription of data</Description>
    <Target>
        <AnyOf>
            <AllOf>
                <Match MatchId="urn:oasis:names:tc:xacml:3.0:function:string-starts-with">
                    <AttributeValue
                        DataType="http://www.w3.org/2001/XMLSchema#string"
                    >
                            -.Domain-
                    </AttributeValue>
                    <AttributeDesignator
                        Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
                        AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                        DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"
                    />
                </Match>
            </AllOf>
        </AnyOf>
    </Target>
</Rule>
```

A.1.2 Policy

```
<Policy
PolicyId="Subscriptions"
Version="1.0"
```

```
RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-unless-permit"
>
    <Description>Subscription policies</Description>
    <Target>
        <AnyOf>
            <AllOf>
                <Match MatchId="urn:oasis:names:tc:xacml:3.0:function:string-starts-with">
                    <AttributeValue
                        DataType="http://www.w3.org/2001/XMLSchema#string"
                    >
                            /v2/subscriptions
                    </AttributeValue>

                    <AttributeDesignator
                        Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
                        AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                        DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"
                    />
                </Match>
            </AllOf>
        </AnyOf>
    </Target>
    <Rule ...></Rule>
    <Rule ...></Rule>
    <Rule ...></Rule>
    <Rule ...></Rule>
</Policy>
```

A.1.3 Policy Set

```
<PolicySet
    xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
    PolicySetId="-.PolicySetID" Version="1"
    PolicyCombiningAlgId="urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:deny-unless-permit"
>
    <Description>-.Description</Description>
    <Target>
        <AnyOf>
            <AllOf>
                <Match MatchId="urn:oasis:names:tc:xacml:3.0:function:string-ends-with">
                    <AttributeValue
                        DataType="http://www.w3.org/2001/XMLSchema#string"
                    >
                            -.Domain
                    </AttributeValue>
                    <AttributeDesignator
                        AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
                        DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true"
                        Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
                    />
                </Match>
            </AllOf>
        </AnyOf>
    </Target>
    <Policy ...></Policy>
    <Policy ...></Policy>
    <Policy ...></Policy>
</PolicySet>
```

A.1.4 Domain

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<domainProperties
    xmlns="http://authzforce.github.io/rest-api-model/xmlns/authz/5"
    externalId="-.ID"
/>
```

A.2 Policy Query

```
<?xml version="1.0" encoding="UTF-8"?>
<Request
    xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
    CombinedDecision="false"
    ReturnPolicyIdList="false"
>
    <Attributes
        Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject
    >
        <Attribute
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
            IncludeInResult="false"
        >
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string"
            >
                -.USSN
            </AttributeValue>
        </Attribute>
    </Attributes>
    <Attributes
        Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
    >
        <Attribute ... ></Attribute>
        <Attribute ... ></Attribute>
    </Attributes>
    <Attributes...></Attributes>
</Request>
```

Bibliography

[1] R. P. Dameri, "Comparing smart and digital city: initiatives and strategies in amsterdam and genoa. are they digital and/or smart?" in *Smart city*. Springer, 2014, pp. 45–88. [Cited on pages v, vii, and 1.]

[2] C. Benevolo, R. P. Dameri, and B. D'auria, "Smart mobility in smart city," in *Empowering organizations*. Springer, 2016, pp. 13–28. [Cited on page 1.]

[3] P. Hall, "Creative cities and economic development," *Urban studies*, vol. 37, no. 4, pp. 639–649, 2000. [Cited on page 1.]

[4] P. Rizwan, K. Suresh, and M. R. Babu, "Real-time smart traffic management system for smart cities by using internet of things and big data," in *2016 international conference on emerging technological trends (ICETT)*. IEEE, 2016, pp. 1–7. [Cited on page 1.]

[5] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel, "Unique in the crowd: The privacy bounds of human mobility," *Scientific reports*, vol. 3, no. 1, pp. 1–5, 2013. [Cited on page 2.]

