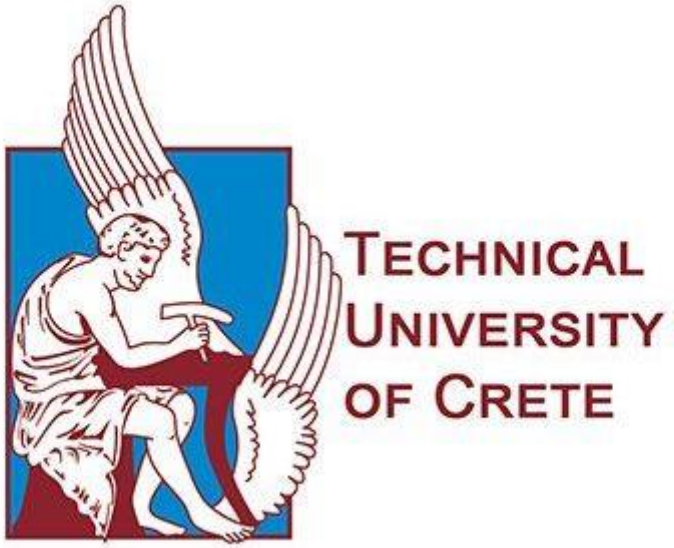
TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGEENIRING



Multi Edge Cloud (MEC) Architecture for supporting Internet of Things (IoT) Applications

Neofytos Zacharia

Committee:
Professor Euripides G.M. Petrakis (Supervisor)
Associate Professor Samoladas Vasilis
Associate Professor Antonios Deligiannakis

Chania, October 2019

Abstract

LINCA, is a distributed master-less IoT system. It is designed and deployed to offer security and high availability services to any IoT organization that wishes to register and provide their services and devices to a wider range of users. Each registered organization in LINCA is realized as a Service Oriented (SOA) Architecture in the cloud as composition of microservices that run in the cloud and can interact with other cloud systems, also registered to the LINCA network. In turn, IoT devices can connect to any LINCA node and a node need no be aware of devices connected to any other node. All organizations in LINCA ecosystem adopt JSON for the description of their devices and provide a search mechanism to search for devices that are connected to any registered cloud and meet the criteria set by their users, such as device ID, device location, device type etc. LINCA follows a 3-tier architecture model where each tier serves functionality for different types of users. The main types of users are System Administrators, Infrastructure Owners and Customers. System Administrators have the right to control and modify their individual cloud system and the users connected to their system. Infrastructure Owners have the permission to install and connect devices of different or similar types in their cloud system. Customers have the right to create subscriptions to devices that are connected to their cloud systems or to devices that are connected to other registered clouds in LINCA, for example to retrieve device measurements. This 3-tier architecture is expandable, by allowing more IoT organizations to connect. LINCA is evaluated using a large number of real and synthetic devices producing massive amounts of data (i.e. sensor measurements). Experimental results show that the system can respond under heavy workloads in real time.

Contents

1. Introduction

1.1. Motivation

Nowadays, there are numerous IoT organizations taking advantage of IoT devices, which can be used to monitor extreme weather conditions or to control the operations of a house, hospital or even an entire infrastructure such as a city, factory etc. However, various issues may arise regarding the way IoT organizations manage their devices when operating independently or in the context of a unified system. In detail, organizations can install and utilize their own IoT devices within certain borders, thereby limiting the use of services offered only to clients registered to the same infrastructure. Furthermore, data collected by each organization's IoT devices are stored locally so that the users of another cloud cannot have a unified view of similar data. For accessing of data in different infrastructures users must register again to each independent infrastructure. This increases the risk of data loss, not only in case of internal system failure within the organization, but also in case of heavy traffic where the centralized system fails to respond to every request.

The development of a distributed system is necessary, where different IoT organizations may connect and interact with each other. Registration is mandatory for an organization to join such a system, as the organization will be able to offer its users even more IoT devices, belonging to the other registered organizations. Another important aspect of this unified system is that registered organizations will be deemed equal and offered the same privileges of high data availability, thus ensuring that their collected data will never be lost. As for security issues, there will be no single master in charge of security, so each registered organization will be responsible for the security of their own devices and services.

The present study intends to build an integrated distributed system with the aforementioned features, relying on and extending iXen, an existing implementation of a cloud-based architecture, in order to facilitate communication and collaboration between individual IoT systems, and satisfy the requirements of a wide range of users.

1.2. Solution

One approach to solve the aforementioned issues involves the use of the LINCA System, which is presented through this thesis. LINCA is an implementation of a distributed IoT system in the cloud, which allows inter-connection and interaction of individual registered organizations thus forming an eco-system which provide users connected to any infrastructure with a unified view of data and services provided by multiple organization given that the user has the necessary authorization (i.e. users must be authorized to access data in a foreign infrastructure).. LINCA is responsible for the data management of registered systems, making them searchable by multiple users among different organizations. Also, LINCA manages the collected data of each

registered organization in a way that ensures high availability. On the other hand, LINCA assigns the responsibility of individual system security to each organization. LINCA functions are available through web-based graphical interfaces. LINCA can manage numerous IoT organizations, as well as the available types of their connected sensors, which are scattered geographically, serving multiple demands of various users.

Data from sensors' measurements is sent to the corresponding organization. Users' interest in these measurements is expressed through queries to the organizations managing the data. Therefore, data will be accessible by users, as long as they are subscribed to the respective sensors and they are granted the necessary authorization by the administration of each specific infrastructure. Infrastructure owners can install devices to their organization, allowing users of either their own organization or of other registered organizations to subscribe, subject to payment. In other words, organizations' sensors registered in LINCA will attract the interest of customers, who will need to pay subscription fees to be able to monitor the measurements of those sensors.

1.3. Contribution

The work builds -upon previous work for iXen [1] system is leveraging principles of Service Oriented Architecture and modern standards of context information management. iXen is an experimental cloud-based configuration of Restful micro-services. Its intention is to tackle the limitations of existing IoT architectures. Also, one of iXen's main priorities is the protection of its services from unauthorized services or user with the OAuth2 mechanism. This research extends iXen and implement a distributed system that connects multiple IoT architectures in a way that offers high availability services, security and an integrated search mechanism.

Each organization in LINCA is represented by a cloud system that is developed in the FIWARE environment. The main characteristics of the approach implemented within the scope of the present thesis are summarized below:

- The design specifications of each registered system are implemented using well-known open source web technologies such as PHP, HTML, JavaScript, as well as cloud-based services provided by the public platform FIWARE, which runs on OpenStack.
- Each registered cloud system in LINCA is developed based on the principles of Service Oriented Architecture (SOA). Each cloud system function acts as a stand-alone service that communicates with others through RESTful interfaces. Service Oriented Architecture simplifies and facilitates the extension of each cloud system, as each separate service can be upgraded or replaced without affecting the operation of the entire architecture. Similarly, new service functionality can be easily added.

- The architecture of a registered cloud system is based on the “secure by design” approach, ensuring that system services are protected in the cloud. This way the services’ REST interfaces are only accessible by the system itself and authorized users.
- The individual systems of LINCA can support sensors that transmit data through different low-level protocols, such as Bluetooth, Zigbee and WiFi, with the assistance of gateways, which transmit data to the cloud using a high-level IP protocol such as http, https or UDP, MQTT and Coap. The connection of each registered cloud system with its sensor interface is achieved through FIWARE's IDAS Back-End Device Management Service.
- In each of LINCA’s cloud systems, data is available in JSON format making data processing independent of device type and communication protocol.
- The project includes a complete design of the system in UML, detailing its specifications.
- Each cloud system in LINCA can easily integrate a business model by assigning its registered users, as well as users registered to other cloud systems, with appropriate roles and specific permissions.
- A key tool used for the implementation of LINCA's architecture is Apache Cassandra Database, which is responsible for data management and enables search by users connected to LINCA’s clouds. Furthermore, Cassandra provides security and high data availability to LINCA’s registered organizations.

To be able to support the previously mentioned features, LINCA relies on a business model where the three main user categories of the infrastructure can each gain value while contributing to the expansion and maximum utilization of the system. The three-layer architecture model adopted by LINCA is depicted in Figure 1, as described in the rest of this section.

- Layer 1 refers to the IoT infrastructure level, where Infrastructure Owners can select the cloud systems on which they will install their IoT devices. The devices of each Infrastructure Owner are connected directly or through gateways to each LINCA cloud system. Infrastructure Owners have the right to register sensors in one of LINCA’s registered cloud systems. The goal of Infrastructure Owners is to attract as many customers as possible from various LINCA cloud systems to whom they will sell data management services.

- Layer 2 is essentially the core of the distributed system, comprising of numerous IoT cloud systems designed according to the architecture of iXen. Each LINCA cloud representing an IoT cloud system is controlled by its own System Administrator, who is in charge of adding and managing the registered users of the particular cloud system along with their assigned roles and permissions. Furthermore, the administrator of an individual LINCA cloud system also monitors the operation and functionality of the system, while also being responsible of assigning permissions to users registered to others LINCA cloud systems who want to gain access to the devices or services of that particular cloud.

Finally, System Administrators have the responsibility of registering their systems to LINCA in order to make them discoverable from users that wish to subscribe.

- Layer 3 includes the Customers of each cloud system within LINCA. They have the rights to subscribe to one or more of the existing cloud systems and register to their corresponding sensor devices so as to collect data regarding the measurements of those IoT systems. Moreover, they can use the system of the cloud they have subscribed to in order to request access to the sensors of other LINCA cloud systems.

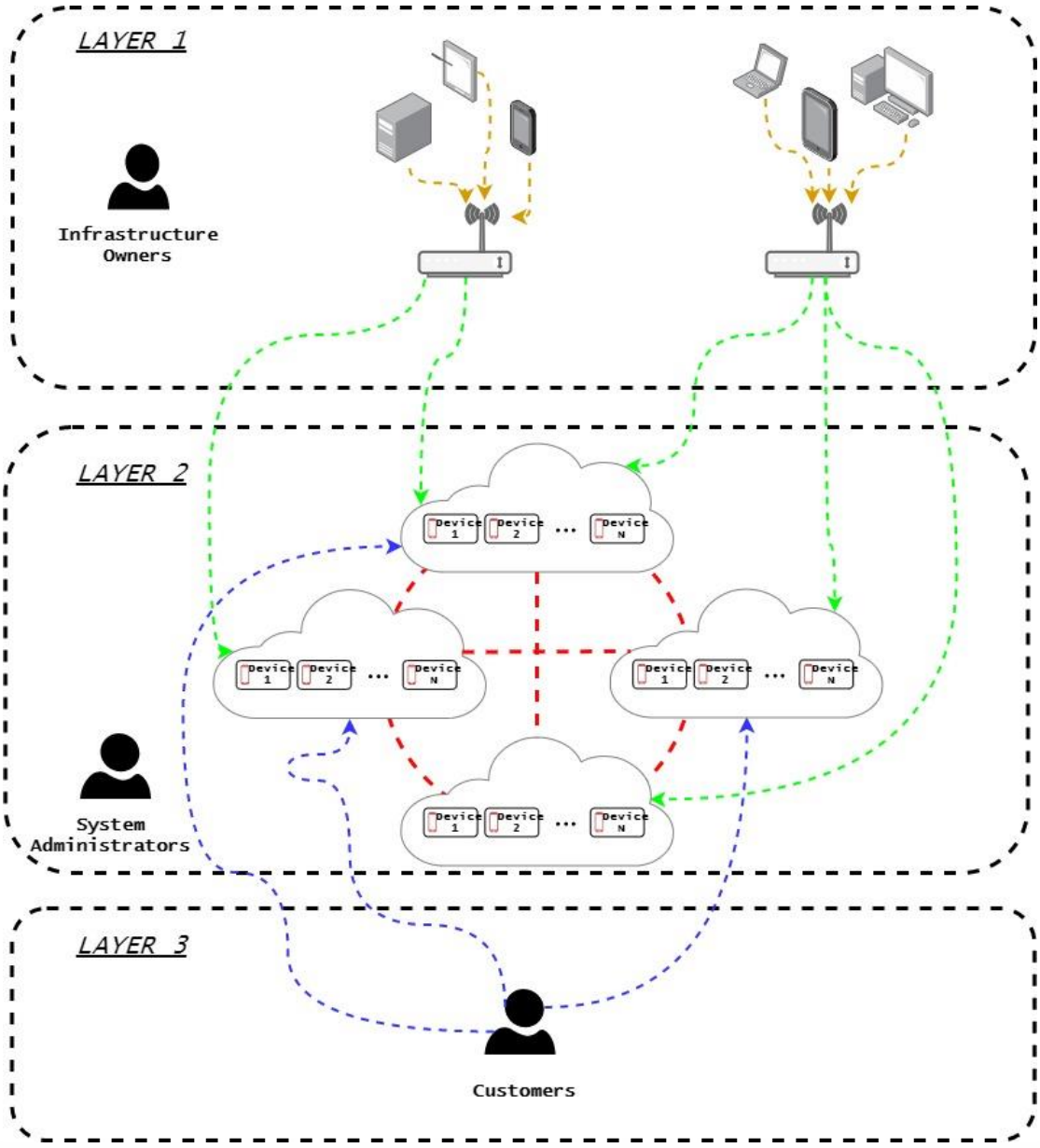


Figure 1 – Three-layer architecture of LINCA

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 enumerates the requirements for the presented system design and outlines functional and non-functional specifications through UML diagrams, while also providing the architecture diagram of a LINCA cloud system.

Chapter 4 describes the services of a LINCA cloud system and explains how different LINCA cloud systems interact with each other according to the requirements of Chapter 3.

Chapter 5 analyzes the performance of the system using real sensor data in order to examine system performance in cases of high data volume and computational load.

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

2. Background

2.1 Service Oriented Architecture

A service-oriented architecture (SOA) [2] is a computer software design enabling the provision of services among application modules over a network. Services built according to the SOA architecture can exchange data without human interaction or code alterations.

A significant feature of a service in SOA is its independence of the technical characteristics of other interacting services. This transparent exchange is accomplished through the implementation of a firmly defined interface, which goes through the required actions to allow inter-service data streams.

Service-oriented architecture (SOA) is based on a number of principles briefly discussed below:

- **Loose Coupling**, meaning minimum interdependency among services, so as to ensure seamless operation even in case of service functionality modifications.

- **Service Abstraction**, which refers to services' ability to conceal the logic behind their functionality from other services or applications. In other words, a service only provides the necessary details about what it does and not the way it does it.

- **Service Reusability**, which demands that logic or functionality is broken down into separate services for maximum reuse. A code written for a particular service should be able to work with multiple application types, without having to rebuild it for each individual application implementation.

- **Service Autonomy**, implying that services have complete knowledge and control over the functionality they implement.

- **Service Composability** refers to the "divide and conquer" approach applied by services, which tend to tackle problems by breaking them down into smaller, more manageable tasks, each implementing an individual business functionality.

- **Service Interoperability**, meaning that they apply common standards enabling different subscribers to use them.

2.1.1 RESTful Web Services

Web services based on REST[1] Architecture are known as RESTful web services, which rely on HTTP protocols to enable communication between client and server applications. The REST architecture handles each content as a resource. Furthermore, a RESTful web service is usually defined by a Uniform Resource Identifier (URI) and performs resource representation using HTTP[2] Methods and JSON format.

The main HTTP methods are GET, POST, PUT and DELETE, referring to the operations of reading, creating, updating and deleting respectively. A RESTful architecture provides a common data model for these four operations, which defines the input to the POST and PUT methods, as well as the output for the GET method, while the HTTP status code indicates operation success or failure.

A RESTful architecture also involves self-descriptive messages, with resources being independent from their representation to allow access to their content in diverse formats, like JSON[3], XML[4] and others.

Finally, it performs stateful interactions via hyperlinks. As all interactions with resources are stateless, each HTTP request includes all required information regarding its execution to ensure that previous communication states do not have to be stored.

2.2 FIWARE Platform for Application Development in Cloud

FIWARE[5] is an open-source middleware platform based on OpenStack[6], supporting cloud-based development and distribution of service-oriented applications. This well-structured platform allows both intra-platform and inter-platform service assembly. The FIWARE platform provides simple but robust APIs facilitating application development, while their specifications are public and free of charge.

2.2.1 FIWARE Services

FIWARE Generic Enablers (GE) provide simple general-purpose platform functions available through REST APIs, which can be used as modules of more complex applications.

The following services were used in the context of this thesis:

- **Identity Management (IdM) – Keyrock**

Identity management is a security and business principle allowing specific individuals to access particular resources under properly defined conditions, regarding the time and reason of access.

[1] https://restfulapi.net/rest-architectural-constraints/

[2] https://www.w3schools.com/whatis/whatis_http.asp

[3] https://www.json.org/

[4] https://www.w3schools.com/whatis/whatis_xml.asp

[5] https://www.fiware.org/about-us/

[6] https://www.openstack.org/

The FIWARE Keyrock[7] Generic Enabler provides an out-of-the-box configuration of the common characteristics of an Identity Management System, enabling other modules to use standard authentication mechanisms in order to accept or reject requests based on industry standard protocols. These characteristics include user access to networks, services and applications, secure and private authentication from users to devices networks and services, authorization, trust and user profile management and privacy-guaranteeing access to personal data.
The Identity Manager is the fundamental module connecting IdM systems at connectivity-level and application level and authorizing third-party services to access personal data stored in a secure environment.

- **PEP Proxy – Wilma**

A PEP Proxy is an endpoint placed in front of a secured resource at a common public location, serving as a protector controlling access to resources. Users or other actors have to provide adequate information to the PEP Proxy for their request to pass through the PEP proxy and reach the actual location of the secured resource, which is unknown to the outside user and could be found in a private network behind the PEP proxy or on an entirely different machine.

FIWARE Wilma[8] is a simple PEP proxy built to work with the FIWARE Keyrock Generic Enabler. When a user attempts to obtain access to the resource behind the PEP proxy, the PEP will send the user's attributes to the Policy Decision Point (PDP), from which it will receive a security decision to enforce (Permit or Deny). Authorized users will barely notice any disruption of access, as the received response is identical to the one they would receive upon direct access to the secured service, whereas unauthorized users receive a 401 Unauthorized response.

- **Authorization PDP – AuthZForce**

For more complicated access control scenarios, an extra mediation microservice is necessary to assess each Permit/Deny policy decision by examining the data provided by the requesting service according to the full set of access control rules.

FIWARE AuthZForce[9] is an advanced access control Generic Enabler offering such an interpretive Policy Decision Point (PDP) according to the XACML standard and providing an API to get authorization decisions based on authorization policies and requests from PEPs.

Rulesets can be updated making security policy maintenance flexible and adaptive to business needs. Additionally, highly extensible language is used to describe the access policy and meet any access control scenario.

[7] http://fiware.github.io/specifications/ngsiv2/stable/

[8] https://catalogue-server.fiware.org/enablers/pep-proxy-wilma

[9] https://catalogue-server.fiware.org/enablers/authorization-pdp-authzforce

- **Publish/Subscribe Context Broker – Orion Context Broker**

The Orion Context Broker[10] is an implementation of the Publish/Subscribe Context Broker GE, providing an NGSI interface through which clients can query and update context information, receive notifications upon context information alterations and register context provider applications.

- **FIWARE Cygnus**

Cygnus[11] is a connector persisting context data originating from Orion Context Broker into other third-party databases and storage systems, like MySQL, MongoDB, DynamoDB and CKAN, to generate a historical view of the context. It accepts NGSI dataflows and stores them in its predefined database. Cygnus can store raw and aggregate data, independent of user database.

- **FIWARE Comet**

The FIWARE Comet[12] stores and retrieves historical raw and aggregated context data registered in an Orion Context Broker instance.

All communications between the Comet and the Orion Context Broker (or any other third party) use standardized NGSI interfaces.

2.2.2 Related Technologies to FIWARE

- **Authorization Protocol – OAuth2**

OAuth[13] is an open-standard authorization protocol offering secure designated access capability to applications, by disallowing the exchange of password data and demanding the use of authorization tokens, the so-called “OAuth2 tokens”, to verify an identity between service consumers and providers. Therefore, it allows end users to approve the interaction between applications on their behalf without having to disclose their credentials.

Additionally, the OAuth2 mechanism is specifically designed to work with HTTP protocol and allows the assignment of OAuth2 access tokens to third parties that have already been identified by an authorization service, such as Keyrock IdM.

- **MongoDB Databases**

MongoDB[14] is an open-source non-relational database management system (DBMS) using a document-oriented database model that supports various forms of data and is suitable for big data applications and other processing jobs involving data that do not fit well in a traditional relational

[10] https://catalogue-server.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker
[11] https://fiware-cygnus.readthedocs.io/en/latest/
[12] https://fiware-sth-comet.readthedocs.io/en/latest/
[13] https://oauth.net/2/
[14] https://www.mongodb.com/

model. Instead tables and rows, the MongoDB architecture comprises of collections and documents.

- **Information Model – NGSIv2**

FIWARE-NGSIv2[15] manages the whole lifecycle of context information, involving updates, queries, registrations, and subscriptions. The NGSIv2 API comprises of a simple information model based on context entities and a RESTful interface for context data exchange through queries, subscriptions and updates.

The key elements of a NGSI information model are:

Entity is any physical or logical object (sensor, user etc.). Each entity is characterized by an entity id and a type e.g. "Sensor".

Attributes are elements of entities and have a name, a type, and a value.

Name (of an attribute) describes the type of property that represents the value of the attribute of the entity.

Type (of an attribute) refers to the data type of the value of the attribute (e.g. Float, Int, String). An attribute can have from one-to-n metadata.

Metadata is a part of an attribute describing the property of the attribute value. Metadata variables, like name, type and value, follow the same rules followed by the corresponding attribute variables.

JSON objects are used for entity representation, applying the syntax rules set by the NGSI standard.

- **Extensible Access Control Markup Language (XACML)**

eXtensible Access Control Markup Language[16] (XACML) is a vendor-independent declarative access control policy language, a processing model and an architecture specifying how to assess access requests according to policy-defined rules and enabling common access control terminology and interoperability. XACML policies are split into a hierarchy of three levels, PolicySet, Policy and Rule.

The PolicySet is a collection of Policy elements containing one or more Rule elements. Each Rule within a Policy is evaluated as to whether it should grant access to a resource - the overall Policy result is defined by the overall result of all Rule elements processed in turn. Separate Policy results are then evaluated against each other using combining algorithms define which Policy wins in case of conflict. A Rule element consists of a Target and a Condition.

.

[15] https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html

[16] https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html

- **Internet of Things (IoT) – Cloud Computing**

Two pioneering technologies, Internet of Things (IoT) and cloud computing, have seen tremendous expansion in recent years [3] , due to the increasing use of smart devices and sensors in many fields, such as healthcare and assisted living, industrial systems and environmental monitoring. Smart devices and sensors operating as Internet-connected data collectors, together with large (mostly cloud) platforms where data is stored for permanent storage and analysis, provide a beneficial environment for modern businesses to broaden their client base.

The concept of marrying the Internet of Things with cloud computing has also provided new opportunities for real-time data accumulation and analysis. Being simple, scalable and affordable, cloud computing has become one of the most preferred platforms for IoT data storage, processing and analysis, with companies selecting to deploy their applications and systems on the cloud to minimize infrastructure, maintenance and operating costs.

2.3 IoT Platform

IoT platforms are the core of IoT architecture, connecting the real and virtual world and enabling communication between entities.

An IoT platform includes the following components:

- **Connectivity and Normalization**

The layer of connectivity incorporates various protocols and data formats into a single "software" interface, guaranteeing device interactivity and proper data reading. A common format and storage location for all data facilitates the management, analysis and monitoring of IoT devices.

- **Device Management**

The device management unit makes sure that the connected entities function properly and that installed software and applications are running correctly with updated versions. Actions performed in this layer include device disposition, remote configuration, management of software and firmware updates and troubleshooting. With thousands of devices composing an IoT-supported system, automation and batch tasks are required to minimize manual labor and related costs.

- **Database**

Data storage is another key feature of an IoT platform, while device data management has made database requirements more complex and demanding, in terms of:

Volume, as the amount of data to be stored can be enormous.

[17] http://cassandra.apache.org/

[18] https://database.guide/what-is-a-column-store-database/

Variety, with diverse devices and types of sensors employing different data formats.

Velocity, often making data flow analysis necessary for instant decision making.

Veracity, or accuracy, since sensors sometimes generate vague and imprecise data.

To meet these requirements, an IoT platform is often combined with a cloud-based database, distributed across numerous sensor nodes, with scalability for big data and capability of storing both structured and unstructured data (SQL and NoSQL respectively).

- **Processing and Action Management**

This IoT platform component involves accumulating data from the connectivity and normalization module and storing it in the database. Event-triggering rules are used in this stage to enable "smart" actions depending on the sensor data. An example of such rule in the case of a smart home could be: "If GPS-based indications show that the distance between a person's smartphone and their home is greater than 5 meters, then all home lights should be turned off."

- **Analytics**

IoT implementations often demand complex analytics to benefit from data streams registered in an IoT platform. For instance, in a smart home, analytics could assist in finding which combination of lights and heating is mostly preferred by the owner during the day and night hours depending on weather conditions.

- **Data Visualization**

Data visualization is vital since it enables pattern and trend identification. Line or pie charts and 2D or 3D models available in administrative toolkits are used to this end.

- **External Interfaces**

In business and corporate implementations, it is significant and beneficial to integrate IoT with existing management tools, ERP systems, and the IT ecosystem in general. Embedded application programming interfaces (APIs), software development kits (SDKs) and gateways are the fundamental mechanisms enabling integration of third-party systems and applications. Therefore, well-defined external interfaces are essential in minimizing related integration efforts and costs.

2.4 Distributed Systems

A distributed system can be simply defined as a group of computers operating together and appearing as a single entity to the end-user [4].

[17] http://cassandra.apache.org/

[18] https://database.guide/what-is-a-column-store-database/

The computers composing a distributed system can be either in physical proximity, interconnected through a local area network (LAN), or geographically sparse and connected via a wide area network (WAN). A distributed system can consist of diverse architecture components like mainframes, servers, workstations, personal computers, minicomputers and so on. Moreover, regardless of their types, these machines operate simultaneously and have a shared state, ensuring that upon failure of a single component, the entire system's uptime will not be significantly affected.

Distributed Systems (DS) have numerous advantages, include the following:

- Node interconnection facilitates data exchange and sharing between DS nodes.
- Scalability enables easy node insertion to the distributed system.
- Seamless operation implies that failure of a single node cannot cause failure of the entire distributed system, instead communication among all other nodes is maintained.
- Multiple sharing of resources across various nodes is possible.

2.5 Distributed Databases

A distributed database comprises of two or more files stored in different servers located either on the same network or on entirely different networks [5]. Database components are stored in multiple physical locations and processing is disseminated among multiple database nodes. Distributed Databases have the following characteristics:

- Databases are logically interrelated and usually compose a single logical database.
- Data is physically stored across multiple nodes. Data in each node is managed by a Database Management System (DBMS) independent of the other nodes.
- Node processors are connected through a network and do not dispose of multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database includes transaction processing, yet does not constitute a transaction processing system.

2.5.1 Apache Cassandra

Apache Cassandra[17] is a NoSQL, wide column store[18], peer-to-peer distributed database running on a server cluster, designed to manage large amounts of data and support high user traffic, i.e. thousands of concurrent users or operations per second. Unlike other master-slave databases, in Cassandra, all nodes in its cluster have an identical role and communicate with

[17] http://cassandra.apache.org/

[18] https://database.guide/what-is-a-column-store-database/

each other equally, while there is no single point of failure[19] guaranteeing increased fault tolerance. Consequently, in case of cluster node failure, other nodes take over to complete the task. An additional advantage of this database is the possibility to add (or remove) a server to (or from) the cluster at any time without requiring downtime. Moreover, high data write speed allows real-time processing of big data.

As mentioned before, Cassandra is a wide column store NoSQL database. This means that it uses tables, rows, and columns, where the names and format of each column can vary from row to row on the same table. The components of Cassandra's data model are keyspaces and column families, also known as tables and columns. A graphic representation of this data model is depicted in Figure 2.

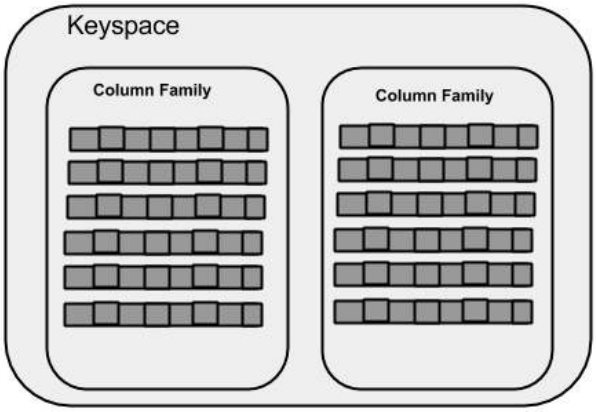


Figure 2 – a graphic representation of a Cassandra Data Model

A keyspace[20] is a container for a list of one or more tables, like a database in a relational database. These tables contained within a keyspace, are also known as column families and comprise of a collection of rows. In Cassandra, a row is the smallest unit of the table that stores data. It consists of a primary (or partition key), identifying a row in a column family (table), and a number of columns associated with it. In turn, A column is Cassandra's basic data structure with two values, namely key or column name and column value. Column key is similar to the concept of a column name in relational databases and uniquely identifies a column in a row, while a column value stores one value or a collection of values. A column family is similar to the concept of a column value in a relational database. Figure 3 shows a column family with its rows of data with the corresponding columns and partition keys.

[19] https://techblog.mdsol.com/2014/06/16/no_single_points_failure.html

[20] https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/cqlKeyspacesAbout.html

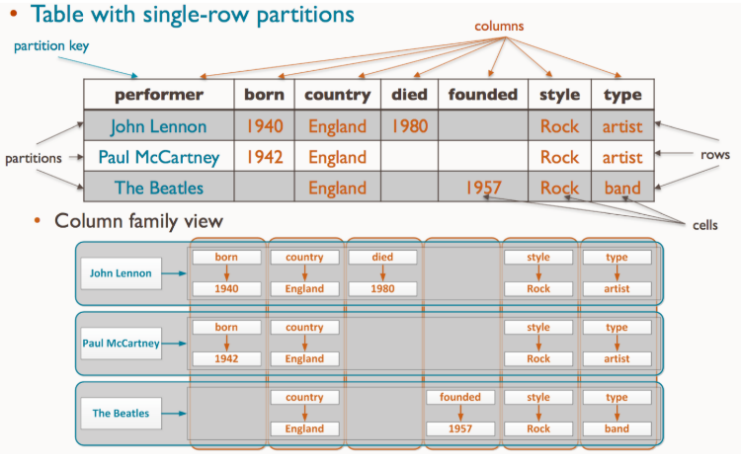


Figure 3 – Column family view

The basic infrastructure component of this database is the node server responsible for storing data. A collection of Cassandra server nodes constitutes a cluster of nodes, known as the Cassandra ring. Interaction between cluster nodes is based on a peer-to-peer communication protocol called Gossip Protocol[21] which propagates information about data and node health. Communication between two nodes involves the provision of information about each node's status, as well as about the latest status of any node with which it had previously communicated. This process allows for failure detection. On the other hand, during start up, a cluster node, the so-called "seed node", uses this protocol to facilitate all other nodes of the Cassandra ring in identifying each other.

