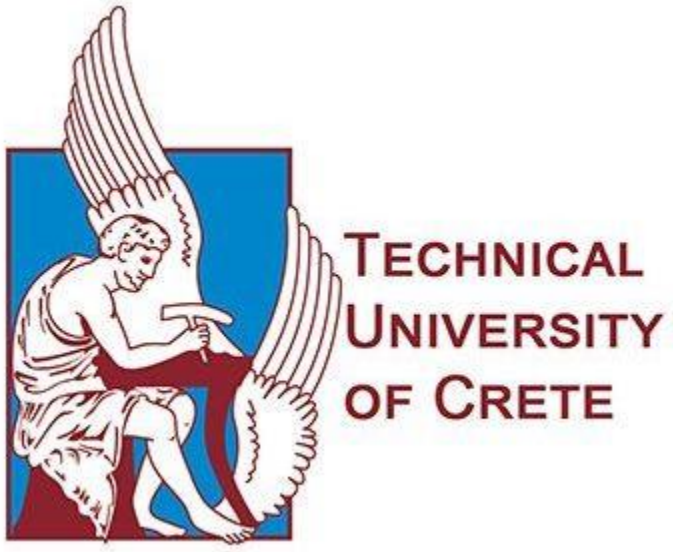
TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



Authorized user access in federated Service Oriented Architectures for the Internet of Things in the Cloud

Kontochristos Ilias

Committee

Supervisor: Prof. Euripides G.M Petrakis

Assoc. Prof. Deligiannakis Antonios

Assoc. Prof. Samoladas Vasileios

Chania, 2020

Abstract

This thesis aims in enhancing the security of a federated Service Oriented Architecture, based on the communication of RESTful micro-services in the cloud. To achieve this goal, HTTPS (HyperText Transfer Protocol Secure) or HTTP over TLS (Transport Layer Security) was incorporated in the communication of the micro-services. In each exposed service, TLS certificates were installed in order to encrypt the communication and guarantee the authenticity of the service. Furthermore, emphasis was put on improving the mechanism for authorization and authentication of the users when they access the system. The authorization code grant of the OAuth 2.0 protocol and OpenID Connect were incorporated in the API of the system. The authorization code grant provides API security and reduces the chance of exposing user credentials by utilizing scoped access tokens. OpenID Connect extends the OAuth 2.0 Protocol and provides enhanced user authentication, ID token validation, and SSO (Single Sign-On) functionality amongst the federated architecture nodes. A series of experiments showed that the added security measures introduce delay into the system. Furthermore, it is demonstrated that different encryption algorithms and key lengths may affect system performance.

Table of Contents

1. Introduction

1.1 Motivation

Internet of Things (IoT) and cloud computing are two highly interconnected technologies. The exponential use of sensors in industrial fields such as energy, healthcare, building management, agriculture, and transportation, as well as the introduction of smart devices facilitating daily activities, have created a vast volume of information which needs to be stored and processed via IoT applications. Due to its characteristics (scalability, affordability, easy maintenance, and accessibility), cloud computing is an ideal platform for deployment of IoT systems which collect, store, process, and analyze IoT data. The majority of these IoT systems follow the principles of the Service Oriented Architecture (SOA) and consist of independent services, communicating over the network. That communication is based on the HTTP protocol and is achieved by exchanging requests and responses between client and server services. The client service makes a request for a resource using a HTTP method (Get, Put, Post, Head, Delete) and the server service replies accordingly. HTTP is based on the assumption that mutual trust exists between the client and the server and has no built-in security measures to ensure communication integrity. Moreover, the communicating services are unable to verify each other's authenticity. Consequently, messages can be intercepted by malicious third parties (man-in-the-middle attacks), who can acquire valuable and sensitive information such as user credentials, credit card numbers and identification of the user. Furthermore, since there is no data validation, the content of the messages can be altered, injected with malware, or redirected to another malevolent service.

Encryption of the communication between two services is essential. In addition, a safe channel and communication method must be established before the exchange of the messages begins. These three components ensure that the content of the messages will remain safe even if they get intercepted by a third party, as they can not be read without possession of the decryption key.

Additionally, encoding guarantees that the messages have not been modified in any way and, as a result, the messages can be discarded by the services if an anomaly is detected. Another important matter that needs to be addressed is service identification. Each service ought to provide a document verifying its authenticity. In order for this document to be considered valid it must be signed by an independent authority that all the services recognize and trust.

iZen is a federated IoT system that consists of a union of equipotent IoT nodes that communicate over the network. Each iZen node leverages the principles of Service Oriented Architectures (SOA) and is implemented as a composition of RESTful micro-services [1]. iZen offers user and organization management, data process storage, and management services, as well as appropriate interfaces for IoT devices installation. iZen services are protected by PEP-PROXY servers that allow access only to authorized users and services. There are three parties of interests in iZen (system administrators, infrastructure owners and customers). System administrators manage their IoT node and enlist it in the iZen federation to make it discoverable from the other nodes. Infrastructure owners register their sensors in a cloud node and sell the data they collect. Finally, the customers can subscribe to sensors in order to receive measurements. The current study uses iZen as the prototype and extends it, in order to address the aforementioned issues.

1.2 Solution

Upgrading HTTP (HyperText Transfer Protocol) to HTTPS (HyperText Transfer Protocol Secure) guarantees communication security. Each exposed service creates a private key, encoded by a strong encryption algorithm, and a certificate that includes information of the service owner (e.g. country, state, corporation, email and domain name). The certificate is sent to a trusted Certificate Authority (CA) which tests and verifies the information and proceeds to sign it. At the beginning of the communication between the services, a handshake is performed. During the handshake, the client and server service will agree upon an encryption algorithm to be used. The server presents its

certificate in order to validate its identity. The certificate contains the server's domain name and the certificate authority that vouches for its authenticity. Finally, the server provides a key for generation of the session keys, that will be used for encryption and decryption, before the actual message transmission begins.

Aiming to minimize the risk of exposing user credentials and to enhance user authorization and authentication, OpenID Connect and the authorization code grant of OAuth 2.0 protocol were implemented in the web API of iZen. Upon receiving a login request, the iZen web application now requests an authorization code from Fiware Keyrock Identity Manager (IDM) and redirects users to its graphical interface, which prompts them to input their email and password. After a successful login, Keyrock IDM responds to the web application with the generated authorization code. The web application makes a second request to the IDM instantaneously, in order to exchange the received authorization code with an access token and an ID token. IDM returns an access token to the web application, alongside a scoped ID token in the form of JSON web token which includes only the necessary user information. The iZen web application performs signature validation when it receives the ID token in order to verify the authenticity of the sender, and to ensure that the user belongs in a trusted iZen node. Additionally, it extracts the necessary user information from the ID token and binds them to the corresponding user session. This procedure is essential, especially when Single Sign On (SSO) functionality is used for login to a remote iZen node. In the aforementioned scenario, the web application will communicate, for authorization and authentication of the user, with the IDM located in the cloud where the user is registered.

Figure 1 presents an abstract view of services communication during user login.

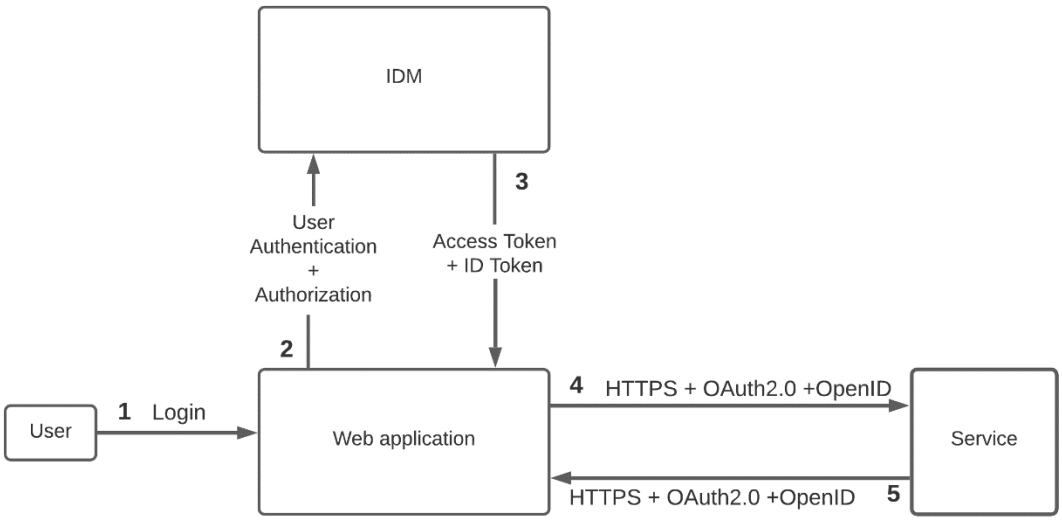


Figure 1: Abstract view of services communication

1.3 Contributions of the Work

The following summarizes the contribution of this work:

HTTPS protocol is enabled in iZen services that communicate over the public network in order to encrypt the communication ensuring its security and integrity.

User login and logout process is altered to follow the authorization code grant type flow, hence the risk to expose user credentials is reduced.

OpenID Connect protocol is enabled which offers improved user authentication and system security utilizing scoped id tokens and the signature validation process.

SSO functionally is incorporated between the iZen nodes allowing users to access remote nodes via their registered IDM account.

1.4 Structure

Chapter 2 provides the knowledge background required for understanding this work and presents the software tools that are used for the completion of this thesis.

Chapter 3 presents iZen's system architecture and briefly explains the functionality of each service.

Chapter 4 identifies the services where the communication occurs over the public network thus the incorporation of the HTTPS protocol is necessary. Additionally, it describes the procedure to generate TLS/SSL certificates, the incorporation of the HTTPS and OpenID Connect protocols and the implementation of authorization code grant type flow, signature token validation and Single Sign-On functionality in the system.

Chapter 5 demonstrates how the added security measures and certificate private keys that are generated by two different encryption algorithms i.e., RSA and Elliptic Curve, which utilize different key lengths, affect the system's performance.

Chapter 6 summarizes the conclusions and Chapter 7 offers recommendations for future work.

2. Background

2.1 HTTP

The Hypertext Transfer Protocol (HTTP) [2] is a stateless application level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. HTTP communication occurs over TCP [3] and port:80 is the designated service port.

HTTP "client": is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests.

HTTP "server": is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method (GET, PUT, POST, DELETE, HEAD, TRACE, OPTIONS, CONNECT, PATCH), Uniform Resource Identifier (URI) [4], and protocol version. The request-line is followed by header fields containing request modifiers, client information, representation metadata, an empty line to indicate the end of the header section, and a message body containing the payload body.

A server responds to client requests by sending one or more HTTP response messages, each beginning with a status line that includes the protocol version, a success or error code, and textual reason phrase, possibly followed by header fields containing server information, resource metadata, representation metadata, an empty line to indicate the end of the header section, and a message body containing the payload body.

2.2 HTTPS

The Hypertext Transfer Protocol Secure (HTTPS) [5],[6] extends HTTP, by encrypting the communication protocol using Transport Layer Security (TLS) [7] or its predecessor Secure Sockets Layer (SSL). As a result, HTTPS is alternatively referred to as HTTP over SSL/TLS. HTTPS ensures data integrity and privacy, and protects the communication against interception by third parties (man-in-the-middle attacks). Furthermore, HTTPS provides authentication of server services, and clients if opted by the server, by utilizing certificates. Certificates contain information about the owner, the domain name of the service, as well as a private key that was encoded using a strong encryption algorithm. Servers send their certificate to a Certificate Authority (CA), i.e., highly trusted third-party organizations, that verify the information and signs it, vouching for the authenticity of the service. HTTPS communication uses a secure channel utilizing the TLS cryptographic protocol and the designated service port is the 443.

The main difference between HTTP and HTTPS communication is the TLS handshake that occurs after the TCP connection has been established and before services start exchanging the actual messages. During the handshake, the server and client discuss which TLS version and cipher suite [8] will be used for encryption. Furthermore, the server service verifies its identity via its public key and the digital signature of the TLS certificate authority. Lastly, the two services generate session keys for symmetric communication encryption after the handshake is over.

The TLS handshake steps while using (RSA), the most common key exchange algorithm named after its creators Rivest Ron, Shamir Adi, and Adleman Leonard, are presented below.

1. Client initiates the handshake with a *ClientHello* message that contains the TLS versions and cipher suites, that client supports, as well as a string of random bytes known as the *Client Random*.

2. Server responds with a *ServerHello* message that contains its certificate, its choice for TLS version and cipher suite that will be used for the communication, and a *Server Random,* its string of random bytes. Optionally, server may request for the client's certificate. If server and client are not compatible a *Handshake Failure* message will be sent instead.

3. Client proceeds to verify the server's certificate with the authority that issued it and responds with its certificate if requested; client will send an empty certificate if he does not possess one. This process guarantee's server authenticity.

4. Server verifies client's certificate. If it receives an empty or invalid certificate, it may choose to continue the communication or interrupt it and respond with *Handshake Failure* message.

5. Client sends another random string of bytes, called the "P*remaster Secre*t", which is encrypted using the public key obtained from the server's certificate and can only be decrypted using the server's private key.

6. Server proceeds to decrypt the *Premaster Secret*.

7. The two sides generate session keys from the Client Random, the Server Random and the Premaster key. The generated session keys should be identical for both sides.

8. Each side sends a *Finished* message that is encrypted with the session key and validates the message that it receives from the other side.

9. Handshake is completed and secure symmetric encryption has been achieved. From this point forward, communication will be encrypted using the session keys.

When the communication ends, the client has to send an "*end of communication"* message, otherwise the server assumes that an error has occurred.

Figure 2 visualize the handshake between a client and a server.

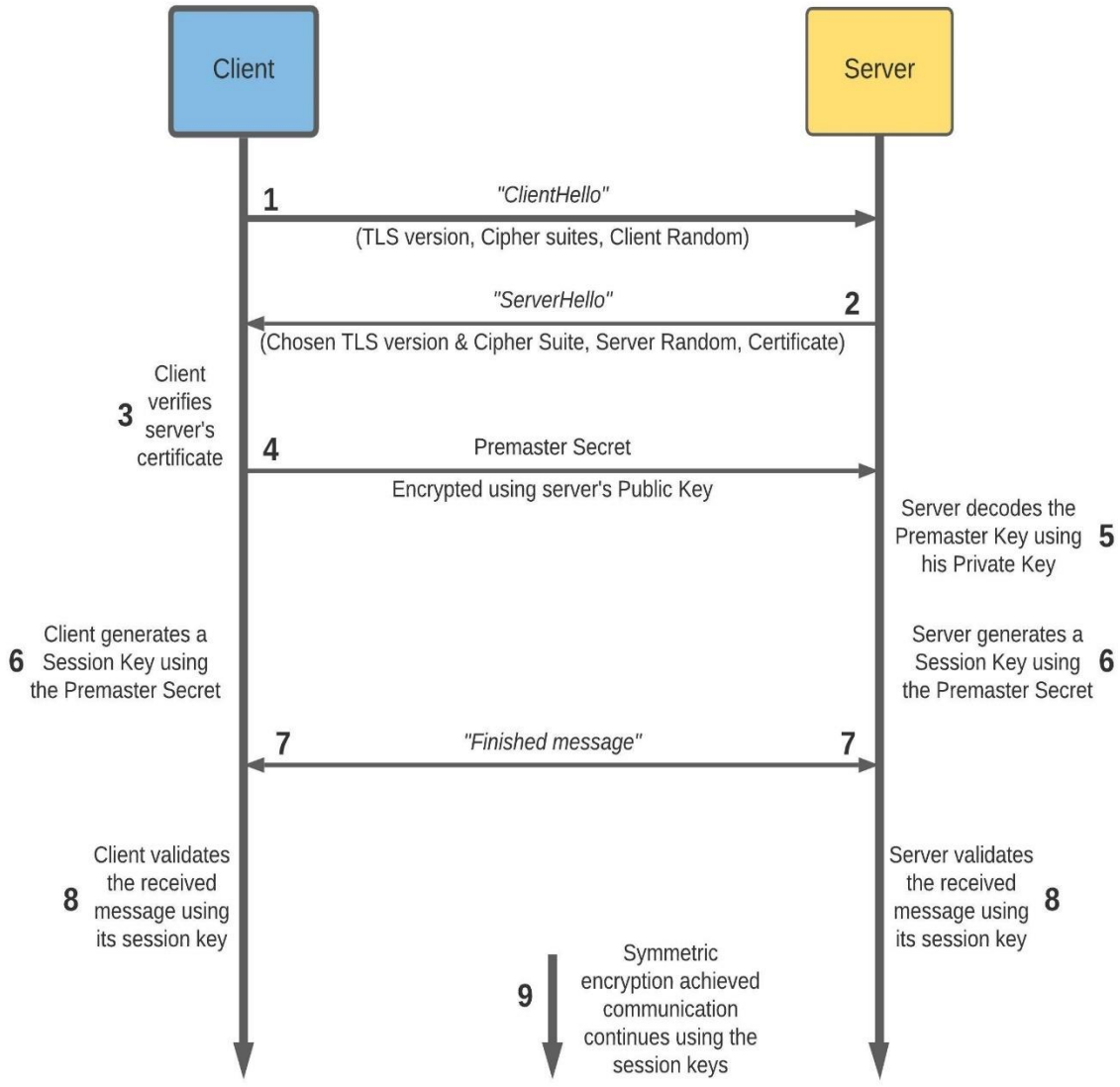


Figure 2: Client and Server Handshake

2.3 OAuth 2.0

OAuth 2.0 [9] is an open standard authorization protocol used to provide secure delegated access to client applications while protecting user's credentials. Authorization servers act as an intermediate between client applications and end users, issuing access tokens to client applications, in order for them to access protected resources. Identity managers issue access tokens only after successfully authenticating the resource owner (end user) and acquire authorization.

2.4 OpenID Connect

OpenID Connect [10] builds on the OAuth protocol and extends it, by utilizing scoped tokens to provide additional user authentication. When OpenID Connect is incorporated, access tokens issued by an authorization server include a scoped ID token. ID tokens contain information such as issuer's identity, issued and expiration time of the token and additional information controlled by scopes, that the end user has accepted to reveal to the client application. Moreover, OpenID Connect offers signature validation of the access tokens and can facilitate Single Sign-On (SSO) functionality among different client applications.

Role of HTTPS, OAuth2.0 and OpenID Connect in communication

All three protocols i.e., HTTPS, OAuth2.0 and OpenID Connect are utilized in services communication as each one implements a distinctive function.

- **HTTPS** encodes the communication guaranteeing the security and the integrity of the exchanged messages.
- **OAuth2.0** provides access tokens for user authorization and authentication in the system.
- **OpenID Connect** enhances the authentication by providing scoped id tokens which contain additional issuer and user information.

2.5 Service Oriented Architecture

Service Oriented Architecture (SOA) [11], [12] is a software design pattern that defines principles based on which services can be used as independent reusable modules, reachable over the network via a well formatted interface. Each service performs a specific and explicitly defined function and provides an interface that other services can use for communication. This communication takes place using standard protocols (SOAP/HTTP or JSON/HTTP) and without the need for human interaction or further code alteration, thus facilitating easy access and fast integration into existing systems.

The main principles of Service Oriented Architecture (SOA) are presented below:

Standardized Service Contract: Services ought to provide a description of their functionality for the other application services.

Service Autonomy & Abstraction: Services control their functionality and how it is implemented but conceal code logic from other services.

Loose Coupling: Services must depend on each other as little as possible, so that any service functionality modification does not hinder the functionality of the whole application.