[6] J. S. Resende, R. Martins, and L. Antunes, "Enforcing privacy and security in public cloud storage," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2018, pp. 1–5. [Cited on pages 3, 29, 31, 51, and 87.]

[7] B. Gedik and L. Liu, "Protecting location privacy with personalized k-anonymity: Architecture and algorithms," *IEEE Transactions on Mobile Computing*, vol. 7, no. 1, pp. 1–18, 2007. [Cited on page 3.]

[8] T. Xu and Y. Cai, "Location anonymity in continuous location-based services," in *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, 2007, pp. 1–8. [Cited on page 3.]

[9] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi, "Geo-indistinguishability: Differential privacy for location-based systems," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 901–914. [Cited on pages 3, 24, 25, 27, 107, and 121.]

[10] M. Cunha, R. Mendes, and J. P. Vilela, "Clustering geo-indistinguishability for privacy of continuous location traces," in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, 2019, pp. 1–8. [Cited on pages 3, 26, 107, 121, and 123.]

[11] P. R. Sousa, L. Magalhães, J. S. Resende, R. Martins, and L. Antunes, "Provisioning, authentication and secure communications for iot devices on fiware," *Sensors*, vol. 21, no. 17, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/17/5898 [Cited on page 4.]

[12] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002. [Cited on pages 6 and 17.]

[13] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Cited on page 7.]

[14] Backblaze, "Erasure coding: Backblaze open sources reed-solomon code," https://www.backblaze.com/blog/reed-solomon/, (Accessed on 04/22/2021). [Cited on page 7.]

[15] "klauspost/reedsolomon: Reed-solomon erasure coding in go," https://github.com/klauspost/reedsolomon, (Accessed on 04/22/2021). [Cited on page 7.]

[16] D. Slamanig and C. Hanser, "On cloud storage and the cloud of clouds approach," in *2012 International Conference for Internet Technology and Secured Transactions*. IEEE, 2012, pp. 649–655. [Cited on page 8.]

[17] J.-M. Bohli, A. Skarmeta, M. V. Moreno, D. García, and P. Langendörfer, "Smartie project: Secure iot data management for smart cities," in *2015 International Conference on Recent Advances in Internet of Things (RIoT)*. IEEE, 2015, pp. 1–6. [Cited on pages xiii, 9, and 10.]

[18] J. L. Hernández-Ramos, A. J. Jara, L. Marín, and A. F. Skarmeta Gómez, "Dcapbac: embedding authorization logic into smart things through ecc optimizations," *International Journal of Computer Mathematics*, vol. 93, no. 2, pp. 345–366, 2016. [Cited on page 10.]

[19] J. M. Bohli, D. Dobre, G. O. Karame, and W. Li, "Privloc: Preventing location tracking in geofencing services," in *International Conference on Trust and Trustworthy Computing*. Springer, 2014, pp. 143–160. [Cited on pages 11 and 12.]

[20] R. Al-Dhubhani, R. Mehmood, I. Katib, and A. Algarni, "Location privacy in smart cities era," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 123–138. [Cited on pages 12 and 15.]

[21] Z. Khan, Z. Pervez, and A. G. Abbasi, "Towards a secure service provisioning framework in a smart city environment," *Future Generation Computer Systems*, vol. 77, pp. 112–135, 2017. [Cited on pages xiii, 12, and 13.]

[22] F. Cirillo, G. Solmaz, E. L. Berz, M. Bauer, B. Cheng, and E. Kovacs, "A standard-based open source iot platform: Fiware," *IEEE Internet of Things Magazine*, vol. 2, no. 3, pp. 12–18, 2019. [Cited on page 14.]

[23] "KeyRock," Jun 2021, [Online; accessed 27. Jun. 2021]. [Online]. Available: https://fiware-idm.readthedocs.io/en/latest [Cited on pages 15 and 116.]