As already mentioned, a table within a keyspace consists of various rows referenced by partition keys. The partitioner[22] is a hash function calculating the hash value of a particular partition key. This value is known as token. The hashing algorithm data mapping to physical cluster nodes, meaning that every range of values (token range) generated from the partition keys through the hashing algorithm[23] is assigned to the corresponding cluster node. Then, the created token will decide which node will receive the first replica of data that the token refers to, while the total number of replicas across the cluster depends on the replication factor[24]. If the replication factor is greater than one, then the placement of the subsequent replicas is determined by the replication strategy. There are two main replication strategies used by Cassandra, the Simple Strategy[25], placing subsequent replicas on the next node in a clockwise routine, and the Network Topology Strategy[26] ensuring that replicas are not stored on the same rack, i.e. a unit that contains multiple servers all stacked one on top of another. In addition, Cassandra uses snitches[27] to discover the overall network overall topology. A snitch determines which datacenters and racks nodes belong to. With this process, Cassandra stores data replicas on multiple ring nodes to guarantee reliability and fault tolerance. Figure 4 below shows the division of a 0 to 100 token range evenly amongst a four-node cluster. Node 1 is responsible for partition key hash values 0-24, Node 2 is responsible for

[21] [22] [23] https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture

[24] https://docs.datastax.com/en/archived/cql/3.3/cql/cql_using/useUpdateKeyspaceRF.html

[25] [26] https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html

[27] http://cassandra.apache.org/doc/latest/operating/snitch.html

partition key hash values 25-49, Node 3 is responsible for partition key hash values 50-74 and Node 4 is responsible for partition key hash values 75-99.

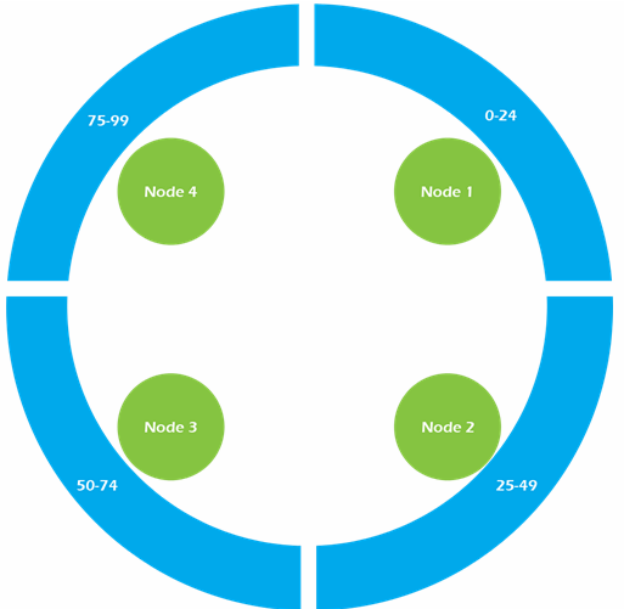


Figure 4 – Nodes of Cassandra rings with their corresponding token range

The client reads or writes requests that can be sent to any node in the cluster. Cassandra is a master-less database, so the client can connect to any cluster node at any given moment. When a client connects to a node with a request, the particular node serves as the controller for the specific client operation and acts as a proxy between the client and the nodes owning the requested data. The controller decides which nodes in the ring should get the request based on cluster configuration.

Depending on the partition key and replication strategy, the controller forwards and replicates data to the respective nodes, which process the request individually. During a node-level write operation, every node initially writes data into the commit log and then writes them into the memtable, which is a write back cache located in the memory. The commit log, located in the disk, is used for restoring the data in case of node failure resulting to data loss in the memtable. Whenever the memtable is at full capacity levels, the data it held is written to the disk's SSTable (Sorted String Table), which is an ordered immutable key value map, providing an efficient way of storing large sorted data segments in a file. Moreover, after data in the memtable are flushed to an SSTable, their corresponding data in the commit log are purged.

Likewise, during a node-level read operation, the client can choose to connect to any node of the cluster ring. The chosen node is called the controller and is responsible for returning the requested data. A partition key is necessary for every read operation and is used by the controller to locate the node where the first replica is located.

Consequently, for every read request, Cassandra reads data from all corresponding SSTables and scans the memtable for any data fragments, which are then merged and returned to the controller. Internally, the SSTables are using a Bloom Filter to check whether the requested partition key is stored in an SSTable. Cassandra uses Bloom Filters to check if any of the SSTables contains the requested partition key, without having to actually read their contents, hence evading expensive I/O operation. Once a read is completed from all contributing nodes, the controller compares the retrieved data. If the replica has an older version of the data, the controller returns the latest version to the client by issuing a read repair command with the older version of the data.

Also, a new node can join the Cassandra Cluster without affecting the function of the other cluster nodes. The auto-bootstrap function in Apache Cassandra is responsible to redistribute the data in Cassandra's cluster when a new node is joining the cluster. Initially, the node that will join the Cassandra cluster is defined as an empty node without data. When this new node starts the auto-bootstrap process, it must contact the cluster seed nodes in order to learn information about the other cluster nodes and the configurations they follow. After it contacts the seed nodes, it informs the Cassandra Cluster that is ready to join the cluster. Immediately, through the consistent hashing algorithm, the node calculates the portion of cluster's data for which will be responsible. In this way the cluster sends to the new node the corresponding portion of data .When the new node receives all the data for which it will be responsible, it informs the cluster that is a part of it and is ready for usage .

2.6 Docker

Docker[28] is a containerization platform that packages applications and all their dependencies together in the form of a docker container to guarantee interoperability, i.e. seamless operation in any environment.

As Figure 5 demonstrates, every application runs on separate containers and has its own dependencies and libraries, ensuring independence of the other applications and providing developers with the necessary security to build applications that will not interfere with one another.

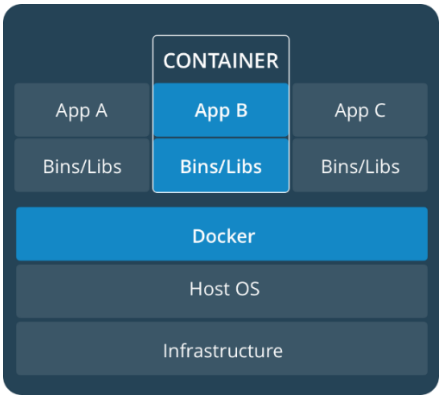


Figure 5 – Docker

[28] https://www.docker.com/

The basic features of Docker are listed below:

- **Dockerfile**[29] is a textual instruction document with all the commands that a user can call on the command line to assemble an image.
- **Docker Images**[30] are the building components of a Docker Container, stored in the Docker Registry, which is either a local user repository or a public repository, like a Docker Hub, permitting multiple users to collaborate in building an application.
- **Docker Container**[31] is a running instance of a Docker Image holding the entire package needed to run the application. As a standardized unit, it can be created on the fly for application or environment deployment.
- **Docker Machine**[32] is a tool enabling a provider to install Docker Engine (Docker's Software) on virtual hosts and manage them easily with docker-machine commands. It allows the creation of Docker hosts on various environments, such as a Windows box or local Mac, a business network, a data center, or even cloud providers like Microsoft Azure and Amazon AWS. Docker-machine commands can help start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to the provider's host.

[29] https://docs.docker.com/engine/reference/builder/

[30] https://docs.docker.com/engine/reference/commandline/images/

[31] https://www.docker.com/resources/what-container

[32] https://docs.docker.com/machine/overview/

3 LINCA System Requirements and Design

3.1 Use Case

LINCA System is consisted of three levels. Each of those levels serves a different functionality for different types of users. The basic types of users in the system are administrators, infrastructure owners and customers. The administrators have the rights to manage their cloud systems that are registered in LINCA and the users of them, while the infrastructure owners have the permissions to install and connect their own devices of the same or different type at the cloud system on which they are registered. The Customers that are connected to their registered cloud system, have the right to subscribe to sensors that are connected to their cloud system but also to sensors that are located to other remote registered cloud systems of LINCA where they are authorized, in order to receive updates of the sensor's measurements.

3.2 Functional and Non-Functional System Requirements

System requirements are defined by separating them into functional and non-functional.

3.2.1 Functional System Requirements

Functional requirements are defined as the processes that must be met by the system and they are directly related to its implementation. To consider the system as fully functional, all the requirements of each type of user must be met. Below, the requirements of each user group is presented separately.

- **Customers**

1. **Sign up** - The user in order to sign up to a cloud system, he/she fills in his/her details, such as name, email, password. In addition, he/she selects the customer option as his/her type of user. When he/she completes the necessary information, the administrator of this cloud system registers him/her to the cloud system as customers and assigns them their respective roles with their corresponding permissions.

2. **Login** - The user enters his/her login details such as email and password, on the login page in order to log in to his/her cloud system. Immediately, an authentication process is performed by the cloud system. If the user is authenticated successfully then he/she can access the cloud system.

3. **Search for available clouds / Subscribe to available clouds** - The graphical interface shows the cloud systems that are registered in the LINCA and that are available to receive subscriptions requests from users that belong to other clouds. When requesting subscription to

these cloud systems, the corresponding administrator decides whether to accept the user's subscription request. If that administrator decides to accept it, he/she will assign to the user who made the request the respective roles with their corresponding permissions, in order to have access to resources of his/her cloud system.

4. **Search for Sensors / Subscribe to Sensors** - The graphical interface shows to the user the available cloud systems that are registered in the LINCA. Then, the user selects the cloud systems he/she wants to know about their connected sensors, along with the corresponding sensors' measurements that the sensors wish to perform. When the user fills in the necessary information, an identification process is performed by the cloud system that user is connected and an authorization process is performed by the cloud systems that he/she wishes to know about their sensors. After the successful identification and authorization of user, this user is able to select the sensors that resulted from the search and create a new subscription to them. Furthermore, when the user selects the sensors that they wish to create a subscription of, the user's sensor subscription list in Cassandra database is updated with the corresponding sensor's information.

5. **View sensors' subscriptions** - Through a graphical interface, a user can view the sensors that they have a subscription to, along with the cloud systems to which they belong, the owner of them, and the date of subscription. This list is located on the distributed database of Cassandra. All of these will be done after the user is successfully authenticated and authorized by the cloud system in which he/she is connected.

6. **View Sensor Current Measurement** - A user, through the graphical interface, can monitor their sensors' current measurements that they subscribed to. This will be done after a user authorization and authentication check has been performed. The identification process is performed by the cloud system that the user is connected to, and an authorized process by the cloud he/she wishes to know about the sensors current measurements. After the successful identification and authorization of the user, they can monitor the current measurement of that sensor of that sensor.

7. **View Sensor Statistical Measurements** - A user, through the graphical interface, can monitor the average, minimum and maximum of the measurements recorded by the sensors that they have subscribed to. Again, this will be done after a user authentication and authorization check has been performed. The identification process is performed by the cloud system that is connected, and an authorization

process by the cloud he/she wishes to know about the sensors statistical measurements. After the successful identification and authorization of user, he/she can monitor the statistical measurement of the selected sensor.

- **Infrastructure Owners**

1. **Sign up** - The user in order to sign up to a cloud system, he/she fills in his/her details, such as name, email, password. In addition, he/she selects the Infrastructure Owner option as his/her type of user. When he/she completes the necessary information, the administrator of this cloud system registers him/her to the cloud system as customers and assigns them their respective roles with their corresponding permissions.

2. **Login** - The user enters his/her login details such as email and password, on the login page in order to log in to his/her cloud system. Immediately, an authentication process is performed by the cloud system. If the user is authenticated successfully then he/she can access the cloud system.

3. **Insertion of Sensor** - The infrastructure owner has the right to insert new sensors into the cloud system that he/she is connected. Specifically, through the graphical interface, the infrastructure owner selects the type of sensor that he/she will register and then defines the sensor's name, identity, and measurements. When the user fills in the necessary information, an identification and authorization process is performed by the cloud system that user is connected. After the successful identification and authorization process of user, he/she can connect his/her sensors to the cloud system with the help of IoT Agent Service. IoT Agent Service is responsible to receive data from the connected cloud's sensors and to forward them to Publish/Subscribe Service. The information of the cloud's sensors is also stored in the distributed Cassandra database, in order to be discoverable by authorized users who belong to other cloud systems of LINCA. The physical device sends its data to the cloud system's gateway that it is connected. The gateway forwards the data it receives to the Sensor Interface Service (IoT Agent) of this cloud system. There, the service analyzes the data and detects which sensor it is referring to. Then Sensor Interface Service updates the corresponding sensor entity in the Publish/Subscribe Service with the current measurements it received.

4. **Edit registered sensors** - The infrastructure owner can edit a connected sensor by updating its entity in Publish/Subscribe

Service. Also, he/she must update the sensor's information that is stored Cassandra database.

5. **Deletion of registered sensors** – The infrastructure owner can delete a connected sensor by deleting the corresponding sensor entity in Publish/Subscribe Service. Also, he/she must delete the corresponding sensor information in Cassandra database.

6. **View registered sensors** - Through a graphical interface, an Infrastructure Owner can view the sensors that registered on his/her cloud system, along with the date of registration. This will be done after the user is authenticated and authorized by the his/her cloud.

- **Administrators**

1. **Login** - The user enters his/her login details such as email and password, on the login page in order to log in to his/her cloud system. Immediately, an authentication process is performed by the cloud system. If the user is authenticated successfully then he/she can access the cloud system.

2. **Insertion of Cloud System**- The administrator has the permission to insert his/her cloud system to LINCA. Specifically, through the graphical interface, the administrator types the information of his/her cloud system, such as name, IP-address, owner and location. When the administrator fills in the necessary information, an identification and authorization process is performed by his/her cloud system. After the successful identification and authorization process of user the cloud's information is stored in the distributed Cassandra database, in order to be discoverable by authorized users who belong to other cloud systems of LINCA.

3. **Creation of cloud users** - The administrator can create a new user profile within his/her cloud system User Identification and Authorization Service and classify them into one of the available user categories, such as customers and infrastructure owners.

4. **Edit cloud users** – If it is necessary, the administrator may edit user's profile information that are stored in the User Identification and Authorization Service of his/her cloud system.

5. **Deletion of cloud users** - If it is necessary, the administrator can delete a user from his/her cloud system by deleting the corresponding user's profile from the User Identification and Authorization Service of his/her cloud system. By deleting a user should simultaneously delete the information entities associated with him/her in the cloud system. In case a customer user is deleted,

his/her subscription to the sensors must be deleted too. In case an infrastructure owner is deleted, the information related to his/her registered sensors to the cloud system should be deleted at the same time.

6. **System Monitoring** - The administrator has the ability, through a graphical interface, to monitor at any time which users and sensors are on his/her cloud system. Also, the administrator can monitor at the same time the workload of virtual machines run by the cloud system services.

3.2.2 Non-functional System Requirements

The fulfillment of these requirements is not necessary for an application to perform its essential functionality. However, their degree of fulfillment also affects the quality of the finished product, especially if it is a commercial application. These requirements include:

- **Performance** - Refers to the response speed of the system under high workload conditions.

- **Security** - It concerns the security of users, such as their secure access to the system and the protection of their identity and personal data. At the same time, it also concerns the protection of the united system, by preventing access to services and data through the network from unauthorized sources (services or users). The infrastructure of the system is developed in order to ensure by the level of architecture, how all requests between its services are properly authorized, excluding unauthorized users and services from accessing system resources online.

- **Usability** - Specifies how easy the system is to use. This category includes features such as graphical interfaces and everything else designed to improve the application experience.

3.3 Class Diagram

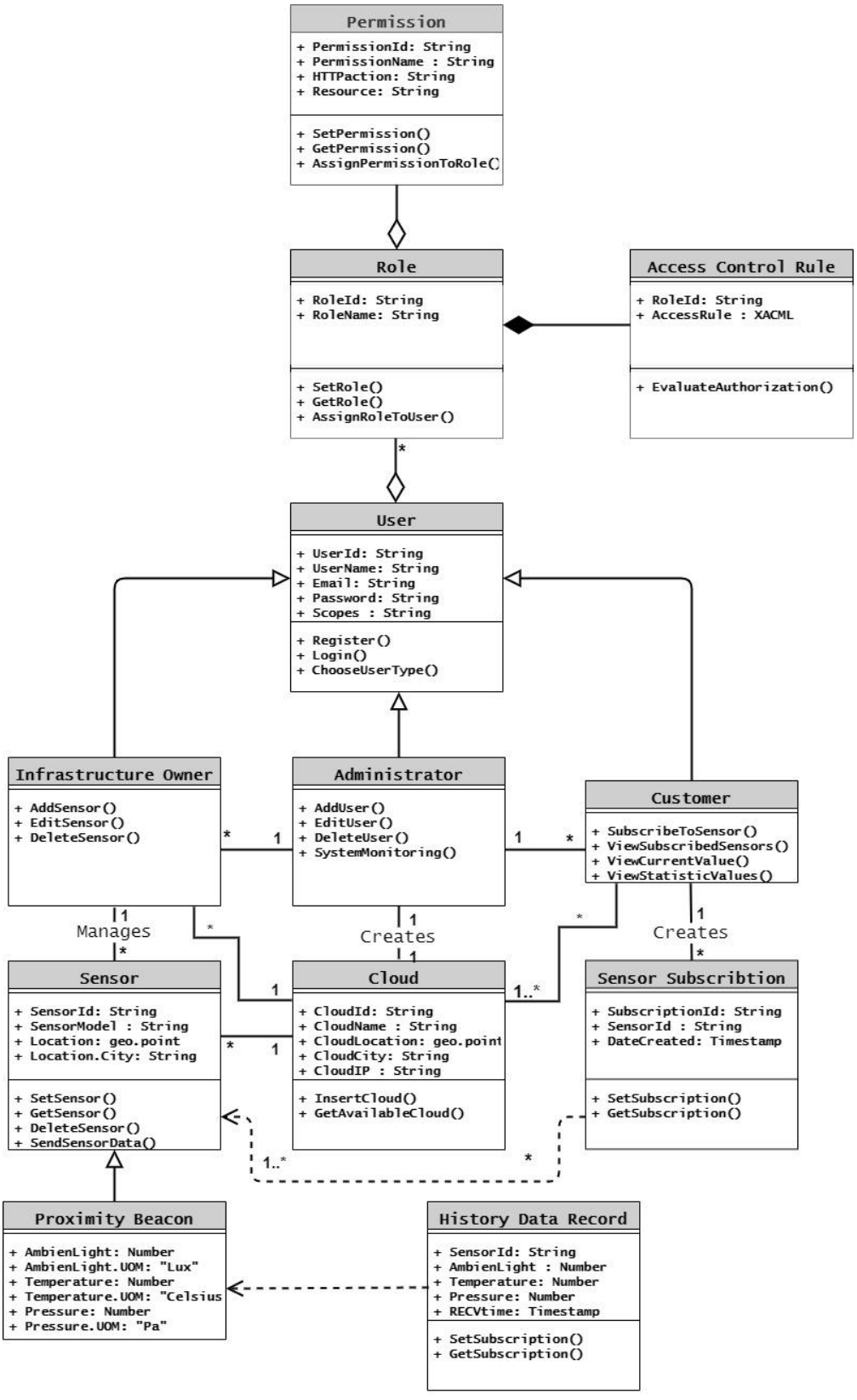


Figure 6- LINCA Class Diagram

In order to understand the functions of each cloud system of LINCA as well as the correlations of individuals services, a detailed description of the classes that make up each cloud system, is illustrated above in Figure 6.

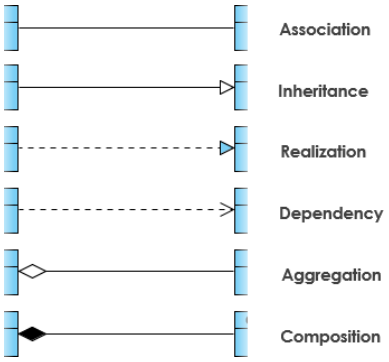


Figure 7 – Correlations between classes of LINCA's cloud system

Each class consists of a header, the attributes and the methods for managing the attributes of this class.

- **Class User** - This class is the generalization of cloud system users. It must be noted that it contains the components that make up each user's profile in the cloud system, such as username, email and password. Once the user has been authenticated by the cloud system, thus successfully login into it, the user continues as an infrastructure owner or a customer, focusing on its corresponding usage scenarios.

- **Class Customer** - The customer is a subclass of the user class. Users of this class focus their attention on the sensors that are registered by the corresponding infrastructure owners. Also, they can search for available cloud systems of LINCA, in order to subscribe to them and get access to their services and devices. In addition, they are able to search for available sensors that are connected to their subscribed cloud systems and subscribe to those sensors that they are interested in. If they subscribe to a sensor, they can monitor its current and statistics measurement.

- **Class Infrastructure Owner** - The infrastructure owner is also a subclass of the user class. A user of this class has the ability to insert / edit / delete sensors in his/her cloud system. One instance of this class is associated with one instance of cloud class.

- **Class Administrator** - The system administrator has the ability to register his/her cloud to LINCA system. Also, they can enter / edit / delete users that are subscribed and registered to his/her cloud system. In addition, administrator has the permission for monitoring of users, sensors, service

workloads of his/her cloud system. The Administrator class is a subclass of the user class.

- **Class Role** - An instance of role class is associated with an instance of permission class. Hence, a user of the cloud system is assigned a role with the corresponding permissions. This way the user inherits the corresponding role's permissions as well. The system automatically assigns a role to a user when they create an account in the cloud system in order to gain access to various cloud system services. The methods used by this class are about creating and assigning roles to a cloud user.

- **Class Permission** - The purpose of the Permission class is to describe the request that the holder has the right to execute. The request made by the user or service consists of an HTTP action such as GET, PUT, POST, PATCH, DELETE , in a resource located at the requested service such as http://cloudB/serviceD/resourceX. This class empowers users, such as infrastructure owners or customers, to access system services. The methods in this class relate to creating a permission and assigning it to a role.

- **Class Access Control Rules** - After permissions are assigned to a role, the access control rule class defines the right of the owner of that role to execute a request. An instance of the Access Control Rule class is associated with an instance of role class. It is responsible for permitting or denying access requests, based on the policy that constitutes the instance of the role. An access control rule follows the XACML standard (eXtensible Access Control Markup Language), where it is stored and maintained in the AuthZForce service.

- **Class Sensor Subscriptions** - Customer of each cloud system can subscribe to a subset of sensors that are connected to cloud systems of LINCA. An instance of the Sensor Subscription class concerns only one customer and contains information on his/her subscriptions to sensors. When a customer chooses to add a new sensor to his/her existing subscriptions list, a unique identifier of the sensor and the date of registration on that sensor, is recorded in a subscription instance. A customer can monitor his/her subscribed sensors through its subscription list.

- **Class Cloud** - This class is the generalization of cloud systems that are registered in LINCA system. When the administrator of each cloud system is registering its cloud system to LINCA must provide the attributes that defines it, such as the cloud ID , cloud name , cloud location, cloud city and cloud ip-address. Each cloud system may include as many sensors instances their infrastructure owner wishes.

- **Class Sensor** - This class is the generalization of different sensor models that are connected to LINCA's cloud systems. When the infrastructure owner of each cloud system is registering a sensor to his/her cloud system, must provide the attributes that defines it , such as the unique sensor ID and the sensor model. Also, the location and the owner of the sensor are automatically recorded by the cloud system. That depends on which cloud system the infrastructure owner logged in (e.g. If the infrastructure owner entered the cloud system that is located to Chania then the sensor's location will be Chania). The cloud system may include as many sensors instances the infrastructure owner wishes.

- **Class Proximity Beacon** - An instance of this class represents sensors of the "Estimote" company, namely "Proximity Beacon" sensors. These sensors can measure the temperature, the ambient light and the atmospheric pressure. There respective units of measurement is Celsius, Lux and Pa.

- **Class Historic Data** - For a sensor that is connected to a cloud system of LINCA, a history of data on changes in its attribute values is maintained. An instance of this class corresponds to only one sensor and contains raw and aggregated time series data for its measurements. The Cygnus service is responsible for this operation. Instance of Historical Data Class is used by the Comet service to extract statistics values, such as average, maximum, minimum, for measurements of each registered sensor. Class methods relate to the retrieval and storage of raw and aggregated data in different sensor instances.

3.4 Use Case Diagram

The functional requirements of customers, infrastructure owners and administrators that are described Section 3.2.1 are presented below in the form of Use Case Diagrams.

Customer through graphical interfaces can browse the available cloud systems that are registered in LINCA and can subscribe to the cloud systems that he/she wishes to access their services and devices. Also, customer user through a search engine can query for sensors that are connected to cloud systems in which he/she is subscribed. In addition, customer can browse a list with his/her subscribed sensors in order to view sensors' current and statistics values. Customer's functional requirements as explained in more detailed in Section 3.2.1 are illustrated in Figure 8.

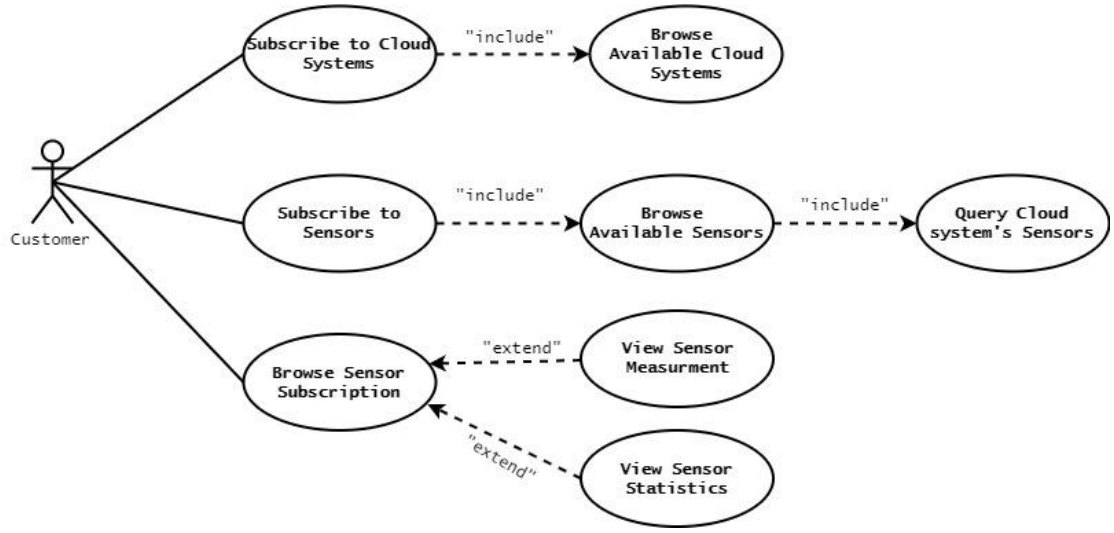


Figure 8 – Customer Use Case Diagram

Accordingly, an Infrastructure Owner through graphical interfaces can register sensors to his/her cloud systems. Also, they can browse a list with his/her registered sensors in order to edit or delete them. Infrastructure owners' functional requirements as explained in more detailed in in section 3.2.1 are illustrated in Figure 9.

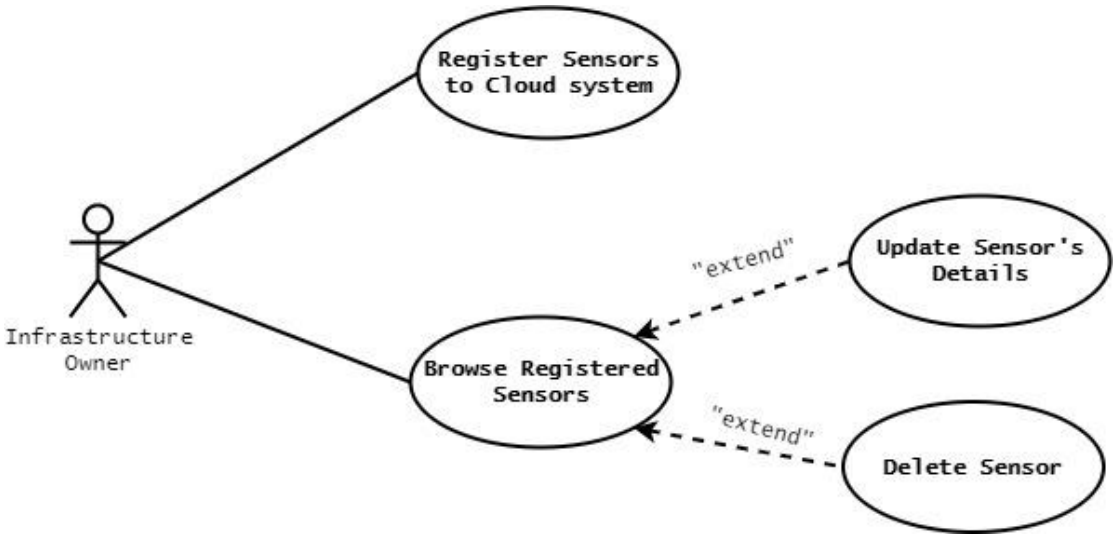


Figure 9 – Infrastructure Owner Use Case Diagram

Lastly, an administrator of cloud system is responsible to register his/her cloud system in LINCA. Also, an administrator of a cloud system is able to insert

/ edit / delete user to his/her cloud system. In addition, they have the ability to monitor the workload of virtual machines run by his/her cloud system services and the ability to monitor connected users' behavior. System Administrators' functional requirements as explained in more detailed in Section 3.2.1 are shown in Figure 10.