Service Reusability & Composability: Application logic must be broken down into smaller pieces so that each service implements a certain functionality and can be reused by another system to fulfill the same purpose. Conversely, services must be able to be combined in order to create a single application.

Service Interoperability: Service interfaces must include common communication standards, so that they can be used by a diverse set of subscribers.

2.6 iZen System

iZen is a federated IoT system that consists of a union of equipotent IoT cloud nodes that communicate over the network. Each node represents a different organization. Each member organization can improve their offered services and increase their potential profit, on the basis of information exchange. Node administrators control the information they share and only authorized users are able to access their system. There are three parties of interest that interact in iZen:

System administrators: are responsible for the proper functioning of their system and monitor its operations. They perform *Create, Read, Update, Delete (CRUD)* operations on users and assign appropriate roles and permissions to them. Lastly, they register their node to the federation to make it discoverable by other nodes.

Infrastructure Owners: possess IoT devices which they can install and register on a cloud node, after receiving the appropriate permission by system administrators. Registered devices are discoverable by all iZen nodes. Infrastructure Owners aim to sell data and data management services to customers.

Customers: subscribe to one or more cloud nodes. They can discover and select IoT devices that interested them and get notified about their measurements. In addition, they can acquire data management services provided by Infrastructure Owners.

iZen nodes are independent, expandable, and secure by design, as each individual service is protected by a *Policy Enforcement Point PEP-PROXY*, conjointly with a *Policy Decision Point* in case more complex policy rules are required. Nodes leverage the principles of Service Oriented Architectures (SOA) and are implemented as a composition of RESTful micro-services by combining

well known technologies such as PHP, HTML and JSON with databases (Cassandra, MySQL, MongoDB) and *Generic Enablers.* The latter are reusable components able to execute specific functionalities and they are accessible via APIs that include standard communication protocols. They are provided by FIWARE [13], an open source platform that aims to contribute in the development and implementation of future internet services and applications.

Popular FIWARE Generic Enablers, that are incorporated in the iZen architecture, are briefly discussed hereafter.

FIWARE Orion Context Broker

Gartner defined context broker as a “service designed to gather reachable context data of a variety of types, sources and velocity. It then applies conditioning, integration, rules and analytics to derive the reduced prepared context data, actionable at a point of business decision by a system or a human.”.

Orion is FIWARE’s implementation of a Context Broker [14] and constitutes the cornerstone of each FIWARE component architecture. It is a NGSIv2 [15] server implementation that manages context process and distribution. Clients can query, update, and register context information or they can subscribe to receive notifications upon designated context change.

FIWARE Keyrock Identity Manager

Identity managers (IDMs) are frameworks consisting of technologies and policies responsible for assigning digital identities to entities (physical users or services). They guarantee that only authenticated and authorized individuals can access protected resources if certain conditions are met.

Fiware Keyrock [16] Identity Manager is the core of system's security and alongside PEP-PROXY Wilma and PDP AuthZforce, it can incorporate security authentication and authorization in a system. Keyrock connects other components at application level and enables them to use standard authentication mechanisms in order to accept or reject requests based on industry standard protocols.

The main services offered by Keyrock's API are:

- User account creation and secure information management.
- Application/service registration and declaration of trusted applications, organizations, and users. Activation of communication protocols or technologies that the application accepts at requests.
- PEP-PROXY registration alongside a policy rule set, in order to protect application access points and allow only authorized authenticated access.
- Organization creation and management. Assigning roles and appropriate permissions to members.

FIWARE PEP PROXY Wilma and PDP AuthZForce

PEP proxies are application endpoints placed in front of individual services or resources in order to protect them from unauthorized access. PEP proxies intercept client requests to the service and perform authentication before permitting or denying access.

For more sophisticated access control, PEP proxies can be combined with *Policy Decision Points (PDP)*. After intercepting a request to the protected service, PEP sends the client's attributes to a PDP which in turn takes the decision to permit or deny access based on relevant registered access policy rule sets. Afterwards, PDP returns its decision to PEP, in order for it to enforced.

FIWARE's generic enablers PEP proxy Wilma [17] and PDP AuthZForce [18] are combined with Keyrock IDM to perform advanced access control to backend services. Keyrock's API enables registration of PEP proxies to secure services. It also facilitates creation of access policy rule sets based on the XACML [19] standard, that utilized by PDP AuthZForce to assess Permit/Deny policy decisions.

FIWARE Short Time Historic Database Comet

Comet [20] is a *Short-Term Historic (STH)* database build on top of MongoDB. Comet is responsible for managing historical raw and aggregated context data registered in an Orion Context Broker instance.

Communication between the Comet database and the Orion Context Broker uses standardized NGSI interfaces.

FIWARE Cygnus

Cygnus [21] is an intermediate in charge of persisting context data from Orion (which is a NGSI source of data) into STH Comet or other third party database that accept NGSI-like context data, in order to create a historical view of the context. Cygnus accepts NGSI data flows and stores them to their predefined appropriate databases.

Cassadra

A key component of iZen's architecture is the Cassandra [22] database. Cassandra is developed by Apache Software Foundation and is a distributed NoSQL database ideal for creation of clusters that replicate and share data between nodes, offering high data availability while guaranteeing no single point of failure. In each iZen cloud node, a Cassandra node exists that manages data and enables search by the registered users.

2.7 Docker

Docker [23] is an open platform offering containerization of applications. It separates applications from infrastructure, as each application is packaged in a loosely isolated environment, called a *container,* alongside its required libraries and dependencies. Thus, docker guarantees application interoperability, while simultaneously allowing multiple containers to run in the same host without interfering with each other, as shown in Figure 3.

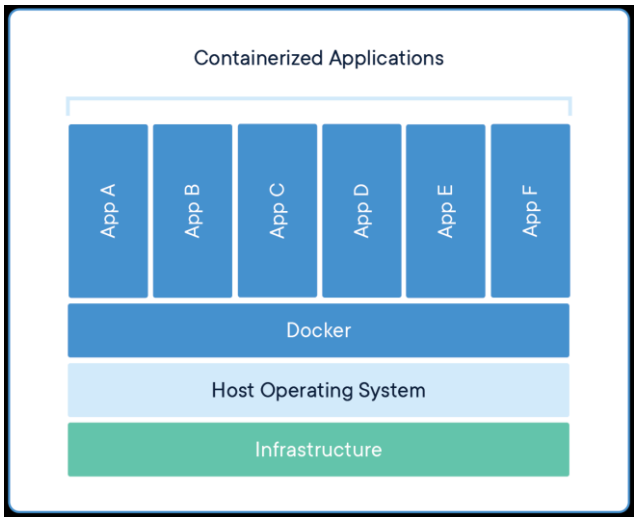


Figure 3: Docker Containerized Applications

Basic concepts of Docker are explained below.

Docker Image [24] : is the building base of a docker container. Developers can create their own images or use and extend images shared by other developers at public registries such Docker Hub.

Dockerfile [25] : is a text document used by docker to automatically build images. It contains all necessary instructions that are needed to assemble an image, in the form of commands. The commands are executed during the container creation.

Docker Container [26] : is a standardized unit containing all necessary libraries, dependencies and environmental variables of an application or service. It represents an instance of a running image.

Docker Volume [27] : is a mechanism for storing persistent data to be exchanged between a container and the host. Volumes create a link between a directory of the container and the host machine, enabling the sharing of stored data. Even if the container is deleted, volumes remain intact and can be bound to another container.

Docker Compose [28] : is a tool for specifying and running applications comprised of multiple containers. Compose utilizes a *YAML* file that defines a container's configuration, inter-connection, volumes and needed environment variables. By utilizing docker compose, developers are able to deploy multi-container applications with a single command.

3. iZen Federated IoT Architecture

iZen cloud nodes are deployed on Virtual Machines (VMs) that use the OpenStack platform. Each VM contains a docker that encloses iZen's system services in individual containers. An iZen node can be reached by users or other nodes via its provided public IP address.

iZen's system containers are deployed utilizing the docker compose tool. Docker's compose YAML file defines the base image used to create each service, the environment variables to pass inside the containers, and the necessary volumes bound to the containers. A private network needs to be created which will be used by containers for communication. Furthermore, a private IP address is assigned to each individual container. Finally, the YAML file describes the mapping of the services. The ports available for requests from users or other services are declared for each container. Docker provides two different port mechanisms for interaction with its container, the publish and the expose mechanism. The publish mechanism enables the assignment of ports to the containers, which are available for requests from outside the Docker. The expose mechanism allocates ports to each container, used for inter-container communication, via Docker's private network.

Consequently, users and node system services can send requests to a service (container), located in a remote cloud node, via the node's public IP address followed by container's publish port. Containers located in the same node, utilize private addresses alongside the exposed ports to communicate inside the established private network of the Docker.

3.1 iZen Architecture

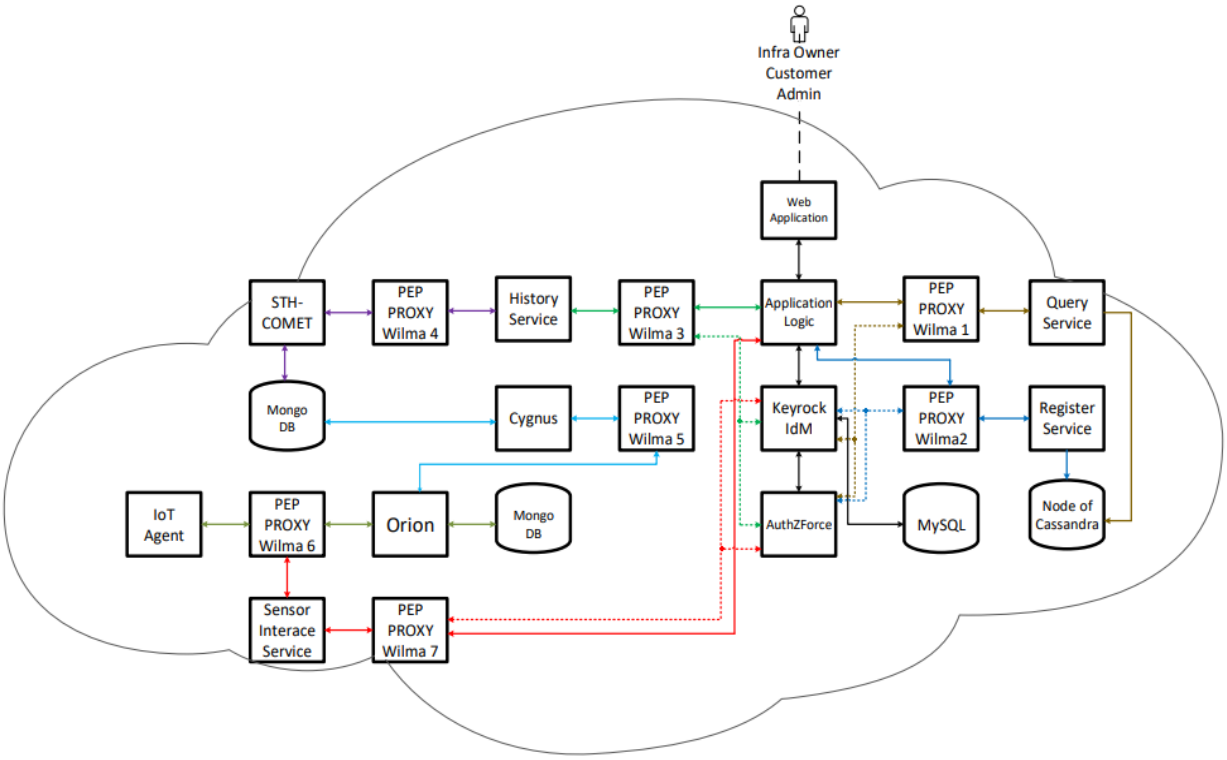


Figure 4: Mapping of iZen services

3.1.1 Front-End Services

Front-end services provide graphical interfaces to facilitate the interaction of cloud users with the iZen node. These services are accessible via the node's public IP and their assigned published port. Communication between these services and the users occurs over the public network.

All front-end services, except from the Identity Manager, are implemented from a newly created Dockerfile. The basic image for the Dockerfile is *7.1.11-apache* acquired from the official repository of PHP. The Dockerfile contains an implementation of Apache web server version 2.4.10 and the libraries for the programming language PHP version 7.1.11 which is used to write

the HTML pages that are served to the users from the front-end services. Additionally, the drivers to support communication with the Cassandra database utilizing PHP 7.1 were installed from DataStax [29]. Lastly, the ports where Apache web server accepts requests were exposed.

The functionalities of each front-end service are briefly mentioned hereafter.

Web application

The web application is the Login endpoint of each iZen cloud node. Users select the cloud node on which they are registered and are redirected to the graphical interface of the node's Keyrock IDM in order to fill in their credentials. After successful authentication Keyrock redirects them back to the web application where they can choose to access either the customers portal or the infrastructure owners portal corresponding to their role in the iZen cloud node. When users select a portal, a role check is performed in order to verify that the user is authorized to access it. Portals are intermediate HTML pages, between the web application and the provided services, which display the available actions for each group of users and redirects them to the corresponding service.

Register Service

The register service provides a graphical interface that can be used by Infrastructure Owners, Admins and Customers to perform the following actions.

- **Infrastructure Owners** can register their sensors by filling the necessary information, i.e., id, name, owner's details and the type of measurements that the sensor provides. Register service inserts the data to the cloud's Cassandra node so that the sensor can be discovered by interested Customers.

- **Customers** can subscribe to selected sensors to receive measurements. Register service updates customer's subscription list in the Cloud's Cassandra node.
- **System Administrators** can fill their nodes information, i.e., location, public IP, and owner's information, in order to register it in the iZen federation and make it discoverable to users. Administrators must acquire authorization to be permitted to register their node.

Query Service

Query service constitutes a search engine that enables users to discover sensors and cloud nodes via its provided graphical interface.

Query service offers fast and easy search by enabling users to filter by cloud node and type of sensor measurement. Before the search begins a permission check is performed to verify that the user is authorized to view the requested cloud's sensors. If the user is authorized, query service retrieves the appropriate sensors from the Cassandra node.

Additionally, users can utilize the service to discover the available cloud nodes registered in the federation and request to subscribe to any of them. A request for subscription is sent to the cloud's node administrator and if he/she approves it, the user can query for the sensors registered in that cloud node.

History Service

History service acts as an intermediate between Comet and the users by providing a graphical interface to interact with it.

Customers and Infrastructure Owners can select from the sensors they are subscribed to or provide and receive historical measurements. Users can specify a time frame that interests them or choose a specific metric of

measurements, e.g., max temperature during a month or average humidity of the week.

Sensor Interface Service

Sensor interface service acts as an intermediate between IoT-Agent and the Infrastructure Owners and enables them to install their sensors in their registered cloud.

Infrastructure Owners fill the characteristics of their sensor in the graphical interface and provide the necessary drivers for their sensor's functionality. Sensor Interface service proceeds to pass the information to the IoT-Agent to complete the installation.

Identification and Authorization Service

The Identification and Authorization Service is implemented based on FIWARE's Keyrock IDM image. It orchestrates system authentication and authorization in services based on the OAuth 2.0 protocol. Via its provided graphical interface users can perform the following actions:

Administrators: register their node's services in Keyrock IDM and declare which services and group of users may have access to them. They can declare which technologies are enabled for communication with the service and add a PEP-PROXY to protect it. Furthermore, they can write rule sets in form of XACML for the Policy Decision Point AuthZForce to assess the requests for access in each service protected by a PEP-PROXY. Finally, administrators define the groups of users that exist in their node and are able to register users in their system and assign them roles and their corresponding permissions.

Customers / Infrastructure Owners: create an account and manage their personal information through Keyrock's graphical interface. They have to make a request to the system administrator in order to join the cloud node. The request contains the intended role they want to receive; the available roles are customer or infrastructure owner. If the administrator accepts it, they are added

to their desired group of users and automatically receive the permissions that correspond to their group.

3.1.2 Back-End Services

Back-end components implement the core functionalities of an iZen system, i.e., data management, storage and retrieval, as well as incorporation of authentication and authorization into the system. These components are based on images provided by FIWARE's official repository, excluding the two MongoDB and the MySQL databases which are based on their own images acquired from the respective official DockerHub repositories. An exception, is the directory database, Cassandra, which is installed directly into the cloud VM and not inside a docker container. Although Cassandra is a back-end component, it may communicate with Cassandra nodes from other clouds via a public network, to exchange data.

Publish/Subscribe Service Orion

This service is implement based on the Orion Context Broker image provided by FIWARE. It is responsible for managing entities based on the NGSI-2 model. Orion receives a HTTP request when a sensor is register in the cloud through the IoT-Agent. It creates a sensor entity with the requested characteristics and stores it in MongoDB. Orion subscribes to the cloud's sensors, in order to receives notifications when the context is altered, and proceeds to inform all subscribers of the particular sensor about the relevant change. Finally, if a customer receives authorization and subscribes to a sensor installed in a remote cloud, Orion proceeds to subscribe to that cloud's Orion instance, in order to receive context changes of the sensor and to inform its customer.

Sensor Data Storage Service Cygnus

This service is implement based on the Cygnus image provided by FIWARE. It utilizes a specialized agent compatible with MongoDB that receives stream flows of data in the NGSI format from Orion Context Broker and proceeds to forward and store them in the historic MongoDB database. The Cygnus-specialized agent subscribes to each of Orion's sensors, so that when a measurement change occurs it gets notified. Upon notification, the agent receives and stores the data in raw and aggregated form in its designated database.

Historic Data retrieval Service Comet

This service is implement based on the Comet image provided by FIWARE. It combines individual time-stamped context data of an entity, that are stored in historic database, in order to create historic view. Moreover, via its RESTful API, it accepts request from the Historic service and retrieves raw and aggregated data from the database.

Authorization Policy Decision Point

This service is implement based on the AuthZForce image provided by FIWARE. It is responsible to assess requests from cloud users to access protected services.

AuthZForce receives a REST request whenever an administrator creates a policy rule, based on the XACML standard, for a PEP proxy and stores it in a different domain for each PEP. Afterwards, whenever a PEP forwards a request for access from a user, PDP checks the rule sets registered in PEP's corresponding domain in order to Permit or Deny access.

Policy Enforcement Point Proxy Server

This service is implement based on the Wilma image provided by FIWARE. The role of Policy Enforcement Points is to ensure that only authorized users or services are able to access protected services. There are two operating scenarios of PEP proxies:

The first scenario occurs when a PEP is an intermediate in the communication of two services. The requesting service must include the master key in the header of the request, in order to be authorized by PEP to access the service. The master key is a secret that was defined by the administrator of the cloud during PEP's creation and each PEP has a different one.

In the second scenario, PEP proxy intercepts the request of a user to the service. In this case, the collaboration of PEP with the Identity Manager and the PDP is necessary to ensure user authentication and authorization. Initially, PEP proxy receives a request for access from a user that includes an OAuth2 token in the header. An OAuth2 token is created during the user's login in the system by the IDM and represents the identity of the user and that he/she is authenticated by the IDM. PEP exports the token and sends it to IDM for validation. IDM verifies token validity using its database and responds with the user's role in the organization. Afterwards, PEP forwards the user's role alongside the desired action and the path of the protected resource to PDP to evaluate them. PDP checks the rule sets inside the domain corresponding to the PEP in order to make a decision to Permit or Deny the request. Finally, PDP returns its decision to PEP which enforces it.