[24] J. Sedayao and I. I. Enterprise Architect, "Enhancing cloud security using data anonymization," *White Paper, Intel Coporation*, 2012. [Cited on page 17.]

[25] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam, "L-diversity: privacy beyond k-anonymity," in *22nd International Conference on Data Engineering (ICDE06)*. IEEE, 2006. [Online]. Available: https://doi.org/10.1109/icde.2006.1 [Cited on page 18.]

[26] J. Domingo-Ferrer and V. Torra, "A critique of k-anonymity and some of its enhancements," in *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 2008, pp. 990–993. [Cited on page 19.]

[27] N. Li, T. Li, and S. Venkatasubramanian, "t-closeness: Privacy beyond k-anonymity and l-diversity," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 106–115. [Cited on page 20.]

[28] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International journal of computer vision*, vol. 40, no. 2, pp. 99–121, 2000. [Cited on page 20.]

[29] C. Dwork, "Differential privacy: A survey of results," in *International conference on theory and applications of models of computation*. Springer, 2008, pp. 1–19. [Cited on page 21.]

[30] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep Learning with Differential Privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 308–318. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978318 [Cited on page 21.]

[31] P. Francis, S. P. Eide, and R. Munz, "Diffix: High-utility database anonymization," in *Annual Privacy Forum*. Springer, 2017, pp. 141–158. [Cited on page 22.]

[32] V. Shmatikov, "k-anonymity and other cluster-based methods," *CS-380S Li, Li, Venkatasubramanian, ICDE*, 2007. [Cited on page 23.]

[33] S. Oya, C. Troncoso, and F. Pérez-González, "Is geo-indistinguishability what you are looking for?" in *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, 2017, pp. 137–140. [Cited on page 26.]

[34] R. Mendes, M. Cunha, and J. P. Vilela, "Impact of frequency of location reports on the privacy level of geo-indistinguishability," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 379–396, 2020. [Cited on pages 26 and 121.]

[35] R. Al-Dhubhani and J. M. Cazalas, "An adaptive geo-indistinguishability mechanism for continuous lbs queries," *Wireless Networks*, vol. 24, no. 8, pp. 3221–3239, 2018. [Cited on page 27.]

[36] M. Kubicka, A. Cela, H. Mounier, and S.-I. Niculescu, "Comparative study and application-oriented classification of vehicular map-matching methods," *IEEE Intelligent Transportation Systems Magazine*, vol. 10, no. 2, pp. 150–166, 2018. [Cited on page 27.]

[37] G. R. Jagadeesh and T. Srikanthan, "Online map-matching of noisy and sparse location data with hidden markov and route choice models," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 9, pp. 2423–2434, 2017. [Cited on pages 27 and 122.]

[38] "Rclone," https://rclone.org/, (Accessed on 04/22/2021). [Cited on page 29.]

[39] R. Mendes, T. Oliveira, V. V. Cogo, N. F. Neves, and A. N. Bessani, "Charon: A secure cloud-of-clouds system for storing and sharing big data," *IEEE Transactions on Cloud Computing*, 2019. [Cited on page 29.]

[40] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "Depspace: a byzantine fault-tolerant coordination service," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 163–176. [Cited on pages 29 and 31.]

[41] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "{SCFS}: A shared cloud-backed file system," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 169–180. [Cited on pages 29 and 31.]

[42] "Keycloak," https://www.keycloak.org/, (Accessed on 05/05/2021). [Cited on page 30.]

[43] M. Vrable, S. Savage, and G. M. Voelker, "Bluesky: A cloud-backed file system for the enterprise," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012, pp. 19–19. [Cited on page 30.]

[44] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Peer-to-Peer Systems*. Springer Berlin Heidelberg, 2002, pp. 53–65. [Online]. Available: https://doi.org/10.1007/3-540-45748-8_5 [Cited on pages 31, 33, 35, and 50.]