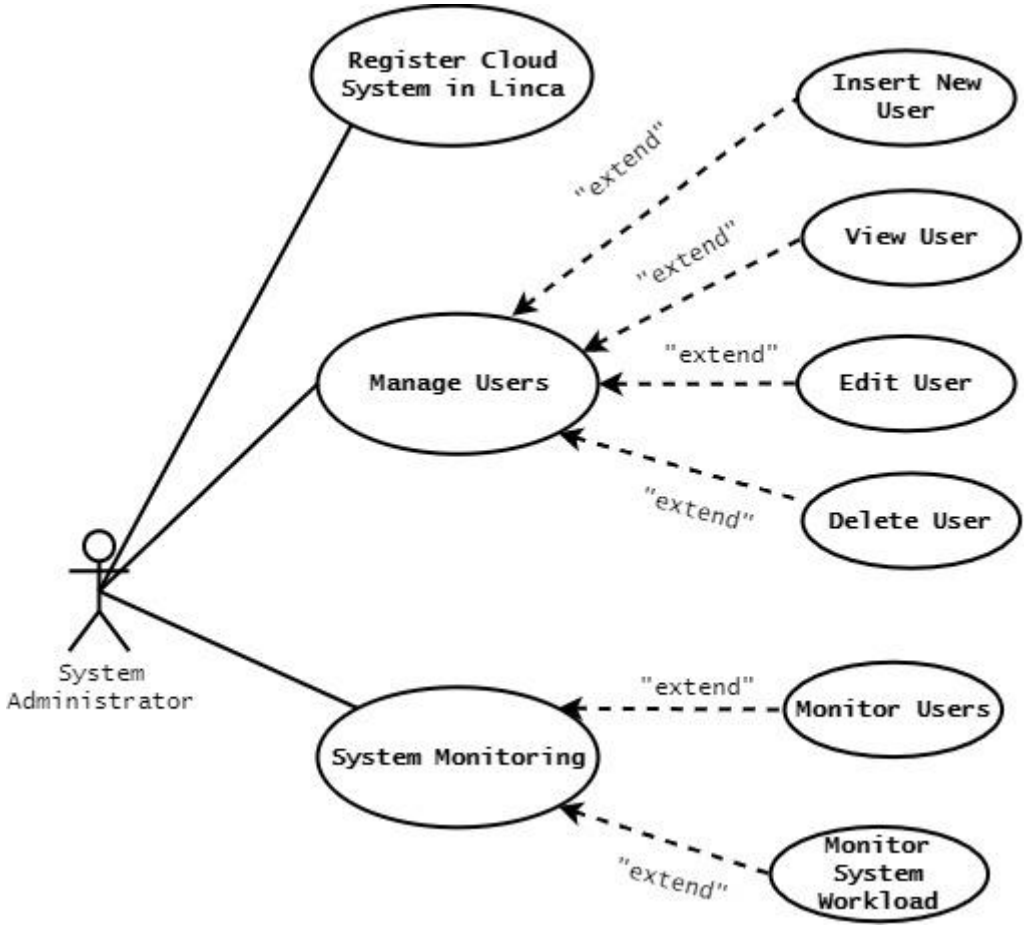


Figure 10 – Administrator Use Case Diagram

3.5 Sequence Diagram

This section shows the sequence diagrams for each type of user. The sequence diagrams below aim to present the most important system's functions for these categories of users.

As local services we define the services that are located to the cloud system in which the user is connected and makes requests and as remote services we define the services that are located to a remote cloud system of LINCA in which the user is interested to access its services.

The main idea of the distributed system is to exploit the features of Cassandras (e.g. replication, master-less, scalable). Each LINCA's cloud system has a node of the Cassandras cluster. A cloud system can become known in the LINCA system by registering to Cassandra. Automatically it becomes discoverable by other cloud system of the distributed system.

All requests made by users through the graphical interfaces to protected services must pass through a common stage. This stage involves user authentication by the local Keyrock. Once the user has been identified by this local service, they can proceed to the next stage, the user authorization.

- **Registration of cloud system in LINCA**

System Administrator of each cloud system is responsible to register his/her cloud system to LINCA system in order to be discoverable from users that wish to subscribe to his/her cloud system. Administrator of each cloud system through the local Web Application can type the information of his/her cloud system, such as name of system, location, ip address and description (a text that describes the cloud system) . Web Application forwards the admin's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic is responsible to add the OAuth2 token of Administrator to his/her initial request and forward it to the local PEP Proxy of Register Cloud Service. Local PEP proxy of Register Cloud Service checks the OAuth2 token in local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Register Cloud Service. Immediately, local PEP Proxy of Register Cloud Service checks in the local AuthZForce Service if the identified user has the permissions to access the local Register Cloud Service. After the successful authentication and authorization of the admin user, his/her initial request is forwarded to the protected service, Register Cloud Service. This service is responsible to insert the information of admin's cloud system to the local node of Cassandra's cluster.

The above procedure is represented as a sequence diagram in Figure 11. The blue dashed lines indicate that the corresponding services are located to local cloud system. The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

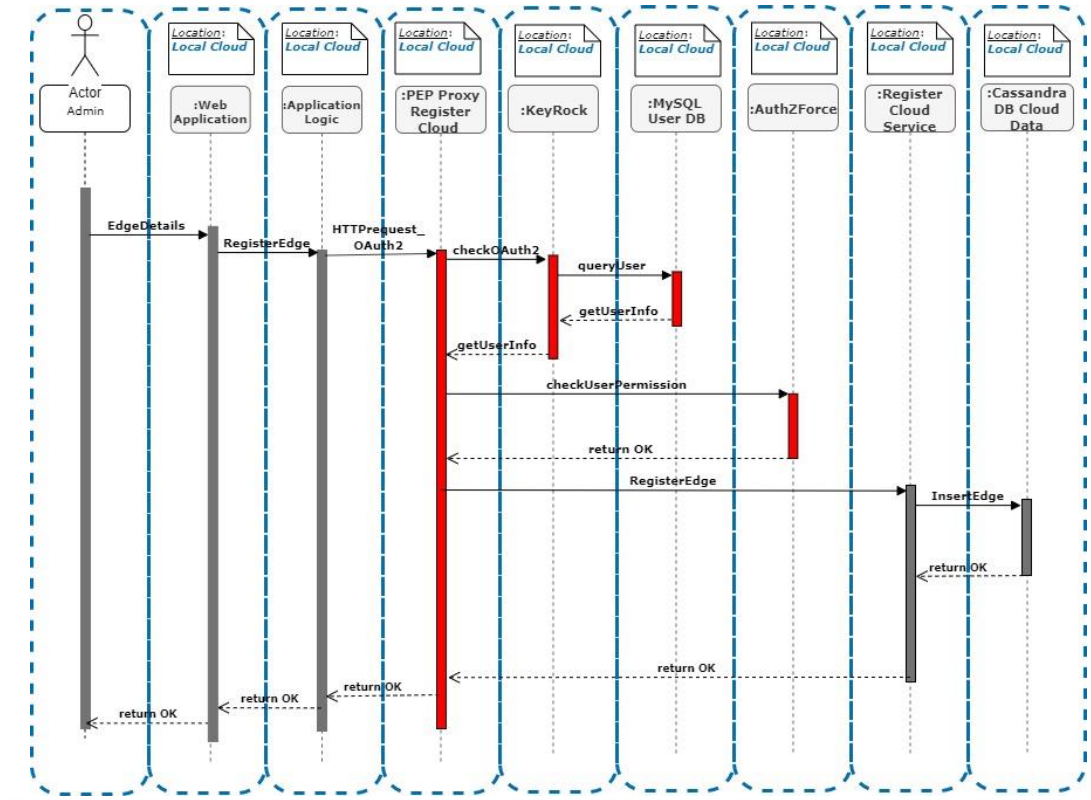


Figure 11 – Registration of cloud system in LINCA Sequence Diagram

[*EdgeDetails*] - Admin via Web Application is typing the information of his/her cloud who wants to register in LINCA.

[*RegisterEdge*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - PEP proxy checks user's permissions in AuthZForce.

[*QueryRegisteredEdges*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , the Register

Cloud Service. If AuthZForce returns “Denied” then the PEP proxy will not forward the initial request.

[*RegisterEdge*] – Register Cloud Service process the request and imports the cloud system’s information in Cassandra DB .

[*return OK*] - In the end, Cassandra DB returns if the insertion was success.

- **Querying for Available Cloud Systems in LINCA**

A customer has the ability to search for available cloud systems that are registered in LINCA in order to subscribed to them. Each cloud system is registered to LINCA by the corresponding administrator as shown in the previous paragraph. Customer through the local Web Application makes a query request to find the available LINCA’s cloud systems. Web Application forwards the customer’s request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Customer to his/her initial request and forward it to the local PEP Proxy of Query Available Clouds Service. Local PEP proxy of Query Available Clouds Service checks the OAuth2 token in local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user’s information of this OAuth2 token to local PEP Proxy Query Available Clouds Service. Immediately, local PEP Proxy of Query Available Clouds Service checks in the local AuthZForce Service if the identified user has the permissions to access the local Query Available Clouds Service. After the successful authentication and authorization of the customer user, his/her initial request is forwarded to the protected service, the Query Available Clouds Service. This service is responsible to query for available LINCA’s cloud systems in the local node of Cassandra’s cluster. This node is responsible for communicating with other Cassandra’s nodes in order to retrieve the data for the user's request. Using Cassandras does not require direct access to foreign systems since it is done directly by Cassandra.

The above procedure is represented as a sequence diagram in Figure 12. The blue dashed lines indicate that the corresponding services are located to local cloud system. The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

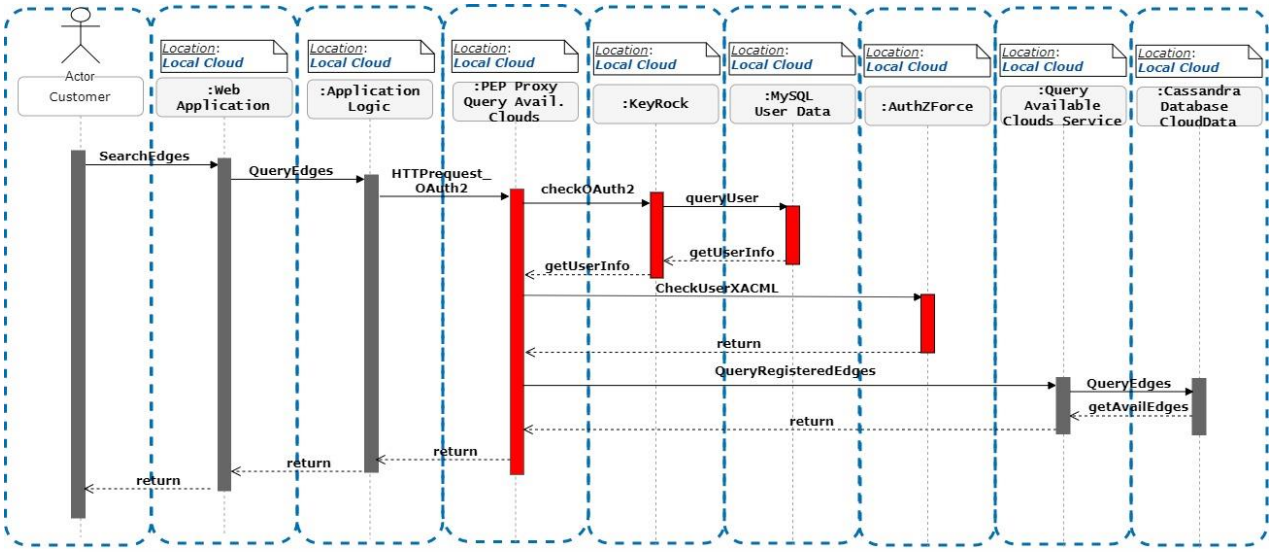


Figure 12 – Querying for LINCA's available Cloud Systems Sequence Diagram

[*SearchEdges*] - User via Web Application wants to search for the Available Clouds systems in LINCA .

[*QueryEdges*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - PEP proxy checks user's permissions in AuthZForce.

[*QueryRegisteredEdges*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , the Query Available Clouds Service. If AuthZForce returns "Denied" then the PEP proxy will not forward the initial request.

[*QueryEdges*] - Query Available Clouds Service process the request and starts querying for available cloud systems in Cassandra DB.

[*getAvailEdges*] - In the end, Cassandra DB returns the available cloud systems.

- **Subscribe to LINCA's cloud systems**

A customer as shown in previous paragraph he/she can query for available clouds that are registered in LINCA. After this process, user can choose the cloud systems that wishes to access their services and devices. Customer through the local Web Application makes a subscription request to the LINCA's cloud system that is interested. Web Application forwards the customer's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Customer to his/her initial request and forward it to the local PEP Proxy of Query Available Clouds Service. Local PEP proxy of Query Available Clouds Service checks the OAuth2 token in local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Query Available Clouds Service. Immediately, local PEP Proxy of Query Available Clouds Service forward the user's subscription request to the cloud system administrator that the customer has chosen to subscribe with. This cloud system administrator must permit or deny the user's subscription request. If this cloud system administrator accepts the subscription request, then he/she creates a role for the requested user in the Keyrock Service which is located to his/her cloud system. In the end, the Keyrock Service creates a XACML file with the permissions of the requested users in AuthZForce which is also located to the chosen cloud system.

The above procedure is represented as a sequence diagram in Figure 13. The blue dashed lines indicate the corresponding services are located to local cloud system. The green dashed lines indicate the corresponding services that are located to the remote cloud system(the system that user is requested to subscribe).The red boxes represent the services that are responsible for the authentication and registration of the requested user.

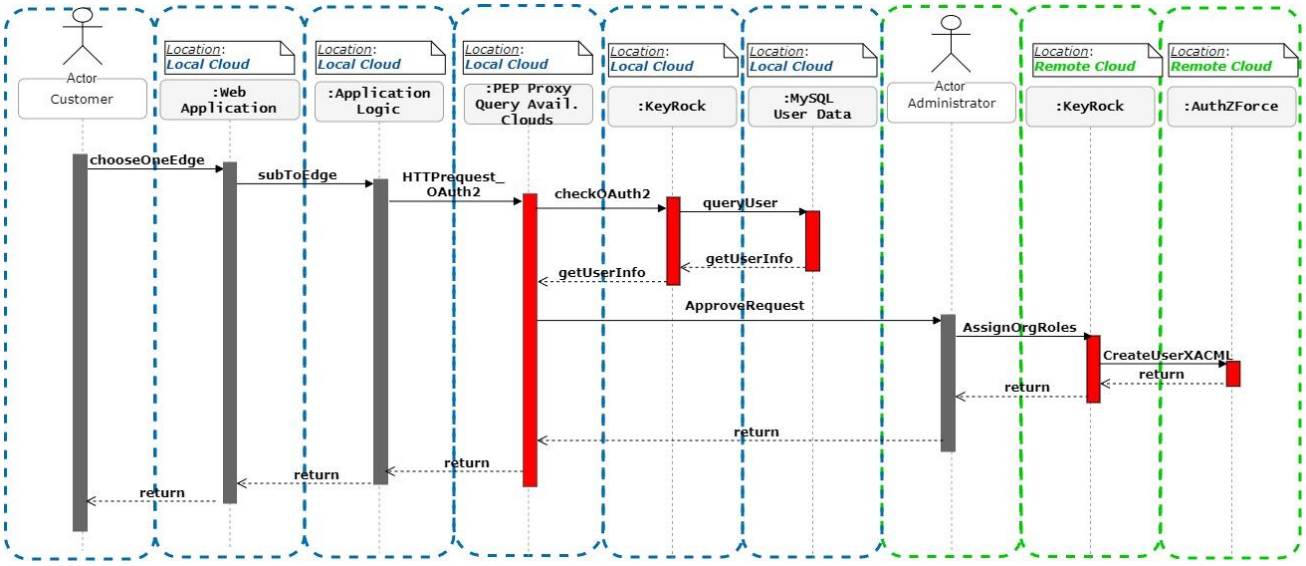


Figure 13 – Subscribe to LINCA's cloud system Sequence Diagram

[*chooseOneEdge*] - User via Web Application chooses the cloud system he/she wishes to subscribe for.

[*subToEdge*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately, Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*AssignOrgRoles*] - Admin of the chosen cloud system processes the request. If he/she accepts the request, then creates a role in his/her Keyrock for the user who makes the request.

[*CreateUserXACML*] - In the end, Keyrock creates a XACML file with the permissions in AuthZForce.

- **Insertion of Sensor in LINCA's cloud system**

Infrastructure owner is responsible to register sensors to his/her cloud system and make them discoverable to users that are connected to different cloud systems of LINCA. The Infrastructure Owner through the local Web Application makes an insert request to register a sensor to his/her cloud system. Web Application forwards the infrastructure owner's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Infrastructure Owner to his/her initial request and forward it to the local PEP Proxy of Register Sensors Service. Local PEP proxy of Register Sensors Service checks the OAuth2 token in local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Register Sensors. Immediately, local PEP Proxy of Register Sensors checks in the local AuthZForce Service if the identified user has the permissions to access the local Register Sensors Service. After the successful authentication and authorization of the customer user, his/her initial request is forwarded to the protected service, the Register Sensors Service. This service is responsible to insert the sensor's information in the local node of Cassandra's cluster in order to be discoverable from subscribed users that are registered in a remote cloud system of LINCA. Also, the Register Sensors Service forwards the sensor's details to local IoT Agent in order to receive data from this sensor. In addition, for this sensor, the IoT Agent creates in the local Publish/Subscribe Service a sensor entity.

The above procedure is represented as a sequence diagram in Figure 14. The blue dashed lines indicate the corresponding services are located to local cloud system. The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

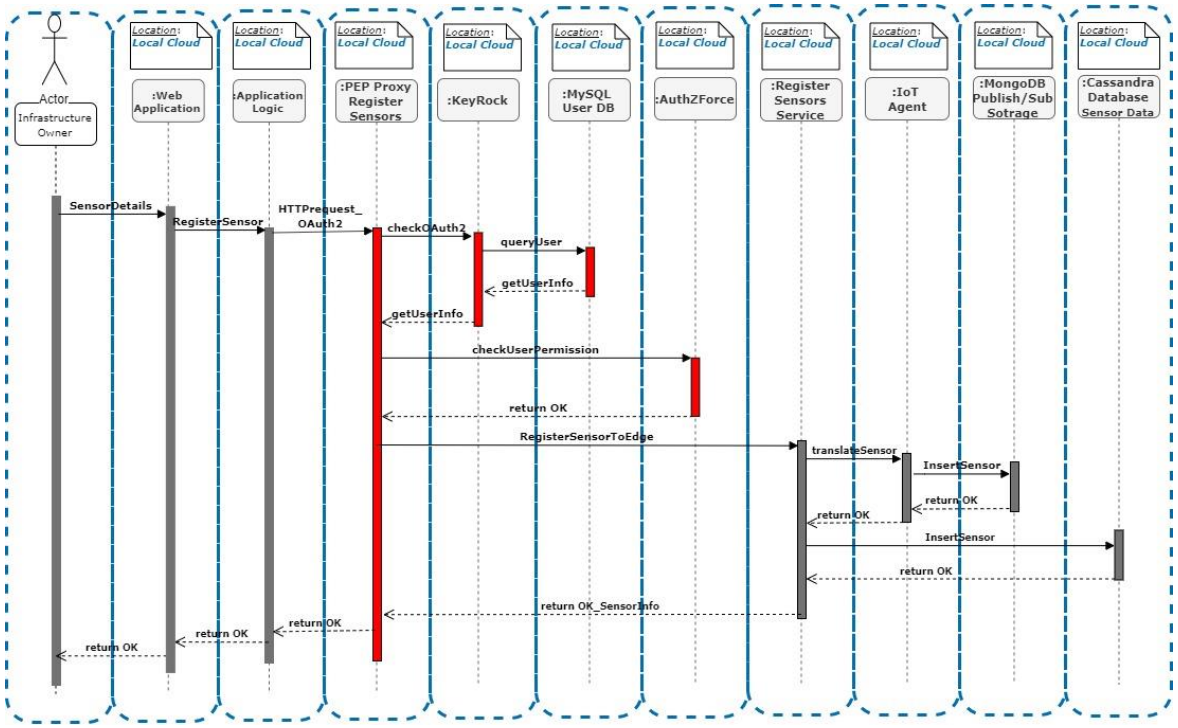


Figure 14 – Insertion of sensor in LINCA's cloud system Sequence Diagram

[*SensorDetails*] - Infrastructure Owner via Web Application is typing the information of sensor that wants to register to his/her cloud system.

[*RegisterSensor*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information.

[*CheckUserXACML*] - PEP proxy checks user's permissions in AuthZForce.

[*RegisterSensorOfEdge*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , the Register Sensors Service. If AuthZForce returns "Denied" then the PEP proxy will not forward the initial request.

[*translateSensor*] - Register Sensors Service processes the request and forwards it to IoT Agent .

[*InsertSensor*] - IoT Agent is responsible to create a sensor entity to Publish/Subscribe Storage where there are other sensors entities. In this way when IoT Agent receives data from the registered sensors, it will

forward to Publish/Subscribe Service to update the corresponding sensor entity.

[*InsertSensor*] - Also Register Sensors Service insert the information of the registered sensor to the local node of Cassandra's cluster, so it can be reached by subscribed users that are registered to remote cloud systems of LINCA.

- **User Authentication-Authorization in LINCA's cloud systems**

The authentication and authorization process of a user is performed in two levels, locally and distributed.

In the first case, the authentication and authorization process is performed locally by a single cloud system of LINCA.

This local authentication and authorization process happens when an administrator of cloud system tries to register his/her system to LINCA, when an infrastructure owner tries to register sensors to his/her cloud system and when a customer is querying / subscribing sensors that are connected to his/her cloud system. Also, this process performed when a customer wants to retrieve current and historical measurements of a subscribed sensor that is connected to his/her cloud system of LINCA.

The user through the local Web Application makes a request to a service in his/her cloud system. Local Web Application forwards the user's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of user to his/her initial request and forward it to the local PEP Proxy who protects the local service that the user wants to access. This local PEP proxy checks the OAuth2 token in the local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy. Immediately, local PEP Proxy checks in the local AuthZForce Service if the identified user has the permissions to access the protected service. After the successful authentication and authorization of the user, his/her initial request is forwarded to the protected service. The local authentication and authorization of user is represented as a sequence diagram in Figure 15a. The blue dashed lines indicate the corresponding services are located to local cloud system.The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

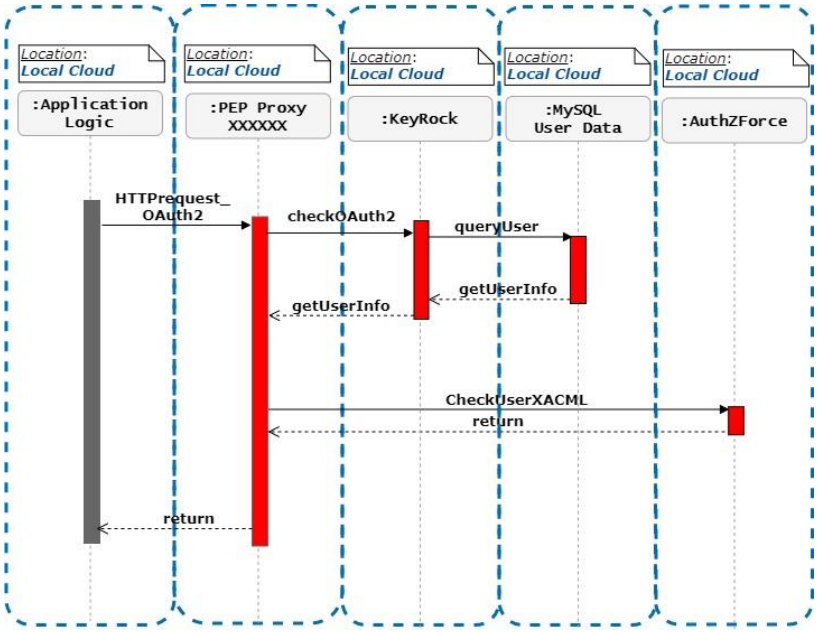


Figure 15a – Local Authentication-Authorization process of User Sequence Diagram

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to user's initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - PEP proxy checks for user's permissions in AuthZForce.

In the second case, the authentication and authorization process is performed distributed by the local cloud system in which the user is connected and the remote cloud system of LINCA in which user want to access its services and devices. As explained in Figure 13 ,when user subscribe to a remote cloud system of LINCA, his/her permissions to this cloud is stored as a XACML file in the AuthZForce Service of this remote cloud system.

This distributed authentication and authorization process happens when a customer is connected to his/her cloud system and he/she tries to query/subscribe to sensors that are connected to a remote cloud system to LINCA. Also, this process performed when a customer wants to retrieve current and historical measurements of a subscribed sensor that is connected to a remote cloud system of LINCA.

The user through the local Web Application is requesting to access sensors that is connected to a remote cloud system of LINCA. The local Web Application forwards the user's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of user to his/her initial request and forward it to the local PEP Proxy. This local

PEP proxy checks the OAuth2 token in the local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy. In this way, the local Keyrock guarantees that the requested user is registered to its system and is valid user. Immediately, local PEP Proxy checks in the remote AuthZForce Service if the identified user has the permissions to access the sensors that are connected to this remote cloud system. After the successful authentication and authorization of the user, he/she can query and subscribe the corresponding remote sensors through the local Query Sensor Service. This service is responsible to route request to the local node of Cassandra in order to retrieve the requested sensors. The distributed authentication and authorization process of user is represented as a sequence diagram in Figure 15b. The blue dashed lines indicate the corresponding services are located to local cloud system. The green dashed lines indicate the corresponding services are located to a remote cloud system. The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

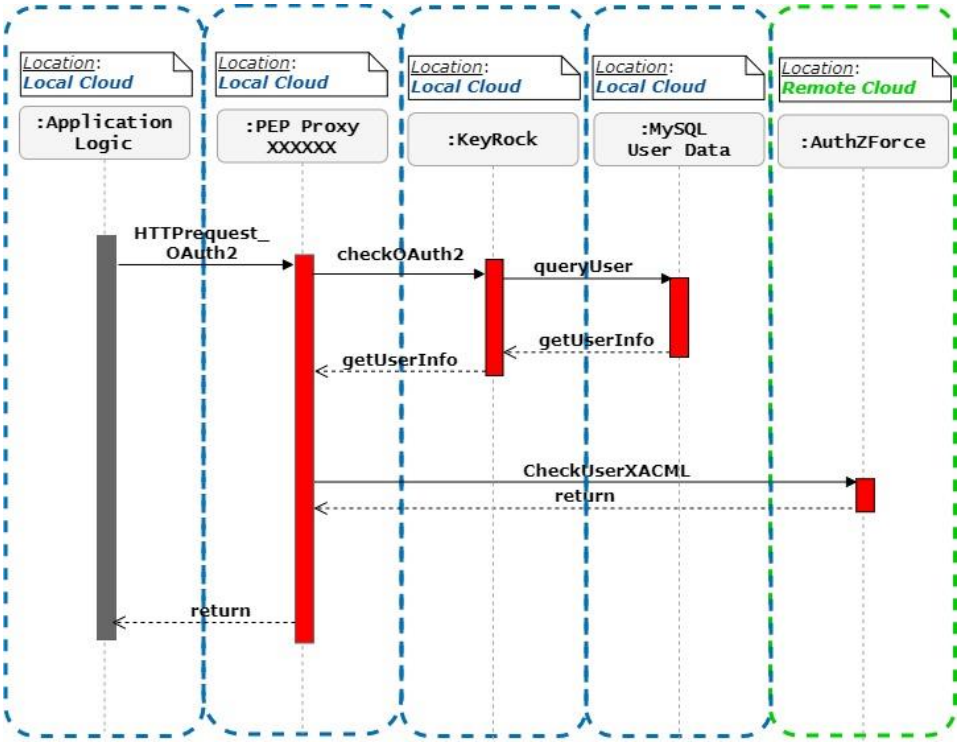


Figure 15b – Distributed Authentication-Authorization process of User Sequence Diagram

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to user's initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - PEP proxy checks for user's permissions in AuthZForce of the requested remote cloud system.

- **Querying sensors in LINCA's cloud systems**

A customer has the ability to search for sensors that are connected to his/her cloud system and in others remote cloud systems of LINCA. Customer through the local Web Application makes a query request to find the sensors in the desired LINCA's cloud system.

Web Application forwards the customer's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Customer to his/her initial request and forward it to the local PEP Proxy of Query Sensors Service. The local PEP proxy of Query Sensors Service checks the OAuth2 token in the local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Query Sensors Service. If the identified user requested sensors that are connected to a remote cloud system, the local PEP proxy Query Sensors checks for user's permissions in AuthZForce Service of the remote cloud system. Else if the user requested sensors that are connected to his/her cloud system, the local PEP proxy Query Sensors checks for user's permissions in AuthZForce Service of user's cloud system. After the successful authentication and authorization of the customer user, his/her initial request is forwarded to the protected service, the local Query Sensor Service. This service is responsible to query in the local node of Cassandra's cluster for sensors that are connected to different cloud systems of LINCA.

The above process is represented as a sequence diagram in Figure 16. The blue dashed lines indicate the corresponding services are located to local cloud system. The green dashed lines indicate the corresponding services that are located to the remote cloud system(the system that user is requested to find its sensors). The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

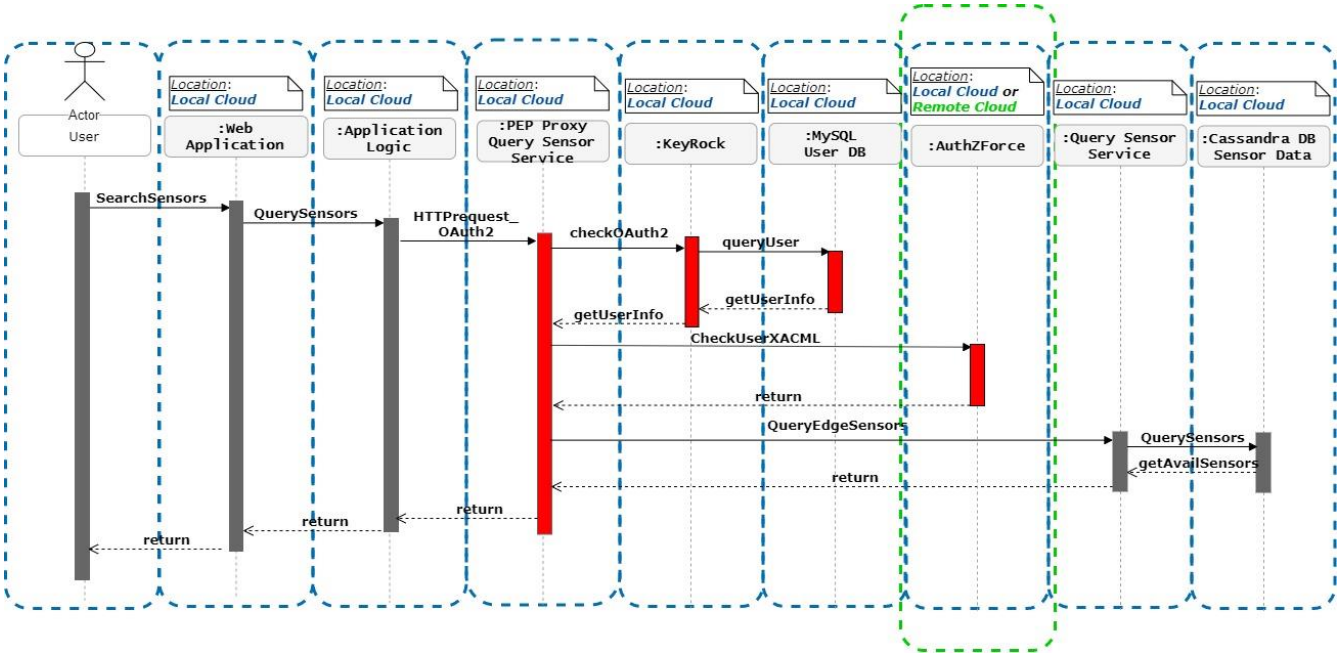


Figure 16 – Querying sensors in LINCA's cloud systems Sequence Diagram

[*SearchSensors*] - User via Web Application chooses the attributes and the cloud systems of sensor who wants to search.