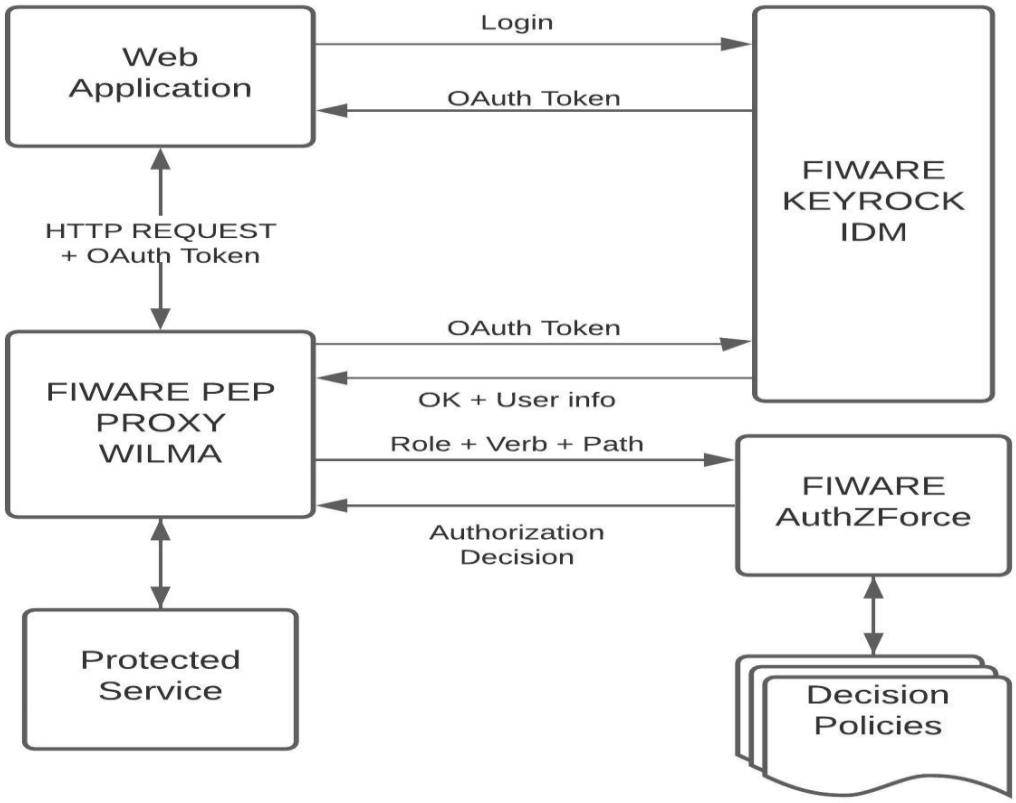


Figure 5:Workflow of authorization of a user's request

4. Implementation

iZen's system services support the HTTP/1.1 protocol for communication. Incorporation of the HTTPS protocol may introduce delay in iZen's system performance. It is essential to recognize points in the iZen system where communication occurs over public networks, i.e., points vulnerable to attacks, as well as points where communication is safe as it is encapsulated inside the iZen system.

This chapter presents the necessary modifications that need to be made in the YAML file of iZen, the necessary configuration for each service, as well as the procedure to generate certificates and encryption keys in order to enable communication over the HTTPS protocol wherever it is necessary. Furthermore, it explains how the authorization code flow of the OAuth 2.0 protocol and OpenID Connect were incorporated in the iZen system for authorization, authentication and additional security. Finally, utilizing OpenID Connect, token signature validation and Single Sign-On (SSO) is enabled through the iZen nodes for the users.

4.1 HyperText Transfer Protocol Secure Incorporation

This section presents the necessary steps to incorporate HTTPS into the system. Parts where the communication occurs over public networks are pinpointed, so that they can be subsequently addressed, in an effort to prevent attacks to the system. Furthermore, the procedure for certificate generation, which are utilized during communication, is demonstrated. Lastly, the modifications inside the docker's compose *YAML* file that took place in order to enable communication over HTTPS are presented.

4.1.1 Components Communicating over public network

Communication of iZen components is based on the HTTP/1.1 protocol. Due to this, upgrading to HTTPS introduces delay in the system's performance. It is essential to recognize and enable HTTPS only for components whose communication occurs over public network, in order to ensure security without heavily affecting performance. The components are presented hereafter.

Front-End Policy Enforcement Points Proxy Servers

PEP proxies (1,2,3 and 7), as shown in Figure 4, are stationed in the front-end. They ensure that only authenticated and authorized users are permitted to access the graphical interfaces of protected services. In a typical scenario, PEP receives a request from a user over the public network, confirms with the PDP and the IDM that the user is authorized, and in the end forwards them to the service through docker's private network. When SSO is utilized to access the node, PEP will communicate with the remote IDM to validate the guest user's access token.

Identification and Authorization Service Keyrock IDM

The graphical interface of the service is reachable through the public network so that administrators can manage their system and users are able to create their account and request entry to their desired group (customers or infrastructure owners). When registered users utilize the Single Sign-On functionality to access a remote iZen cloud, remote services request, through the public network, for an Oauth2 token to authenticate the user. The aforementioned function is described in detail in section 4.2.2.

Web application

Web application comprises the gateway to the iZen system, allowing user access through the provided graphical interface, over public networks. The web application authenticates the users and creates a user session with the necessary parameters for the system functionalities. Furthermore, the web application redirects users to the appropriate service, where they can perform their desired actions, if they are authorized to access the service. Consequently, incorporation of the HTTPS protocol in the web application is necessary.

Directory Database Cassandra

Directory database Cassandra is a back-end component accessible by the cloud's services through the docker's private network. Although it is a back-end service, in order to perform its function, it is necessary to communicate for data exchange with the Cassandra node, placed in each iZen cloud. The inter-node communication occurs over the public network; thus, incorporation of TLS/SSL is essential.

Figure 6 presents the mapping of system's services. Red-colored lines indicate communication occurring over public networks where HTTPS protocol must be enabled, while green-colored ones indicate communication through the private network that occurs over HTTP.

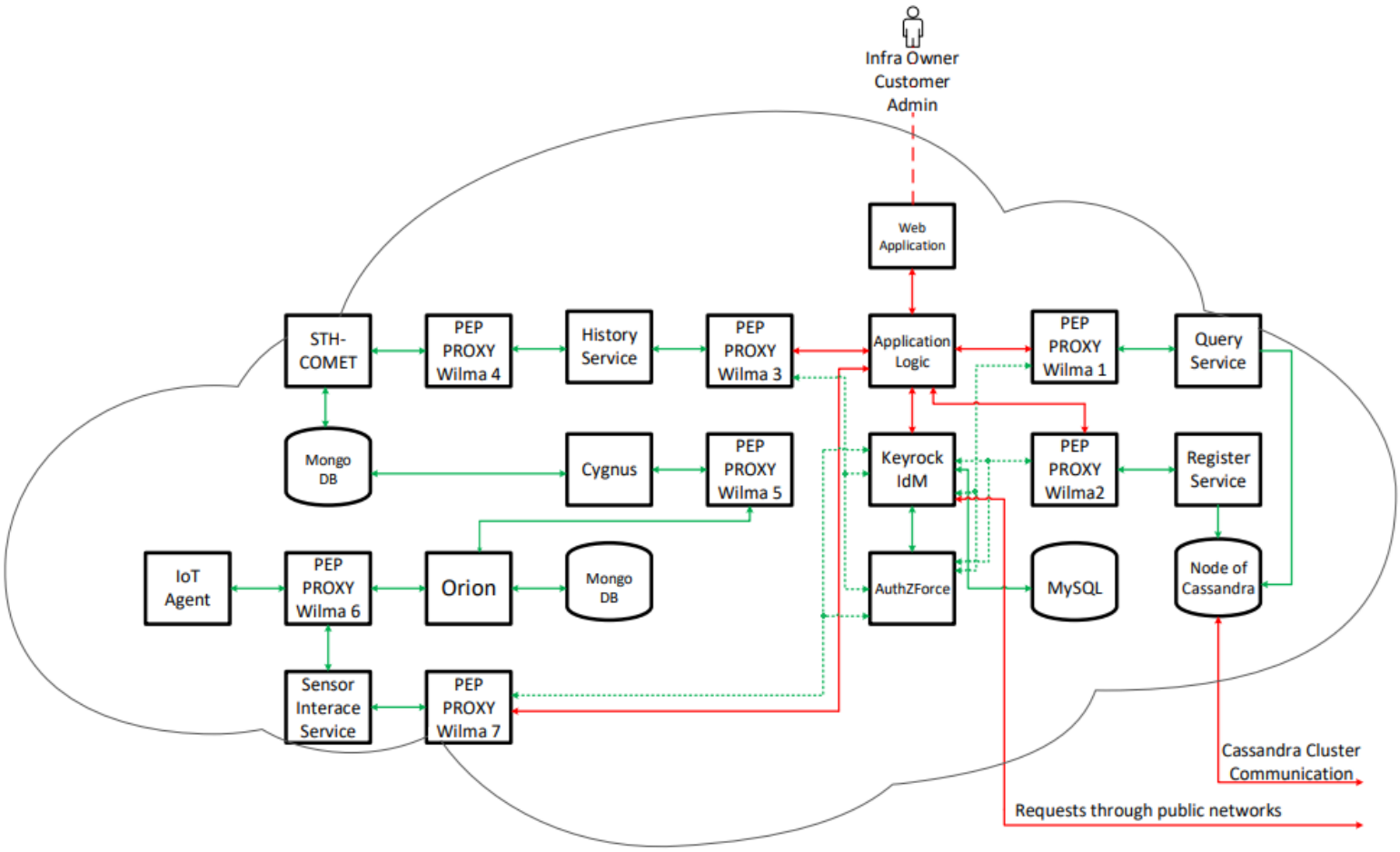


Figure 6: Distinction between public and private network communication

4.1.2 Certificate Creation

Certificates are core components of the HTTPS communication. They digitally bind an organization name with an encryption key and can be presented to client services to validate the authenticity of the server service. Each server service has to own a certificate in order to be able to communicate over the TLS/SSL protocol. A certificate verifies the owner of a domain and can be used by multiple services in that domain.

OpenSSL can be utilized for the creation of Certificate Authorities (CA), cryptographic keys, certificate sign requests, and finally the certificates. OpenSSL [30] is an open source toolkit that includes various cryptography-related libraries and allows users to perform TLS/SSL tasks in the form of terminal commands

Private Certificate Authority Creation

In this thesis, certificates are used for research and not for production purposes. For that reason, an official well-known CA is not needed and instead a private CA is used for signing certificate requests in order to subsequently generate certificates.

The first step for the creation of a private CA is the generation of a private key. Afterwards, a Certificate Sign Request (CSR) is generated, that contains all essential information of the certificate authority. Finally, the private key is used to self-sign the CSR in order to create the CA certificate. In this thesis, the above actions are implemented through CLI commands, using the OpenSSL toolkit.

The steps for the certificate authority creation and the parameters used in each command are described below.

Certificate Authority's Private Key Generation

Initially, the private key of the certificate authority is generated using the CLI command:

openssl genpkey -algorithm RSA -des3 -out CA-private-key.pem -pkeyopt rsa_keygen_bits:4096

- *genpkey*: OpenSSL command to generate a private key with the given parameters.
- *-algorithm RSA:* specifies which algorithm will be used for the public key generation. RSA is the standard algorithm for creation of

encryption keys. Other popular algorithms are RSA-PSS, EC, ED25519.

- *-des3:* indicates that OpenSSL must encrypt the private key utilizing the *Triple Data Encryption Algorithm (3DES)* [31] which applies the *Data Encryption Standard* (*DES) [32]* algorithm three times to each block of data. 3DES ensures that the key will remain safe even in the occasion that is stolen by a third party as it can not be used without decryption. When OpenSSL executes this command, it prompts the user to provide a password to be used.
- *-out CA-private-key.pem*: indicates where the produced key with be stored; the file *CA-private-key.pem* is used in this case. *PEM* is the most common format to store certificates and their keys. The content of the file is Base64 Enconded ASCII. Other popular formats are DER and PKCS#12. OpenSSL provides the necessary commands to convert one format to another, if a service requires a specific format.
- *-pkeyopt rsa_keygen_bits: 4096:* specifies the length of the generated key in bits; 4096 bits in this case. Encryption algorithms and key sizes affect system performance during communication. Different encryption algorithms require different key lengths to provide the same level of security.

Certificate Sign Request Generation

After generating the private key, a certificate sign request (CSR), that contains the necessary information of the domain owner, and the public key must be created and sent to an official CA. The CA reviews and confirms the included information and proceeds to digitally sign it, vouching for its validity. This procedure creates a certificate chain; meaning that each certificate is signed by a trusted CA. The command to generate the CSR is presented below.

openssl req -new -key CA-private-key.pem -out CA-csr.pem

- *req:* OpenSSL command for CSR creation. The generated CSR is in PKCS#10 binary format which is compatible with the X.509 format that will be used for the certificates.
- *-new:* generates a new CSR and prompts the user to fill the necessary information.
- *-key CA-private-key.pem:* Designates the path of the file containing the private key
- *-out CA-csr.pem:* Designates the output file of the command.

When the previous command is executed, OpenSSL requests from the user to fill the following information that will be included in the CSR and the certificate.

Country Name: The official two letter code of the country where the organization is located.

State or Providence Name: The full name of the state or the providence where the organization is located.

Locality Name (e.g., city): The city where the organization is located.

Organization Name (e.g., company): The full name of the organization.

Organization Unit Name (e.g., section): The department of the organization.

Common Name (e.g., fully qualified host name): The fully qualified domain name that the certificate will bind to. Alternatively, user may fill the public IP address of the host.

Email address: The email address of the host owner.

The host owner may choose to left some of these fields blank but some client services might consider the certificate invalid if it has empty fields.

Self-Signed Certificate Generation for the Certificate Authority

When an official CA receives a CSR, it reviews the included data and verifies that the host actually owns the relevant domain name that it received. The CA then proceeds to generate a certificate for the client that includes the digital signature of the CA. Finally, the CA sends the certificate to the client.

In order to create a certificate for the private CA that is used in this thesis, the CSR is signed by the private key that was previously generated. Thus, the private CA will have a self-signed certificate. In section 3.2.3 it is explained how the certificate of the private CA can be stored in the designated directory inside each service, alongside the certificates of the well-known CAs, in order to validate the created certificates during communication between services.

For the generation of the self-signed certificate the following OpenSSL command is used.

openssl x509 -in CA-csr.pem -out CA-certificate.pem -req -signkey CA-private-key.pem -days 365

- *x509:* OpenSSL command for certificate generation following the x509 standard. A certificate based on the x509 format contains a public key and the information of the owner. It is digitally signed by a CA or it can be self-signed. The x509 command of OpenSSL can be used to decode a x509 certificate in order to see its content.
- *-in CA-csr.pem:* indicates the certificate sign request file.
  -out CA-certificate.pem: indicates the output file where OpenSSL will store the certificate as well as its format.
- *-req:* indicates to OpenSSL that the input is a certificate sign request.
- *-signkey CA-private-key.pem:* indicates the key to be used to self-sign the certificate sign request.
- *-days 365:* indicates that the certificate is valid for 365 days. A certificate can be created to be valid for up to three years (1095 days).

When the certificate expires, the owner must make a renewal request to the CA that signed it, or create a new certificate.

Services Certificate Creation

OpenSSL provides various algorithms for the private key generation. In this thesis, two different encryption algorithms are studied, Rivest–Shamir–Adleman(RSA) [33] and Elliptic-Curve (EC) [34]). The influence of the two algorithms in system performance, during HTTP communication over TLS/SLL, is quantitatively measured and compared through a series of experiments.

Services Private Key Generation

For the generation of service private keys, utilizing the RSA algorithm, the same OpenSSL command is used as in the section about the certificate authority's private key creation. The only difference is the name of the key.

openssl genpkey -algorithm RSA -des3 -out Service-private-key.pem -pkeyopt rsa_keygen_bits:2048

For creation of private keys utilizing the Elliptic Curve algorithm, the following OpenSSL command is used:

openssl ecparam -genkey -name prime256v1 -out Service-private-key.pem

- *ecparam:* indicates to OpenSSL to generate elliptic curve parameters, which are used by the ECC algorithm.
- *-genkey:* is the OpenSSL command to generate a private key with the given parameters.
- *-name prime256v1:* indicates to OpenSSL which elliptic curve to use and the number of the bit prime field. Bit prime field is the length of the private key.
- *-out Service-private-key.pem:* indicates to OpenSSL where to store the generated private key and in which format.

Services Certificate Sign Request Generation

After creating the service private keys, a CSR, containing the public key and the information of the service owner, must be created. The CSR is sent to the private CA in order to be signed. The same OpenSSL command is used as in the section about the certificate authority's CSR creation. Of course, this time the private key of the service is provided.

openssl req -new -key Service-private-key.pem -out Service-csr.pem

Similarly to the previous usage of the command in this thesis, OpenSSL prompts the user to fill the necessary information of the service owner. In the field of the fully qualified domain name, the public IP of each VM in OpenStack that hosts an iZen system is declared.

Services Certificate Generation

Finally, for certificate generation, the CSR must be signed by the private CA. The below OpenSSL command is used for this task.

openssl x509 -req -in Service-csr.pem -CA CA-certificate.pem -CAkey CA-private-key.pem -CAcreateserial -out Service-cert.crt -days 365

Compared to the OpenSSL command used to create the CA's certificate the following parameters have been added.

- *-CA CA-certificate.pem*: indicates to OpenSSL the CA certificate that will be used to sign the CSR.
- *-CAkey CA-private-key.pem:* indicates to OpenSSL the private key of the CA that will be used to sign the CSR.
- *-CAcreateserial:* provides a unique serial number to the generated certificate. Each certificate issued by a CA must contain a serial number.

Figure 7 below demonstrates the content of a certificate after it has been decoded utilizing the x509 command of the OpenSSL.

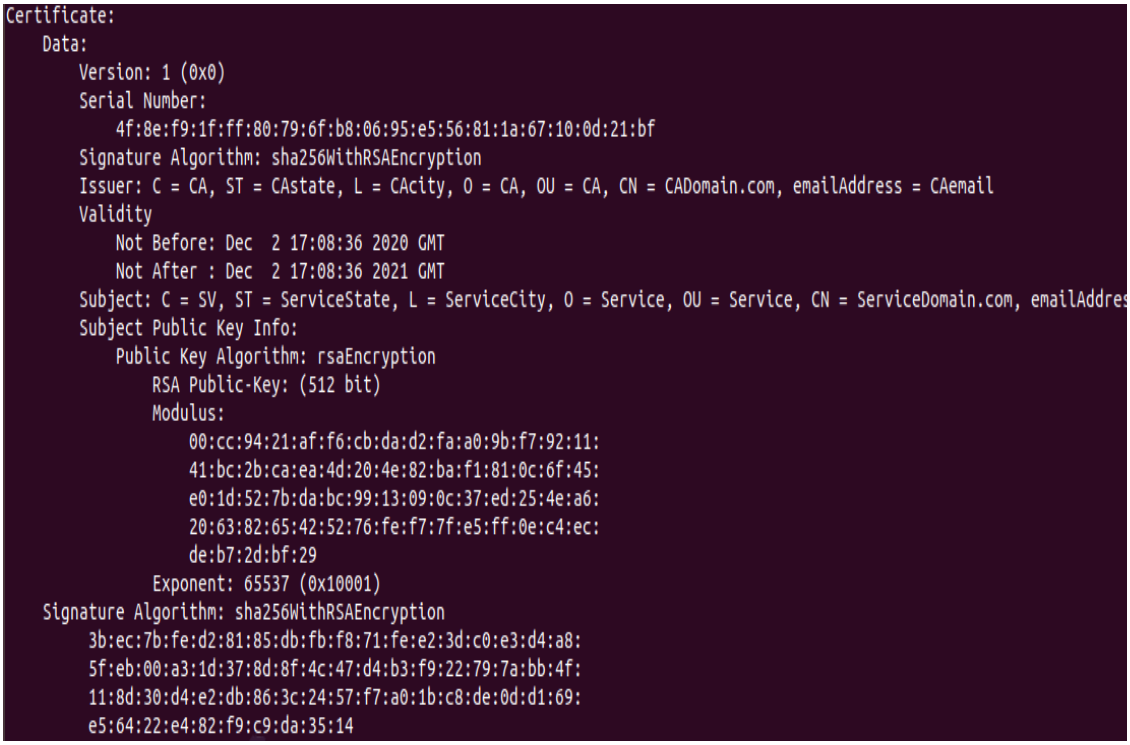


Figure 7: Service certificate content with a 512-bit RSA Public-key

The generated certificates and private keys must be installed in the designated store of each service.