[45] R. Fantacci, L. Maccari, M. Rosi, L. Chisci, L. M. Aiello, and M. Milanesio, "Avoiding eclipse attacks on kad/kademlia: an identity based approach," in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–5. [Cited on page 31.]

[46] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on*

Operating systems design and implementation, 2006, pp. 307–320. [Cited on pages xiii, 34, 37, 40, 50, and 82.]

[47] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 2006, pp. 31–31. [Cited on pages 35 and 50.]

[48] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, 2007, pp. 35–44. [Cited on page 35.]

[49] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317. [Cited on page 39.]

[50] D. S. Roselli, J. R. Lorch, T. E. Anderson *et al.*, "A comparison of file system workloads." in *USENIX annual technical conference, general track*, 2000, pp. 41–54. [Cited on pages 39 and 112.]

[51] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE, 2004, pp. 4–4. [Cited on page 39.]

[52] "richwolski/bluesky: Bluesky file system from michael vrable, goeff voelker, and stephan savage," https://github.com/richwolski/bluesky, (Accessed on 04/22/2021). [Cited on page 41.]

[53] "Oauth 2.0 bearer token usage," https://oauth.net/2/bearer-tokens/, (Accessed on 06/10/2021). [Cited on page 43.]

[54] "Access tokens - oauth 2.0 simplified," https://www.oauth.com/oauth2-servers/access-tokens/, (Accessed on 06/10/2021). [Cited on page 43.]

[55] "Home - fiware-idm," https://fiware-idm.readthedocs.io/en/latest/, (Accessed on 06/10/2021). [Cited on page 44.]

[56] S. Karthikeyan, R. Patan, and B. Balamurugan, "Enhancement of security in the internet of things (iot) by using x. 509 authentication mechanism," in *Recent Trends*

in Communication, Computing, and Electronics. Springer, 2019, pp. 217–225. [Cited on page 44.]

[57] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, "Dtls based security and two-way authentication for the internet of things," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2710–2723, 2013. [Cited on page 44.]

[58] C.-S. Park, "On certificate-based security protocols for wireless mobile communication systems," *IEEE Network*, vol. 11, no. 5, pp. 50–55, 1997. [Cited on page 44.]

[59] R. Oppliger, *SSL and TLS: Theory and Practice*. Artech House, 2016. [Cited on page 44.]

[60] Y. Chung, "Distributed denial of service is a scalability problem," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 1, pp. 69–71, 2012. [Cited on page 49.]

[61] P. R. Sousa, J. S. Resende, R. Martins, and L. Antunes, "Secure provisioning for achieving end-to-end secure communications," in *International Conference on Ad-Hoc Networks and Wireless*. Springer, 2019, pp. 498–507. [Cited on page 53.]

[62] M. Gentilal, P. Martins, and L. Sousa, "Trustzone-backed bitcoin wallet," in *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems*, 2017, pp. 25–28. [Cited on page 56.]

[63] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish—subscribe systems," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014. [Cited on page 66.]

[64] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003. [Cited on page 66.]

[65] "Porto fiware é sucesso no mundo - portal de notícias do porto. ponto." https://www.porto.pt/pt/noticia/porto-fiware-e-sucesso-no-mundo, (Accessed on 05/30/2021). [Cited on page 67.]

[66] "Developers catalogue - fiware," https://www.fiware.org/developers/catalogue/#core, (Accessed on 05/30/2021). [Cited on pages xiii and 68.]

[67] "Ha architecture and deployment - fiware-orion," https://fiware-orion.readthedocs.io/en/master/admin/extra/ha/index.html, (Accessed on 05/31/2021). [Cited on pages xiii, 69, and 70.]

[68] "Authzforce (community edition) - xwiki," https://authzforce.ow2.org/, (Accessed on 06/01/2021). [Cited on pages 71 and 116.]

[69] "extensible access control markup language (xacml) version 3.0," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, (Accessed on 06/02/2021). [Cited on page 71.]