[*QueryEdges*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information.

[*CheckUserXACML*] - If user requested sensors that are connected to a remote cloud system then PEP proxy checks for user's permissions in AuthZForce of the remote cloud. Else PEP proxy checks for user's permissions in AuthZForce of local cloud system.

[*QueryRegisteredEdges*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , the Query Sensor Service of current cloud. If AuthZForce returns "Denied" then the PEP proxy will not forward the initial request.

[*QuerySensors*] - After the success authentication and authorization of the user, Query Sensor Service starts querying for requested sensors in local node of Cassandra DB .

[*getAvailSensors*] - In the end, Cassandra DB returns the requested sensor.

- **Subscribe to Sensors in LINCA's cloud systems**

A customer as shown in previous paragraph he/she can query for sensors that are connected to different cloud systems of LINCA. After this process, user can choose the sensors that wishes to subscribe.

Customer through the local Web Application makes a subscription request for the sensors that wishes to subscribe. Web Application forwards the customer's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Customer to his/her initial request and forward it to the local PEP Proxy of Query Sensors Service. The local PEP proxy of Query Sensors Service checks the OAuth2 token in the local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Query Sensors. If the identified user requested to subscribe to sensors that are connected to a remote cloud system, the local PEP proxy Query Sensors checks for user's permissions in AuthZForce Service of the remote cloud system. Else if the user requested to subscribed to sensors that are connected to his/her cloud system, the local PEP proxy Query Sensors checks for user's permissions in AuthZForce Service of user's cloud system. After the successful authentication and authorization of the customer user, his/her initial request is forwarded to the protected service, the local Query Sensor Service. If the requested sensors are connected to a remote cloud system, then the Query Sensor Service is responsible to subscribe the local Orion Context Broker to the remote Orion Context Broker for the requested sensors. In this way when the remote Orion Context Broker receives updates for the subscribed sensors, it will forward them to the local Orion Context Broker. Also Query Sensor Service is responsible to update user's subscription list in the local node of Cassandra's cluster.

The above process is represented as a sequence diagram in Figure 17. The blue dashed lines indicate the corresponding services are located to local cloud system. The green dashed lines indicate the corresponding services that are located to the remote cloud system(the system that user is requested to find its sensors). The red boxes represent the services that are responsible for the authentication and authorization of the requested user.

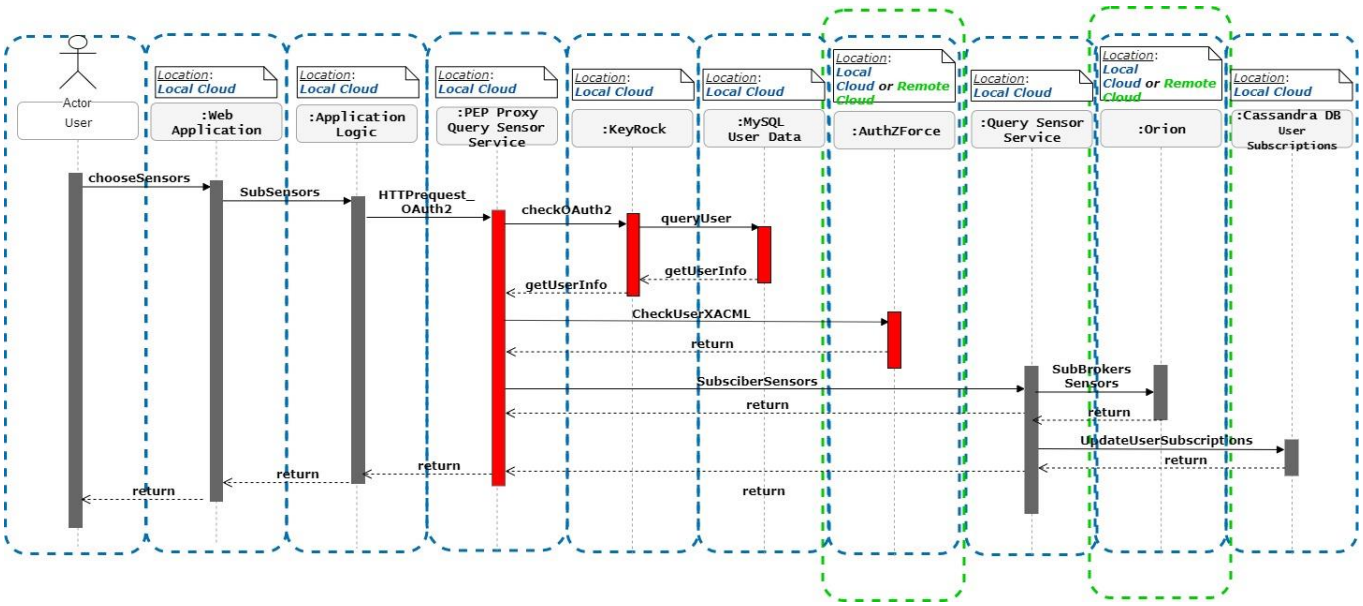


Figure 17 – Subscribe to sensors of LINCA's cloud systems Sequence Diagram

[*chooseSensors*] - User via Web Application after the process of searching , he/she chooses the sensors who wants to subscribe for.

[*SubSensors*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - If user requested to subscribe to sensors that are connected to a remote cloud system then PEP proxy checks for user's permissions in AuthZForce of the remote cloud. Else PEP proxy checks for user's permissions in AuthZForce of local cloud system.

[*SubscribeSensors*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , the Query Sensor Service . If AuthZForce returns "Denied" then the PEP proxy will not forward the initial request.

[*SubBrokersSensors*] - If the requested sensors are connected to a remote cloud system, then the Query Sensor Service is responsible to subscribe the local Orion Context Broker to the remote Orion Context Broker for the requested sensors.

[*UpdateUserSubscriptions*] . In the end, a request is routed to the local node of Cassandra DB to update the user's subscriptions list.

- **Retrieve Statistic/Current Value of a Sensor in LINCA's cloud system**

A customer as shown in previous paragraph he/she can subscribe to sensors that are connected to different cloud systems of LINCA. After this process, user can retrieve statistic and current value of the his/her subscribed.

Customer through the local Web Application makes a request for the the sensor that wishes to retrieve its current value or a statistic value. Web Application forwards the customer's request to the local Application Logic in order to route it to the appropriate services. Then the local Application Logic add the OAuth2 token of Customer to his/her initial request and forward it to the local PEP Proxy of Comet Service. The local PEP proxy of Comet Service checks the OAuth2 token in the local Keyrock in order to identify who is making this request. After, the local Keyrock returns the user's information of this OAuth2 token to local PEP Proxy Comet Service. If the identified user requested to retrieve statistic/current value of sensor that is connected to a remote cloud system, the local PEP proxy Comet checks for user's permissions in AuthZForce Service of the remote cloud system. Else if the identified user requested to retrieve statistic/current value of sensor that is connected to his/her cloud system, the local PEP proxy Comet checks for user's permissions in AuthZForce Service of user's cloud system. After the successful authentication and authorization of the customer user, his/her initial request is forwarded to the protected service, the local Comet Service. Comet processes the request and starts querying in History DB for the current/statistic value of the chosen sensor.

The above process is represented as a sequence diagram in Figure 18. The blue dashed lines indicate the corresponding services are located to local cloud system. The green dashed lines indicate the corresponding services that are located to the remote cloud system (the system that user is requested to find its sensors). The red boxes represent the services that are responsible for the authentication and authorization of the requested user

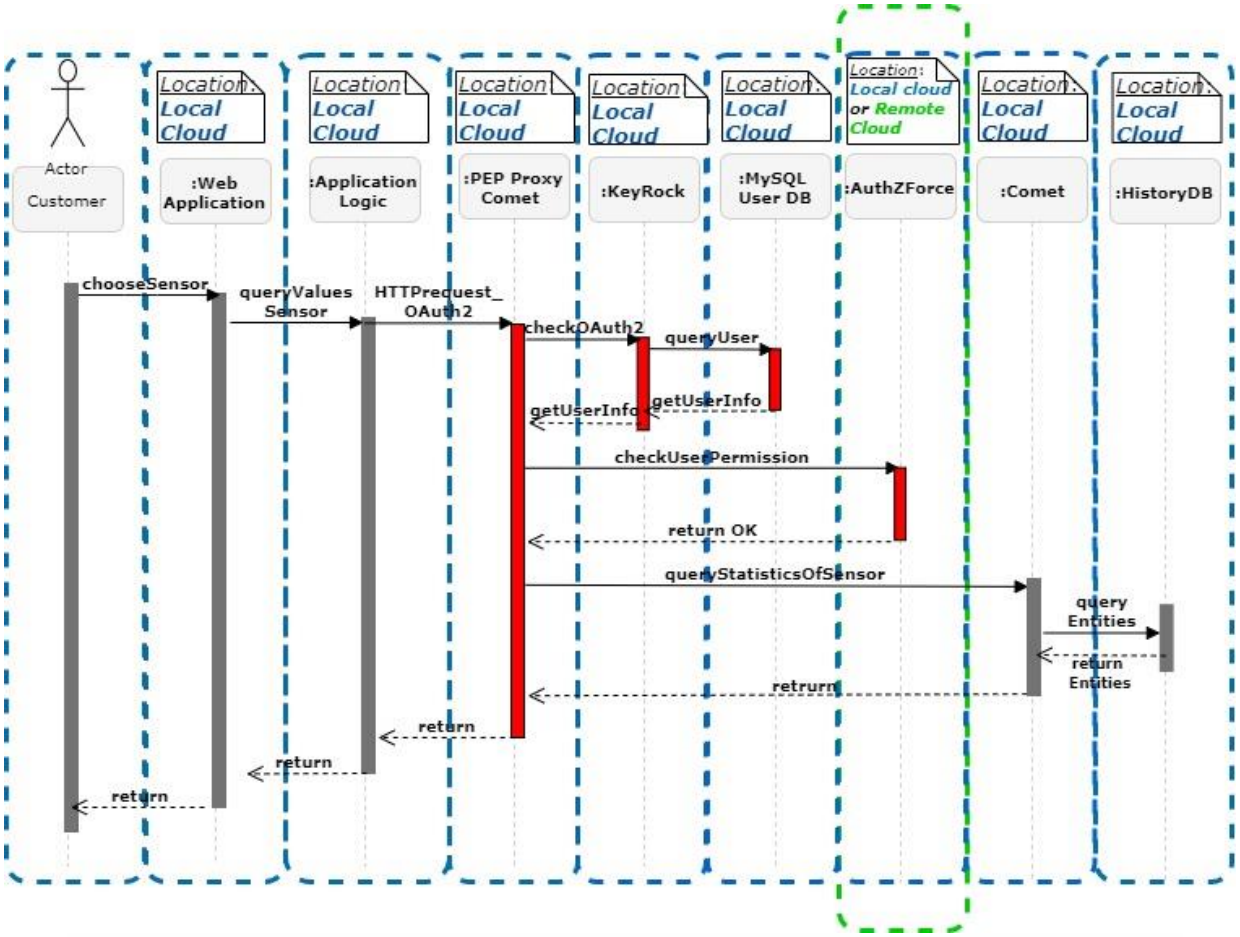


Figure 18 – Retrieve Current/Statistic Value of sensor in LINCA's cloud system Sequence Diagram

[*chooseSensors*] - User via Web Application chooses from subscribed sensors list , the sensors who wants to retrieve current value/statistics.

[*queryValueSensors*] - Web Application forwards the request to Application Logic.

[*HTTPrequest_OAuth2*] - Application Logic adds User OAuth2 token to the initial request and then forwards it to PEP Proxy.

[*checkOAUth2* , queryUser , getUserInfo] - PEP proxy checks User OAuth2 token in Keyrock. Immediately , Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then Keyrock returns to PEP proxy the user's information,

[*CheckUserXACML*] - PEP proxy checks for user's permissions in AuthZForce.

[*queryStatisticsOfSensors*] - If AuthZForce returns "Permit" then PEP proxy forwards the initial user's request to the protected service , Comet . If

AuthZForce returns “Denied” then the PEP proxy will not forward the initial request.

[*queryEntities*] - Comet processes the request and starts querying in History DB for current /statistic value of the chosen sensor.

[*getStatisticsValueOfSen*] – History DB returns the requested information to user.

3.6 Architecture Diagram

LINCA is a unified master-less distributed system. The term “master-less” means that all its clouds which it is consisted of are equal and similar. These clouds consist of several components which implement front-end and back-end services. Each cloud of LINCA is based on SOA architecture principles as explained above in section 2.1. This extends the micro-services of every cloud system to loosely coupled micro-services which can be developed, deployed and maintained independently. As a result, in the case of an error in one micro-service the whole cloud system does not necessarily stop its functionality.

In addition, LINCA’s clouds can communicate with one another in order to satisfy the requirements of the different categories of users as described in section 3.2. This communication is achieved through RESTful communication over an HTTP protocol as described in section 2.1.1.

In order to describe the architecture of the LINCA system, all is needed is to describe one of the clouds that are contained in the system due to the equal and similar characteristics they all have. Initially, the main services of the abstract LINCA’s cloud-level architecture will be given in section 3.6.1 . Once a general idea of what the LINCA’s cloud-level abstract architecture main services is given, a more detailed analysis of the LINCA architecture will be presented in section 3.6.2.

3.6.1 LINCA’s Abstract Cloud-Level Architecture Diagram

The Figure 19 shows an abstract view of the architecture of an cloud that is contained in LINCA system.

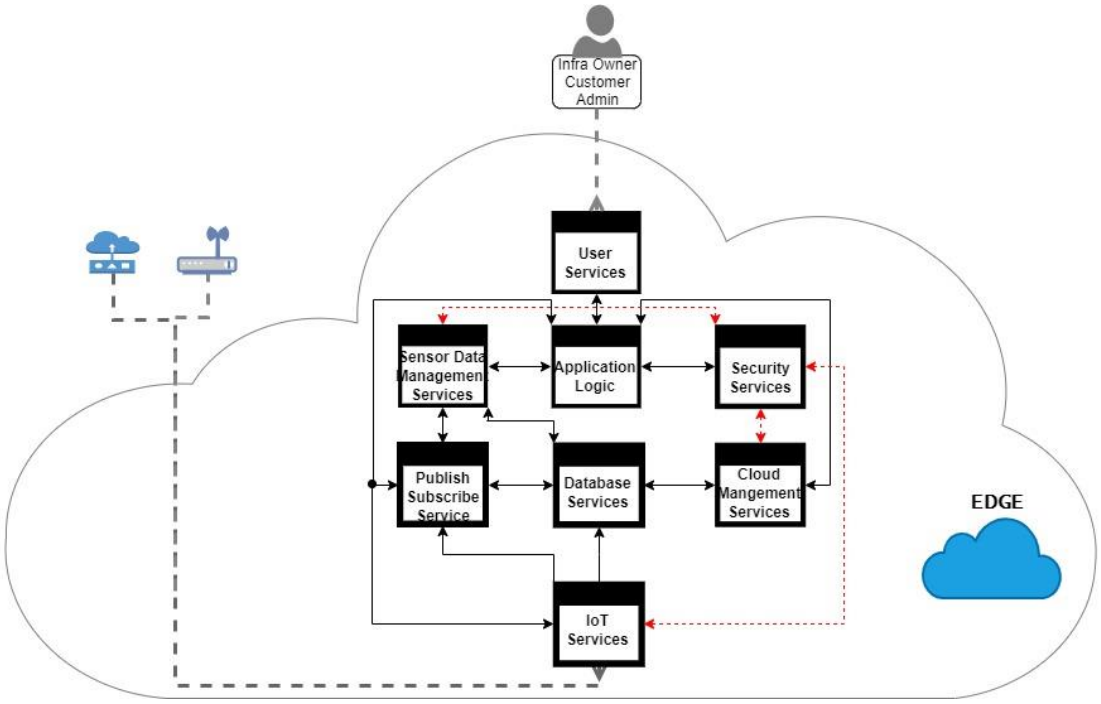


Figure 19 – LINCA's Abstract Cloud-Level Architecture Diagram

This cloud is consisted of the following main services :

1. **User Services** – This service is responsible for the graphical interface of different type of users.

2. **Application Logic** – This service includes the code for orchestrating individual services, so that the cloud system can implement the specified functionality

3. **IoT Services** – At this type of service the IoT devices are connected. They send their measurements to each other through gateways. Also, this service is responsible for registering the IoT devices to its cloud system and querying for the IoT devices that are in every LINCA's cloud system.

4. **Database Services** – These services are consisted of different types of databases that each one of them is connected with other services of the cloud system in order to provide storage functionalities.

5. **Cloud Management Services** - These services are used by the administrator, if they wish to register their cloud system to LINCA. Also, these services are used by the Customer users for querying for available LINCA's cloud systems that are registered to it by the corresponding administrators.

6. **Sensor Data Management Services** – These services are responsible for storing data to the Database Services that are received from the Publish/Subscribed Service. Also, they can retrieve history data from the Database Services.

7. **Publish/Subscribe Service** – This type of service acts as the context broker for IoT devices entities. These entities are stored in the Database Services in the form of JSON. Also, with the assistance of this type of service, users can subscribe to IoT devices that are located in LINCA's cloud systems in order to receive their measurements

8. **Security Services** - The purpose of these services is to prohibit users and services from using system functions

The arrows in Figure 19 represent the RESTful inter-communication of these services over the HTTP protocol. To illustrate this in figure 19, the red arrows represent the connection of security service with the rest of the services.

3.6.2 LINCA's Cloud-Level Architecture Diagram

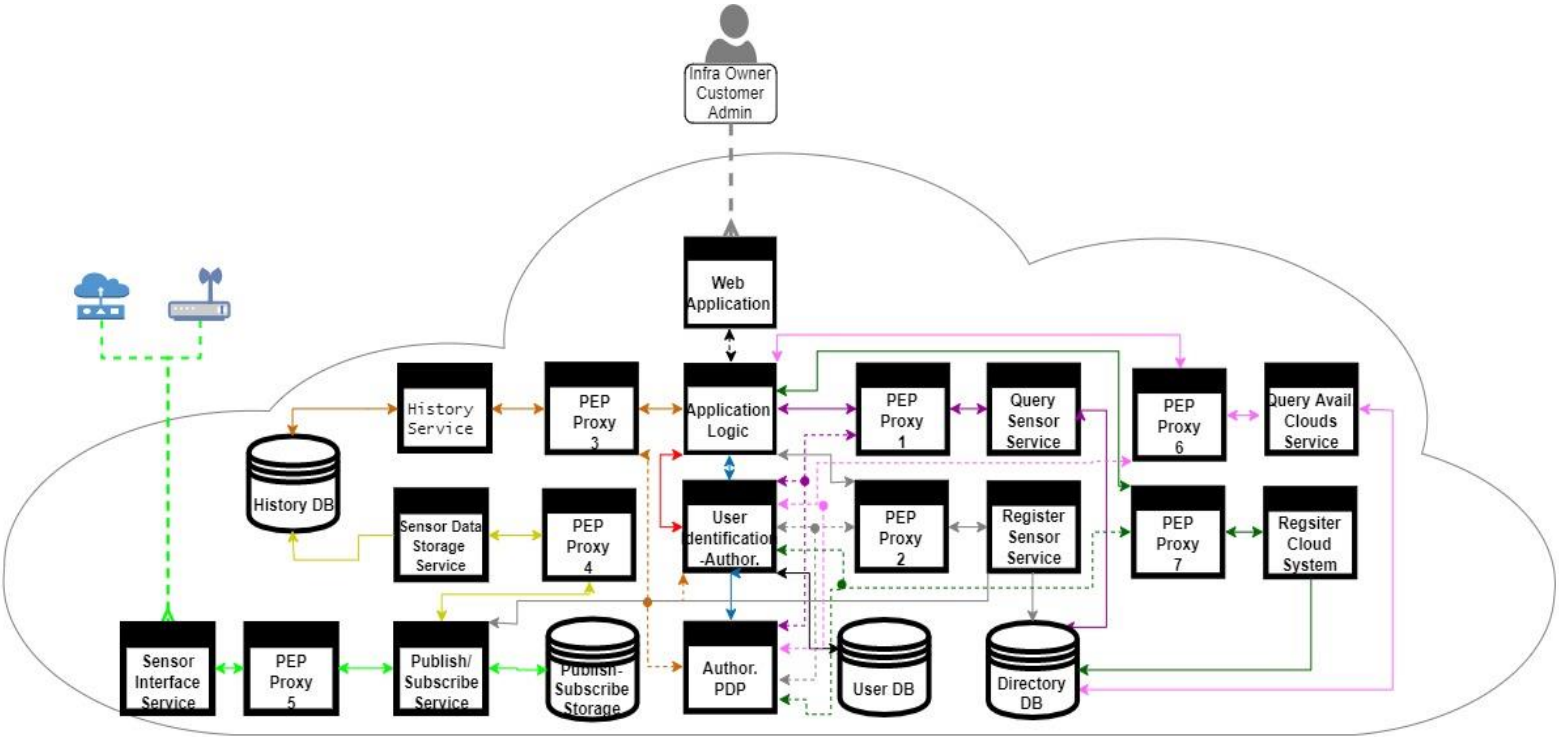


Figure 20 – LINCA's Cloud-Level Architecture Diagram

The Figure 20 illustrates an even more detailed view of cloud architecture. The arrows represent the RESTful inter-communication of the above cloud system services over HTTP protocol. For a better explanation of this, each cloud's function is represented with a different color, as described below :

- Red – Identification and Authorization of User for logging into the system.
  [*Application Logic , User Identification – Authorization , User DB*]

- Blue – Assigning roles and permissions to a user.
  [*Application Logic , User Identification – Authorization , User DB , Authorization Policy Decision Point*]

- Light Green – Connecting different types of sensors to the system.
  [*Things, Gateway, Sensors Interface Service , Policy Enforcement Point Proxy 5, Publish/Subscribe Service , Publish/Subscribe Storage*]

- Pink – Querying Available Clouds in LINCA system.
  [*Application Logic, Policy Enforcement Point Proxy 6, Query Available Clouds Service , Directory DB*]

- Purple – Querying Sensors in LINCA system.
  [*Application Logic, Policy Enforcement Point Proxy 1, Query Sensor Service , Directory DB*]

- Gray – Registering Sensors to its cloud system and in LINCA system.
  [*Application Logic, Register Sensor Service , Policy Enforcement Point Proxy 2 , Publish/Subscribe Service, Directory DB*]

- Dark Green – Registering cloud system in LINCA system.
  [*Application Logic, Register Cloud Service , Policy Enforcement Point Proxy 7 , Directory DB*]

- Yellow – Storing sensor data and measurements.
  [*Data Storage Service , Policy Enforcement Point Proxy 4 , Publish/Subscribe Service*]

- Orange – Retrieving history data from sensors.
  [*Data Storage Service , Policy Enforcement Point Proxy 3 , History Service*]

- **Black Dashed** – Communication between user and system through web interfaces.
  [*Web-Application , Application Logic*]

- Purple dashed / Orange Dashed / Gray Dashed / Pink Dashed / Dark Green Dashed – Checking user permissions in Query / History / Register service respectively.
  [*Policy Enforcement Point Proxy 1 / 2 / 3 , User Identification – Authorization , User DB , Authorization Policy Decision Point*]

➢ **Database Service**

This type of service consists of all the different types of database that exist in a single LINCA's cloud system. Each of these databases is connected to services in order to store the provided data. History DB is used to store sensor data from Sensor Data Storage Service and provide the stored data to the History Service. In addition, User DB is used to store the user's profile information which is provided from User Identification Service. Furthermore, Directory DB is used to maintain the sensors information that exist in the overall LINCA system but also it is used to store all the cloud systems' information that is registered in the LINCA system. The Publish/Subscribe Storage is used to store the sensor entities of its own cloud system. Below, a more detailed description about these databases is shown:

- **History DB** - This is a non-relational database that contains raw and aggregated historical data measurements for all the sensors connected in the system. Sensor Data Storage Service undertakes to receive data streams from the sensors' measurements. The data it receives is stored as raw and aggregated in History DB in order to maintain history of sensors' measurements. Also, the History Service provides REST methods for retrieving information from the History DB.

- **User DB** – This is a relational database used by User Identification and Authorization Service to store user's profile information. In more detail, when a user is sign up to an cloud system the information that provided such as name, surname, email, password, user category , are stored in User DB. With this information User identification and Authorization Service is able to identify a user that tries to log in its cloud system.

- **Directory DB** – Directory DB is a non-relational distributed database. Each registered cloud system in LINCA consists a node of this distributed database in order to make insert and query request. It is used by Register Sensor Service to store sensor's information that they are connected to its cloud system. Also, is used by Register Cloud Service to store information about its clouds systems. In this way , all information about sensors and clouds that exist in LINCA system is stored in this distributed database in order to be discoverable from authorized users that are connected to a remote LINCA's cloud system. In addition, Query Sensor Service and Query Available Clouds Service provide REST methods for

retrieving information sensors and clouds of LINCA that are maintained in Directory DB. Furthermore, this database is used to store a list of the user's subscribed sensors, as a result when infrastructure owner delete a sensor from its cloud and it is existing to user's subscribed sensor list, to be removed.

- ***Publish/Subscribe Storage*** – This is a non-relational database that uses the Publish and Subscribe Service to store the information of the entities it manages. Each entity that is managed by this service is represented by one JSON object. This object follows the syntax rules set by NGSI standard as described in Section 2.2.2 .

➢ **Security Service**

The purpose of the security service is to prohibit unauthorized users and services from using the system functions. In Figure 20, it is noted that each service has its own Policy Enforcement Point Proxy server. These proxy servers in collaboration with the User Identification-Authorization Service and the Authorization Policy Decision Point service are responsible for the system's security. The intercommunication of these services is analyzed below:

- ***User Identification-Authorization Service***

When registering with the system, the user defines the features that make up their personal profile such as name, email, password.

Therefore, the user in order to log in to the system, they must first enter their email and password at the login page. Then, the login request is routed to the User Identification-Authorization Service via its RESTful interface. As long as the information provided is valid, the service creates an OAuth2 token that encrypts the user's identity. Afterwards, the user is logged in. The User Identification-Authorization Service is then able to confirm who the user is and what their roles-permissions are. This is achieved through a REST request that contains the user's OAuth2 token. All the user's profile information is stored in the User DB as described in Section 3.6.2.

- ***Authorization Policy Decision Point Service (Authorization PDP)***

This service is responsible for approving or rejecting access requests made by the users to the cloud system services. A decision, either permit or deny, can be made based on the Access Control Rules (XACML), which results from the applicable access policies that the roles within the User Identification -Authorization Service are recommended. A Policy Rule is created by the user's role that exists within the User Identification -Authorization service. The access control rules are created in the Authorization Policy Decision Point Service and they follow the XACML standards as explained in Section 2.2.2. This is achieved, through the RESTful communication with the User Identification-Authorization Service.

- ***Policy Enforcement Point Proxy Server (PEP Proxy Server)***

A PEP Proxy server is a server that acts as an intermediary for the user's requests and the resources that the user wants to access with it. Each service that has resources that are unlikely to be accessible by unauthorized services or users, has a local PEP Proxy server that undertakes to receive and to forward request to it.

The PEP Proxy Server demands the received HTTP request header to contain one of the following tokens:

- **OAuth2 token** - A valid OAuth2 token was created by the User Authentication and Authorization service upon logging in and corresponds to a user.

- **Master Key** - A secret code that is specified when initializing the Policy Enforcement Proxy Server. Each different Policy Enforcement Point Server has its own unique master code.

If the above tokens are not contained in the HTTP request header, then the request is rejected.

As it is presented in Figure 20, any services that are not eligible to publish its REST interface, such as the Publish/Subscribe Service, Sensor Data Storage Service, History Service, Query Sensor Service, Query Available Clouds Service, Sensor Interface Service, Register Sensor Service, Register Cloud Service, work with a PEP Proxy Server.

In total, seven different PEP Proxy Servers are used in the architecture as shown on Figure 20. Each one runs on its own docker container. By using the OAuth2 mechanism as described in Section 2.2.2, the user's authorization can be dynamically configured depending on its user's category as explained in Section 3.1. According to the above, the user's access requests must bear in their header the OAuth2 token, that they received upon logging in.

The PEP Proxy Servers 1, 2,3, 6 and 7, as shown in Figure 20, because they act as an intermediary for the user's request and the services that they provide protection, they need the user's unique OAuth2 token in the header of their HTTP request in order to authorize and authenticate the user through their collaboration with User Identification - Authorization Service and Authorization Decision Point.

On PEP Proxy Servers 4 and 5, as shown in Figure 20, because they act as intermediary for their protected services and the services that they want gain access, the service requests header contains the Master key code of the PEP Proxy Server. Hence, only the corresponding PEP Proxy is responsible for the security of their protected services because there is no reason to identify or configure authorization of different ranks of users.

