DT0222: Software Architecture

a.y. 2017-2018

https://app.schoology.com/course/1179370010/

Henry Muccini

University of L’Aquila, Italy

Masaccio – Monitoring for urbAn SAfety with the IoT

Deliverable D3 v2 Template

Date	15/01/2018
Team ID	VAS

Team Members		
Name and Surname	Matriculation number	E-mail address
Valentina Cecchini	*255596*	*valentina.cecchini@student.univaq.it*
Stefano Valentini	*254825*	*stefano.valentini2@student.univaq.it*
Andrea Perelli	*254758*	*andrea.perelli@student.univaq.it*

Table of Contents

- Code can be found in the "*implementation*" directory.
- CAPS files can be found in the *"CAPS" directory.*
- *All the diagrams/models are provided in full resolution in the "UML" and "CAPS" directories.*
- *Demo videos that shows the execution of the implemented services can be found at the following links:*
  - **Storage (D2)** https://youtu.be/RT7HMrQBuiI
  - **Actuation (D3)** https://youtu.be/YfUbq6JpVns

Challenges/Risk Analysis

(new ones in red)

Risk	Date the risk is identified	Date the risk is resolved	Explanation on how the risk has been managed
Critical messages delivery times (in case of disaster)	13/11/2017	20/11/2017	We plan on deploying redundant servers (located in a different geographic zone, which host the application) that will go online in case of active server’s failures.
Big data storage	13/11/2017	17/11/2017	We are using a dedicated NoSQL database to handle that amount of data.
Sensors failures	13/11/2017	24/11/2017	We plan on using redundancy of sensors, which will start working in case of failure (of the active sensors).
Learning of the Kafka framework	13/11/2017	15/12/2017	We are using Kafka framework to develop the system: we discovered that it is quite simple to learn and use, even if to exploit its advanced features we had to go deeper into the documentation.
Multi-database integration	20/11/2017	28//11/2017	We are using 2 databases at the same time in order to store data: the relational one stores structural information and the NoSQL stores raw data (as the two are not communicating and they contain different kind of data we do not have synchronization problems).
Install devices to existing buildings	04/01/2018	10/01/2018	We could have had problems with the installation of sensors and actuators in existing rooms but: • the door actuators are only installed in standard doors (the anti-panic doors that are installed in common rooms will not be equipped with actuators as these rooms are not access-restricted) • people counters, cameras and smoke sensors are simply mounted on walls and attached to the power grid/network considering those scenarios there should not have problems.

List of Assumptions

Assumption	Description
Network availability	We assume that we have network coverage across the monitored areas.
Microcontrollers	We assume that we can use a microcontrollers to manage each sensor.

Application Domain

As in D2.

List of Services

(new ones in red)