4.1.3 PEP proxy Identity Manager and Web Application Modifications

The PEP proxy servers and the identification and authorization service are implemented in containers based on images provided by FIWARE. During the creation of the containers, the necessary modules for HTTP communication over TLS/SSL are installed but they are not enabled as HTTP is the default protocol for service communication.

FIWARE provides specified environment variables that facilitate the containers inter-connection and the enabling of the HTTPS protocol in these

services. The environment variables are declared in the *YAML* file that the docker compose tool uses for the creation of the containers and their values are passed in the configuration file of their service, when the container is created.

In order to incorporate the HTTPS protocol in the communication of the services, the modules responsible for HTTPS must be enabled. Moreover, HTTPS-specific port declaration is necessary. As the certificates and the private keys are persistent data, volumes have to be mounted in the appropriate directories of each service.

Identity Manager's Modifications

Initially, the identity manager is configured, via the provided environment variables located inside the YAML file, to use the HTTPS protocol for communication over the public network and HTTP for the communication inside the docker. The utilized environment variables are mentioned below.

- IDM_HTTPS_ENABLED=true: enables the module inside the container of IDM that is responsible for HTTPS communication.

- IDM_HTTPS_PORT=3443: declares the published port that only accepts request based on the HTTPS protocol. The graphical and the REST interface of IDM will be reachable through the public network via this port.

- IDM_PORT=3005: declares the exposed port that will be used by the services to reach IDM through the private network. This communication occurs over the HTTP protocol.

After declaring the environment variables, it is necessary to publish the HTTPS-communication port and expose the HTTP port in the YAML file.

expose:

- "3005"

ports:

- "3443:3443"

As a last step, two new volumes must be created. The first volume is mounted in the directory where the service stores the CA certificates. The second volume is mounted in a new directory in order to store the service's certificate and private key. The volumes need to be declared in the YAML file.

volumes:

./CAcert:/etc/ssl/certs

./IDMcert:/opt/fiware-idm/certs

CA's certificate is placed inside the *CAcert* file of the VM while the service's certificate and private key are stored in the *IDMcert* file.

PEP Proxy Modifications

The front-end PEP proxies (1,2,3 and 7) as shown in Figure 4 must be modified to accept HTTPS requests via the public network, decode them, and forward them to their protected service. These PEPs communicate, via docker's private network, with the identity manager and the policy decision point, in order to provide authentication and authorization of the users.

The necessary changes are implemented in the YAML file utilizing the environment variables below.

- ***-PEP_PROXY_HTTPS_ENABLED=true:*** enables the module inside the container of the PEP that is responsible for HTTPS communication.
- ***-PEP_PROXY_HTTPS_PORT=:*** indicates the PEP's HTTPS port that receives requests through the public network.
- ***-PEP_PROXY_APP_SSL_ENABLED=false:*** indicates that PEP's protected service does not communicate over the SSL protocol, thus the requests must be decoded by the PEP before it forwards them to the service.
- ***PEP_PROXY_PORT=:*** indicates the PEP's HTTP port used for communication through the docker's private network.

- ***PEP_PROXY_IDM_SSL_ENABLED=false:*** indicates that the communication between the PEP and the IDM will not occur over SSL.
- ***PEP_PROXY_IDM_PORT=3005:*** indicates IDM's HTTP port.
- ***PEP_PROXY_AZF_PROTOCOL=http:*** indicates that the PEP will communicate with the AuthZForce PDP over the HTTP protocol.

PEP's HTTPS port must be published, in order to receive HTTPS request through the public network, and the HTTP port must be exposed for the inter-container communication. Furthermore, the creation of two volumes is necessary for storage of the CA's certificate and the private key and certificate the of the PEP.

volumes:

./CAcert:/etc/ssl/certs

./PEPcert:/opt/fiware-idm/cert

Web Application Modifications

The necessary modifications to enable communication over SSL/TLS in the web application are implemented in the *Dockerfile* that is used to create the web application image and in the *docker-compose.yaml file* that creates the containers of the iZen node in the docker.

Inside the Dockerfile, two additional commands have been added. The first one *"RUN a2enmod ssl"* enables the Apache web server module responsible for handling SSL/TLS communication. The second command *"RUN a2ensite default ssl"* indicates to the Apache web server to utilize the default-ssl.conf file for the site. The configuration file has been altered to indicate the path to the previously generated service certificate and key. Finally, port 443 is exposed, which is the designated port to handle HTTPS requests.

The file docker-compose.yaml file must also be altered. The HTTPS port 443 is published in order for the container to receive HTTPS requests from the users through the public network. Three additional volumes, shown below, are mounted in the container, for storage of the service key, service certificate, and CA's certificate.

volumes:

./CAcert:/etc/ssl/certs

./ServiceCert/private:/etc/ssl/private

./ServiceCert/cert:/etc/ssl/certs

4.1.4 Cassandra Directory Database Modifications

Cassandra database uses SSL/TLS certificates to offer *client-to-node* encryption and *node-to-node* encryption. In iZen's system client to node communication occurs via the secure private network, thus it will not be enabled.

Cassandra stores the CA's certificate in a dedicated directory inside a *truststore file*, and the node's private key and certificate in a *keystore file.* It secures these files by requiring a password to permit the access to them. Consequently, a new CA certificate and nodes certificates must be generated that will include the above passwords. For the generation of the certificates OpenSSL toolkit is used. Afterwards, *java keytool [35],* a tool provided by java that facilitates certificate and key management, is utilized to place the CA's certificate in a truststore and the node's private key and certificate in a keystore and then deposit them to their respective directories.

The procedure to enable node-to-node encryption is described below.

Cassandra's Certificate Authority Creation

Initially, a configuration file is created that defines the key pair configurations and includes a password and the necessary information of the CA. This file is named CA.conf and its format and content is shown below.

[req]

distinguished_name = **CA_DM**

output_password = **rootca_password**

prompt = no

default_bits = 2048

[**CA_DM**]

C = **CC**

O = **org_name**

OU = **cluster_name**

CN =**CA_CN**

where:

- **CA_DM:** The distinguished name of the Certificate Authority
- **rootca_password:** The password for the generated file that will be used to sign certificates. For this root CA the password will be cassandra.
- **CC:** The official two letter country code of the CA.
- **org_name:** The organization name of the CA.
- **cluster_name:** The name of the cluster that is formed from each cassandra node in the iZen's cloud systems.

- **CA_CN:** The common name for the root CA. For this CA the CN will be CassandraCA.

After creating the configuration file, a private key and a self-sign certificate will be generated utilizing the OpenSSL toolkit with the below command.

openssl req -config CA.conf -new -x509 -keyout CAkey.key.pem -out CAcert.crt.pem -days 365

The new parameter *-config CA.conf* indicates to Openssl to utilize the information inside the configuration file to fill the fields of the CA certificate.

At last, utilizing the java keytool, a new truststore will be created that will contain the CA's certificate. This truststore have to be installed on each cassandra node to verify incoming connections. Official CAs certificates are preinstalled in truststores in the cassandra. The java keytool command to generate the truststore and its parameters are explained below.

keytool -alias CassandraCA -keystore CAtruststore.jks -importcert -file CAcert.crt.pem -keypass cassandra -storepass cassandra -noprompt

- *-alias CassandraCA:* indicate the common name of the CA and is used by the keytool for identification when importing the certificate in the keystore.

- *-keystore CAtruststore.jks:* indicates the keystore name to be created and the store type. The default store type is JKS while other common types are JCEKS and PKCS12.
- *-importcert:* indicates to keytool to import the certificate file.
- *-file CAcert.pem:* indicates to keytool the file containing the certificate.
- *-keypass cassandra:* The keypass that is used to protect the private key.
- *-storepass cassandra:* The storepass that is used to access the keystore.
- *-noprompt:* keytool automates the procedure and use the default options in order to not prompt the user.

Cassandra's node Certificate Generation

Each Cassandra node must own a certificate that is signed by the CA and is store inside a keystore in the appropriate directory.

Java keytool is used for the generation of a private key and the certificate for the node. Afterwards, the above pair is stored inside the keystore. Java keytool's command that implements the aforementioned tasks alongside its parameters are described hereafter.

keytool -genkeypair -keyalg RSA -alias IZENnode1 -keystore IZENnode1-keystore.jks -keypass cassandra -storepass cassandra -validity 365 -keysize 2048 -dname "CN=IZENnode1, OU= IZENcluster, O=IZEN, C=GR" -ext "san=ip:node_ip_address"

- *-genkeypair:* keytool command for the generation of a private key and a certificate with the mentioned parameters.
- *-keyalg RSA:* indicates to keytool to use the RSA algorithm for the generation of the private key.
- *-validity 365:* indicates that the certificate will be valid for 365 days.

- *-keysize 2048:* indicates the size of the generated key i.e., 2048.
- *-dname:* indicates to keytool the information of the node that must be contained inside the certificate.
- *-ext "san=ip:node_ip_address":* indicates the IP of the Cassandra node.

Utilizing keytool a certificate sign request that includes the generated node certificate from the keystore will be created.

keytool -alias IZENnode1 -keystore IZENnode1-keystore.jks -keypass cassandra -storepass cassandra -certreq -file IZENnode1.csr

- *-certeq:* keytool command for the generation of a certificate sign request.
- *-file IZENnode1.csr:* indicates the output file of the sign request.

The generated CSR must be signed by the CA's public key for the creation of a valid node certificate. The below OpenSSL command is used for this task.

***Openssl -req -CA CAcert.crt.pem -CAkey CAkey.key.pem -in IZENnode1.csr -out IZENnode1.crt.signed -days 365 -CAcreateserial -passin pass:cassan*dra**

Compared to the OpenSSL command that was used to generate the PEP and IDM certificates, an additional parameter has been added.

- *-passin pass:cassandra:* this parameter indicates to OpenSSL the password to access the CA's certificate.

Finally, the generated signed node certificate alongside the CA's certificate must be stored into the keystore in order to create a certificate trust chain.

Keytool -alias IZENnode1 -keystore IZENnode1-keystore.jks -import -file IZENnode1.crt.signed -keypass cassandra -storepass cassandra -noprompt

Keytool -alias CassandraCA -keystore IZENnode1-keystore.jks -import -file CAcert.crt.pem -keypass cassandra -storepass cassandra -noprompt

Cassandra's Configuration File Modifications

After the generation and the storage of the node's certificates in the appropriate keystores, Cassandra's configuration file, *cassandra.yaml,* is edited in order to enable node-to-node SSL encryption. The section of variables that enables node-to-node SSL encryption already exists in the default cassandra.yaml file but it is commented, thus it is inactive. The aforementioned variables that need to be enabled and their edited values in order to activate inter-node SSL communication are mentioned hereafter.

server_encryption_options:

internode encryption: rack : when the variable's value is set to rack Cassandra will enable node-to-node SSL encryption.

keystore: /home/cassandra/certs/IZENnode1-keystore.jks : indicates to Cassandra the path to the node's keystore.

keystore_password: cassandra : indicates to Cassandra the necessary password to access the keystore.

truststore: /home/cassandra/certs/CAtruststore.jks : indicates to Cassandra the path to the node's truststore.

truststore_password: cassandra : indicates to Cassandra the necessary password to access the truststore.

The above procedure is repeated for each Cassandra node in the system to enable node-to-node SSL encryption. It is essential that the same

private CA signs all the certificates of the nodes in order for them to be accepted as valid by other nodes. In the case that an official well-known CA is used to sign and create the node's certificate, that CA's certificate must be added in the same keystore in the node with the generated node certificate for the creation of a certificate trust chain.

4.2 Authorization Code Grant, OpenID Connect and Single Sign-On Implementation

In this thesis, iZen system is extended in order to reduce the chances of exposing users credentials and improve the authentication and authorization of its users during their access to the system. This section describes the implementation of the mechanisms that were incorporated in the system to achieve this goal.

4.2.1 Authorization Code Grant Implementation

FIWARE's Keyrock IDM complies with the OAuth 2.0 protocol and provides all four different grant types. Authorization code grant type is the best practice [36] and is promoted by companies as it offers better security compared to the other types. The procedure to incorporate authorization code grant type for the access of the users in the system is described hereafter.

When an administrator registers his cloud in the IDM, he/she must register each system's service as an application. The fields mentioned below are crucial for IDM's functionality and are utilized to secure user's access.

- ***URL:*** The administrator declares the applications URL. Only OAuth requests from this URL are accepted by IDM.
- ***Callback URL:*** The user agent will be redirected to this URL when the OAuth flow is finished.
- ***Sign-out Callback URL:*** The user will be redirected to this URL when he/she signs out from the application.
- ***Grant Type:*** The administrator chooses the OAuth grant type that will be enabled in the application. When the administrator selects the

authorization code grant type two unique identifiers, the Client ID and the Client Secret are generated.

Now that the application is registered and the necessary variables have been declared the steps of the authorization code grant type and how each step is implemented are described below.

1. A user accesses the graphical interface of the web application and chooses to connect with his IDM account. The web application sends a *GET* request to /oauth2/authorize endpoint of the IDM requesting a *code.* The request must have in its header the parameters (*response_type=code* which indicates to IDM to respond with a code, the *Client ID* that was created when the administrator registered the web application in the IDM, and the *redirect uri* which indicates the location where the user agent will be redirected with the generated code).
2. The user is redirected to IDM's graphical interface and provides the email and the password of his IDM account. IDM initially confirms that the incoming request is from the URL that was declared during the application registration and that the received redirect_uri matches the declared callback URL. Then, it proceeds to search the registered users in its database in order to authenticate the user. After successfully authenticating the user, IDM creates an authorization code. Finally, it responds with the HTTP status code 302 Found, the generated code and redirects the user to the indicated callback URL.
3. When the web application receives the authorization code, it must exchange it with an *access token.* This time, the web application makes a POST request to the /oauth2/token endpoint of the IDM. In the header field of the request an *Authorization Basic* header is included. This header is the identity of the service and its value is generated by joining the two unique identifiers, Client ID and Client Secret, with the symbol ":" between them (i.e client_id:client_secret) and encrypting them using the base64 method (i.e. base64(client_id:client_secret)). The body of the request includes the parameters (grant_type=authorization_code which indicates

to IDM to respond with an access token, the received code from the previous request, and the redirect uri).

4. IDM validates the identity of the application by decoding the authorization header and confirming that the client id and client secret match the generated ones during the registration of the application. It proceeds to generate an access token and stores it in his database. Furthermore, it creates a user session that includes the oauth sign in. At last, it responds with the HTTP status code 200 OK in the header field of the response, and the main body, which is in JSON format, and contains the generated access token, indicates that the type of the token is bearer, declares the expiration time of the access token, and provides a refresh token which can be exchanged by the application to automatically refresh the access token if it expires while the user continues his session.
5. When the web application receives the IDM's response, it extracts the access token and binds it to the user's session. The access token signifies the user's identity and it needs to be provided to the PEPs in order for them to authenticate the user and authorize his access to the protected resources.

If the user has previously sign in IDM though the provided graphical interface, IDM would have already created a session for the user, due to this, it will not prompt the user to fill in his credential, and it will instantly respond with a code at the step 2 mentioned above.

When the user desires to log out of the web application, the user sessions that are created inside the application and the IDM must be deleted. The flow that occurs during the user's log out is described below.

1. The user clicks the logout button in the application. This triggers the application to send a *DELETE* request to */auth/external_logout* endpoint of the IDM. The request includes the client id, facilitating the IDM to find and delete the user's session.

2. The IDM finds the application's domain inside his database, based on the provided client id, and deletes the stored session of the user. It proceeds to redirect the user agent at the */Destroy_session.php* endpoint of the web application, which is the designated Sing-out Callback URL.
3. The web application destroys the parameters of the stored user session and redirect the user to the log in page of the application.

The complete authorization code grant type flow that occurs during the users login and logout is presented in the Figure 8.

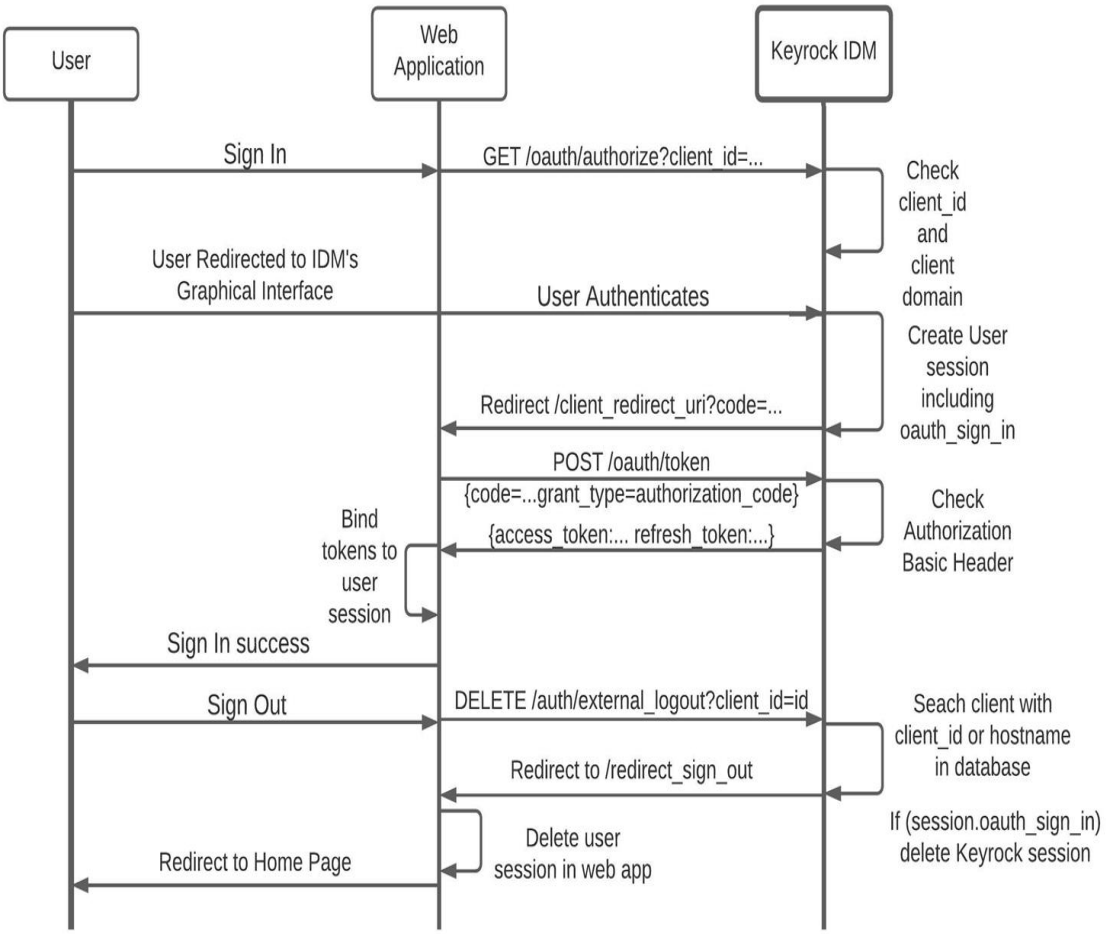


Figure 8: Authorization code grant type flow

4.2.2 OpenID Connect, Signature Validation and Single Sign-On Implementation

OpenID connect builds upon the OAuth 2.0 protocol and extends it in order to offer improved user authentication and authorization in the system. Furthermore, it improves the security of the system as it facilitates the creation of a signature validation mechanism for the generated tokens. In the latest version of Keyrock IDM, FIWARE incorporated the necessary functionalities that allow users to enable OpenID connect in their applications through the graphical interface provided by Keyrock.