In conclusion, when a user is signed up to a LINCA's cloud system, their profile information and permissions are stored and maintained in the Identification-Authorization Service and the Authorization PDP Service,

respectively. When the user routes a login request to the cloud system that they signed up before, Identification-Authorization Service checks if the requested user belongs to its cloud system. If the identification is successful, an OAuth2 token, which defines the identity of this user in the cloud system, is returned in order to make an access request to the services of the cloud system without exposing their personal information. Then, when the logged in user routes an access request to an cloud's service they must provide in the request's header their OAuth2 token that they received when they logged in. The PEP proxy that "protects" the service which the user requested access to, must receive first the user's request in order to check who the user is and what permission it has, with the help of the Identification-Authorization Service and the Authorization PDP Service. More specifically, the PEP proxy extracts the OAuth2 token from the user's request and asks the Identification-Authorization Service who the user is and what their roles are. After the Identification-Authorization Service identifies the user, it returns to the PEP proxy the user's profile information and roles. Next, the PEP Proxy based on the information that it received from the Identification-Authorization Service, asks the Authorization PDP if this user has the permission to access the service that the PEP proxy protects. If the Authorization PDP returns "Permit" then the PEP Proxy forwards the user's initial request to its protected service. Otherwise, if the Authorization PDP return is "Denied", the PEP proxy will not forward the request to its protected service. The above process is represented as workflow in Section 3.5.1 and as an abstract view in Figure 21 below:

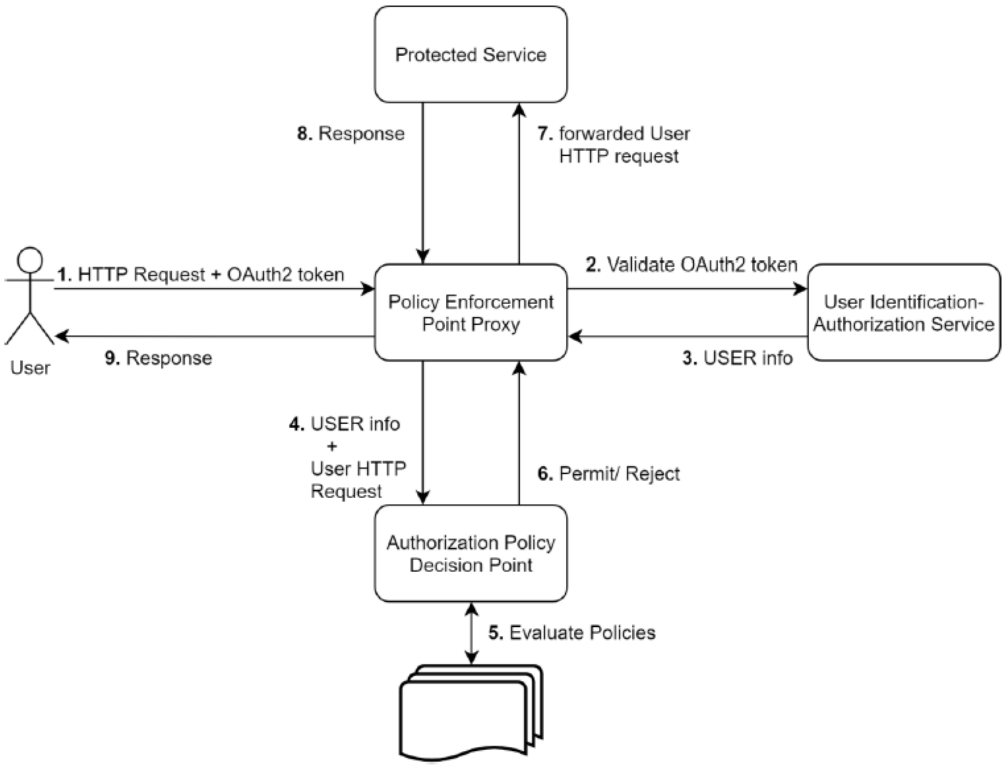


Figure 21 – PEP Proxy Server Function using OAuth2 token

If a user log in to a LINCA's cloud system and wants to access a service that is located in a remote cloud system of LINCA, it follows procedure that it will be describe in Section 4.2.

- **Publish/Subscribe Service**

The Publish/Subscribe Service acts as a mediator for the "Sensor" entities. These entities are stored in the Publish-Subscribe Storage database in form of JSON as described in Section 2.1 .

The JSON representation of the "Sensor" entity contains information about its features:

- Unique sensor identifier (consisting of its name and unique code).
- Device's model.
- Sensor's owner.
- Name of the city in which it is located.
- Measuring characteristics (e.g. Temperature, humidity, pressure).
- Unit of measurement (e.g. Celsius, Pa, LUX).

- **Sensor Data Management Service**

This service is responsible for storing data to Database Service that are received from Publish/Subscribe Service. Also, they can retrieve history data from Database Service too. It consists of two micro-services as follows:

- ***Sensor Data Storage Service***

This service is responsible for collecting data that are derived from the sensor measurements. These measurements are administered by the Publish/Subscribe Service. The aim of this service is to maintain a history record of the measurements of each sensor in the History DB. By using the subscription feature of the Publish/Subscribe Service , the Sensor Data Storage Service is subscribed to all its sensor entities. Each new sensor measurement that results, triggers one update event from Publish Subscribe Service to the endpoint of the Data Storage Service. As a result, the service receives data from all measurements of each particular sensor. The service is able to store the received data to a wide range of different types of databases like MongoDB, MySQL etc. The data received by this service through the subscription feature, is stored in History DB with two different tactics :

- Raw – The data is stored as raw in the measurement's history of each particular sensor.

- Aggregated – This data is statistical values deriving from the combination of the new arrival data , with the existing historical sensor data. The statistical values are :

- Maximum value between all samples of a sensor measurements in the last day/month/year.
- Minimum value between all samples of a sensor measurements in the last day/month/year.
- The sum of all samples for a sensor's measurements in the last day/month/year.

- ***Sensor Data Storage Service***

This service is the RESTful interface of the system's historical database. It is connected to the History DB and provides REST methods for retrieving raw and aggregated historical data of the system's sensor measurements.

➢ **IoT Service**

This is the service where the IoT devices are connected to and they send their measurements through gateways. Furthermore, this service is responsible for registering that the IoT devices to its cloud system and querying for the IoT devices that are in every LINCA's cloud system. This type of service consists of the following micro-services :

- ***Query Sensor Service***

The purpose of this service is to provide the users a search engine to find sensors that are connected in their subscribed cloud systems. This service is responsible for translating a user's demand and searching for the requested sensors in the Directory DB. If the user searches for sensors that are only connected to their cloud system, the identification and authorization process that is described above is done before the search in the Directory DB. But, if the user searches for sensors that are located in other LINCA's cloud systems the identification and authorization of the user follows the process that will be explained in Section 4.2.

- **Register Sensor Service**

This service is connected to the Directory DB and it provides the REST methods for inserting sensors. It stores the sensors' information in the Directory DB in order to be discoverable by the local users and the authorized remote users. Also, it provides the REST methods for inserting sensors to the Publish/Subscribe Service in order to forward its sensors' data to the Sensor Data Storage Service, which are received from Sensors Interface Service.

- **Sensor Interface Service**

The purpose of this service is to be able to support different types of sensors that transmit data through the UltraLight 2.0 protocol. Data is obtained from the gateways that are connected to the system's Sensor Interface Service. The Sensor Interface Service receives the physical sensor's data from the

gateways and updates the corresponding sensor that is maintained in the Publish Subscribe Service. In this way, the sensor entities in the system are constantly updated with the current measurements of the physical sensors.

- **Cloud Management Service**

This service is used by the administrators of cloud systems, if they wish to register their cloud system to LINCA. Also, it is used by the Customer users for querying for available LINCA's cloud systems that are registered to LINCA system by the corresponding administrators. This type of service consists of the following micro-services:

- **Query Cloud Service**

The purpose of this service is to present to the users the available remote clouds that are registered in LINCA. After the identification and authorization process that is described above is done, this service is responsible to search for the LINCA's clouds in Directory DB. Afterwards, the users can choose the remote clouds systems that they are interested in and request subscription from the corresponding cloud administrator. If a remote administrator accepts the user's subscription request then this administrator creates and stores to their cloud system, the roles and permission of this user. In this way, each cloud's administrator can manage the roles and permissions of the remote users that are subscribed to their cloud system.

- **Register Cloud Service**

This service is connected to the Directory DB and it provides the REST methods for inserting the cloud's information to it . This service stores its cloud's information in the Directory DB in order to be discoverable by the authorized remote users that wish to subscribe to their cloud system. If the administrator accepts the user's subscription request that they received from a remote LINCA's cloud system, then it creates and stores to their cloud system the roles and permissions for that remote user. In this way, the administrator of this cloud can manage the roles and permissions of the remote users that are subscribed to their cloud system.

- **Application Logic**

Application Logic includes the code for orchestrating individual services, so that the cloud system can implement the specified functionality.

- **User Service**

This service consists the Web Application of the cloud system. It is considered part of the Application Logic as it contains the code needed to implement the graphical interfaces of the cloud system.

4 LINCA System Implementation

LINCA is a unified master-less distributed system consisting of different cloud systems. Each LINCA's cloud system is deployed as a Virtual Machine (VM) which are provided from IntelliCloud that is based on OpenStack. The individual services of each cloud system except from Directory DB are deployed as docker container. Directory DB was deployed as individual service in every cloud system. Also, the deployment of these services is based on the following technologies:

PHP - PHP is a programming language for creating dynamic web pages. A PHP page is processed by a compatible Web server, such as the Apache Server, in order to produce the final content in real-time, which will either be send to the user's browser in an HTML format or transmitted to another PHP script. The Apache servers' images are used to produced Apache servers' containers for the Back-End to handle most REST requests between different services. Therefore, the PHP and some of its extensions are used. More specifically:

- **cURL** - cURL is a PHP library that allows data transfers between services using various protocols, such as DICT, FTP, FTPS, HTTP, HTTPS etc.. The library is used in order to call the HTTP protocol methods, such as POST, GET, DELETE, PATCH, PUT, directly through the PHP code.

- **CassandraDB PHP Library** – This library has the role of a driver to manage the distributed Apache Cassandra Database through a PHP code. The API provided by this library enables basic Cassandra functions, such as command queries, writes, updates etc. The CassandraDB PHP library is used in order to insert and search the LINCA's sensors and the cloud systems, where they are stored the Directory DB. Also, this library is used in order to update the user's subscriptions list that is stored in Directory DB as well.

The section 4.1 below, is a presentation of the mapping of LINCA's cloud systems services, as shown in Figure 22, to services of FIWARE's catalogue. Also, in Section 4.2 the interaction between LINCA's clouds systems is presented.

4.1 Implementation of LINCA's Cloud System Services

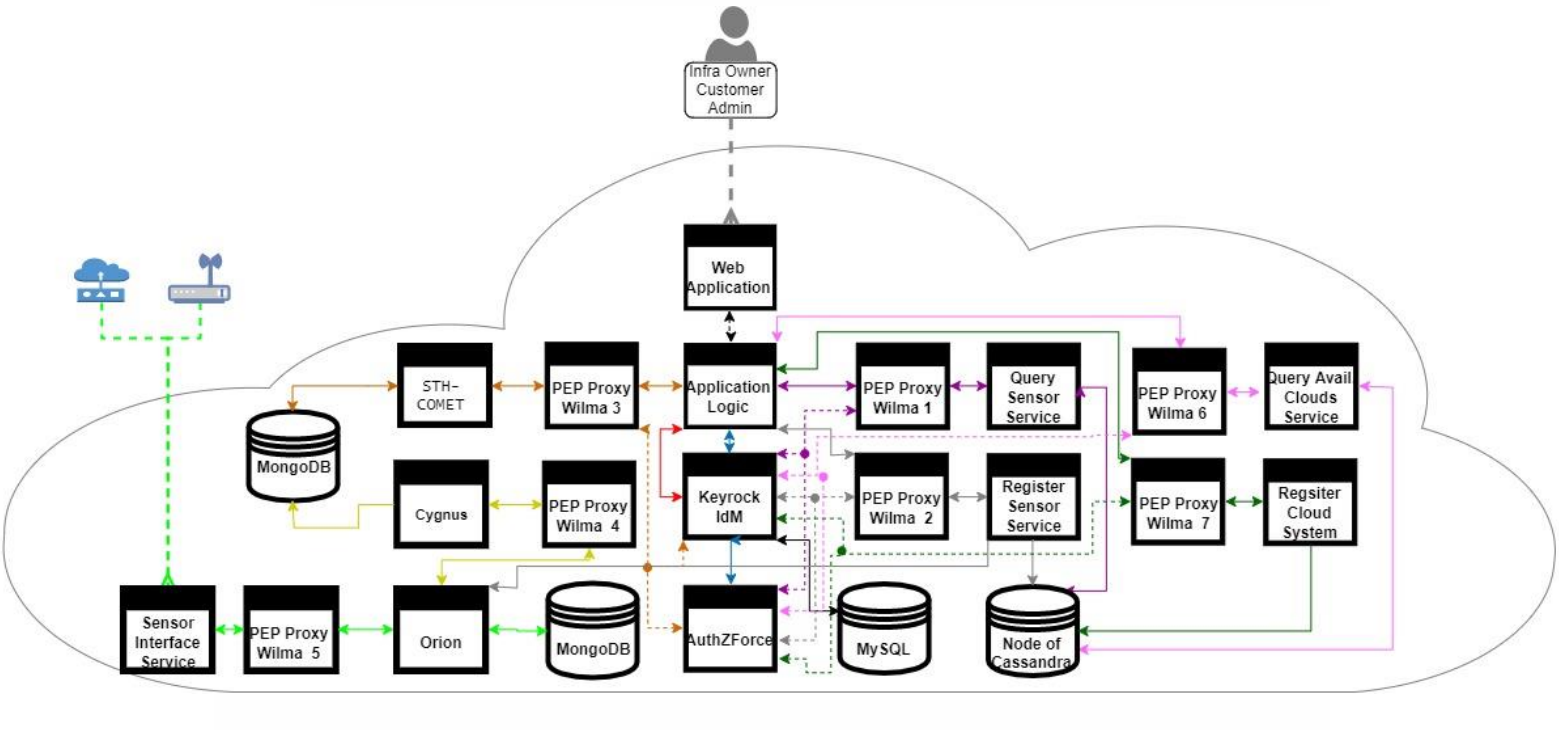


Figure 22 – Mapping LINCA's cloud system services with FIWARE's Services

LINCA's individual cloud systems are developed on Intellicloud virtual machines (VMs) that use the OpenStack platform. Each cloud system (VM) has a private and a public IP Address. The communication of individual LINCA cloud systems is done using the private IP address while the public ip address is used for the interaction of cloud systems with their users.

Within each cloud system there is a docker that hosts containers with the corresponding system services. The containers are connected to an internal network created by the docker and can communicate with each other via internal ip addresses. The docker uses two ports mechanisms to interact with its containers, the expose and the publish mechanism. The publish mechanism can assign a port to a container in order to receive requests from services outside the local docker container. The expose mechanism can assign one or more ports to a container to communicate only with containers that are connected to the local docker network.

Therefore, to access a cloud service, a user must use the public ip address of the cloud in combination with the publish port assigned to that service.

Communication between a system service (container) with a service (container) of another system is carried out using the private ip addresses of the individual systems along with the publish ports of the corresponding system services.

The docker which hosts system services in the form of containers can be installed on any computer. DockerHub, which is a publish repository of images, offers images of FIWARE services to develop into docker containers. In conclusion, LINCA implementation can be deployed on a group of computers that can communicate with ip addresses under the same network and have the docker installed. The mapping of cloud system services with FIWARE services is as follows:

- **User Identification and Authorization Service**

This service is a docker container running Keyrock IDM image[34]. Keyrock IDM is provided from the FIWARE service and uses the OAuth2 authorization protocol to authenticate users and provide authorization for access to services of LINCA's cloud system. The authentication and authorization of users who wishes to access services of a remote cloud system is described in section 4.2.

- Registration of the cloud system in Keyrock IDM

The system administrator registers the cloud system in the Keyrock IDM service. The new system registration is done with the assistance of the graphical environment provided by Keyrock IDM. When the cloud system is registered, the Keyrock IDM creates two unique identifiers related to this system. These two identifiers are named, client_id and client_secret. After joining the above two identifiers with the ":" symbol between them (i.e. client_id:client_secret) and encrypting them with the base64 method (i.e. base64 (client_id: client_sercret)) a new identifier will be created, and it will be called "Authorization_Basic". For Keyrock IDM, this identifier is the identity of the cloud system. A connection request to the system must necessarily include the "Authorization_Basic" in its header.

- Registration of a new user in the registered cloud system

The user that is interested in accessing the cloud system should first create an account with the Keyrock IDM service on that cloud. To do so, they must complete a registration form where they must enter their details as well as the user's group which they want to join. Once this form is completed correctly, the local administrator will either accept or reject the user's registration request. If it is accepted, the Admin will rank them in the user category they have chosen. The user with "Customer" category is assigned the role of "Customer" while the user in the "Infrastructure Owner" category is assigned the role of "Infrastructure Owner". The authentication process begins when the user fills in the login details (email and password) and requests access to the cloud system. In more detail, a REST request is executed from the local cloud Application Logic endpoint to the local cloud Keyrock IDM service. The header of this request includes the "Authorization_Basic" wich is produced by Keyrock IDM during the registration. The body contains the user's login information. Upon successful authentication, the local cloud Keyrock IDM will return an OAuth2

token to the endpoint of the local cloud Application Logic and the user will be able to access the cloud system. A session is then created on the user's server that stores the user's OAuth2 token (i.e. $ _SESSION [OAuth2_token]). This is where the first stage of user authentication ends. Next, the cloud application logic generates a second REST request to the local Keyrock IDM that contains the user's OAuth2 token. Local Keyrock returns to the local Application logic, the user roles in the cloud system. Once this is done, if the user's role is "Customer" , a $ _SESSION ["User_category"] with the value "Customer" will be created if the user's role is "Infrastructure Owner" , a $ _SESSION ["User_category"] with the value "Infrastructure Owner" will be created. This ensures that the graphical interfaces for the "Customer" category are accessible only to users with an active session ($ _SESSION ["User_category"]) with the "Customer" value. The same applies to graphical interfaces for "Infrastructure Owners" . If a user attempts to access a graphical interface that does not correspond to their user category, then the system routes them to the original graphical interface where they must enter their login details again.

- User authorization on registered cloud system

Users' access policies to resources are developed by the local Keyrock IDM in the form of roles-permissions. More specifically, an in-service permission defines the right of its owner to execute a specific REST request in that service.

  - Example: The "Customer_Access" permission specifies the HTTP request with GET method at URL: http://localhost/CustomerPortal.

A role holds some permissions. The user assigned to this role also receives the corresponding permissions.

  - Example: The "OrdinaryCustomer" role includes the "Customer_Access" permission. So, the owner of the "OrdinaryCustomer" role has the right to make an HTTP request with GET method at URL: http://localhost /CustomerPortal

The user with the "Customer" role has the following permissions :

  - Use of the Query Available Clouds Service to search available clouds. (*POST request http://localhost/getAvailableClouds.php*)
  - Use of the Query Sensors Service to search sensors. (POST request *http://localhost/getDesiredIDs.php)*
  - Use of the Query Sensors Service to search the list with user's subscribed sensors. (POST request *http://localhost/getMySensors.php)*
  - Use of the Query Sensor Service to subscribe to a sensor. (POST request *http://localhost/subcreate.php) .* Also, an HTTP request (POST request *v2/Entities)* is routed to Publish/Subscribe in order to receive updates of the subscribed sensor's measurements.
  - Use of the History Service to view statistic values of a subscribed sensor. (POST request *http://localhost/getHistoryValue.php).*

- Use of the History Service to view current value of a subscribed sensor. (POST request *http://localhost/getCurrentValue.php).*

The user with the "Infrastructure Owner" role has the following permissions :

- Use of the Sensor Interface Service to register a sensor's drivers to the local cloud system. (POST request *http://localhost/AgentParser.php) .*
- Use of the Register Sensor Service to register a sensor to Directory DB (Cassandra DB), in order to be discoverable from authorized users from other clouds. (POST request *http://localhost/RegisterSensor.php).*
- Use of the Query Sensors Service to search the list with user's registered sensors. (POST request *http://localhost/getInfraSensors.php).*
- Use of the History Service to view statistic values of a sensor. (POST request *http://localhost/getHustoryValue.php).*

- Use of the History Service to view current value of a sensor. (POST request *http://localhost/getCurrentValue.php).*

The HTTP methods of the Keyrock IDM service discussed in this section are described in the REST Table 1 :

Method	URL Method	Request Header	Request Body	Method Descr.
POST	/auth2/token	Authorization: base64 (client_id: client_secret)	{ &username="username" &password= "password" }	A valid username and password must had given. Returned an OAuth2token.
POST	/v1/auth/tokens		{ "name": "admin_username" "password":"admin_password" }	Admin information is given . Returned an access identifier:*X_subj_token* .
POST	/v1/applications/ client_id/ users/user_id/ roles/role_id	X-Auth-token:X_subj_token		This method corresponds “role_id” to user with id “user_id”.
GET	/user?access_token= {OAuth2 token}	Authorization: base64 (client_id: client_secret)		Receives an OAuth2 token in its URL. If it is valid , user’s information is returned (User id, User Roles, Permissions …)

REST table 1 – Keyrock IDM

- **Authorization Policy Decision Point (Authorization PDP)**

This service is a docker container running AuthZForce image[35]. This service is provided by the FIWARE catalogue and it is the Authorization Policy Decision Point (PDP) service of the local cloud system. The aim of this service is to make decisions, permit or deny, about the user’s access requests. The decision it makes, is based on access rules that are stored in the service. This access rules follow the XACML standard.

A role that is registered in local cloud Keyrock IDM service is associated with one of the stored AuthZForce access rules. An access rule describes how the user request must be standardized for approval. AuthZForce has the following RESTful interfaces :

[35] https://hub.docker.com/r/authzforce/server

- Creating a new role in the local cloud Keyrock IDM service, automatically triggers a REST request from the local Keyrock IDM to the AuthZForce Service. This service will create a new access rule that is associated with the new role from the Keyrock.

- The local cloud Policy Enforcement Policy (PEP) Proxy Servers forward to AuthZForce the user's access request, in order to be evaluated for approval or rejection.

- **Policy Enforcement Point Proxy Server (PEP Proxy Server)**

This service is a docker container running Wilma image[36]. Wilma PEP proxy provided by the FIWARE catalogue and is implemented in order to corporate with the local Keyrock IDM and the local AuthZForce PDP. This collaboration provides protection to others local services of the cloud system from local users. The purpose of any PEP Proxy Wilma is to "protect" their respective services from unauthorized users and services:

- Publish/Subscribe Service
- Sensor Data Storage Service
- History Service
- Register Sensor Service
- Register Cloud Service
- Query Sensors Service
- Query Available Clouds Service

The PEP Proxy Servers 1, 2, 3, 6 and 7, as shown in Figure 20, because they act as an intermediary for the user's request and services that they provide protection, need the user's unique OAuth2 token in the header of its HTTP request in order to authorize and authenticate user through its collaboration with User Identification-Authorization Service and Authorization Decision Point as shown in Figure 23.

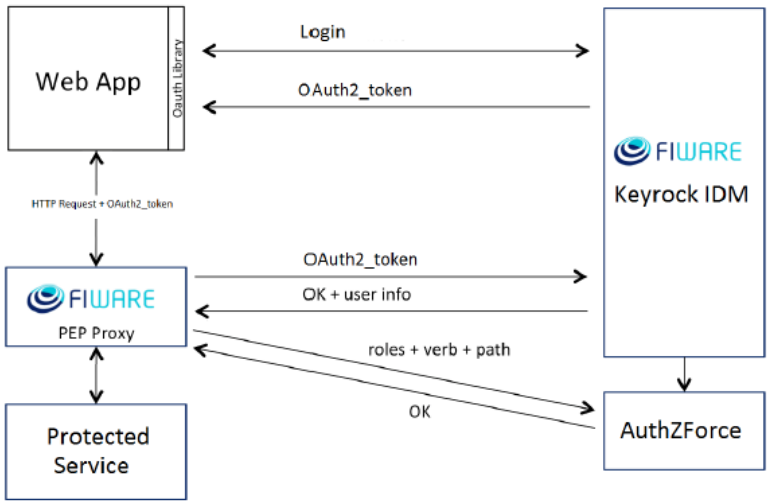


Figure 23 – PEP Proxy Wilma using OAuth2 token

[36] https://hub.docker.com/r/fiware/pep-proxy/

On PEP Proxy Servers 4 and 5, as shown in Figure 20, because they act as intermediary for their protected services and services that want access to them, the service requests header contains the Master key code of the PEP Proxy Server, as shown in Figure 24

Figure 24 – PEP Proxy Wilma using Master Key

The authentication and authorization of users who wishes to access services that are located to a remote LINCA's cloud system, it will be describe in section 4.2.

- **Publish/Subscribe Service**

This service is a docker container running Orion Context Broker image[37]. This service is provided by the FIWARE catalogue. It operates according to the NGSI-2 model data model for managing context information through its RESTful interface. The service functions are described as follow:

- ***Create/Update Entities***

The task is to design the Orion Context Broker to maintain the NGSI entities that describe:

A. Every different sensor in local cloud

When a sensor is registered at the local end, an HTTP request is routed to create a new NGSI "sensor" entity in Orion Broker. This request is implemented using the POST method. The body of this request includes the following as shown in Figure 24.

[37] https://hub.docker.com/r/fiware/orion/

```
{
    "type": "Sensor",
    "isPattern": "false",
    "id": "urn:ngsi-ld:tl:beacon:001",
    "attributes": [
        {
            "name": "Temperature",
            "type": "Integer",
            "value": " "
        },
        {
            "name": "TemperatureUOM",
            "type": "Celsius",
            "value": " "
        },
        {
            "name": "AmbientLight",
            "type": "Integer",
            "value": " "
        },
        {
            "name": "AmbientLightUOM",
            "type": "LUX",
            "value": " "
        },
        {
            "name": "city",
            "type": "string",
            "value": "Thessaloniki"
        },
        {
            "name": "Infra",
            "type": "Ownership",
            "value": "urn:ngsi-ld::001"
        },
        {
            "name": "TimeInstant",
            "type": "ISO8601",
            "value": " "
        }
    ]
        }
    ],
    "updateAction": "APPEND"
},
```


Figure 24 – Create/Update request Orion Context Broker

In detail, the features included in the Figure 24 are as follows:

- **Id** – Unique Identifier of the sensor.
- **Type** – Refers to the type of the NGSI entity. The value “Sensor” indicates that is a sensor entity.
- **Attributes** – Attributes for a particular sensor.
  - Name – name of the attribute of the sensor.
  - Type – type of sensor’s attribute.
  - Value – value of senor’s attribute.
  - updateAction – Contains the value “Append”. This means that the above request when is made for the first-time acts as creation request. This request creates an entity as it is described in its body. In the case where the entity already

exists, the request acts as a update request (changing the existing entity).

B. Every different sensor in foreign cloud

When a user subscribes to a sensor that is located at a foreign cloud, the local Orion Context Broker subscribe to the foreign Orion Context Broker to receive updates for these sensors. Thus, the foreign Orion Context Broker creates a sensor entity to the local Orion.

- ***Subscription Entities***

Through subscriptions to entities, Orion Context Broker triggers updates on any changes (the "ONCHANGE" condition) that occur in an entity's attributes. The update is sent to a predefined - by the subscriber - URI via a REST request using POST method. The request body contains the change information which is described with the NGSI-2 information model. The entity subscription function is used by the local Data Storage Service. Having the role of subscriber, it receives notifications at its endpoint about changes in sensor measurements and stores them in the system's historical database.

```
{
"description": "Notify Cygnus of all context changes",
"subject": {
  "entities": [
    {
      "idPattern": "urn:ngsi-ld:tl:beacon:001"
    }
  ]
},
"notification": {
  "httpCustom": {
              "url": "http://147.27.50.197:5051/notify",
               "headers": {"X-Auth-Token": "thismagickeyforcygnus"}
  },

  "attrsFormat": "legacy"
},
"throttling": 5
}'
```

Figure 25 – Subscribe request to Orion Context Broker

In detail the features included in the Figure 25 are as follows:

- **Entities** – The entities we subscribe to.
- **IdPattern** – This is the only sensor ID we want to create a subscription for.
- **Notifications** – This is a feature that contains the information about the updates it receives.

- httpCustom : This is an http request that we modify as we want.
- url – Indicates the final destination of the updates it receives. In this case it's the PEP Proxy of the Data Storage Service.

- **Throttling** - This is a variable that specifies the frequency at which updates will be sent to the specified endpoint.

The following REST table 2 shows the REST methods that implement all the functions of the Orion Context Broker as discussed in this section:

Method	URL Method	Request Header	Request Body	Method Descr.
POST	/v2 /Entities	Fiware-ServicePath: /Sensors	Diagram 2.1.3.2.1	Creation of «Sensor» entity.
GET	/v2 /Entities /{entity id}	FiwareServicePath: /Sensors		Recovery of entity with ID "entityid" which is located in «Sensor» entities.
DELETE	/v2 /Entities /{entity id}	FiwareServicePath: /Sensors		Deletion of entity with ID "entityid" which is located in «Sensor» entities.

REST table 2 – Orion Context Broker

- **Sensor Data Storage Service**

This service is a docker container running Cygnus image[38]. The Cygnus service is based on the "Apache flume40" architecture and provides various agents (Agents) responsible for collecting NGSI data streams and storing them in a predefined (external) database. An agent consists of a listener who is responsible for receiving the data, a "channel" where the listener forwards the data, and a "sink" that "receives" the data from the channel in order to store them in an external database . The components of Cygnus is shown in Figure 26.

[38] https://hub.docker.com/r/fiware/cygnus-ngsi/

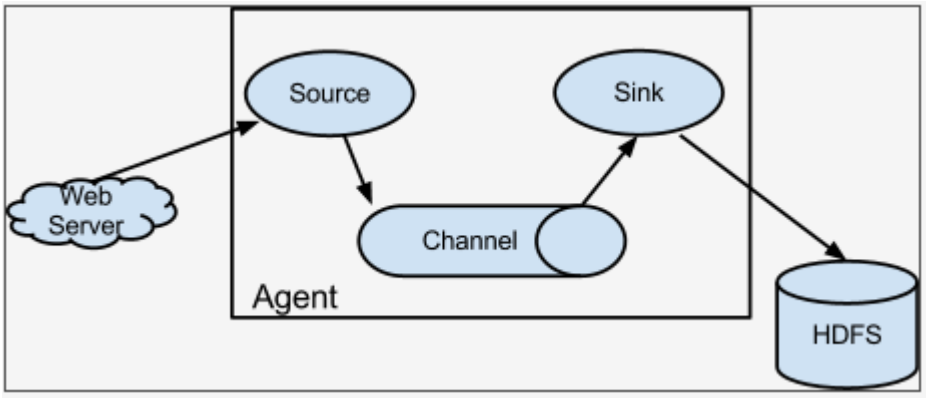


Figure 26 – Cygnus

Cygnus provides various specialized agents to collect and maintain NGSI data in the following database - repository services:

- HDFS Hadoop - file sharing system.
- MySQL - relational database.
- CKAN - Open Data Platform.
- MongoDB - NoSQL database for documents.
- Kafka - Subscription Message Broker.
- DynamoDB - Cloud-based NoSQL database from Amazon Web Services.
- CarTo - Database specializing in geographic data.

For example, in the scenario where we need to store NGSI feeds in a DynamoDB database, we will need to use a specialized "DynamoDB agent" of the Cygnus service. Similar to the MongoDB database storage scenario, we will need a specialized "MongoDB agent". In this work we have put in place a specialized agent (Cygnus) for storing raw and aggregated data in the historical database of the MongoDB system.

By using the subscription function of the Orion Context Broker service, the agent subscribes to all Orion sensor entities. So, with every change that happens to the value of a sensor attribute, an update is triggered from Orion to the endpoint of the agent subscriber. The agent thus receives all measurements of the system's sensors each time they occur.

It then has the responsibility of storing the measurements as raw and aggregated data in the historical database. In this way the agent maintains a time series data for the measurements of each different sensor.