[70] "Replication — mongodb manual," https://docs.mongodb.com/manual/replication/, (Accessed on 06/02/2021). [Cited on page 76.]

[71] C. Bachechi and L. Po, "Traffic analysis in a smart city," in *IEEE/WIC/ACM International Conference on Web Intelligence-Companion Volume*, 2019, pp. 275–282. [Cited on page 80.]

[72] "Faq: Concurrency — mongodb manual," https://docs.mongodb.com/manual/faq/concurrency/#faq-concurrency, (Accessed on 06/01/2021). [Cited on page 82.]

[73] "Apache jclouds® :: Home," Mar 2021, [Online; accessed 20. Jun. 2021]. [Online]. Available: https://jclouds.apache.org [Cited on page 87.]

[74] B. Porter, J. van Zyl, and O. Lamy, "Maven – Welcome to Apache Maven," Jun 2021, [Online; accessed 22. Jun. 2021]. [Online]. Available: https://maven.apache.org [Cited on page 88.]

[75] "Running Orion from command line - Fiware-Orion," Jun 2021, [Online; accessed 22. Jun. 2021]. [Online]. Available: https://fiware-orion.readthedocs.io/en/master/admin/cli/index.html#command-line-options [Cited on page 88.]

[76] "Home - fiware-cygnus," Jun 2021, [Online; accessed 1. Jul. 2021]. [Online]. Available: https://fiware-cygnus.readthedocs.io/en/latest [Cited on page 88.]

[77] telefonicaid, "fiware-cygnus: Connecting Orion Context Broker and Cygnus," Jul 2021, [Online; accessed 1. Jul. 2021]. [Online]. Available: https://github.com/telefonicaid/fiware-cygnus/blob/master/doc/cygnus-ngsi/user_and_programmer_guide/connecting_orion.md [Cited on page 88.]

[78] "Pricing | Cloud Storage | Google Cloud," Jun 2021, [Online; accessed 22. Jun. 2021]. [Online]. Available: https://cloud.google.com/storage/pricing [Cited on page 89.]

[79] "Preço Amazon S3 - AWS," May 2021, [Online; accessed 22. Jun. 2021]. [Online]. Available: https://aws.amazon.com/pt/s3/pricing [Cited on page 89.]

[80] "Preços dos Azure Storage Blobs | Microsoft Azure," Jun 2021, [Online; accessed 22. Jun. 2021]. [Online]. Available: https://azure.microsoft.com/pt-pt/pricing/details/storage/blobs [Cited on page 89.]

[81] "Production-Grade Container Orchestration," Jun 2021, [Online; accessed 20. Jun. 2021]. [Online]. Available: https://kubernetes.io [Cited on page 90.]

[82] "HAProxy Technologies | The World's Fastest and Most Widely Used Software Load Balancer," Jun 2021, [Online; accessed 21. Jun. 2021]. [Online]. Available: https://www.haproxy.com [Cited on page 92.]

[83] "keepalived," Jun 2021, [Online; accessed 21. Jun. 2021]. [Online]. Available: https://github.com/acassen/keepalived [Cited on page 92.]

[84] "Creating a cluster with kubeadm," Mar 2021, [Online; accessed 21. Jun. 2021]. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm [Cited on page 93.]

[85] "About Calico," Jun 2021, [Online; accessed 23. Jun. 2021]. [Online]. Available: https://docs.projectcalico.org/about/about-calico [Cited on page 95.]

[86] "Rook," Jun 2021, [Online; accessed 21. Jun. 2021]. [Online]. Available: https://rook.io [Cited on page 95.]

[87] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *Journal of Cryptographic Engineering*, vol. 5, no. 2, pp. 141–151, 2015. [Cited on page 97.]

[88] "Elliptic Curve Integrated Encryption Scheme (ECIES) with AES," Jun 2021, [Online; accessed 26. Jun. 2021]. [Online]. Available: https://asecuritysite.com/encryption/ecc3 [Cited on page 97.]