When the cloud administrator enables the OpenID connect, Keyrock IDM responds when requested with an access token and an additional Jason Web Token (JWT), the id_token. A JWT consists of three parts, the *Header*, the *Payload*, and the S*ignature*. Each part is encoded in JSON and they are joined by the symbol "." between them (i.e. json_encode(Header).json_encode(Payload).json_encode(Signature)). The decoded content of each part of the id_token is in JSON format and includes the information below.

Header: declares which algorithm is used for the signature or the encryption, and what type of token the id_token is.

Payload: contains information of the organization where the user is a member and its role in that organization. Furthermore, it provides the user's email, username and unique identifier inside the system. Lastly, it provides information of the token itself i.e. (the issuer of the token, the subject that the token refers to, the audience that the token is intended for, the expiration time of the token and the time that the token is issued).

Signature: is a keyed hash value generated using the HMAC method. The parameters of the method are: a hashing algorithm i.e., sha256, the message to be hashed which consists of the base64URL encoded header joined with the base64URL encoded body by the symbol "." between them, and a shared secret key used for generating the HMAC variant of the message digest.

The secret is generated when the administrator enables the OpenID connect protocol in the application. The token's signature is a mean to verify the integrity of the claims inside the token and validate its authenticity.

The authorization code grant type flow is slightly different when the OpenID protocol is enabled. The altered steps of the flow compared to the previous section are described hereafter.

Step 1: The web application additionally includes the *scope=openid* parameter in the GET request to the /oauth/token endpoint of the IDM. This parameter indicates to IDM that the OpenID connect protocol is used and that the IDM must respond with an id token that includes the claims mentioned above in the *payload* section.

Step 4: The IDM responds with the access token and the JWT id token. The signature of the id token is created using the shared secret that was generated when the administrator enabled the OpenID connect protocol in the registered web application in the IDM.

Step 5: The web application extracts the access token and the id token from IDM's response. It proceeds to split the id token in order to receive each part i.e. (header, payload, signature) and performs signature validation utilizing the shared secret. If the signature is valid, the web application allows the access of the user in system and binds both the access token and the id token in the user's session as they will be utilized for the user's authentication and authorization afterwards.

The steps 2 and 3 of the login flow as well as the logout flow remain the same.

The complete authorization code grant type flow that occurs during the users login and logout while the OpenID protocol is enabled is presented in Figure 9.

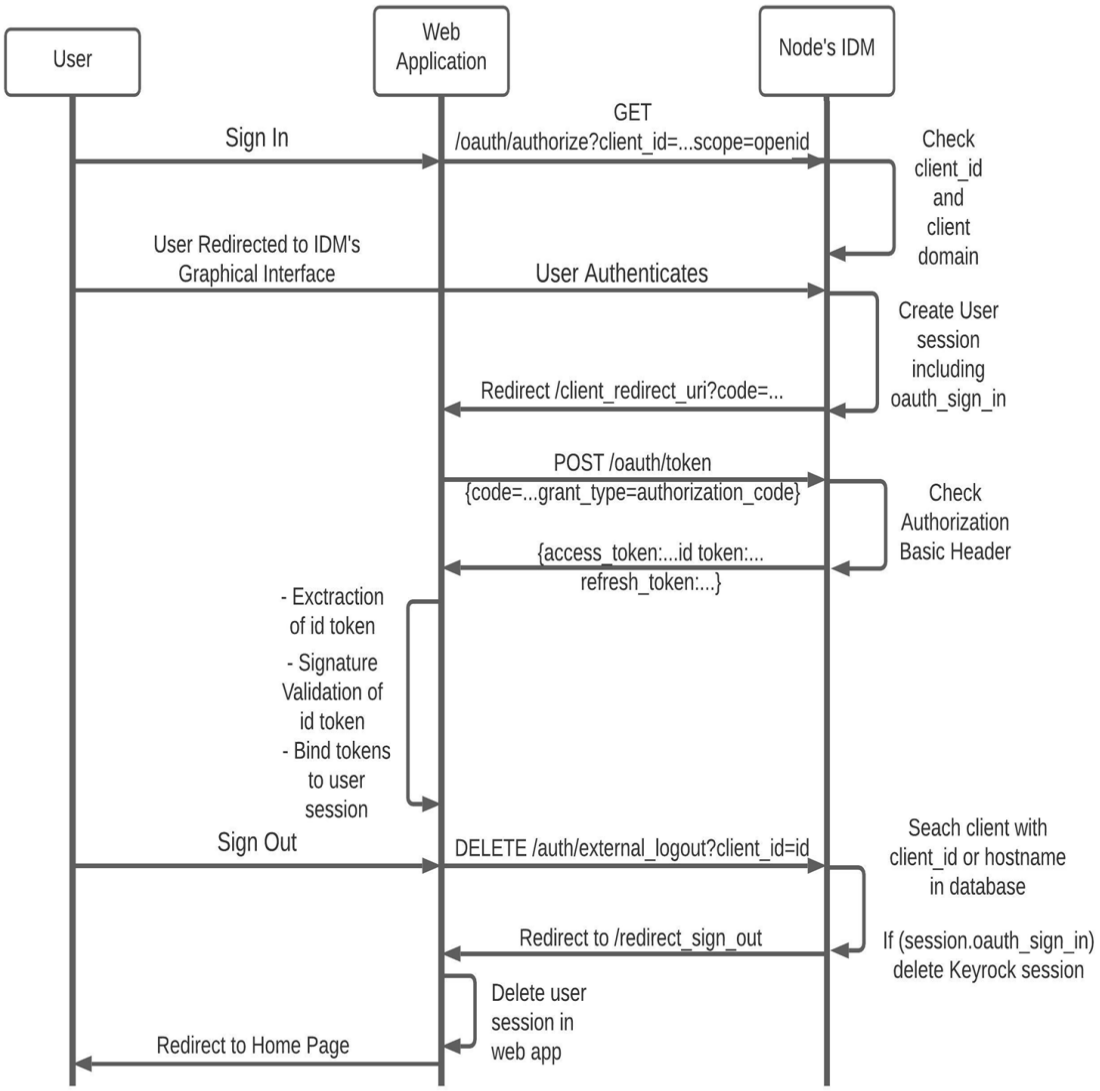


Figure 9: User login and logout flow when OpenID is enabled

Signature Validation Process

Signature validation guarantees the integrity of the id token's content and that the sender is the IDM and not a third-party service that imposes as the IDM. In order to implement the signature validation a secret that is known only to the IDM and the application is utilized. When the web application receives an id token, it proceeds to generate an "expected" signature based on the token's header, payload and the shared secret. If the expected signature matches the

received signature of the id token, the token is considered valid, thus the user may access the system. Conversely, if the two signatures do not match, the access is denied and the user is redirected to the initial login page. The implementation of the signature validation in the web application of the system is described hereafter.

1. The web application extracts the id token from the IDM's response. It splits the id token at the symbol "." in order to receive the JWT token's three parts i.e.(header, payload, signature) and stores them into variables. The parts are base64 encoded.
2. The header and the payload are decoded into text by utilizing the function base64_decode() of the PHP programming language.
3. For the generation of the expected signature the header and the payload must be encoded in base64URL format. PHP does not include a function for base64URL encoding, thus a new function is created for this purpose. The new function named base64urlEncode receives as input a text and replaces the symbols "+", "/" and "=" with "-", "_" and " " respectively. Then, it encodes the text using the PHP function base64_encode(). The output variable is in base64URL format.
4. After encoding the header and the payload in base64URL format the expected signature can be created. For this purpose the hash_hmac function of PHP is utilized. This function generates a keyed hash value using the HMAC method. It receives as input the hashing algorithm, the data and a HMAC variant of the message digest. The hashing algorithm utilized is the *sha256*, the data consist of the base64URL encoded header joined with the base64URL encoded payload with the symbol "." between them i.e. (base64urlEncode(Header).base64urlEncode(Payload)), and the HMAC variant is the shared secret.
5. Finally, the web application compares the expected signature with the received signature. If the two signatures match the id token is

considered valid as only someone who possesses the secret i.e., the IDM, can generate this signature.

Single Sign-On implementation

Incorporation of the OpenID connect protocol in the communication of the system provides the necessary mechanisms for the implementation of single sign-on functionality among the iZen nodes. The JWT id token includes all the essential information to authenticate and authorize a user in the system. Additionally, the signature validation guarantees the integrity of the claims that the id token contains, and verifies that the token is generated from a trusted IDM.

In order to explain the single sign-on functionality in this section the two interacting iZen nodes will be referred as *node A* and *node B.* The user is registered in the node A, therefore this node provides the id token for the user authentication and authorization. Node B is the remote node where the user desires to have access to.

For the implementation of the single sign-on functionality among node A and node B, the administrators must perform the following actions.

The administrator of node A must register a new application in his IDM. The *URL, Callback URL, and Sign-out Callback URL* of this application will indicate the appropriate URL's at the domain of the node B. Keyrock IDM ensures that only requests from node's B domain will be accepted to this application. Additionally, the administrator designates that the application's grant type is authorization code grant in order to generate the unique identifiers of the application, i.e., client id and client secret. Moreover, he/she enables the OpenID connect protocol in the application, therefore IDM will respond with an id token when it receives a code. Finally, the id token secret that was generated when then administrator of node A enabled the OpenID connect protocol and the unique identifiers of the application are shared with the administrator of node B.

The administrator of node B stores the received id token secret and application identifiers inside its web application. Furthermore, he/she must

modify the application to redirect the users to node's A graphical interface of the IDM to fill their password and email, in order to receive a code and exchange it for the id token. Moreover, the logout procedure must be modified to send a *DELETE* user session request to the appropriate IDM. Finally, the web application must identify if the guest user is a customer or an infrastructure owner, and bind the respective role i.e. (guest customer or guest infrastructure owner) to the user's session. Consequently, the administrator must create the aforementioned roles in his organization and assign them the appropriate permissions. Hence, the administrator is able to control the resources which the guest users are authorized to access.

The flow of a guest user's login to a remote node is briefly discussed hereafter.

1. The guest user from node A visits the web application of node B and selects to login with his registered account in node A. The web application sends a *GET* request to /oauth2/authorize endpoint of the IDM located at node A requesting a *code.* The request includes the client id of the newly registered application that accepts requests from node B and the scope=openid parameter. The redirect URI in the request indicates the HTML page of the node B.
2. The user is redirected to the graphical interface of node's A IDM to fill in his credentials. The IDM verifies that the request is coming from a trusted domain and authenticates the user. It responds to the indicated redirect URI with an authorization code.
3. The web application receives the authorization code and makes a POST request to the /oauth2/token endpoint of the IDM located at node A requesting to exchange the received code for an access token and an id token. The *Authorization Basic* field of the header consists of the client id and client secret that the administrator of node B received and stored in the web application from node's A administrator.

4. Node's A IDM decodes the *Authorization Basic* field of the header and validates the identity of the application. It proceeds to generate an access token and an id token of the authenticated user, which is signed using the shared token secret. It stores a user session in his database and responds to the web application with the generated tokens.
5. Node's B web application extracts the id token from the response. It proceeds to split the token and performs signature validation. If the signature is valid, it extracts the user's role in node A from the claims of the id token. If the user is a customer the web application will assign the role of the *guest customer* in node B, while if the user is an infrastructure owner it will assign the role of the *guest infrastructure owner.* Finally, the web application creates a user session and binds the access token, the id token and the user's new role to it. When the user desires to access a protected resource, the PEP sends the user's new role to the PDP for assessment of the request based on the rule set that the administrator registered for this role. Furthermore, if a service requires some user's credentials for its functionality, it extracts them from the id token.

When the user desires to logout from the node's B system the web application sends a *DELETE* request to */auth/external_logout* endpoint of the IDM located in node A. IDM deletes the stored user session from its database and redirects the user agent to the indicated sign-out callback URL at node B. Afterwards, node's B web application deletes the user session that was created in the web application, and redirects him/her to the login home page.

The complete flow when a guest user logins and logouts to a remote node is presented in the Figure 10 below.

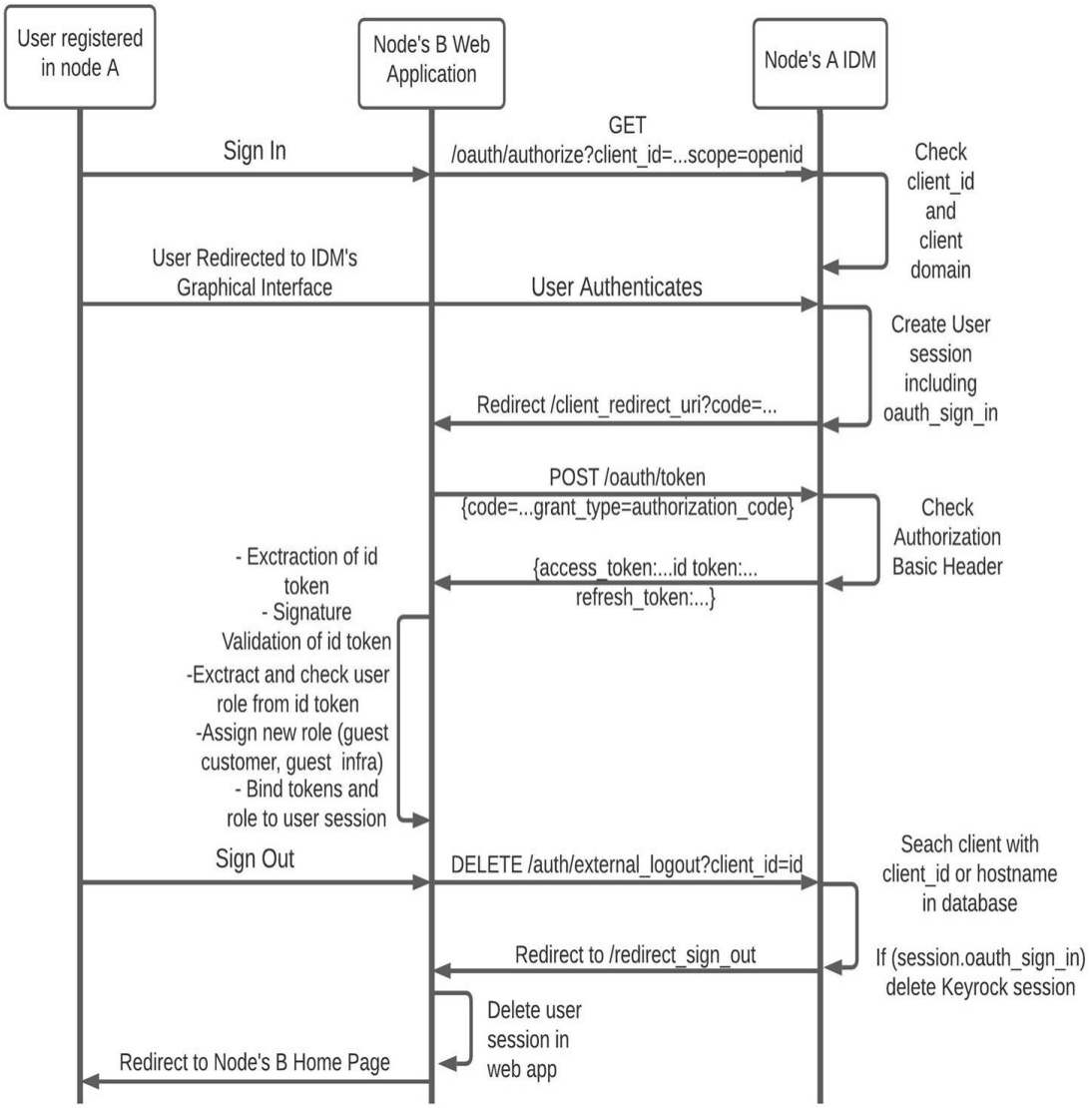


Figure 10: Single Sign-On login and logout flow

5. Performance Evaluation

A series of experiments were conducted in order to evaluate the performance of the system after enabling the HTTPS and the OpenID connect protocols in communication, as well as incorporating the authorization code grant, the signature validation mechanism and the Single Sign-On functionality among the cloud nodes in the system.

5.1 Infrastructure for Conducting Experiments

In total, three cloud node systems were used for the conduction of the experiments. Two of them were developed in virtual machines on the Intellicloud of Technical University of Crete at Chania. One cloud node was developed in a private physical machine located at Athens in order to take realistic measurements that include the network delay.

The technical features of the virtual machines and the physical machine are as follows:

CPU	4 VCPU
MEMORY	8GB
HDD	80GB
OS	Ubuntu 18.04 LTS

Figure 11: VMs Technical Features

CPU	Intel i7-4CPU:510U @ 2.00GH, 2 Cores, 4Logical Processors
MEMORY	8GB
SSD	500GB
OS	Ubuntu 18.04 LTS

Figure 12: Physical Machine Technical Features

5.2 Apache Bench Tool

Apache Bench [37] is a load testing and benchmarking tool for HTTP and HTTPS servers. It can simulate a large number of requests to be handled by a service and it allows the user to set a concurrency, i.e., the number of requests that the service must execute at the same time.

The two metrics used in this study are:

Time taken for tests i.e., the total time that the service needs to execute all the requests at a specified concurrency.

Time per request mean i.e., the average time that the service needs to execute a batch of requests at a specified concurrency.

Apache Bench was installed in each VM and the physical machine, in order to evaluate the system performance.

In each of the following experiments, Apache Bench was utilized to simulate 2000 service requests. Each experiment was repeated for five different concurrencies: 1, 50, 100, 150 and 200.

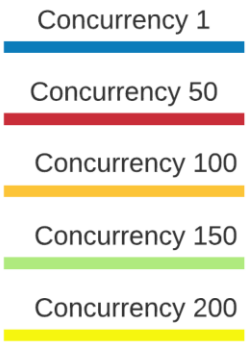


Figure 13: Color coding of concurrency, used in the diagrams below.

The performed experiments take into consideration the network delay. It is measured that the included delay is 10 to 11ms per request.

5.3 Experiment 1 – HTTP vs HTTPS

This experiment evaluates the response time of the web application and compares the performance penalty of using HTTPS compared to HTTP.

Details: A user that lives in Athens visits the home page of the web application which is located at a cloud node at Chania with IP address 147.27.60.43 and its published port is 8060. The requests are simulated from the Apache Bench tool installed in the machine located at Athens, hence network delay is included in the measurements. When HTTPS protocol is enabled, the certificate installed in the web application contains a 2048-bit public key that was generated using the RSA encryption algorithm.