In order to store aggregated information :

For each different sensor, different variables are maintained in the database that relate to:

- The maximum value of the sensor measurements for the last month / day / hour.
- The minimum value of the sensor measurements for the last month / day / hour.

- The sum of the sensor measurements for the last month / day / hour.

Example :

Suppose that the maximum temperature of the sensor with "urn: ngsi-ld: tl: beacon: 001" identifier for this month is kept in the variable "MAX_temperature_ urn: ngsi-ld: tl: beacon: 001_September" within the history DB and has the value "34". The agent receives an alert with the new " urn: ngsi-ld: tl: beacon: 001" sensor temperature measurement set to "36". The agent will update the variable for the maximum sensor temperature for this month from "34" to "36".

With this tactic, a request to retrieve the maximum / minimum / average measurement of a sensor for any month / day / hour can be executed instantly once there is a Pro-aggregation. Otherwise, we would have to recover a fairly large number of raw metrics between the time frame we are interested in (maybe thousands of metrics) and export - at that time - with some MAX / MIN / AVERAGE algorithm to the desired value (Much more time consuming).

At this point, as the operation of a Cygnus agent has become more understandable, it is worthwhile to note one more positive thing that it offers on an architectural level. As explained at the beginning of the section the agent consists of a listener, a channel and a "sink". The channel acts as a temporary repository of data received by the listener (the size of the channel memory is set when the agent is initialized), Sink undertakes to "retrieve" the temporary data in the channel and store it in the external database. This way, a failed record (eg network delay, database overload, etc.) can be repeated without losing data.

The REST method provided by the service for receiving and storing NGSI feeds is described in the following REST table 3.

Method	URL Method	Request Header	Request Body	Method Descr.
POST	/notify			Endpoint of the subscriber of this service. Subscribed data streams are sent for storage.

REST table 3 - Cygnus

- **History Service**

This service is a docker container running Comet image[39]. FIWARE-COMET is a service provided by FIWARE. It is locally linked to the MongoDB (History DB).

With its RESTful interface, it retrieves raw and aggregated historical information which is stored in the MongoDB (HistoryDB).

This information has been stored on a historical basis by the Cygnus agent as we saw in the previous section. The REST API of the service is described in the following REST table 4:

Method	URL Method	Request Header	Request Body	Method Descr.
GET	/STH/v1/contextEntities/ type/ Sensor/ id/ {Sensor_id} /attributes/{temperature} &LastN=1			Request for retrieving the current value of temperature from a sensor with the ID “Sensor_Id”.
GET	/STH/v1/contextEntities/ type/ Sensor/ id/ {Sensor id} /attributes/{temperature}/ aggrMethod={max/min/avg} &aggrPeriod= {Hour} &dateFrom ={2019-09-01T00:00:00.000Z} &dateTo ={2019-09-2T23:59:59.999Z}			Request for retrieving the min/max/sum from historical data of a sensor with the ID “Sensor_Id”. The “aggrMethod” takes values like “sum”, “max”, “min”. The “aggrPeriod” takes values like “month”, “day”, “hour”.

REST table 4 – STH Comet

- **Sensor Interface Service**

This service is a docker container running IoT-Agent image[40]. This service is provided by FIWARE and it aims to provide:

- Provide a sensor insertion mechanism to the infrastructure owners according to their specifications so that customer can find the registered devices.
- Recovery and translation of data sent by the sensors and then forwarded to the Publish Subscribe service.

The sensor is inserted by using the graphical interface where it gives the infrastructure owner the choice to select the cloud they wish to insert the sensor

[39] https://hub.docker.com/r/fiware/sth-comet/

[40] https://hub.docker.com/r/fiware/iotagent-ul/

and then to complete the sensor characteristics. This allows the infrastructure owners to easily insert sensors at the clouds that are authorized.

- **Query Sensors Service**

This service is a docker container running Apache Server image[41]. The purpose of the service is to provide the users with a specified search engine so the users can find the devices they need.

The sensors can be selected by the user, firstly by choosing their cloud that they are located at, through the graphical interface. In addition, they can choose the type of sensors' measurements. In this way, the user can easily search for sensors in LINCA's clouds. Before the search starts, an authentication and authorization process is executed, in order to check if user is authorized at the LINCA's clouds that wants to search for their connected sensors. The service communicates with the Directory DB to retrieve sensors from the requested LINCA's clouds.

- **Query Available Clouds Service**

This service is a docker container running Apache Server image[42]. In this service a user can search for any available clouds that are registered in LINCA system. They can choose in which one they want to subscribe to. If the corresponding cloud admin accept the user's subscription request, then the user can query for the sensors in admin's cloud.

- **Register Sensor Service**

This service is a docker container running Apache Server image[43]. The purpose of this service is to provide users with a specified graphical interface so users can register their sensors to their connected LINCA's cloud system.

In the graphical interface, Infrastructure Owners can fill the information that are related to their sensors, such as name, id, owner details and type of measurements. Also, this service communicates with the Directory DB in order the infrastructure Owners to insert their sensors information to it. In this way authorized users can search for sensors that are connected in their authorized LINCA's cloud systems.

This service is also used to update the customer's subscription list in Directory DB. For example, when a customer creates a subscription to a sensor, a request is routed from Register Sensor Service to Directory DB to update the user's subscription list.

- **Register Cloud Service**

This service is a docker container running Apache Server image[44]. In this service, a user-admin can register his/her cloud's information in LINCA in order to be discoverable by others remote authorized users . An authorized

[41] [42] [43] [44] https://hub.docker.com/r/webdevops/php-apache

user means that it made a subscription request to this admin's cloud and this admin accept it and created for this user roles-permissions

4.2 LINCA's Cloud Systems Interaction

LINCA's is a master-less distributed system that is consisted of with identical and equal cloud systems. These clouds systems can interact with each other through RESTful communication in order to satisfy the demands of user categories that described in Section 3.1. More specifically, LINCA's cloud systems interact with each other when:

- A Customer user is searching for the available cloud systems that are registered in LINCA.
- A user wants to subscribe to remote cloud systems in order to access their services and devices.
- Authentication and Authorization process of user takes place, in order to get access to remote services
- A Customer user is searching for sensors that are connected to remote cloud systems.
- A customer is subscribed to sensors that are connected to remote cloud systems and their data bust be fetched by user's cloud system in order to provide the sensors' updates to the subscriber.

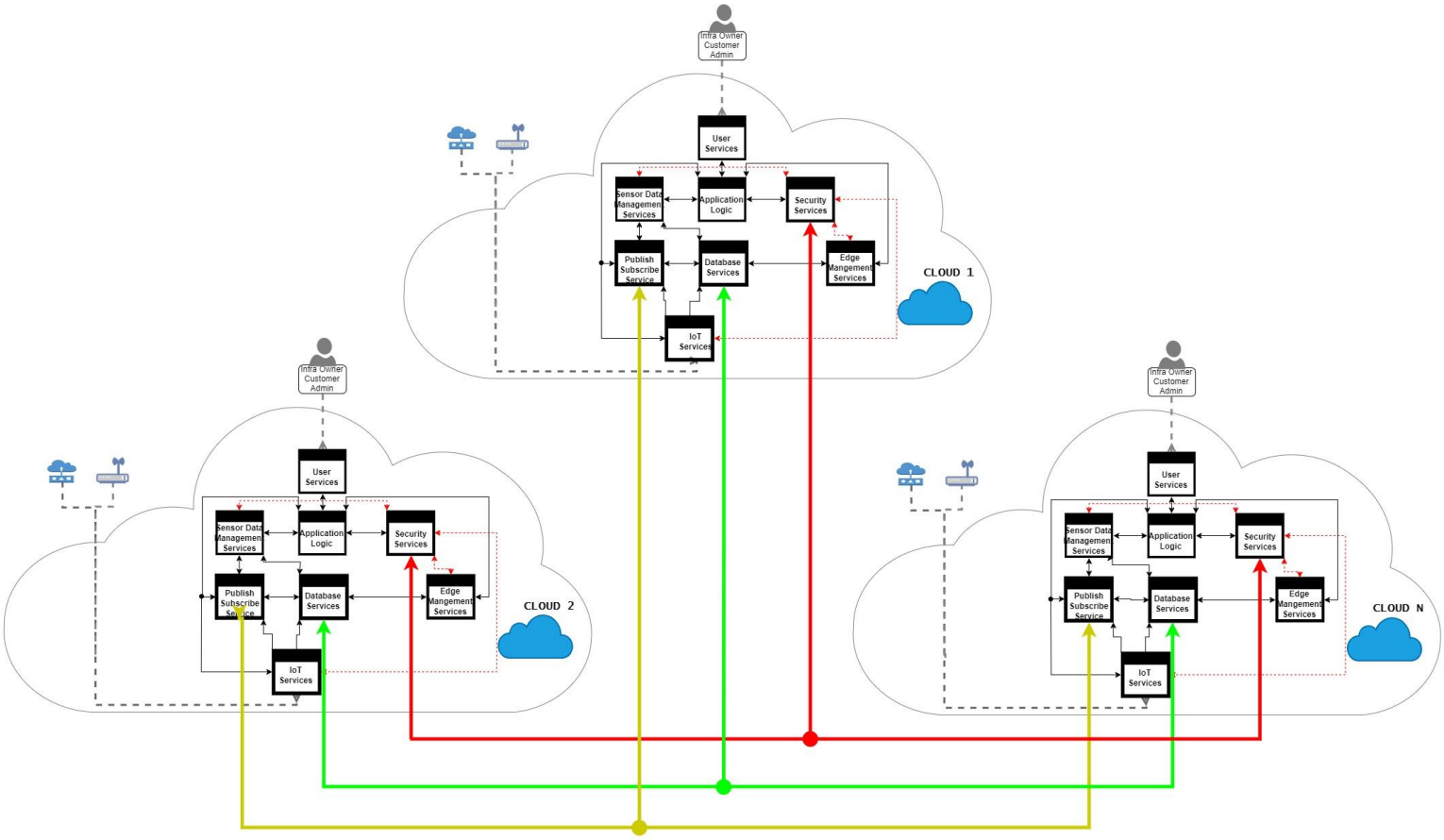


Figure 27 – LINCA's cloud systems interaction

The Figure 27 illustrates a view of the LINCA's clouds system interaction. Arrows represent the RESTful communication over HTTP protocol of:

- Services that are in the same cloud system.
  - Red Dashed Arrows : communication of security service with the other services as described in section 4.1.
  - **Black Arrows :** communication between the rest services of the cloud system as described in section 4.1.

- Services that are in different cloud system.
  - Red Arrows : communication between security services of each cloud system in order to register, authenticate and authorize user in a remote cloud system.
  - Green Arrows : communication between Database Services of each cloud system. More specifically , Directory DB of each cloud system is a node of Cassandra's ring. After the authorization and authentication process user can seek sensors that are connected to LINCA.
  - Yellow Arrows : communication between Publish/Subscribe Services of each cloud systems in order to retrieve updates of subscribed sensors' measurements.

The connection of services of the same cloud system is explained in section 4.1. In addition, we will describe the communication between remote services(services that are located in different LINCA's cloud systems).

- **Interaction of Security Services**

Each Security Service is consisted from 4 micro services as described in Section 3.6.2. These micro services are the User Identification and Authorization Service, Policy Enforcement Point Proxy Server (PEP Proxy), and Authorization Policy Decision Point (Authorization PDP).

As shown in Figure 13 ,when user subscribe to a remote cloud system of LINCA, his/her permissions for this cloud is stored as a XACML file in the AuthZForce Service of this remote cloud system.

In this way, the cloud system that the user is connected, and he/she routes requests to services of remote cloud system, is responsible for the user authentication and the remote cloud system that user wants to access its services , is responsible for user authorization because it maintains user's permissions in its Authorization PDP Service.

A security service communicate with other security service of different cloud system when a customer through his/her cloud system tries to query or

subscribe to sensors that are connected to a remote cloud system to LINCA. Also, this process performed when a customer wants to retrieve current and statistic value of a subscribed sensor that is connected to a remote cloud system of LINCA.

When a user is trying to log in to his/her cloud system the local Identification-Authorization Service checks if the requested user belongs to its cloud system. If the identification is successful, an OAuth2 token is returned, which defines the identity of this user in his/her cloud system.

When the logged in user wishes to query or subscribe sensors that is connected to a remote cloud system of LINCA, must provide in the request's header his/her OAuth2 token that received when he/she logged in. The local PEP proxy Query Sensor receives first the user's request in order to check to its cloud system who the user is and what permission it has to the remote cloud in wich the requested sensors are connected. Actually, the PEP proxy Query Sensor extracts the OAuth2 token from the user's request and asks the Identification-Authorization Service which is located to user's cloud system (the cloud system that is connected and can route request) who the user is. After the Identification-Authorization Service identifies the user, it returns to the local PEP proxy Query Sensor the user's profile information. Next, the local PEP Proxy Query Sensor based on the information that it received from the Identification-Authorization Service, asks the Authorization PDP Service that is located to the remote cloud, if this user has the permission to query or subscribe sensors of this cloud. If the Authorization PDP returns "Permit" then the local PEP Proxy Query Sensor forwards the user's initial request to the local Query Sensor Service. Otherwise, if the Authorization PDP return is "Denied", the PEP proxy will not forward the request. Then, the local Query Sensor Service is searching in the local node of the Cassandra (Directory DB) the sensors of the remote cloud. The above process is represented as workflow in Section 3.5 and as an abstract view in Figure 28 below:

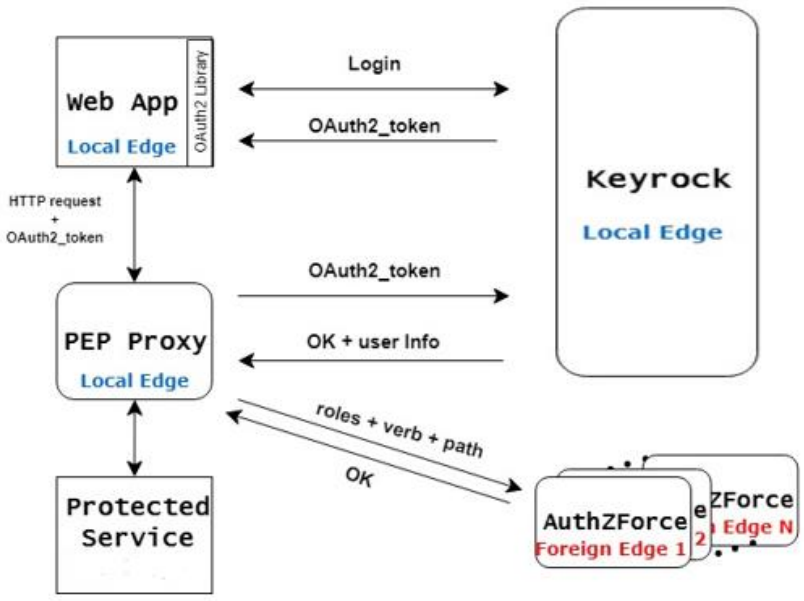


Figure 28 – PEP Proxy using OAuth2 between remote LINCA's cloud systems

- **Interaction of Public/Subscribe Services**

Each Publish/Subscribe Service consists the Orion Context Broker Service ,as described in Section 3.6.2 . The communication between Publish Subscribe Services that are located to different clouds is explained in the following Scenario :

- Scenario

  - An Infrastructure Owner is connected to CLOUD 1, as shown in Figure 27, and is authorized to register its sensors, "Sensor1" and "Sensor2" to its cloud system. As result, these sensors send data to Sensors Interface Service through gateways in ordered to forwarded to Publish/Subscribe Service. The two services that were mentioned above are located to CLOUD 1.
  - User A is connected to CLOUD 2 and he/she routes a subscription request to CLOUD 1.
  - We assume that the Admin of CLOUD 1 accepts the request and he/she assign roles and permission to User A. As a result, user A has the permission to query and subscribe to sensors that are connected to CLOUD 1.
  - User A through his local system (CLOUD 2) routes request in order to subscribe sensors "Sensor1" and "Sensor2" that are located to CLOUD 1.
  - After the authentication and authorization mechanism that explained above (Interaction of Security Services), user A is successfully

subscribed to these sensors. More specifically, Publish/Subscribe Service of CLOUD 1 is subscribed to Publish/Subscribe of CLOUD 2 for the “Sensor1” and Sensor2”.

When “Sensor1” or “Sensor2” send new measurements to their Publish/Subscribe Service (CLOUD 1) , then this service forwards the new measurements to the subscribers of these sensors. So, Publish/Subscribe Service of CLOUD 1 will forward the measurements to Publish/Subscribe Service of CLOUD 2. In addition, Publish/Subscribe Service of CLOUD 2 will forward these measurements to the Sensor Data Storage Service of CLOUD 2 in order to store them in History DB. This procedure is repeated for every new measurement from “Sensor1” and “Sensor2”.

- **Interaction of Database Services**

Each Database Service consists a Directory DB. This database is an implementation of Cassandra Database. Each Directory DB that is located to every cloud system of LINCA corresponds to a node of Cassandra’s ring. The communication in Cassandra ring is described in Section 2.5.1. Authorized users can query LINCA’s sensors through the local node of Cassandra Cluster. Unauthorized user cannot get access to the any node of Cassandra’s ring because the only way to get access to them is through Register Sensor Service, Register Cloud Service, Query Sensor Service ,Query Available Clouds Service which are protected by Security Services.

4.3 Docker and Virtual Machine Interaction

LINCA's individual cloud systems are developed on Intellicloud virtual machines (VMs) that use the OpenStack platform. The individual services of each cloud system are deployed as docker containers except from Directory DB (Cassandra) which is a deployed as a service in the Virtual Machine (VM), as shown in Figure 29. Each cloud system (VM) has a private and a public IP Address. The communication between cloud-cloud and cloud-user is done by using the public IP address.

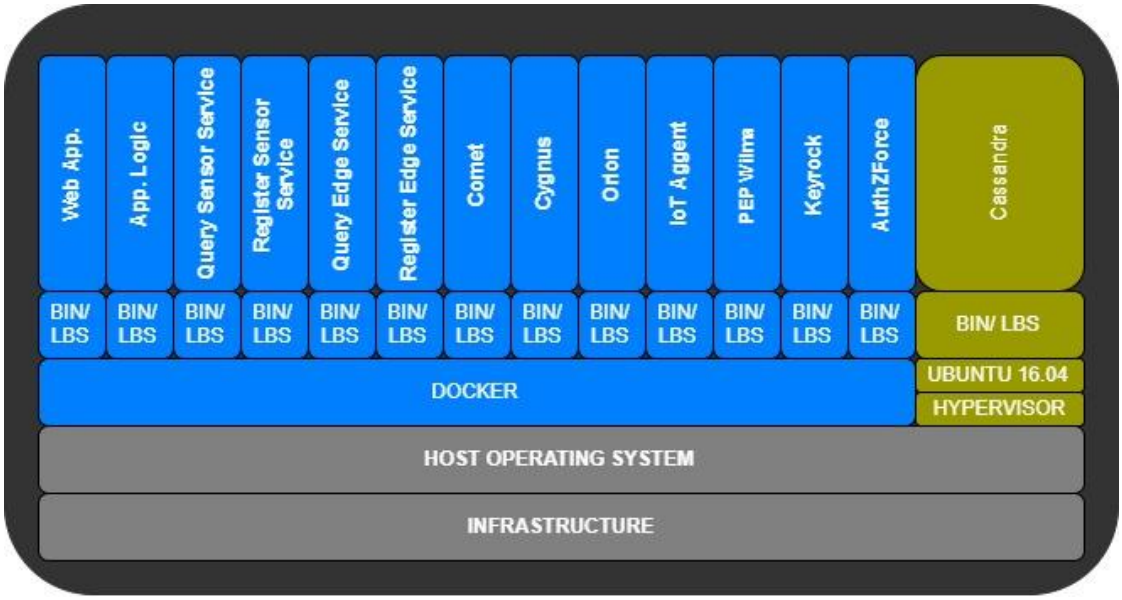


Figure 29 – Docker of LINCA's cloud system

The containers are connected to an internal network created by the docker and can communicate with each other via internal ip addresses. The docker uses two ports mechanisms to interact with its containers, the expose and the publish mechanism. The expose mechanism can assign one or more ports to a container to communicate only with containers that are connected to the internal docker network. The Figure 30 represents the communication of docker containers (cloud services) over the internal docker network. Containers of this docker network can communicate with each other via internal ip addresses (e.g. Container 1 internal ip address is 172.18.1.5) along the exposed ports (e.g. Container 1 exposed port is 8060). Also, a docker container (e.g. Container 2) can exposed one or more ports.

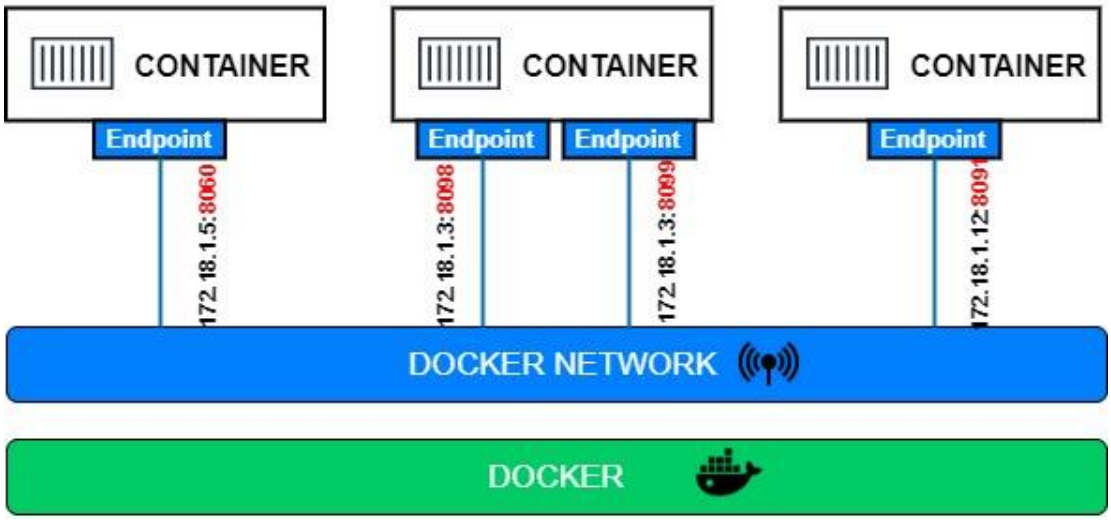


Figure 30 – Docker Network

The publish mechanism is a way of binding host machine port to a running docker container port. In other words, when a port of docker container is published, it is mapped to a specific port of the host machine. In this way docker containers can receive requests from services that are outside the docker internal network.

Therefore, a user in order to access a cloud service that is deployed as docker container, he/she must use the public ip address of the cloud system (VM) along with the published port of the docker container in which the service is running. The Figure 31 shows that the published port 8061 of Container 4 and is mapped to the host port 80. As a result, the Container 4 can receive requests from services outside the internal docker network. The requests can be sent through the public ip of the host machine along with the published port of Container 4 (e.g. 147.27.50.200:8061).

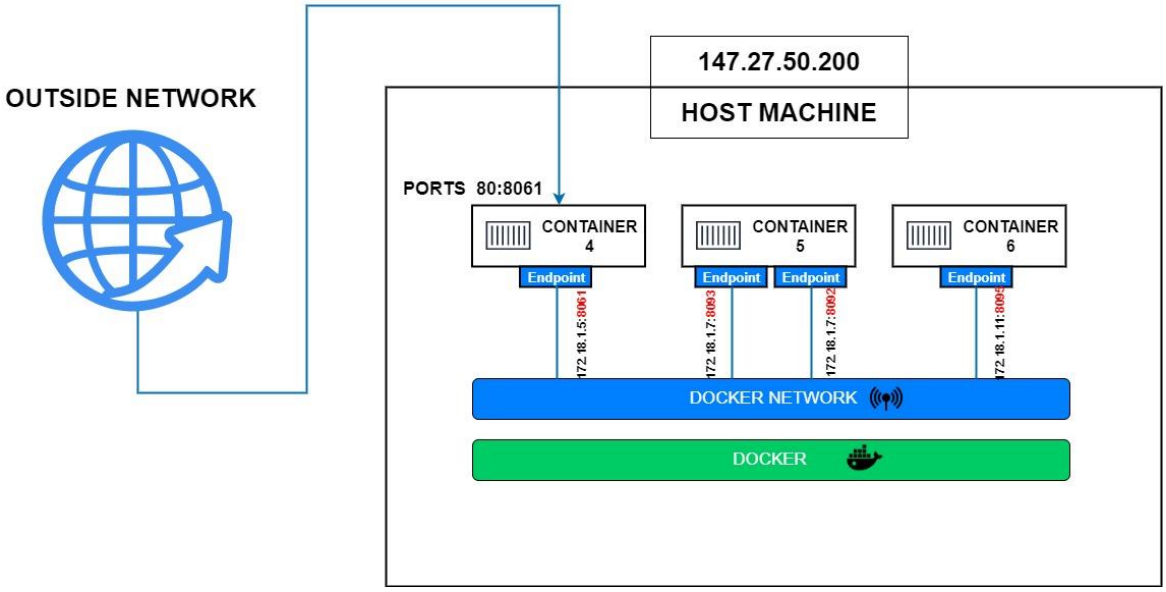


Figure 31 – Communication of docker container with outside world

The docker service can be installed on any computer. Also, DockerHub, which is a publish repository of images, offers images of FIWARE services in order to develop them as docker containers. In conclusion, LINCA can be implemented deployed on any group of computers that can communicate under the same network and have the docker service installed.

5. Back – End Performance

Each of the cloud systems was developed on Intellicloud of Technical University of Crete. In total, 3 virtual machines are used. Each of this machine consists a docker machine and Apache Cassandra Database. In every docker machine there are 22 docker containers where the following services are executed:

- Web Application
- Application Logic
- Query Sensor Service
- Query Available Cloud Service
- Register Sensor Service
- Register Cloud Service
- STH-Comet Service
- Cygnus Service
- Orion Context Broker Service
- Sensor Interface Service
- PEP Proxy for each protected service,7 in total.
- Keyrock IDM
- AuthZForce
- MySQL Database
- MongoDB, 2 in total.

The technical features of the above virtual machines are as follows:

CPU	4 VCPU
Memory	8GB
HDD	80GB
OS	Ubuntu 16.04 LTS

The Apache Benchmark tool was installed and used in each virtual machine, in order to determine the performance of the system under real conditions. This tool can create quite a few simultaneous requests. Also, it can create heavy workloads on each system service individually by specifying the number of requests to be served and how many of them will be executed at the same time.

In each of the following experiments, 2000 requests are made to each system's service. These requests are repeated with different number of concurrencies. The measurements refer to the execution time per request that occurred and are divided into categories according to their concurrency. These categories are as follows:

As local services we define the services that are located to the cloud system in which the user is connected and makes requests and as remote services we define the services that are located to a remote cloud system of LINCA in which the user is interested to access its services.

5.1. Experiment 1 – Query Sensors in Local Cloud System

Scenario - The user through the graphical interface is querying for sensors based on desired features, that are connected to his/her local cloud system. A local cloud system defines the cloud where the user logs in and makes requests. For this experiment, the local cloud is called Athens with ip address http://147.27.50.200.

Services - User via Web Application chooses the attributes of the sensors that wishes to search and are connected to his/her local cloud. The Web Application forwards the request to Application Logic. After, the Application Logic adds User OAuth2 token to the initial user's request and then forwards it to the local PEP Proxy that "protects" the local Query Sensor Service. Next, this PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then it returns to local PEP proxy the user's information. In addition, PEP proxy checks for user's permissions in local AuthZForce PDP Service. If local AuthZForce returns "Permit" then the PEP proxy forwards the initial user's request to the protected service, the local Query Sensor Service. If local AuthZForce returns "Denied" then the PEP proxy will not forward the initial request. After the successful user authentication and authorization, the local Query Sensor Service starts to query Directory DB for the desired sensors that are connected to the local cloud. In the end, Directory DB returns the desired sensors that are connected to the local cloud. The above workflow is represented in Section 3.5.

Details - The request examined in this experiment, concerns finding sensors at the user's local cloud (ip address: http://147.27.50.200) that measure temperature and pressure. The query was made on a collection of 900 sensor entities (virtual, in order the response to be realistic) with only three of them meeting the query criteria.

REST - POST http://147.27.50.200:8060/getDesiredIDs, with request body (city=Athens && measurement= Temperature && measurement = Pressure).

Results - The results for the execution time per request are listed in the following Figure 32.

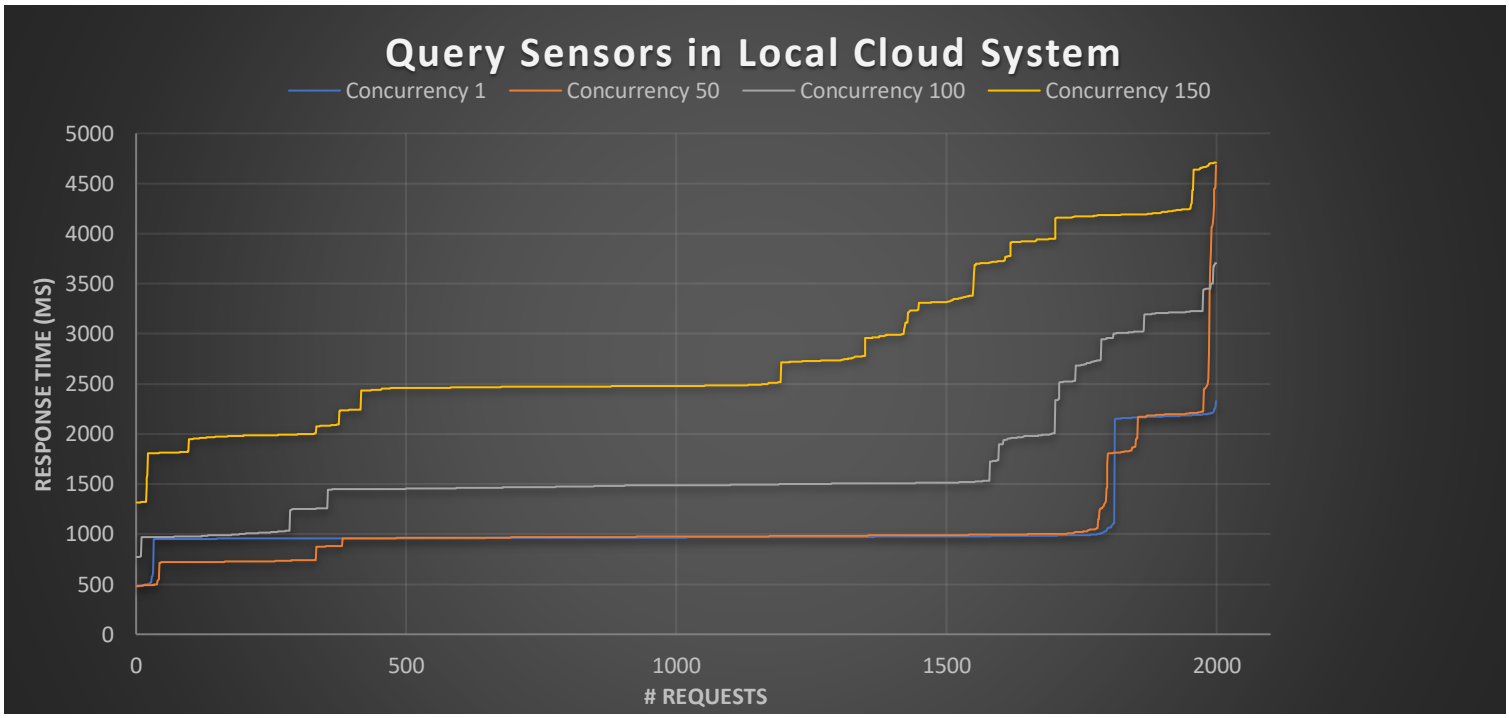


Figure 32 – Execution time per request for querying sensors in Local Cloud System

5.2. Experiment 2 - Query Sensors in Remote Cloud System

Scenario - The user through the graphical interface is querying for sensors based on desired features, that are connected to a remote cloud system. A remote cloud system defines the cloud in which user is not directly connected but is authorized to query for its sensors through user's local cloud. For this experiment, the local cloud is called Athens with ip address http://147.27.50.200 and the remote cloud is called Chania with ip address http://147.27.50.199.