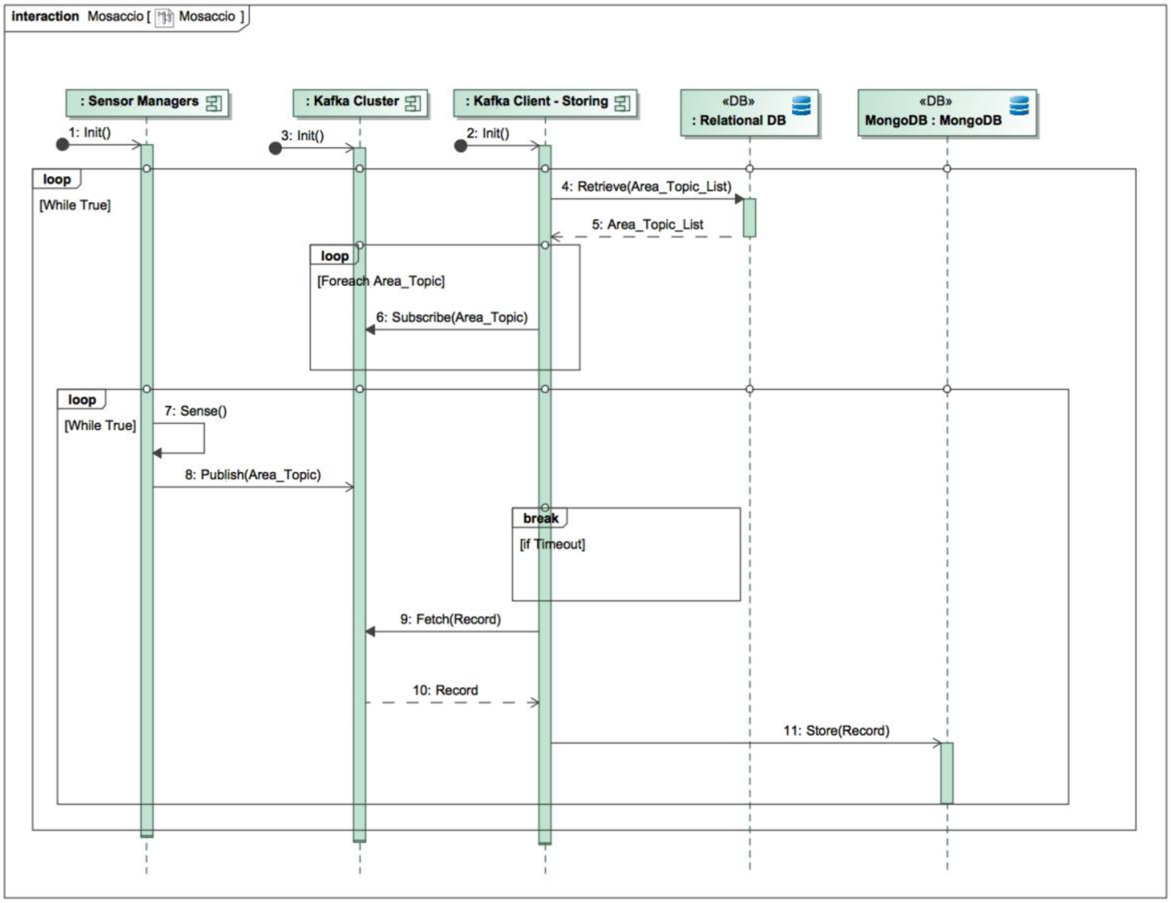
The following services have been confirmed and updated after an interview with *Rocco Matricciani*.

- **Access control** (only certain users are allowed to enter certain areas, by area we mean rooms, floors, etc.)
- **Security monitoring**
  - Air quality control (with respect to, e.g. chemical labs)
  - Smoke detection
  - Firefighting measures
- **Alarm dispatching** (sirens, loudspeakers) in case of emergencies
  - First responders’ communication service
- **Smart Information service:** provides situational aware information about the monitored area.
- **Crowd monitoring:** analyse the fluxes of peoples inside the UnivAQ’s structures to improve lectures/events scheduling and rooms assignments.
  [this one has been reported to be a high interest service by UnivAQ’s personnel]
- **Data Storing and Analysis:** all the gathered data Is stored to be analyzed. The results of such analysis will allow adjustments that will improve the organization of the University. (room assignments, people flows, schedule adjustments, etc.).
- **Attendance Registration**: allow students to register their attendance to mandatory courses by using their personal card. This allow the personnel to check if a certain student has achieved its amount of presences in a mandatory course.

UML Static and Dynamic Architecture View

Static view is the same as D2.

Dynamic view for the **Kafka Client - Storing** has been updates as follows:



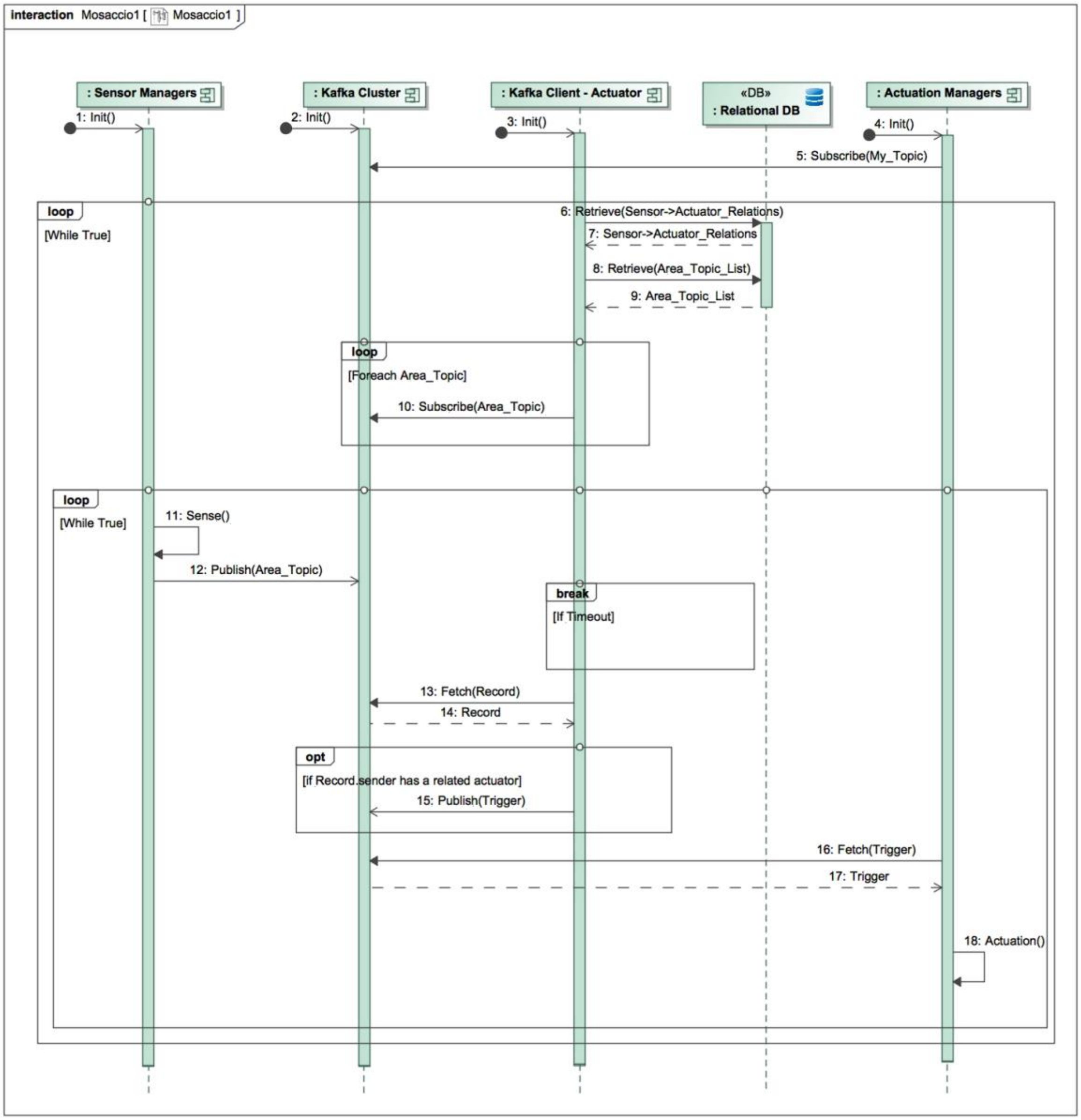
As correctly stated in the correction of the Deliverable 2 we have fixed the Storing service sequence diagram by adding a break in the inner loop that interrupts the execution when the timeout is reached.

The publishing of the messages has been modified in order to be asynchronous as stated in the previous deliverable (in the “Future improvements” section).

Dynamic view of the system for the **new** implemented service: **Actuation**

The actuation service consists of the triggering of some actuators when a certain sensor senses a certain event (e.g. a user slides his card on a card reader -> the door opens).
The implementation of this service allows us to further test our architecture, adding components and seeing how they can work together.