REST: When HTTP is enabled GET requests are send to http://147.27.60.43:8060. Resultant times measured for HTTP protocol are presented in the diagrams below.

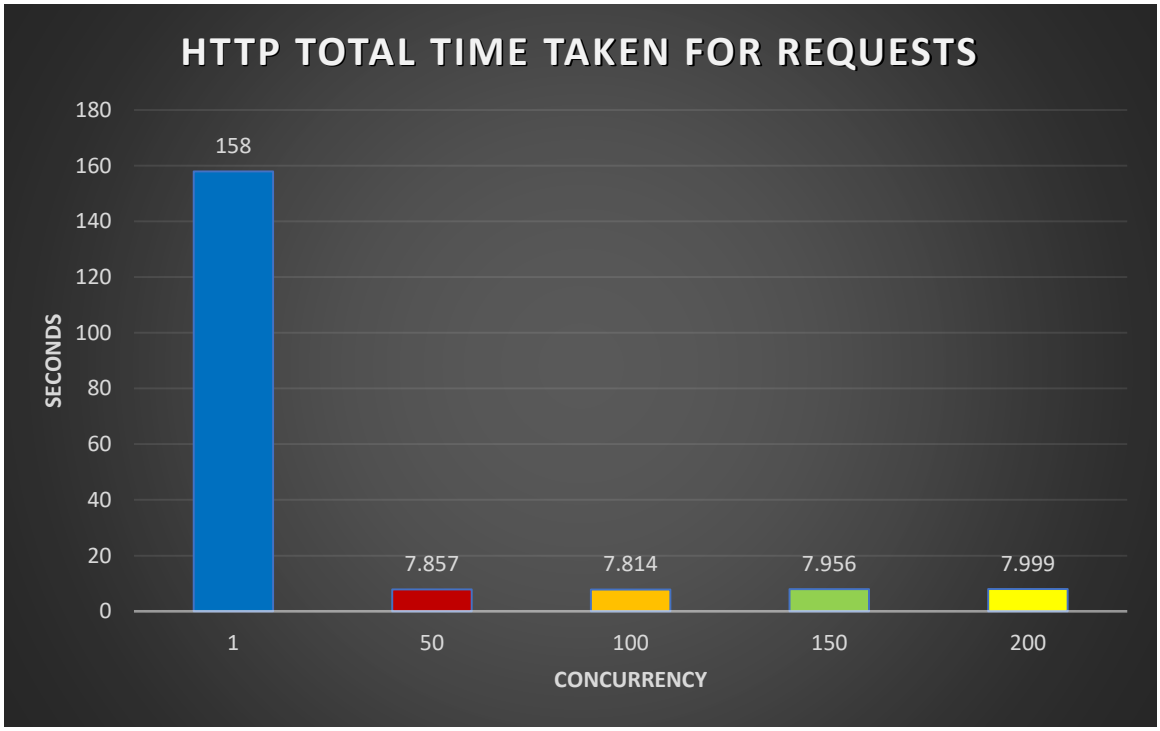


Figure 14: Total Time Taken for requests while HTTP protocol is utilized

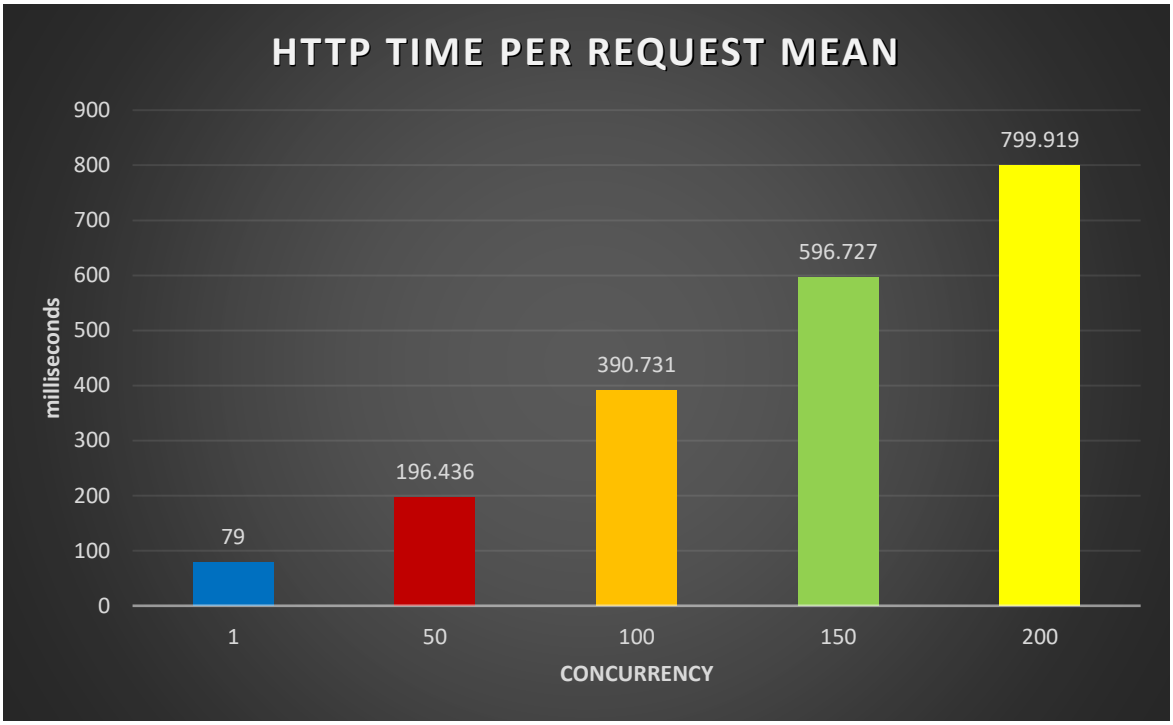


Figure 15: Time per request mean while HTTP protocol is utilized

REST: When HTTPS is enabled, GET requests are send to https://147.27.60.43:8060. Results are presented in the diagrams below.

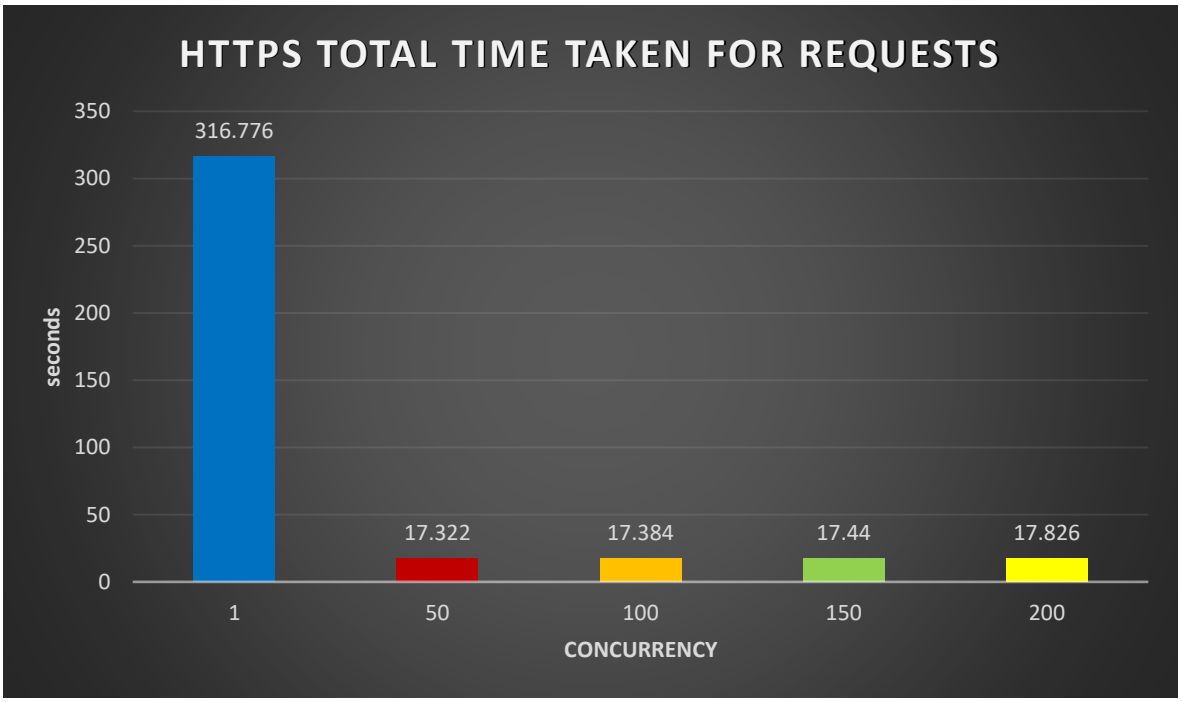


Figure 16: Total Time Taken for requests while HTTPS protocol is utilized

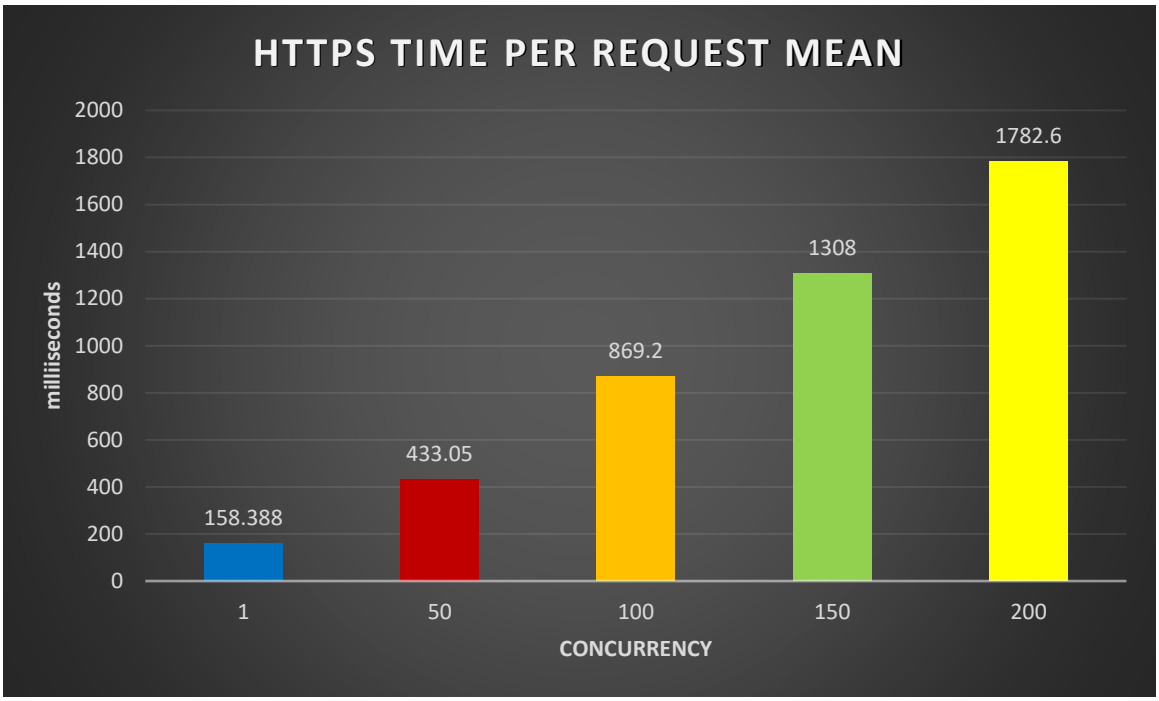


Figure 17: Time per request mean while HTTPS protocol is utilized

The Diagrams below compare the total time taken and the time per request at each concurrency between HTTP and HTTPS.

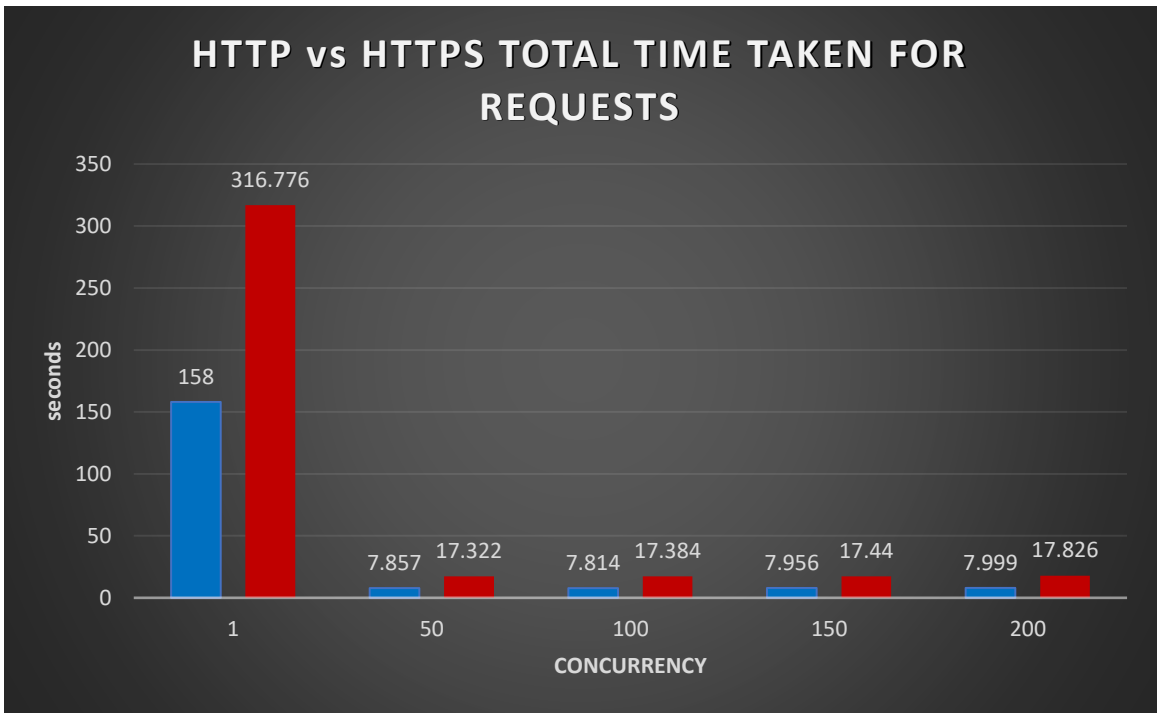


Figure 18: Total Time Taken for request comparison between HTTP and HTTPS

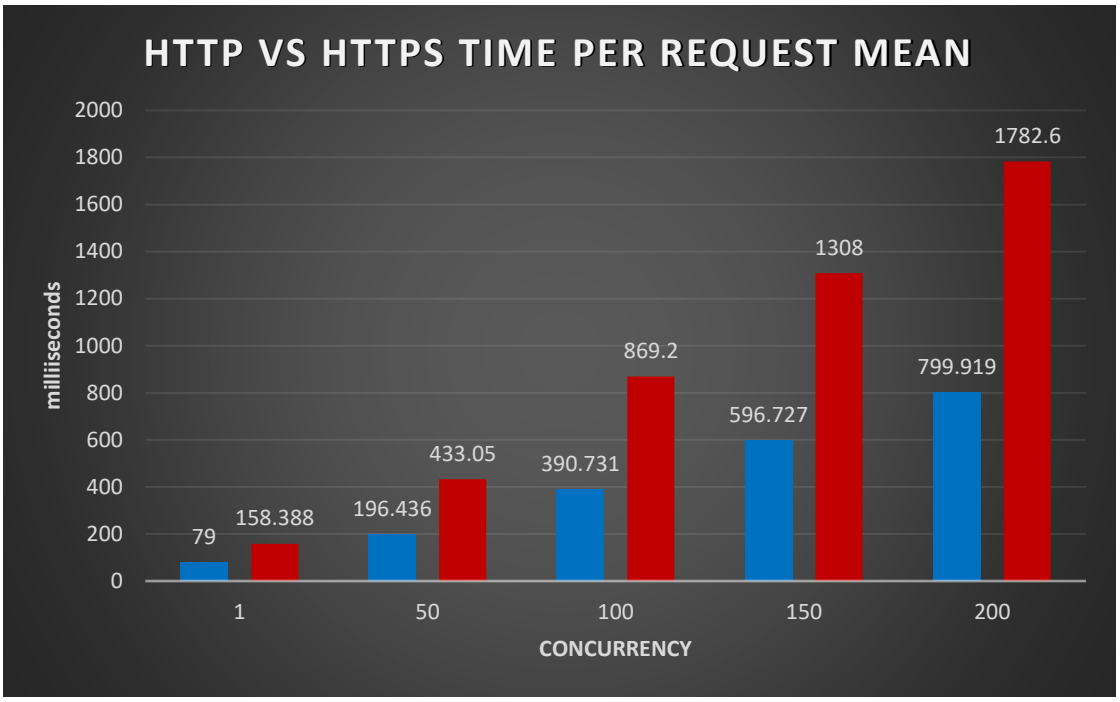


Figure 19: Time per request mean comparison between HTTP and HTTPS

The comparison of the metrics demonstrates that enabling the HTTPS protocol in the web application introduces significant delays, as the required time to execute the same number of requests at the same concurrency doubles.

5.4 User login to remote cloud node utilizing the Single Sign-On (SSO) functionality

When a user initiates a login at a remote cloud node utilizing the SSO functionality provided by OpenID Connect, a two-part communication occurs between the node's web application and the IDM where the user is registered.

In the first part of the communication, the web application requests an authorization code from IDM and redirects the user to its graphical interface. IDM prompts the user to fill in their email and password. After successfully authenticating the user, IDM responds with an authorization code to the web application.

In the second part, the web application sends a request to IDM, that includes the previously acquired authorization code, in order to exchange it for an access token and an id token. IDM responds accordingly. The received access token is utilized for authorization of the user, while the id token is for authentication as it includes all necessary user and issuer information.

Experiment 2 measures the response time of the remote IDM, when the web application requests an authorization code. Experiment 3 measures the time needed for IDM to exchange the previously generated authorization code for an access token and an id token.

5.4.1 Experiment 2 – Request for authorization code while SSO is utilized for user login to remote node

The second experiment quantifies the response time when the web application requests an authorization code from an IDM, which is located at a remote cloud node. As the two communicating services are located in different cloud nodes, the HTTPS protocol is utilized.

Details: A user registered at a cloud node located at Chania initiates a login utilizing the Single Sign-On functionality, from the cloud node located at Athens. The web application, in the Athens cloud node, sends a request for authorization code at the cloud IDM at Chania. The IP address of the node at Chania is 147.27.60.43 and the IDM published port is 3443. IDM's certificate contains a public key that was generated using the RSA encryption algorithm and its length is 2048 bits.

REST: The Athens web application sends a GET request at https://147.27.60.43:3443/oauth2/authorize. The request header contains (1) the parameter "response_type=code", (2) the client ID of the registered application at Chania's IDM, which accepts requests from the Athens node, (3) the "scope=openid" parameter, and (4) the redirect URI.

The measured metrics are presented in the diagrams below.