Services - User via local Web Application chooses the attributes of the sensors that wishes to search and the remote cloud that their connected. The local Web Application forwards the request to local Application Logic. After, the local Application Logic adds User OAuth2 token to the initial user's request and then forwards it to the local PEP Proxy that "protects" the local Query Sensor Service. Next, this PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then it returns to local PEP proxy the user's information. In addition, local PEP proxy checks for user's permissions in AuthZForce PDP Service that is located to the remote cloud that user wants

to find its devices. If remote AuthZForce returns “Permit” then the local PEP proxy forwards the initial user’s request to the protected service, the local Query Sensor Service. If remote AuthZForce returns “Denied” then the PEP proxy will not forward the initial request. After the successful user authentication and authorization, the local Query Sensor Service starts to query Directory DB for the desired sensors that are connected to the remote cloud. In the end, Directory DB returns the desired sensors that are connected to remote cloud. The above workflow is represented in Section 3.5.

Details - The request examined in this experiment, concerns finding sensors at the user's remote cloud (ip address: http://147.27.50.199) that measure temperature and pressure. The query was made on a collection of 900 sensor entities (virtual, in order the response to be realistic) with only three of them meeting the query criteria.

REST - POST http://147.27.50.200:8060/getDesiredIDs, with request body (city=Chania && measurement= Temperature && measurement = Pressure).

Results - The results for the execution time per request are listed in the following Figure 33.

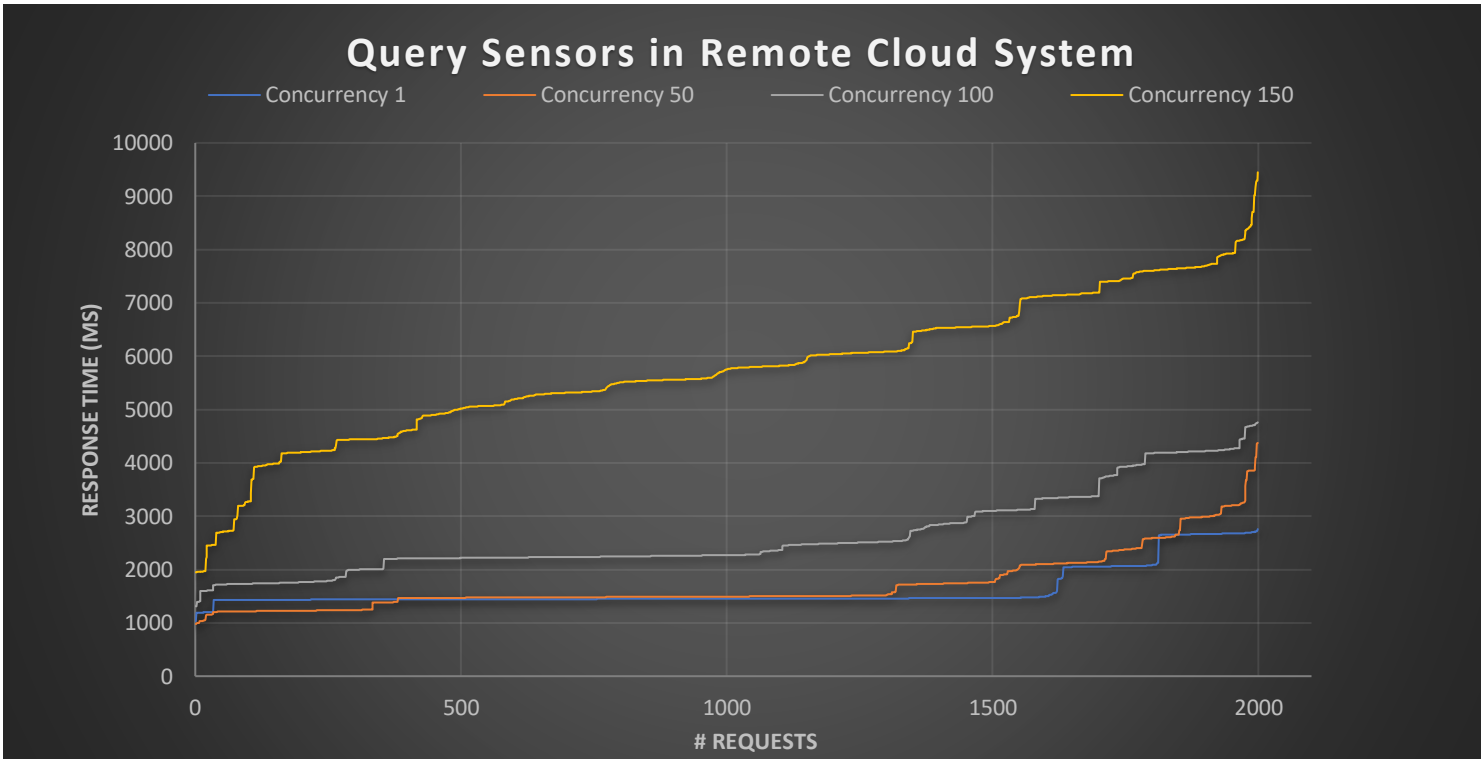


Figure 33 – Execution time per request for querying sensors in Remote Cloud System

By computing the average execution time per request of Experiment 1 and Experiment 2, we notice a difference between their average execution time of

each concurrency category. This difference is due to the connection time that Experiment 2 needs to authenticate-authorize a user in a remote cloud and to query for sensors that are connected to this remote cloud. The results for the average execution time per request of Experiment 1 and Experiment 2 are shown in the following Figure 34.

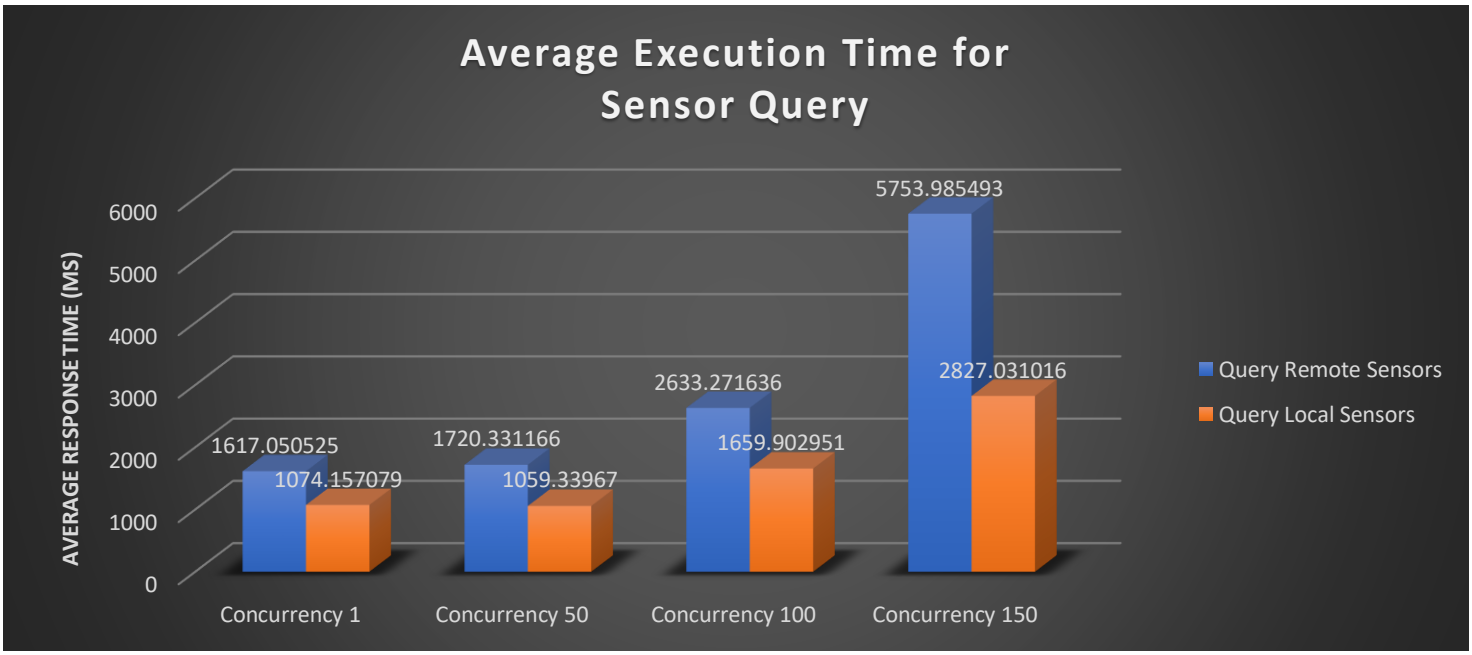


Figure 34 – Average execution time per request for querying sensors in Local Cloud System and in Remote Cloud system

5.3. Experiment 3 – Retrieve Maximum Value of Local Sensor

Scenario - The user through the graphical interface requests the maximum temperature measurements of his/her subscribed sensor “urn:ngsi-ld:t:beacon:1”, every hour of the last 24 hours. This sensor is connected to user’s local cloud system.

Services – User via local Web Application chooses from subscribed sensors list the sensor “urn:ngsi-ld:t:beacon:1” in order to retrieve its maximum temperature measurements of every hour of the last 24 hours. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user’s information. Local PEP proxy checks for user’s permissions in local AuthZForce PDP. If local AuthZForce returns “Permit” then local PEP proxy forwards the initial user’s request to the protected service, local STH-Comet. If local AuthZForce returns “Denied” then the local PEP proxy will not forward the initial request. Comet processes the request and starts querying

in History DB for the maximum temperature measurements of the chosen sensor. At last, History DB returns the requested information. The above workflow is represented in Section 3.5.

REST - GET "147.27.50.200:8666/STH/v1/contextEntities/type/Sensor/id/urn:ngsild:t:beacon:1/attributes/Temperature?aggrMethod=max&aggrPeriod=hour&dateFrom=2019-08-24T10:25:00.000Z&dateTo=2019-08-25T10:25:00.000Z".

Results - The results for the execution time per request are listed in the following Figure 35.

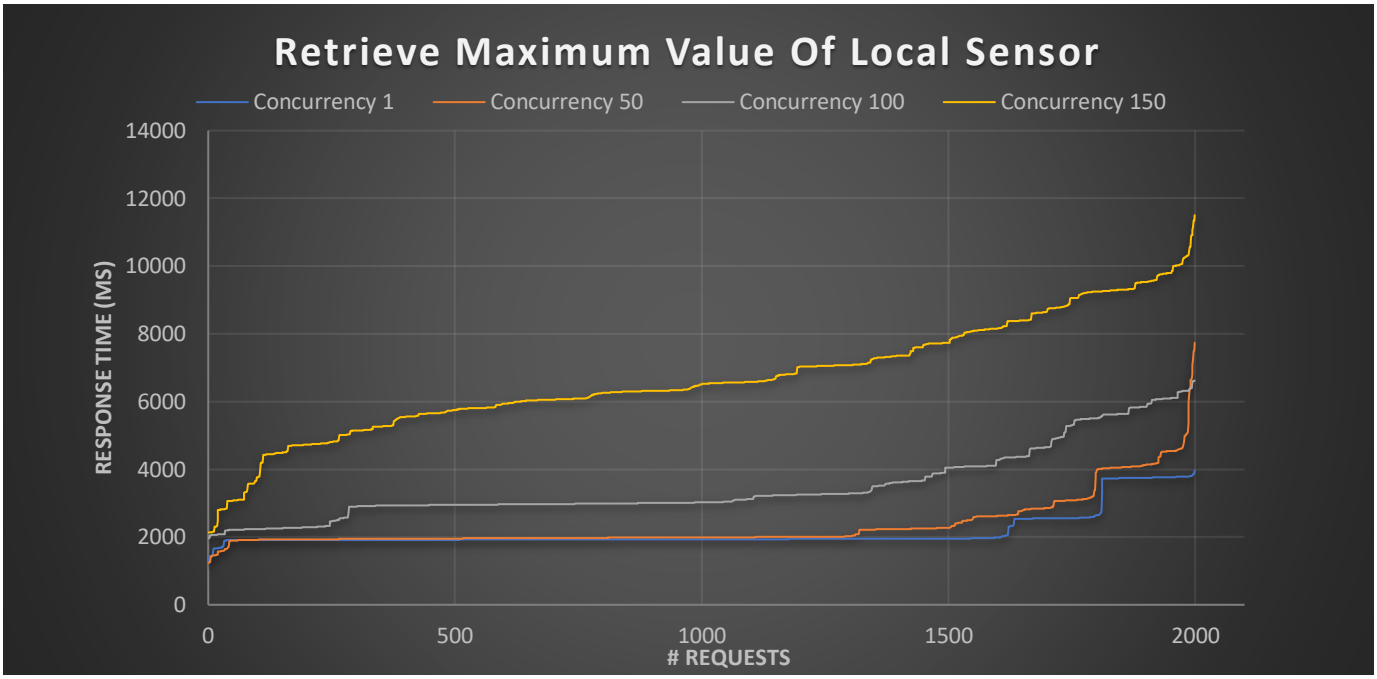


Figure 35 – Execution time per request for retrieving maximum value of sensor

5.4. Experiment 4 - Retrieve Maximum Value of Remote Sensor

Scenario - The user through the graphical interface requests the maximum temperature measurements of his/her subscribed sensor "urn:ngsi-ld:t:beacon:251", every hour of the last 24 hours. This sensor is connected to a remote cloud system.

Services – User via local Web Application chooses from subscribed sensors list the sensor "urn:ngsi-ld:t:beacon:251" in order to retrieve its maximum temperature measurements of every hour of the last 24 hours. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user's information. In addition, local PEP proxy checks for

user's permissions in AuthZForce PDP Service that is located to the remote cloud in which the subscribed sensor is connected. If remote AuthZForce returns "Permit" then local PEP proxy forwards the initial user's request to the protected service, local STH-Comet. If local AuthZForce returns "Denied" then the local PEP proxy will not forward the initial request. Comet processes the request and starts querying in History DB for the maximum temperature measurements of the chosen sensor. At last, History DB returns the requested information. The above workflow is represented in Section 3.5.

REST - GET "147.27.50.200:8666/STH/v1/contextEntities/type/Sensor /id/urn:ngsild:t:beacon:251/attributes/Temperature?aggrMethod=max&aggrPe riod=hour&dateFrom=2019-08-24T10:25:00.000Z&dateTo=2019-08-25T10:25:00.000Z".

Results - The results for the execution time per request are listed in the following Figure 36.

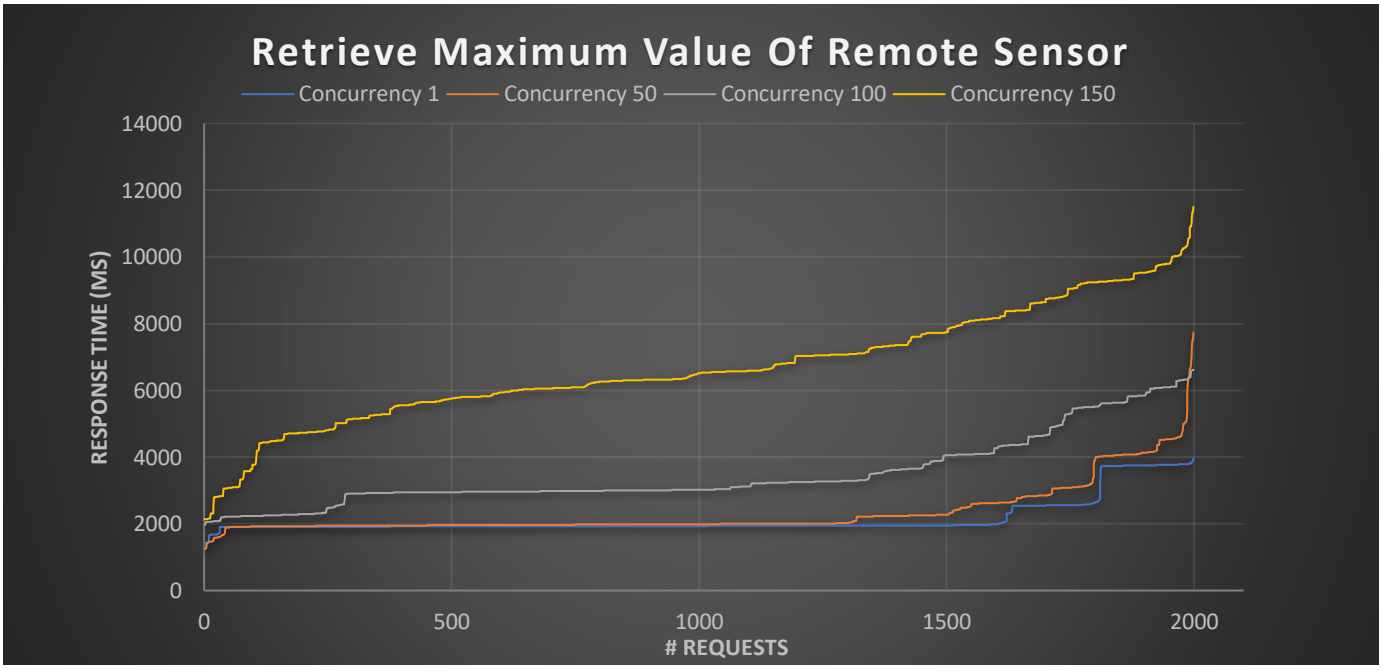


Figure 36 – Execution time per request for retrieving maximum value of sensor

By computing the average execution time per request of Experiment 3 and Experiment 4, we notice a difference between their average execution time of each concurrency category. This difference is due to the connection time that Experiment 4 needs to authenticate-authorize a user in a remote cloud. The results for the average execution time per request of Experiment 3 and Experiment 4 are shown in the following Figure 37.

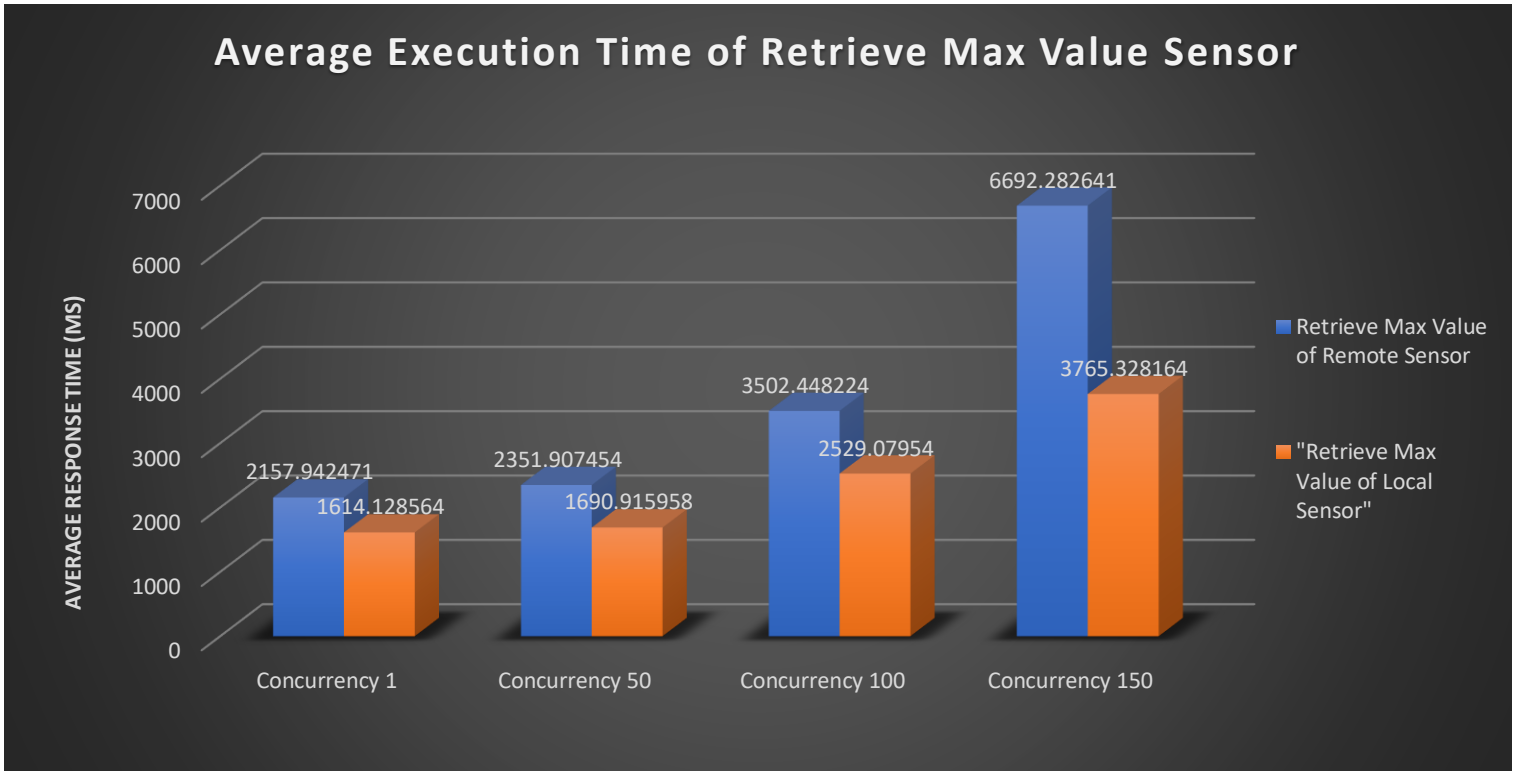


Figure 37 – Average execution time per request for retrieving max value of sensor in Local Cloud System and in Remote Cloud system

5.5. Experiment 5 - Retrieve Current Value of Local Sensor

Scenario - The user, through the graphical interface, requests the current temperature measurement of his/her subscribed sensor “urn:ngsi-ld:t:beacon:13”. This sensor is connected to user’s local cloud system.

Services – User via local Web Application chooses from subscribed sensors list the sensor “urn:ngsi-ld:t:beacon:13” in order to retrieve its current temperature measurements. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user’s information. Local PEP proxy checks for user’s permissions in local AuthZForce PDP. If local AuthZForce returns “Permit” then local PEP proxy forwards the initial user’s request to the protected service, local STH-Comet. If local AuthZForce returns “Denied” then the local PEP proxy will not forward the initial request. Comet processes the request and starts querying in History DB for the current temperature measurements of the chosen sensor. At last, History DB returns the requested information. The above workflow is represented in Section 3.5.

REST - GET “147.27.50.200:8666/STH/v1/contextEntities/type/Sensor /id/urn:ngsi-ld:t:beacon:13/attributes/Temperature?LastN=1".

Results - The results for the execution time per request are listed in the following Figure 38.

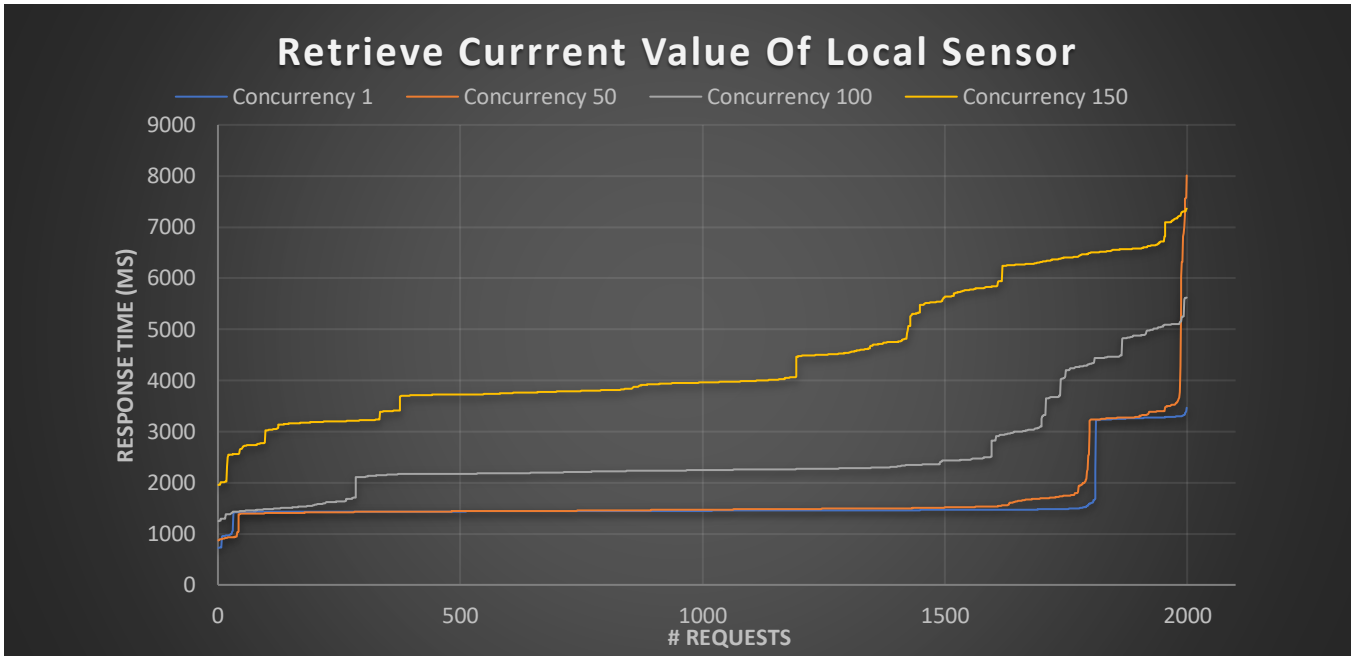


Figure 38 – Execution time per request for retrieving current value of local sensor

5.6. Experiment 6 - Retrieve Current Value of Remote Sensor

Scenario - The user, through the graphical interface, requests the current temperature measurement of his/her subscribed sensor “urn:ngsi-ld:t:beacon:251”. This sensor is connected to a remote cloud system.

Services – User via local Web Application chooses from subscribed sensors list the sensor “urn:ngsi-ld:t:beacon:251” in order to retrieve its current temperature measurement. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user’s information. In addition, local PEP proxy checks for user’s permissions in AuthZForce PDP Service that is located to the remote cloud in which the subscribed sensor is connected. If remote AuthZForce returns “Permit” then local PEP proxy forwards the initial user’s request to the protected service, local STH-Comet. If local AuthZForce returns “Denied” then the local PEP proxy will not forward the initial request. Comet processes the request and starts querying in History DB for the current

temperature measurement. At last, History DB returns the requested information. The above workflow is represented in Section 3.5.

REST - GET “147.27.50.200:8666/STH/v1/contextEntities/type/Sensor /id/urn:ngsi-ld:t:beacon:251/attributes/Temperature?LastN=1".

Results - The results for the execution time per request are listed in the following Figure 39.