[89] "What is Yubico OTP?" Jun 2021, [Online; accessed 26. Jun. 2021]. [Online]. Available: https://developers.yubico.com/OTP [Cited on page 98.]

[90] YubicoLabs, "yubikey-val," Jun 2021, [Online; accessed 27. Jun. 2021]. [Online]. Available: https://github.com/YubicoLabs/yubikey-val [Cited on page 98.]

[91] "Validation Protocol V2.0," Jun 2021, [Online; accessed 27. Jun. 2021]. [Online]. Available: https://developers.yubico.com/yubikey-val/Validation_Protocol_V2.0.html [Cited on page 98.]

[92] "OTPs Explained," Jun 2021, [Online; accessed 28. Jun. 2021]. [Online]. Available: https://developers.yubico.com/OTP/OTPs_Explained.html [Cited on page 98.]

[93] "yubikey-personalization-gui," Jun 2021, [Online; accessed 28. Jun. 2021]. [Online]. Available: https://developers.yubico.com/yubikey-personalization-gui [Cited on page 100.]

[94] "zxing-android-embedded," Jun 2021, [Online; accessed 23. Jun. 2021]. [Online]. Available: https://github.com/journeyapps/zxing-android-embedded [Cited on page 105.]

[95] Yubico, "yubikit-android," Jun 2021, [Online; accessed 23. Jun. 2021]. [Online]. Available: https://github.com/Yubico/yubikit-android [Cited on page 105.]

[96] "MapView | Google Play services | Google Developers," Apr 2021, [Online; accessed 24. Jun. 2021]. [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/maps/MapView [Cited on page 106.]

[97] chatziko, "location-guard," Jun 2021, [Online; accessed 24. Jun. 2021]. [Online]. Available: https://github.com/chatziko/location-guard [Cited on page 107.]

[98] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 179–192. [Cited on page 108.]

[99] S. N. Srirama and M. Liyanage, "Tcp hole punching approach to address devices in mobile networks," in *2014 International Conference on Future Internet of Things and Cloud*. IEEE, 2014, pp. 90–97. [Cited on page 108.]

[100] bitletorg, "weupnp," Jun 2021, [Online; accessed 25. Jun. 2021]. [Online]. Available: https://github.com/bitletorg/weupnp [Cited on page 109.]

[101] "Filebench," Jun 2021, [Online; accessed 20. Jun. 2021]. [Online]. Available: https://github.com/filebench/filebench [Cited on page 112.]

[102] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "Nfs version 3: Design and implementation." in *USENIX Summer*. Boston, MA, 1994, pp. 137–152. [Cited on page 112.]

[103] "PEP Proxy - Wilma - Fiware-Pep-Proxy," Jun 2021, [Online; accessed 15. Jul. 2021]. [Online]. Available: https://fiware-pep-proxy.readthedocs.io/en/latest [Cited on page 116.]

[104] "Home - Fiware-Orion," Jul 2021, [Online; accessed 15. Jul. 2021]. [Online]. Available: https://fiware-orion.readthedocs.io/en/master [Cited on page 116.]

[105] "Wireshark · Go Deep." Jul 2021, [Online; accessed 15. Jul. 2021]. [Online]. Available: https://www.wireshark.org [Cited on page 118.]

[106] valhalla, "valhalla," Jul 2021, [Online; accessed 16. Jul. 2021]. [Online]. Available: https://github.com/valhalla/valhalla [Cited on page 121.]

[107] "Welcome to the QGIS project!" Jul 2021, [Online; accessed 16. Jul. 2021]. [Online]. Available: https://qgis.org/en/site [Cited on page 121.]

[108] S. Benedetto and E. Biglieri, "Viterbi algorithm," *Principles of Digital Transmission: With Wireless Applications*, pp. 807–815, 2002. [Cited on page 121.]