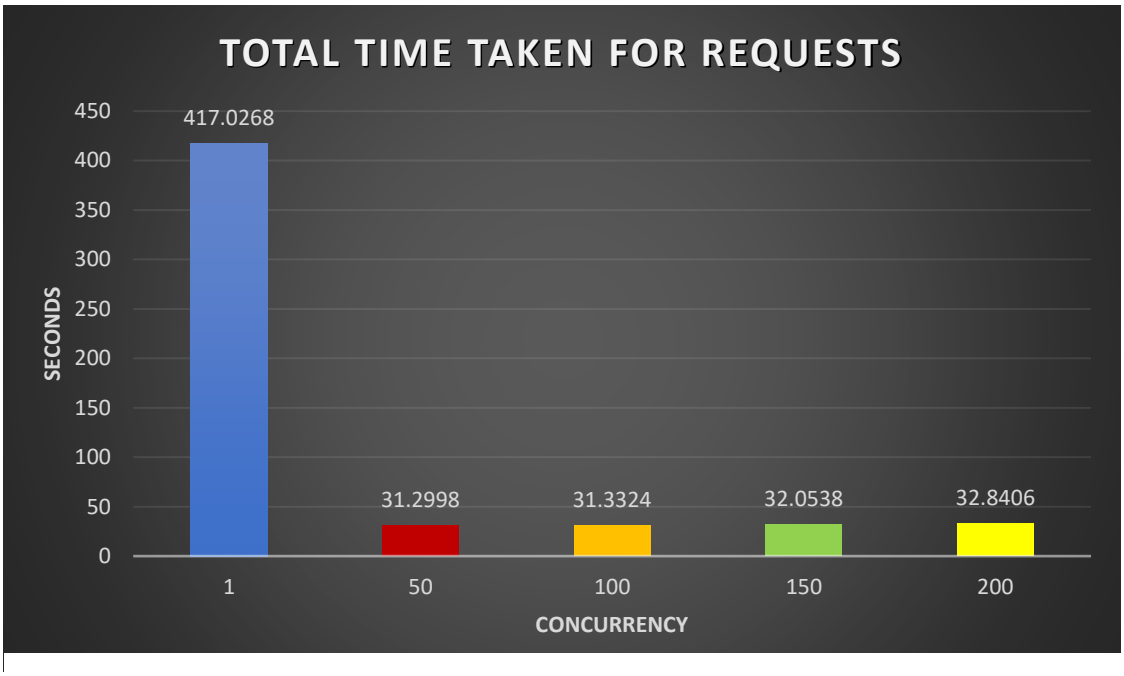


Figure 20: Total Time Taken for requests when the web app requests for authorization code while SSO is utilized

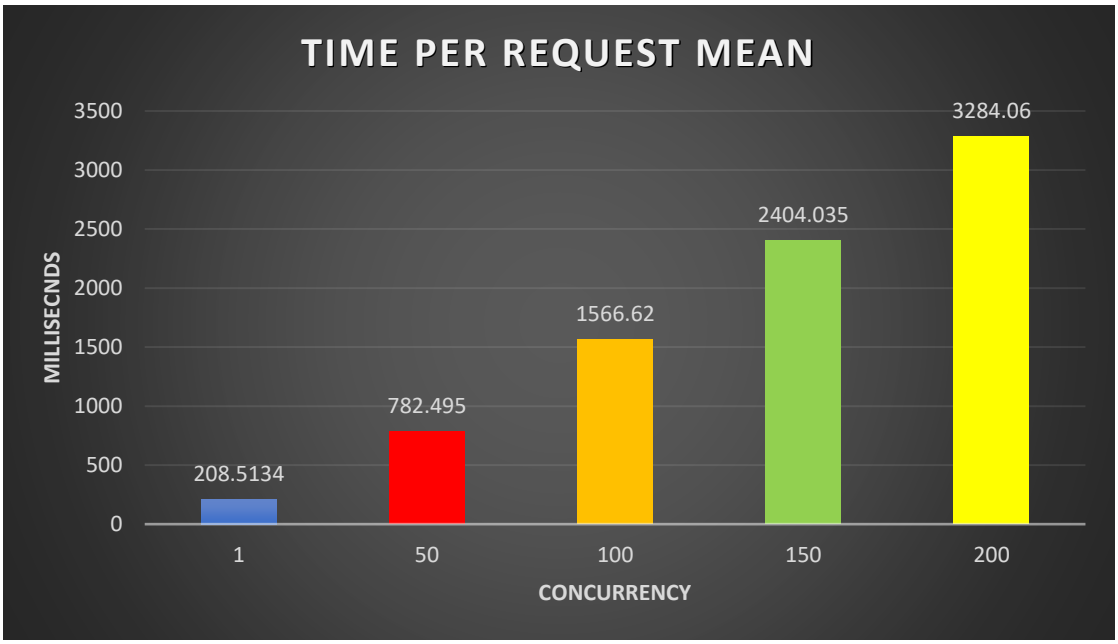


Figure 21: Time per request mean when the web app requests for authorization code while SSO is utilized

5.5.2 Experiment 3 – Exchange of authorization code for access token and id token while SSO is utilized

In the third experiment, the response time is measured when the web application makes a request to exchange the previously acquired authorization code for an access token and an id token from an IDM located at a remote cloud node. As the two communicating services are located in different cloud nodes, the HTTPS protocol is utilized.

Details: A user registered at a Chania cloud node initiates a login to the cloud node located at Athens, utilizing the Single Sign-On functionality. The user was previously redirected at Chania's IDM graphical interface to fill his credentials and was successfully authenticated by the IDM. Subsequently, the IDM responded with an authorization code for Athens's web application. The web application of Athens's cloud node sent a request at Chania's cloud IDM, in order to exchange the received code for an access token and an ID for the authorization and authentication of a guest user.

The IP address of the Chania node is 147.27.60.43 and the published IDM port is 3443. IDM's certificate contains a public key that was generated using the RSA encryption algorithm and its length is 2048 bits.

REST: Athens's web application sent a POST request to https://147.27.60.43:3443/oauth2/token. The request header contains the *Authorization Basic* field consists of the client id and client secret that the administrator of Athens's node received and stored in the web application from Chania's node administrator. The body of the request contains, in JSON format, the field "grant_type=authorization_code", the received authorization code, and the redirect_uri.

The measured results are presented in the diagrams below.

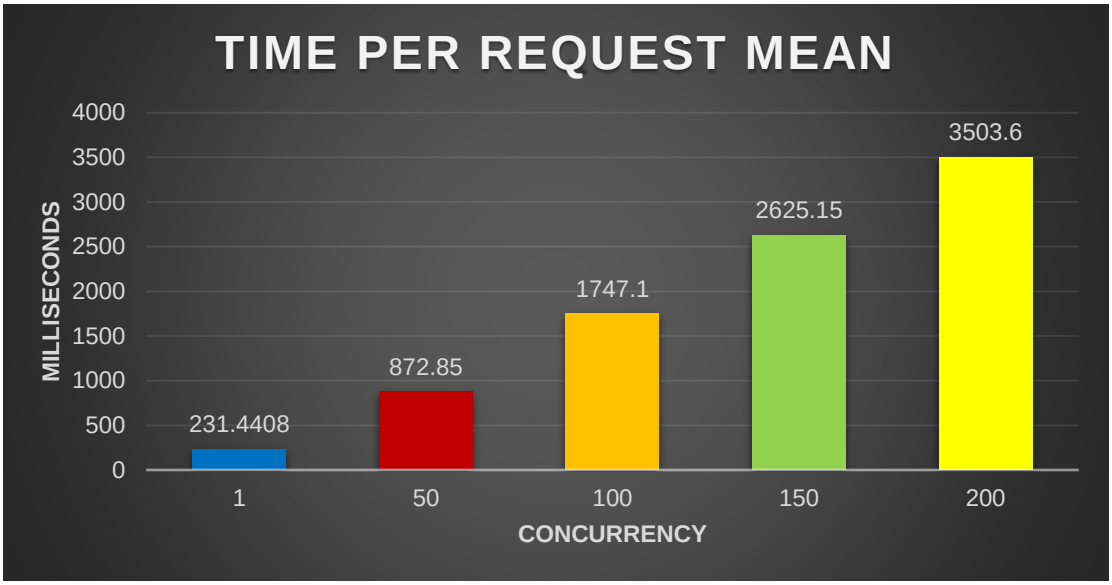


Figure 22: Time per request mean when the web app requests for Access and ID token while SSO is utilized

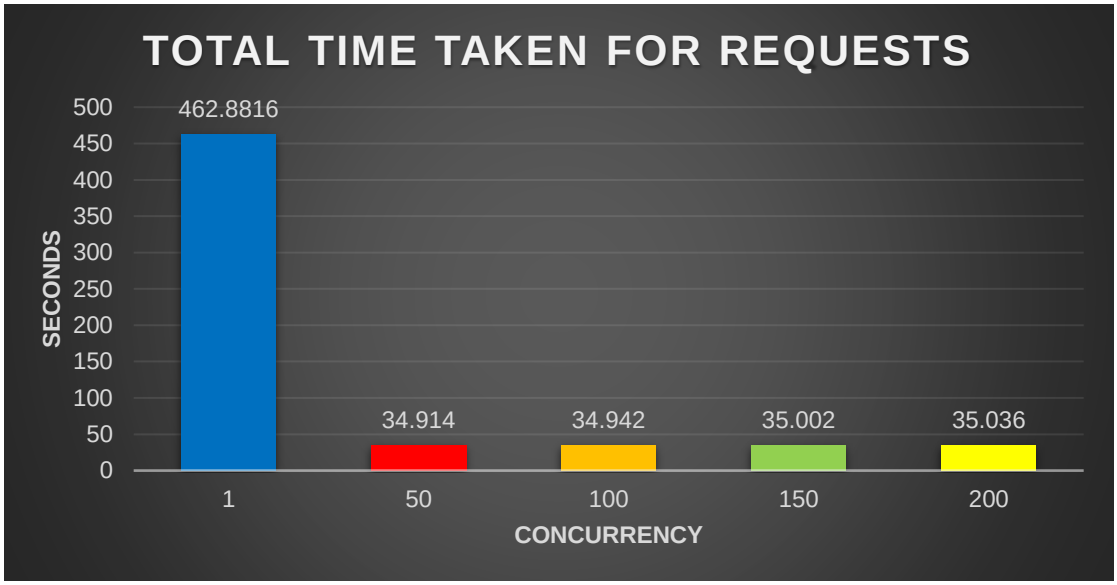


Figure 23: Total Time Taken for requests when the web app requests for Access and ID token while SSO is utilized

5.6 Experiment 4 – RSA Encryption Algorithm Evaluation

This experiment evaluates the response time of the web application when the HTTPS protocol is enabled and the application's public key, that is contained in the certificate, was generated utilizing the RSA encryption algorithm.

Details: The web application is located at a cloud node at Chania with IP address 147.27.60.43 and its published port is the 8060. Utilizing the RSA encryption algorithm, two public keys with different lengths, *2048* and *4096* bits, were generated and respectively installed in the web application, in order to study how different key lengths, affect system performance. The requests are simulated from the Apache Bench tool, installed in the machine located at Athens, in order to take into consideration, the network delay.

The diagram below compares the total time needed for the web application to execute all the requests at each concurrency, when the two RSA keys, with different lengths, were utilized.

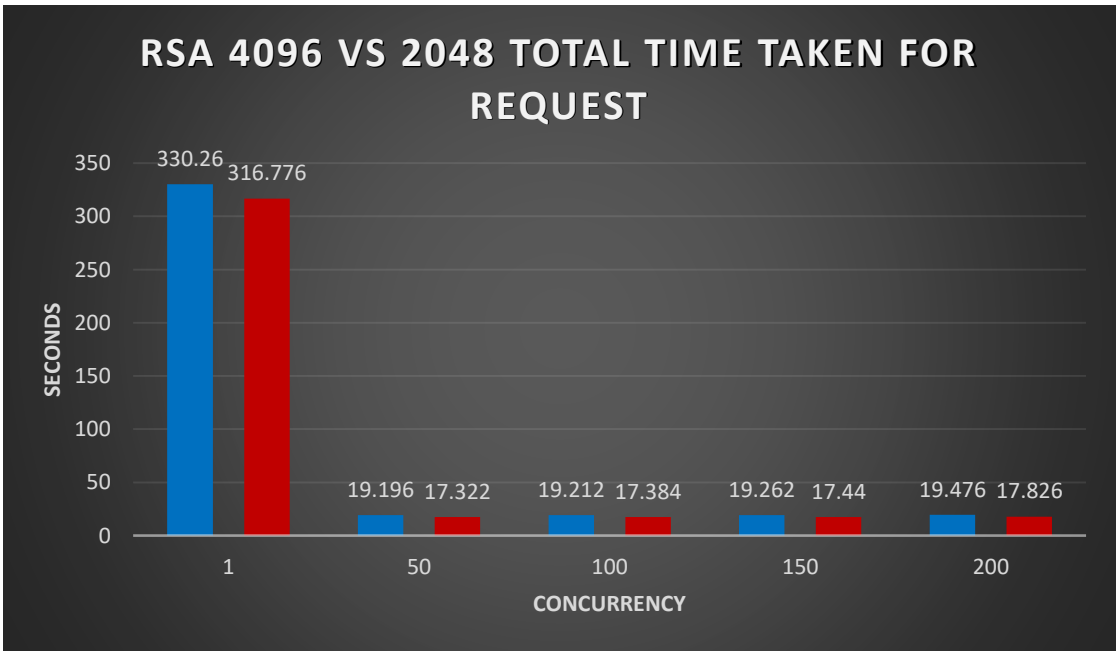


Figure 24: Total time taken for requests comparison between the RSA keys

The following diagram compares the average time that the web application needed to execute a batch of requests at each concurrency when the two RSA keys with different lengths were utilized.

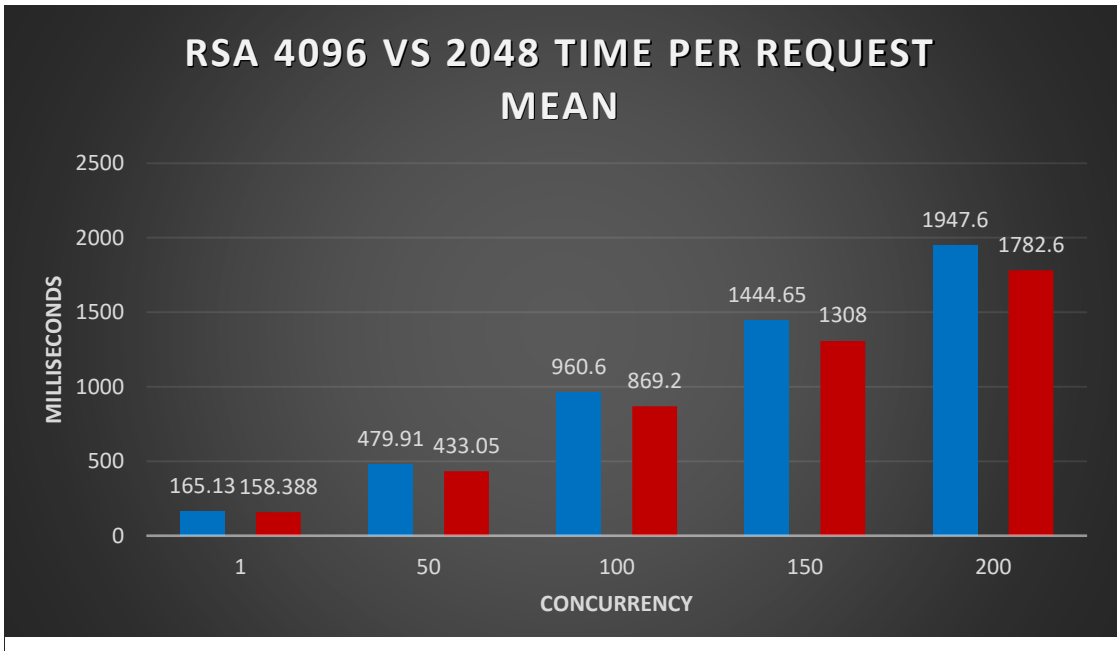


Figure 25: Time per request mean comparison between the RSA keys

As expected, the measurements when the 4096 bits length key is utilized are higher, compared to the respective measurements of 2048 bits length key.

It is also worth mentioning that 2048 bits is the minimum key length for the certificate's key. Companies promote the usage of 4096-bit length keys for additional security, especially when the service manages sensitive information.

5.7 Experiment 5 – Elliptic Curve Encryption Algorithm Evaluation

This experiment is identical to the previous one, with the only difference being that the keys were generated utilizing the elliptic curve encryption algorithm. The two different keys have lengths 256 and 384 bits. A 256 bits ECC key provides the same security with a 3072 bits RSA key, while a 384 bits key is equivalent to a 7680 bits RSA key.

The diagram below compares the total time needed for the web application to execute all the requests at each concurrency, when the two ECC keys with different lengths were utilized.

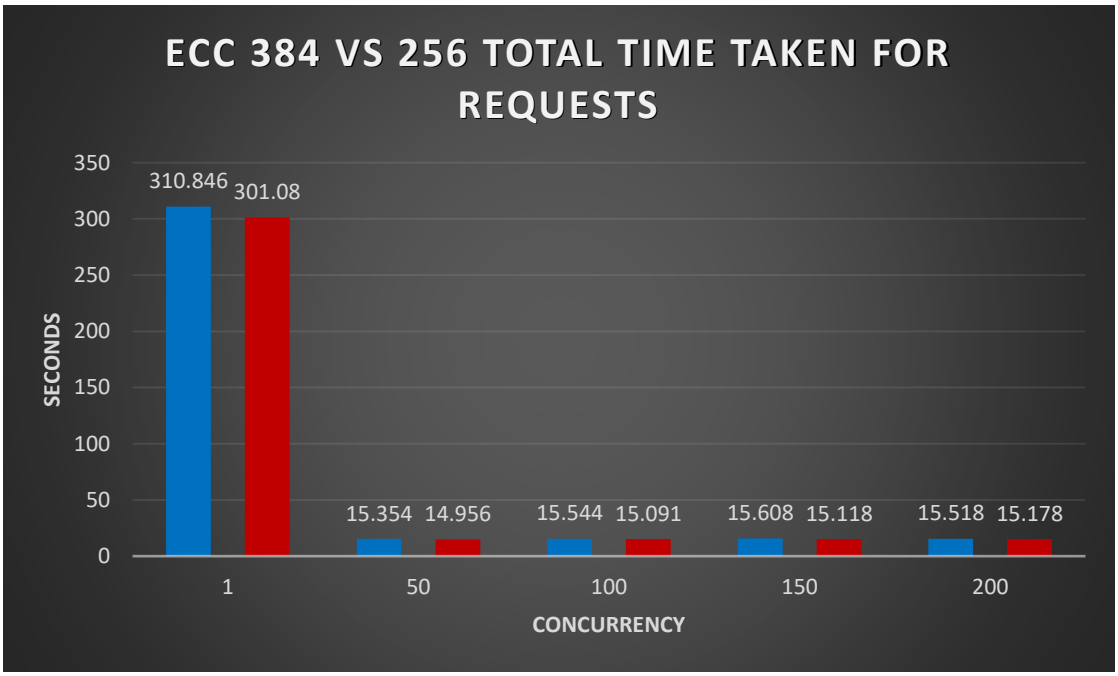


Figure 26: Total time taken for requests comparison between the ECC keys

The next diagram compares the average time that the web application needed to execute a batch of requests at each concurrency when the two keys with different lengths were utilized.