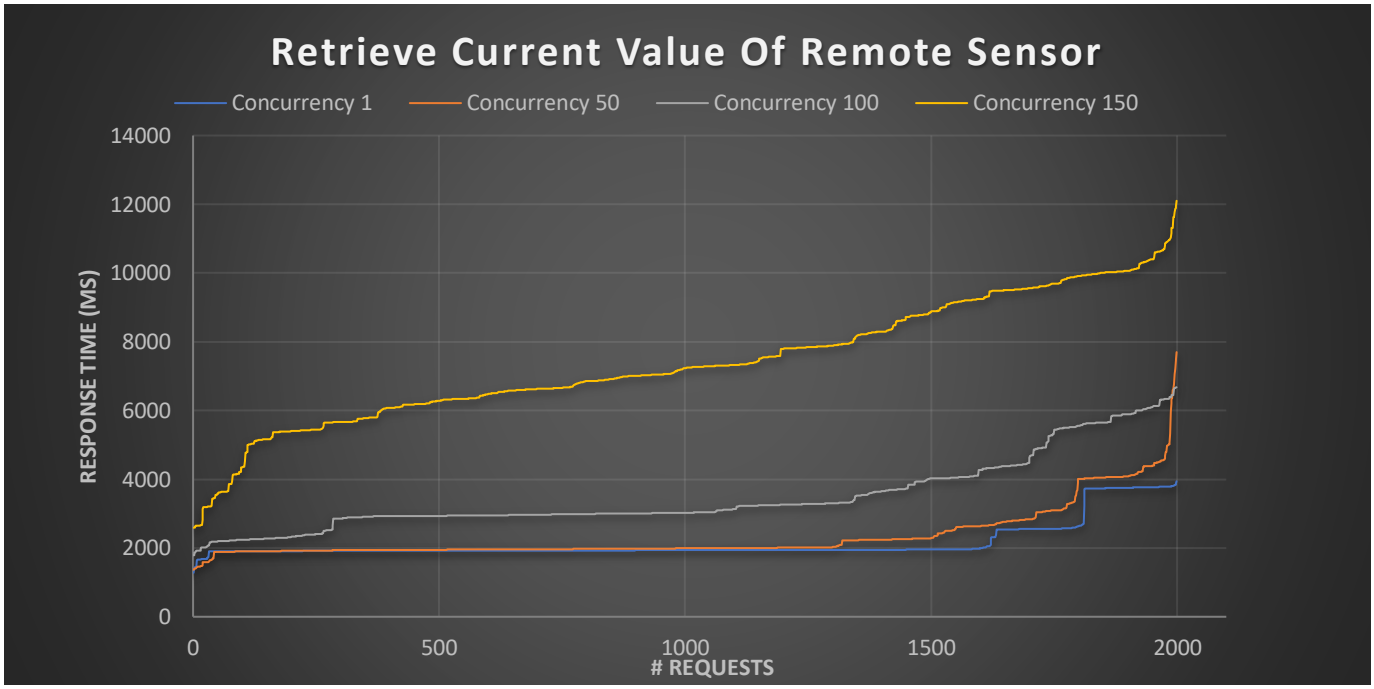


Figure 39 – Execution time per request for retrieving current value of remote sensor

By computing the average execution time per request of Experiment 5 and Experiment 6, we notice a difference between their average execution time of each concurrency category. This difference is due to the connection time that Experiment 6 needs to authenticate-authorize a user in a remote cloud. The results for the average execution time per request of Experiment 5 and Experiment 6 are shown in the following Figure 40

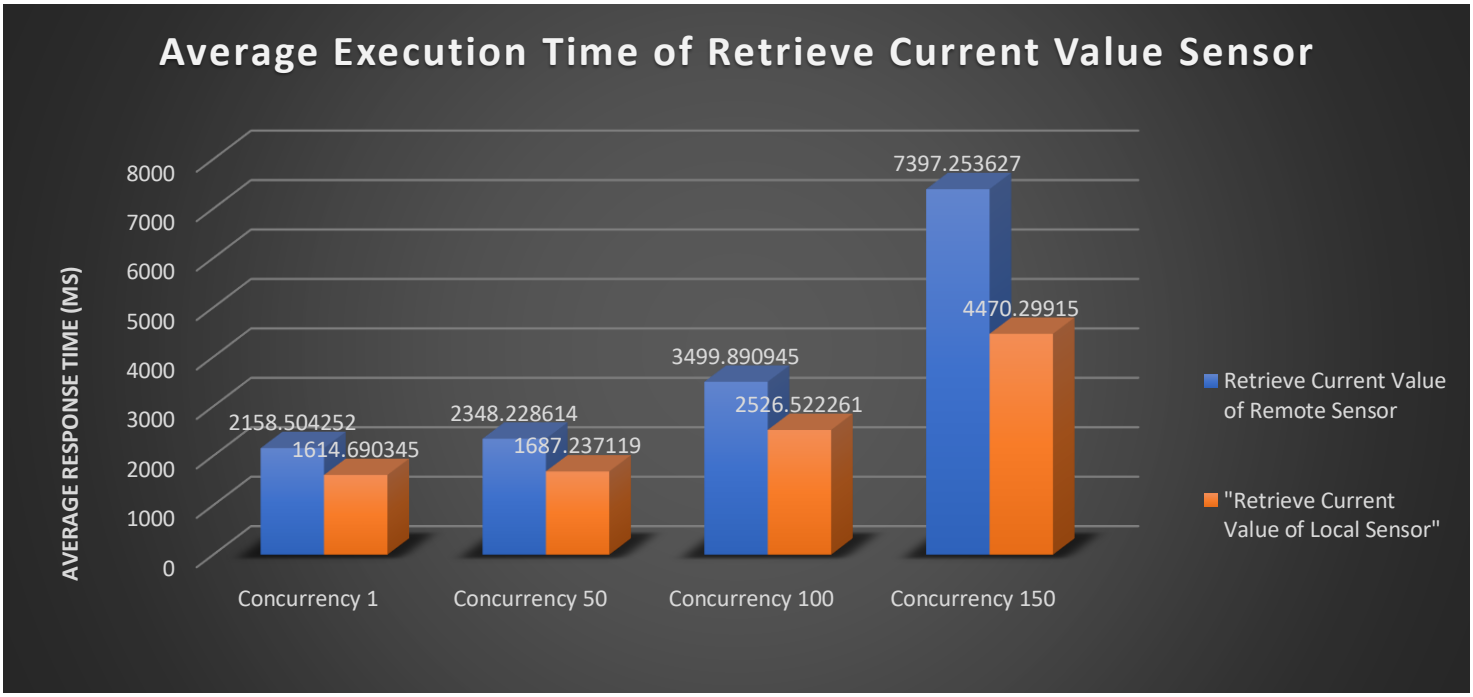


Figure 40 – Average execution time per request for retrieving current value of sensor in Local Cloud System and in Remote Cloud system

5.7. Experiment 7 – Subscribe to Local Sensor

Scenario - The user that is connected to his/her local cloud system, wants to subscribe to a sensor that is connected to his/her local cloud. The ID of this local sensor is “urn:ngsi-ld:t:beacon:1”. The user routes a subscription request to the local service that is responsible to subscribe the user to the requested local sensor. Before this, user’s request must pass through the security services of his/her local system. Local cloud system is the system in which the user is connected and he/she has the permissions to query and subscribe local sensors. For this experiment, the local cloud is called Athens with ip address http://147.27.50.200.

Services – User via local Web Application after the process of querying sensors, he/she chooses to subscribe to sensor “urn:ngsi-ld:t:beacon:251” who is connected to his/her local cloud system. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user’s information. Then local PEP proxy checks for user’s permissions in local AuthZForce PDP. If local AuthZForce returns “Permit” then local PEP proxy forwards the initial user’s request to the protected service, the local Query Sensor Service. If AuthZForce

returns “Denied” then the PEP proxy will not forward the initial request. Also Query Sensor Service is responsible to update user’s subscription list in the local node of Cassandra’s cluster. The above workflow is represented in Section 3.5.

REST - POST http://147.27.50.200/Subcreate, with body request (user=athens@customer.com , city= Athens, sensor = urn:ngsi-ld:t:beacon:1).

Results - The results for the execution time per request are listed in the following Figure 41.

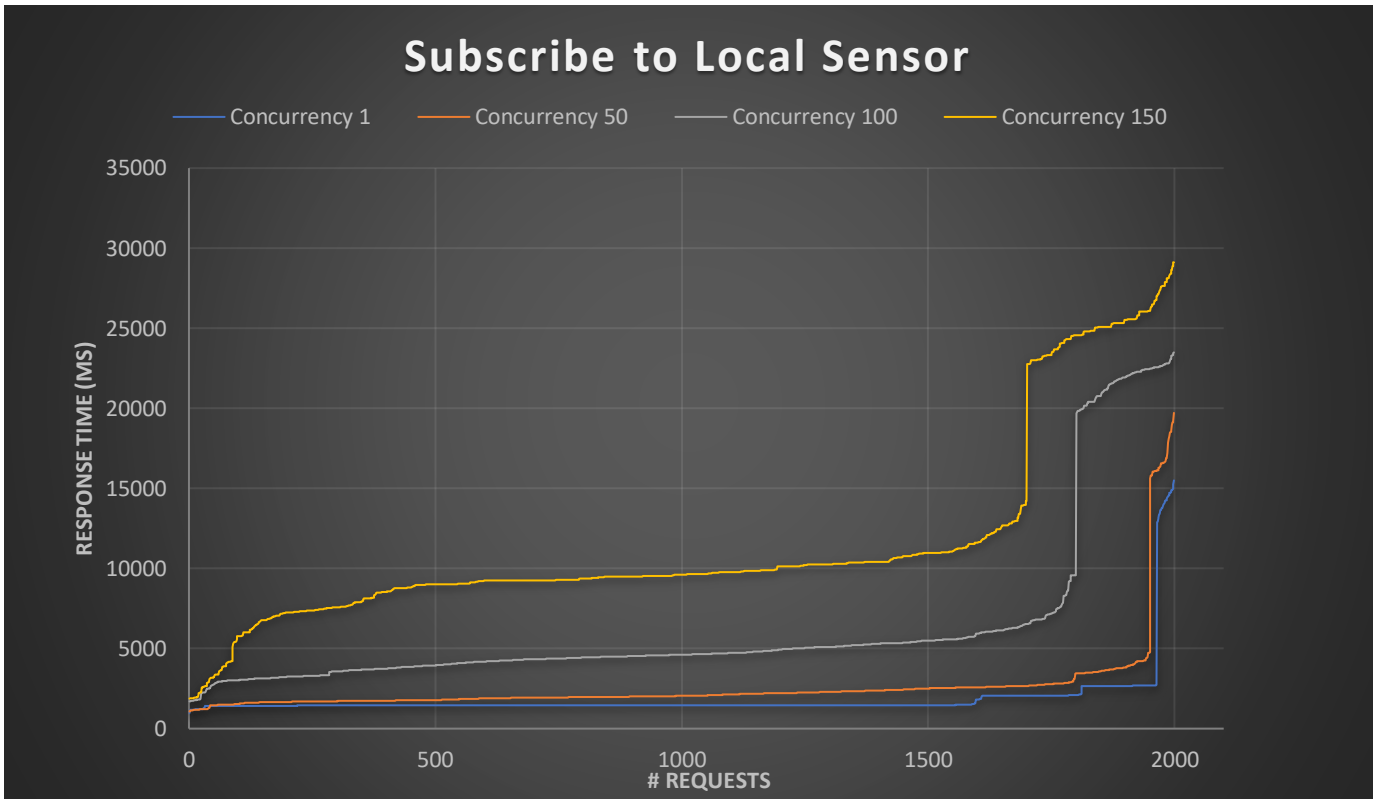


Figure 41 – Execution time per request for subscribing sensor in local cloud system

5.8. Experiment 8 – Subscribe to Remote Sensor

Scenario - The user that is connected to his/her local cloud system, wants to subscribe to a sensor that is connected to a remote cloud system. The ID of this remote sensor is “urn:ngsi-ld:t:beacon:256”. The user routes a subscription request to the local service that is responsible to subscribe the user to the remote requested sensor. Before this, user’s request must pass through the security services of his/her local system and through the security services of remote cloud in which the requested sensor is connected. Local cloud system is the system in which the user is connected and he/she routes subscription request to the remote cloud. For this experiment, the remote cloud is called Chania with ip address

http://147.27.50.199 and the local cloud is called Athens with ip address http://147.27.50.200

Services – User via local Web Application after the process of querying sensors, he/she chooses to subscribe to sensor “urn:ngsi-ld:t:beacon:256” who is connected to a remote cloud system. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user’s information. Then local PEP proxy checks for user’s permissions in the remote AuthZForce PDP Service that is located to the remote cloud system in which the requested sensors are connected. If remote AuthZForce returns “Permit” then local PEP proxy forwards the initial user’s request to the protected service, the local Query Sensor Service. If remote AuthZForce returns “Denied” then the PEP proxy will not forward the initial request. The local Query Sensor Service is responsible to subscribe the local Orion Context Broker to the remote Orion Context Broker for the sensors that user wants to subscribe. In this way when the remote Orion Context Broker receives updates from the requested sensors , it will forward them to the local Orion Context Broker.Also Query Sensor Service is responsible to update user’s subscription list in the local node of Cassandra’s cluster. The above workflow is represented in Section 3.5.

REST - POST http://147.27.50.200/Subcreate, with body request (user=athens@customer.com , city=Chania, sensor=urn:ngsi-ld:t:beacon:256).

Results - The results for the execution time per request are listed in the following Figure 42.

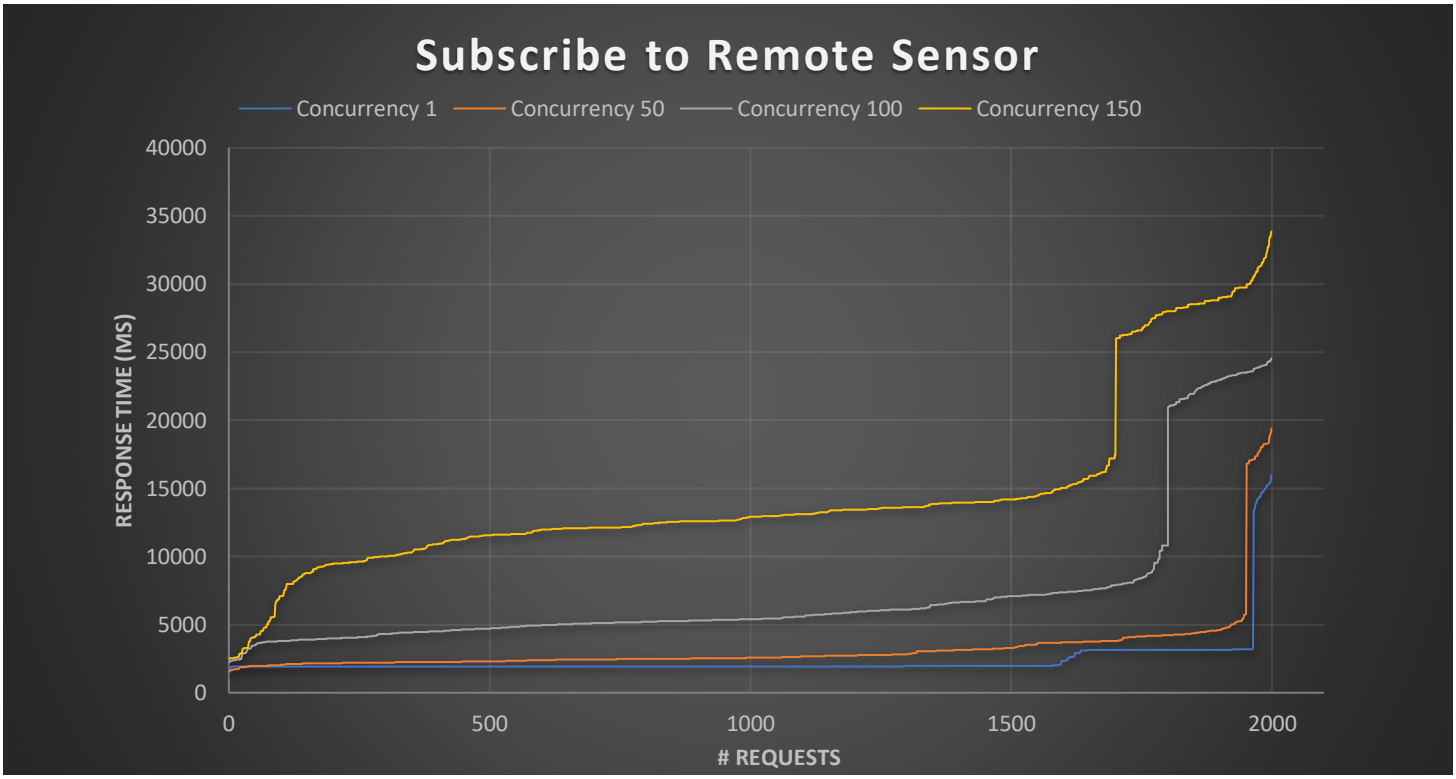


Figure 42 – Execution time per request for subscribing sensor in remote cloud system

By computing the average execution time per request of Experiment 7 and Experiment 8, we notice a difference between their average execution time of each concurrency category. This difference is due to the connection time that Experiment 8 needs to authenticate-authorize a user in the remote cloud. The results for the average execution time per request of Experiment 7 and Experiment 8 are shown in the following Figure 43.

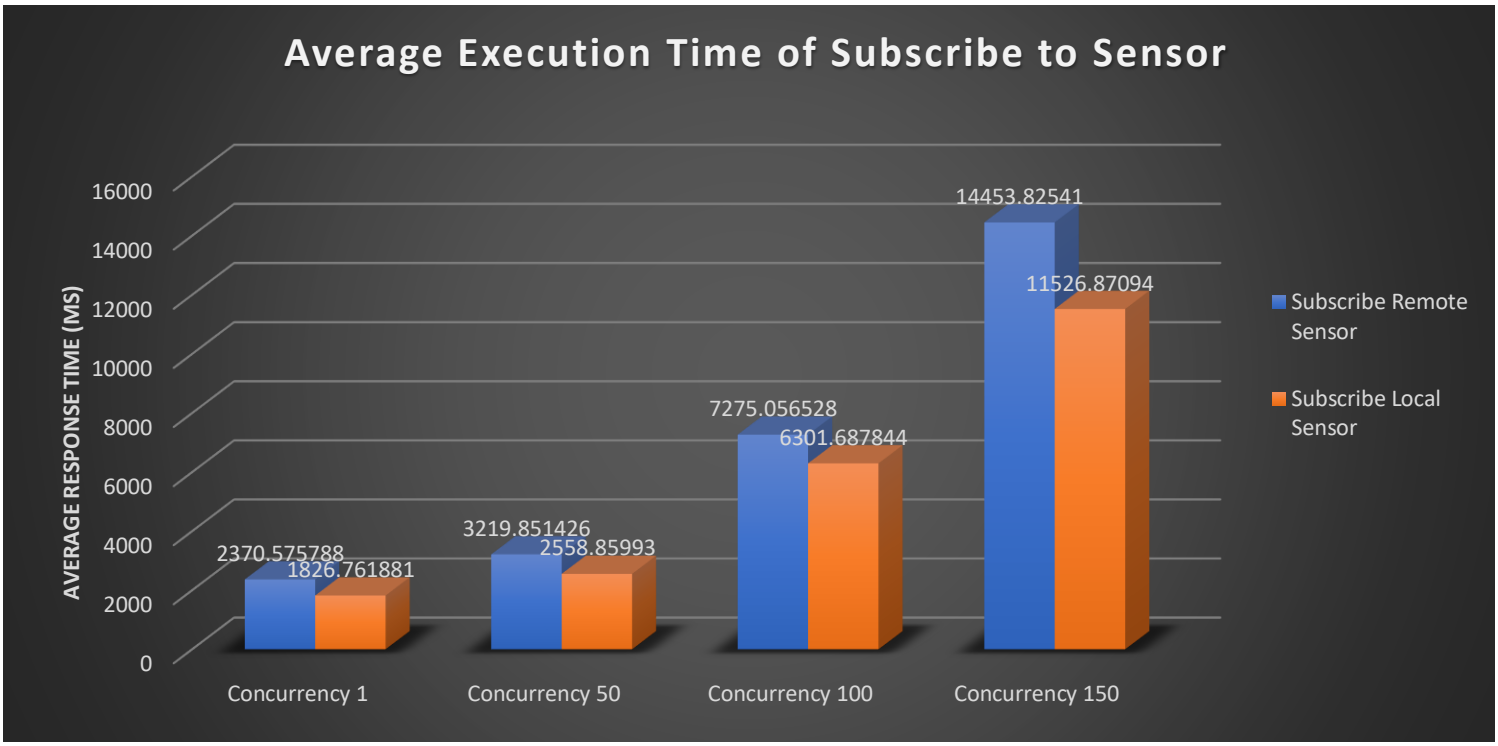


Figure 43 – Average execution time per request for subscribing sensor in Local Cloud System and in Remote Cloud system

5.9. Experiment 9 – Register Cloud System to LINCA

Scenario – An administrator of a cloud system wants to register his/her cloud in the LINCA ecosystem. In order to do this, he/she must route request to the appropriate local service that is connected to the local node of Cassandras cluster. This service will register the cloud system's information to Cassandra cluster in order to be discoverable from other existing cloud systems of LINCA. Before the above process the admin's request must pass through the security services of his/her system. The name of his/her cloud is called Athens with ip address 147.24.50.200.

Services – Admin via local Web Application is typing the information of his/her cloud who wants to register in the LINCA system. Local Web Application forwards the request to local Application Logic. Local Application Logic adds User OAuth2 token to the initial request and then forwards it to local PEP Proxy. Local PEP proxy checks User OAuth2 token in local Keyrock. Immediately, local Keyrock checks its database if user exist with the corresponding OAuth2 token. If user, exist then local Keyrock returns to local PEP proxy the user's information. Local PEP proxy checks user's permissions in local AuthZForce. If local AuthZForce returns "Permit" then local PEP proxy forwards the initial

user's request to the protected service, the local Register Cloud Service. If local AuthZForce returns "Denied" then the local PEP proxy will not forward the initial request. Local Register Cloud Service process the request and imports the cloud's information in Directory DB in order to be discoverable from remote users. In the end, Directory DB returns if the insertion was success. The above workflow is represented in Section 3.5.

REST - POST http://147.27.50.200/InsertNode with body request (name=Athens, address= http://147.27.50.200, longitude=27.8, latitude=18.1).

Results - The results for the execution time per request are listed in the following Figure 44.

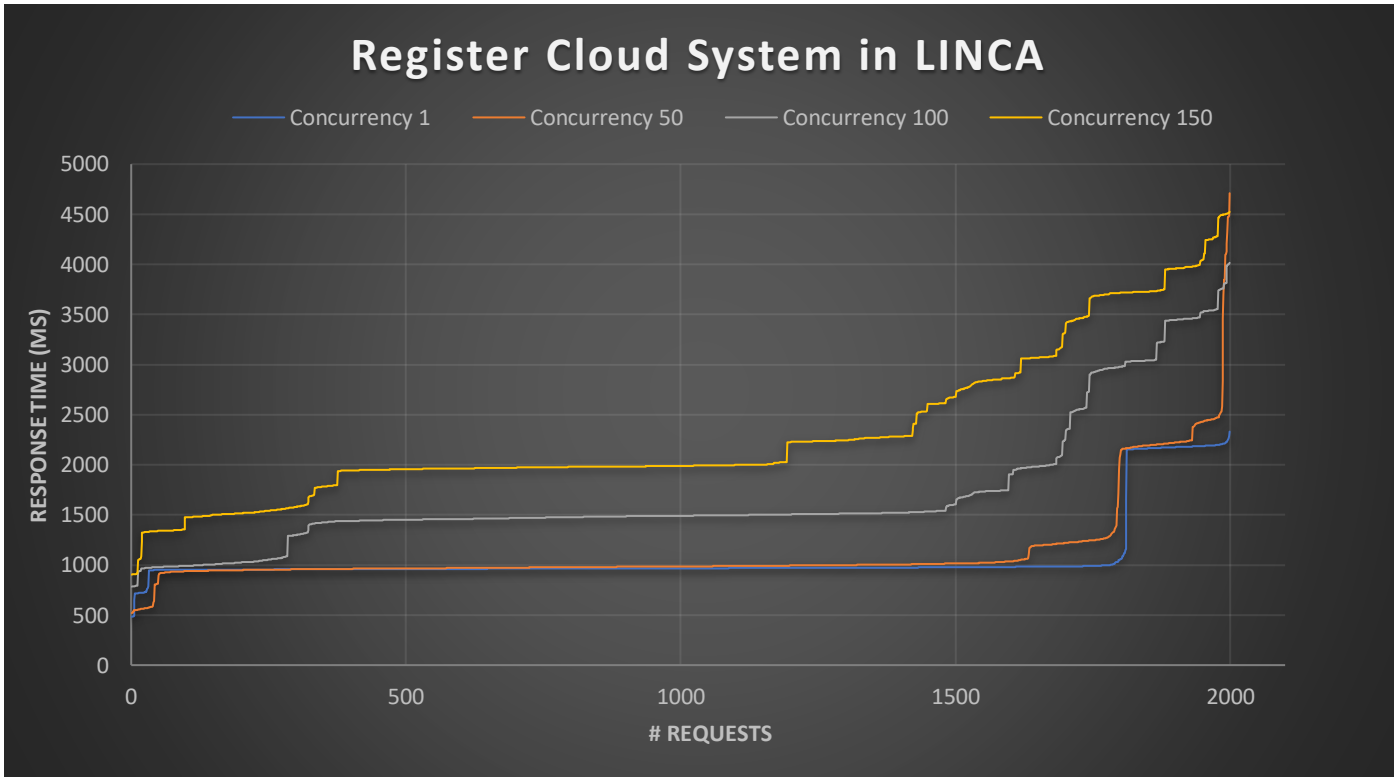


Figure 44 – Execution time per request for inserting a Cloud System in LINCA

The results for the average execution time per request of Experiment 9 and are shown in the following Figure 45.

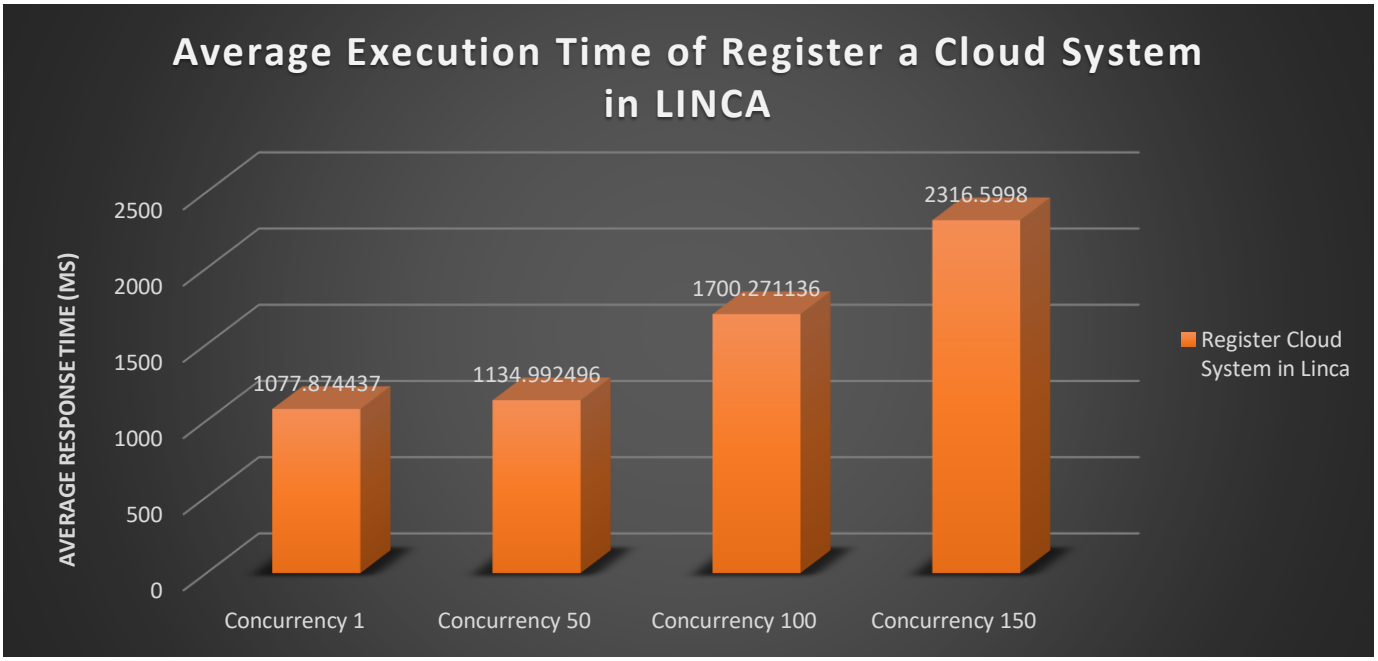


Figure 45 – Average execution time per request for registering a Cloud System in LINCA

6. Conclusions

For this thesis, we introduced and implement a master-less distributed system, named LINCA. The development of LINCA in the cloud computing through FIWARE has the advantage of using its services. Keyrock Identity Manager , Wilma PEP Proxy, AuthZForce provide security to LINCA's cloud systems as a result to the overall LINCA. Orion Context Broker provides subscriptions features in order to retrieve sensors that are connected to its cloud system or in a remote cloud system. Cygnus is responsible to store and maintain the sensors that are received from Orion Context Broker and the STH-Comet is responsible to retrieve history data measurements of these stored sensors.

The sensor interface service as implemented is not a generic solution for sensor interfaces in the system as it is adapted to work only for "Proximity Beacon" sensor devices of "Estimote" company. Its development was carried out with the intention of demonstrating the system using the physical devices we had at our disposal (Proximity Beacons).

Also, one of the main components of LINCA that make it a master-less distributed system is Apache Cassandra. Cassandra is a master-less distributed database consists of nodes that form a cluster , called Cassandra ring. Each of these nodes is located at every LINCA's cloud system. In this way , users can query for sensors that are connected to its cloud system or to remote cloud systems. In addition, users can search for the available cloud systems that are registered in LINCA system.

Of course, the use of Service-Oriented Architecture (SOA), and more specifically the use of RESTful services, assist with the communication between services and thus the development of the service that is responsible for the orchestration of services in each cloud system. A great advantage of this architecture was the flexibility to use different programming languages for each cloud system operation as well as ease in modification of individual cloud's services without affecting the whole system.

Summarizing our conclusions , LINCA is a system where different cloud systems can interact with others clouds under security in order to provide their sensors and services. Each of this system is capable of handling large number of devices while the functions it provides are executed in real time even for large number of users.

7. Future Work

The design and implementation of the system can be considered that meets the Functional System Requirements that are explained in Section 3.2.1, by achieving the development of a fully scalable, distributed, master-less IoT System for cloud system management and the management of their connected devices following the three architectural model. However, the work as it has been done within a specific time as a result to give place for some issues as future work:

- **Deployment of LINCA system in Kubernetes** - Kubernetes automates deploying, scaling and managing the individual containerized applications on a cluster of virtual servers. Kubernetes also lets you automatically handle networking, storage, logs, alerting for all particular containers.
- **Deployment of Sensor Interface based on Back End Device Management (IDAS) of FIWARE** – A general solution for secure connection of devices to their cloud systems can be the FIWARE Back End Device Management – IDAS instead of the existing sensor interface service that are located in every cloud system. IDAS has the ability to receive data and translate specific IoT protocols (LoRaWAN0.1, HTTP, MQTT, CoaP) into the NGSI information protocol, that is the FIWARE data representation-exchange standard. In addition, the IDAS service has its own protection mechanism as it only receives data from the registered physical devices that the infrastructure owners registered to their cloud systems.
- **Integration of Algorithm to exploit stored device data** - This can be achieved by developing a service and integrating to it a new algorithm , which will exploit the available data of the devices that are stored in LINCA system in an interesting way.
- **Change communication protocol to HTTPS** - All requests between system services are handled using the HTTP protocol. An important improvement in system security is that to change the communication protocol to HTTPS due to it is a more secure protocol for the transmission of "sensitive" information.

8. References

[1] Koundourakis X.: Design and Implementation of Service Oriented Architecture for Deploying IoT Applications in the Cloud, Diploma Thesis, School of Electrical and Computer Engineering, Technical University of Crete, February 2019.(Technical Report TR-TUC-ISL-02-2019)

[2] Mike P. Papazoglou, Willem-Jan van den Heuvelm, Service oriented architectures: approaches, technologies and research issues, in: The VLDB Journal (2007), doi: 10.1007/s00778-007-0044-3.

[3] Euripides G.M. Petrakis, Stelios Sotiriadis, Theodoros Soultanopoulos, Pelagia Tsiachri Rentaa, Rajkumar Buyyac, Nik Bessis Internet of Things as a Service (iTaaS): Challenges and solutions for management of sensor data on the cloud and the fog, doi:10.1016/j.iot.2018.09.009.
http://www.intelligence.tuc.gr/~petrakis/publications/iTaaS.pdf

[4] Krishna Nadiminti, Marcos Dias de Assunção, and Rajkumar Buyya, Distributed Systems and Recent Innovations: Challenges and Benefits (2014)

[5] Swati Gupta, Kuntal Saroha, Bhawna, Fundamental Research of Distributed Database, IJCSMS International Journal of Computer Science and Management Studies, Vol. 11, Issue 02, Aug 2011