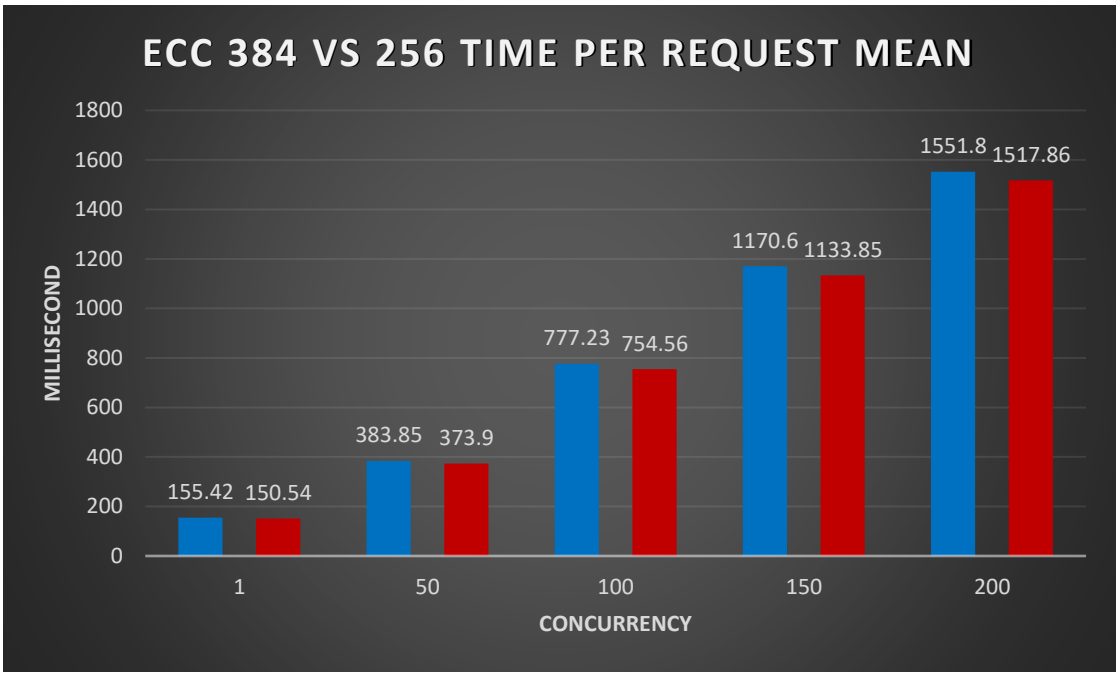


Figure 27: Time per request mean comparison between the ECC keys

5.8 RSA and Elliptic Curve Encryption Algorithms Comparison

The measured values for each utilized key, at each concurrency are compared and the results comparing performance are presented in the diagrams below.

The diagram below demonstrates the total time needed for the web application to execute all the requests at each concurrency for all the utilized keys.

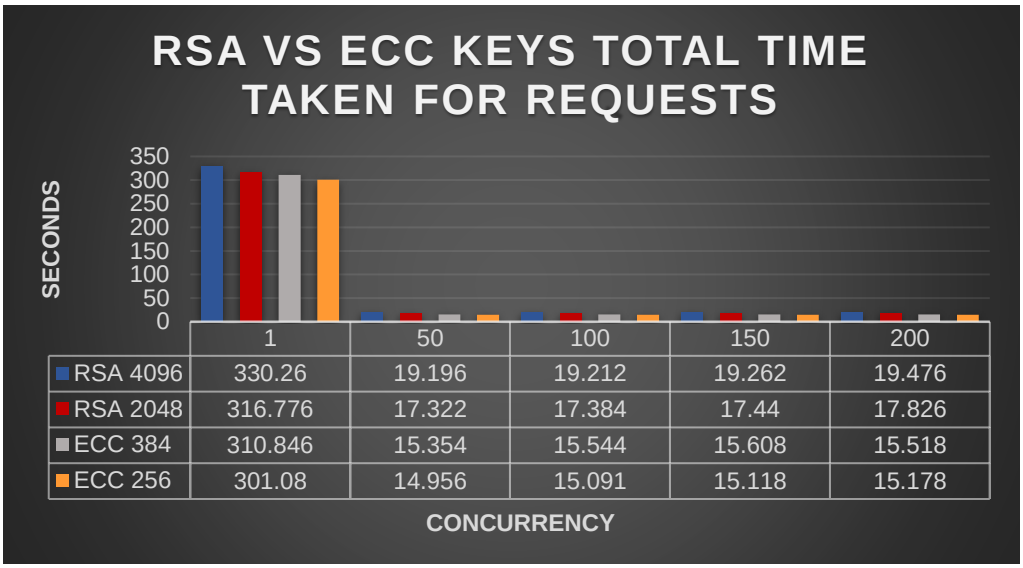


Figure 28: Total Time Taken for requests comparison between all keys

The following diagram demonstrates the average time that the web application needed to execute a batch of requests at each concurrency for all the utilized keys.

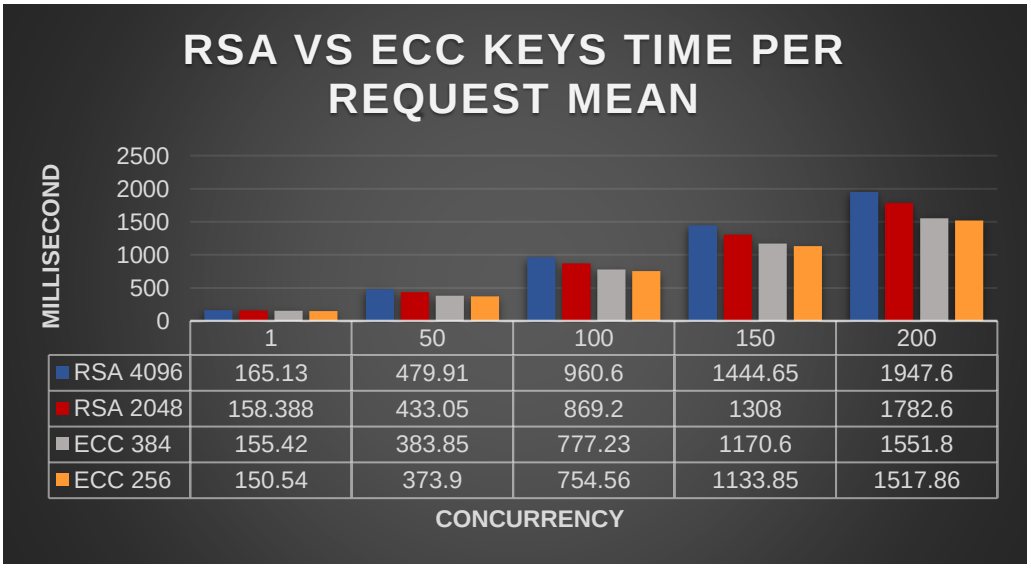


Figure 29: Time per request mean comparison between all keys

The comparison shows that the keys generated with the EC encryption algorithm outperforms the keys generated with the RSA encryption algorithm. Characteristically, even when the 384 bits length ECC key is utilized the system's performance is greater compared to when 2048 bits length RSA key is used.

Finally, the best system performance is observed when the 256-bit ECC key is utilized.

5.9 Experiment 6 - Register Sensor to a Remote Node while SSO is utilized

The last experiment studies the response time of the system when an infrastructure owner, which has utilized the SSO functionality to login to a remote node, registers a sensor to the node.

In this scenario, the user is registered in the Athens node. Utilizing the SSO functionality they have successfully logged into the Chania's node web application. The web application has successfully authenticated the user and it has bound an access token to them, as well as an id token and the guest infrastructure owner role. The user is redirected to the graphical interface of the *Register Service* in order to register their sensor. When the user sends the request, PEP extracts the user's role in the node, i.e., guest infrastructure owner, and forwards it alongside the requested resource to the PDP. The PDP assesses the request based on the stored rule set for the role and returns its decision to the PEP. If the permission is *Permit,* the PEP forwards the request to the register service. The latter extracts the sensor information from the request body and registers the sensor to the Cassandra directory database. Lastly, register service responds to the user with a *success message* or a *failure message* depending on the outcome of the user's request.

Details: The requests are sent from the physical machine located in Athens; hence the measurements include the network delay. The IP address of Chania's node is 147.27.60.43 and the PEP published port, that protects the register service, is 8066. The PEP's certificate contains a public key that was generated using the RSA encryption algorithm and its length is 2048 bits.

REST: The user sends a POST request to https://147.27.60.43:8066/RegisterSensor. The body contains the email of the user, the sensor's name and the type of the measurement the sensor receives, e.g., temperature.

The measured times are presented below.

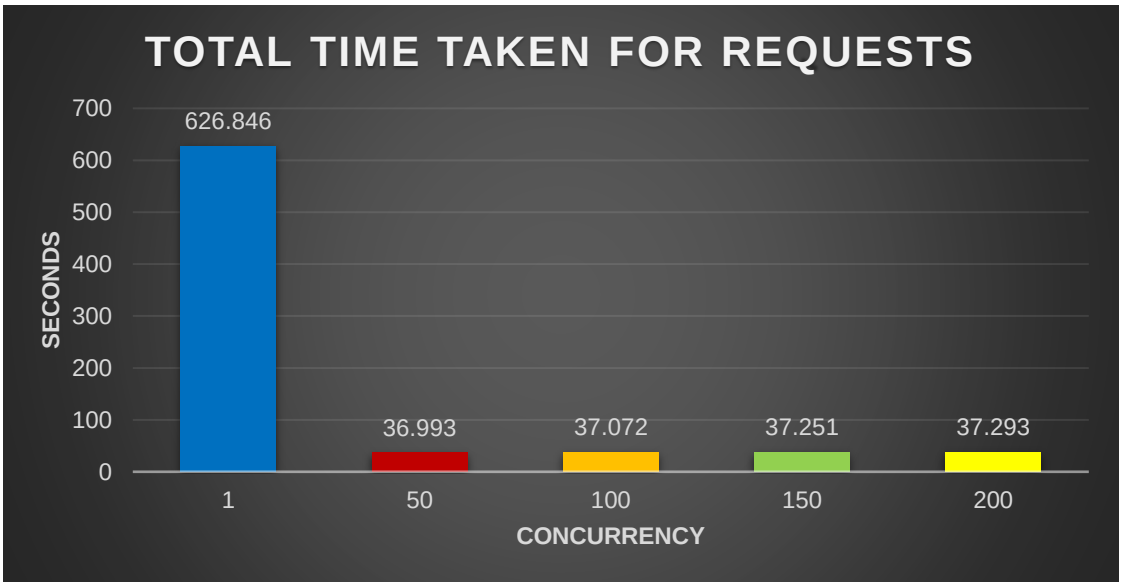


Figure 30: Total Time Taken for requests to register a sensor to a remote node

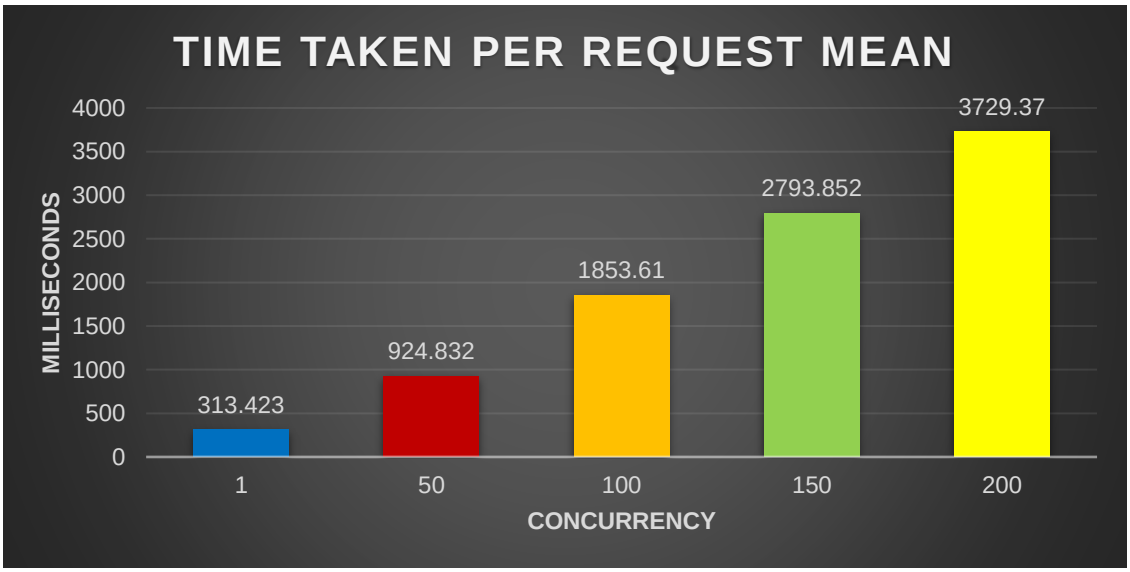


Figure 31: Time per request mean to register a sensor to a remote node

6. Conclusions

In this thesis, different security authentication and authorization mechanisms were implemented in a master-less federated cloud node architecture, in order to improve the system of each node in these areas.

Specifically, the HTTPS protocol was incorporated in crucial components of the system, for which communication occurs over public networks. Usage of the HTTPS protocol reinforced system security, as it encodes the communication utilizing encryption algorithms. To this end, two underlying encryption algorithms, i.e., RSA and ECC with different key lengths, were utilized in order to evaluate their effect in system performance. The experimental measurements demonstrated that enabling HTTPS introduced significant delay in the system, due to the time penalty introduced by the encryption process that occurs between two services before the communication begins. Moreover, quantitative evaluation revealed that amongst tested keys, the one generated by the ECC encryption algorithm with 256-bit length provides the best system performance.

The authentication code grant type of the OAuth2.0 protocol was incorporated in the system in order to reduce the chance of exposing user credentials. In addition, the OpenID Connect protocol, which extends the OAuth2.0 protocol, was enabled. OpenID Connect improves user authentication and authorization in the system by providing scoped ID tokens. In addition, it facilitated the creation of a token signature validation mechanism which guarantees the authenticity of the tokens and the integrity of the claims that they contain.

Finally, Single Sign-On (SSO) functionality, amongst federated nodes, was implemented. SSO was based on the OpenID Connect protocol and it allows registered users to login and use services of remote cloud nodes, on which they are not registered, via the IDM account of their node. In order to authenticate and authorize the user, remote cloud nodes request ID tokens from the user’s

registered node. These ID tokens contain all necessary user information required for system functionality. Furthermore, administrators of remote cloud nodes can control to which resources the guest users have access to, via rule sets.

7. Future Work

This thesis succeeded in reinforcing overall system security, as well as user authentication and authorization, by utilizing modern techniques and solutions. Additionally, it provided an implementation of Single Sign-On functionality, among federation cloud nodes. However, there are still areas for improvement and further research, which could not be covered in this study, due to time constrains or current technological limitations. Proposed future directions are outlined below.

Upgrading HTTPS/1.1 protocol to HTTPS/2.0 protocol – HTTPS/2.0 utilizes multiplexing of the requests and the responses. Due to this, it can greatly improve system performance, while guaranteeing security of the communication.

Securing the communication between the system and the IoT sensors – IoT sensors are characterized by their low energy demand and their relative low processing power. Due to these characteristics, incorporation of the HTTPS protocol in order to secure the communication between the sensors and the system is a demanding challenge, due to the associated computational complexity of encryption.

Modification of the system to replace Cassandra database – Cassandra is a high-performance distributed database and constitutes a core component of the system, as a Cassandra node is installed in every cloud node, enabling the exchange of information between them. Nonetheless, Cassandra poses several system security risks either due to service errors or due to malicious edge owners. A potential flaw can compromise the information stored in all nodes, while restoring information is time consuming and has large cost implications for the edge owners. Replacement of Cassandra with a no-distributed, no-SQL database requires modifications in the implementation of system services. Formation of the federation can be based on the OpenID Connect protocol and the Single Sign-On functionality that was implemented in this thesis.

References

[1] N. Zacharia, K. Tsakos, and E. G. M. Petrakis, “iZen: Secure Federated Service Oriented Architecture for the Internet of Things in the Cloud BT - Advanced Information Networking and Applications,” 2020, pp. 1189–1200.
[2] “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.” https://tools.ietf.org/html/rfc7230.

[3] “What is TCP/IP?” https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/.

[4] “Uniform Resource Identifier (URI): Generic Syntax.” https://tools.ietf.org/html/rfc3986.

[5] “HTTP Over TLS.” https://tools.ietf.org/html/rfc2818 (HTTPS).

[6] “TLS Security 5: Establishing a TLS Connection.” https://www.acunetix.com/blog/articles/establishing-tls-ssl-connection-part-5/.

[7] “The Transport Layer Security (TLS) Protocol Version 1.2.” https://tools.ietf.org/html/rfc5246.

[8] “Cipher Suites in TLS/SSL (Schannel SSP).” https://docs.microsoft.com/en-us/windows/win32/secauthn/cipher-suites-in-schannel.

[9] “Oauth 2.0.” https://oauth.net/2/.

[10]“OpenID Connect Protocol.” https://auth0.com/docs/protocols/openid-connect-protocol.

[11]M. Papazoglou and W.-J. Heuvel, “Service oriented architectures: Approaches, technologies and research issues,” *VLDB J.*, vol. 16, pp. 389–415, Jul. 2007, doi: 10.1007/s00778-007-0044-3.

[12]M. Khanbabaei and M. Asadi, “Principles of service-oriented architecture and web services application in order to implement service-oriented architecture in software engineering,” vol. 5, pp. 2046–2051, Nov. 2011.

[13]“What is FIWARE?” https://www.fiware.org/about-us/.

[14]“FIWARE-ORION.” https://fiware-orion.readthedocs.io/en/master/.

[15]“FIWARE-NGSI v2 Specification.” https://fiware.github.io/specifications/ngsiv2/stable/.

[16]“IDENTITY MANAGER - KEYROCK.” https://fiware-idm.readthedocs.io/en/latest/.

[17]“PEP PROXY WILMA,” [Online]. Available: https://fiware-pep-proxy.readthedocs.io/en/latest/.

[18]“WELCOME TO AUTHZFORCE’S OFFICIAL DOCUMENTATION.” https://authzforce-ce-fiware.readthedocs.io/en/latest/.

[19]“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.

[20]“WELCOME TO THE FIWARE SHORT TIME HISTORIC (STH) - COMET DOCUMENTATION.” https://fiware-sth-comet.readthedocs.io/en/latest/.

[21]“CYGNUS.” https://fiware-cygnus.readthedocs.io/en/latest/.

[22]“What is Cassandra?,” [Online]. Available: https://www.datastax.com/cassandra.

[23]“What is Docker?” https://opensource.com/resources/what-docker.

[24]“Docker Images.” https://docs.docker.com/engine/reference/commandline/images/.

[25]“Dockerfile reference.” https://docs.docker.com/engine/reference/builder/.

[26]“What is a Container?” https://www.docker.com/resources/what-container.

[27]“Use volumes.” https://docs.docker.com/storage/volumes/.

[28]“Overview of Docker Compose.” https://docs.docker.com/compose/.

[29]“DataStax.” https://www.datastax.com/.

[30]“OpenSSL.” https://www.openssl.org/.

[31]D. Cherry and T. Larock, “2 - Database Encryption,” D. Cherry and T. B. T.-S. S. Q. L. S. Larock, Eds. Boston: Syngress, 2011, pp. 27–71.

[32]E. Conrad, S. Misenar, and J. Feldman, “Chapter 3 - Domain 3: Security engineering,” E. Conrad, S. Misenar, and J. B. T.-E. H. C. (Third E. Feldman, Eds. Syngress, 2017, pp. 47–93.

[33]“PKCS #1: RSA Cryptography Specifications Version 2.2.” https://tools.ietf.org/html/rfc8017.

[34]“Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier.” https://tools.ietf.org/html/rfc8422.

[35]“keytool - Key and Certificate Management Tool.” https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html.

[36]“OAuth 2.0 Security Best Current Practice draft-ietf-oauth-security-topics-12.” https://tools.ietf.org/id/draft-ietf-oauth-security-topics-12.html.

[37]“ab - Apache HTTP server benchmarking tool.” https://httpd.apache.org/docs/2.4/programs/ab.html.