The execution is considered from the point of view of 1 sensor that triggers 1 actuator (we have more sensors/actuators in the demo video).



At the beginning, the **Sensor Manager**, the **Kafka Cluster**, the **Kafka Client – Actuator** and the **Actuation Manager** are initialized (are running).

The **Actuation Manager** subscribes to its **Actuator Topic** and waits for a triggering message.
The **Kafka Client - Actuator** retrieves from the relational database the list of the pairs *sensor -> actuator* (from the table **sensors_actuators**). This allows the client to know on which **Area Topics** it needs to be subscribed.

In fact, through a dedicated query the **Kafka Client - Actuator** only subscribes to the **Area Topics** in which there is at least 1 sensors that triggers an actuator.
Once the **Kafka Client - Actuator** retrieves the list of **Area Topics** from the relational database it subscribes to them.
In the meantime, the **Sensor Manager** is running and starts sensing the data from its sensor, once the data is sensed It asynchronously publishes it on its topic.

Then the **Kafka Client - Actuator** receives the message, check whether or not the sensor that sent it is related to an actuator and in this case, it asynchronously publishes it to the respective **Actuation Topic**, it otherwise ignores the message.

The **Actuation Manager** that was listening to its topic then receives the message sent by the **Kafka Client - Actuator** and triggers the actual actuator (the physical device).

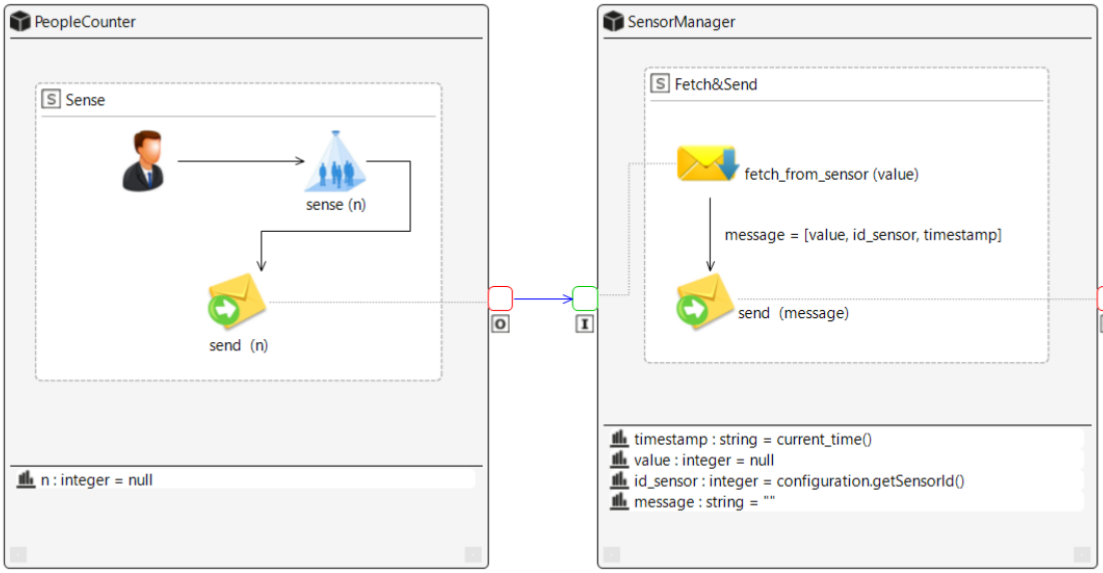
Just like the **Kafka Client - Storing**, described on deliverable 2, the **Kafka Client - Actuator** restarts avery X seconds, refreshing the data coming from the databases, so it is easy and immediate to add more sensors/actuators to the system without manual intervention.

CAPS Architecture View

CAPS SAML

In this section, we present some of the devices that are used in our architecture.
All the representations are quite minimal because all the logic and the checks are provided by server(s) and so the received/sent messages do not need refactoring (or do need in a minimal part); that is an intended behavior as we want that the critical operations are executed on the most reliable part of the system.

People Counter

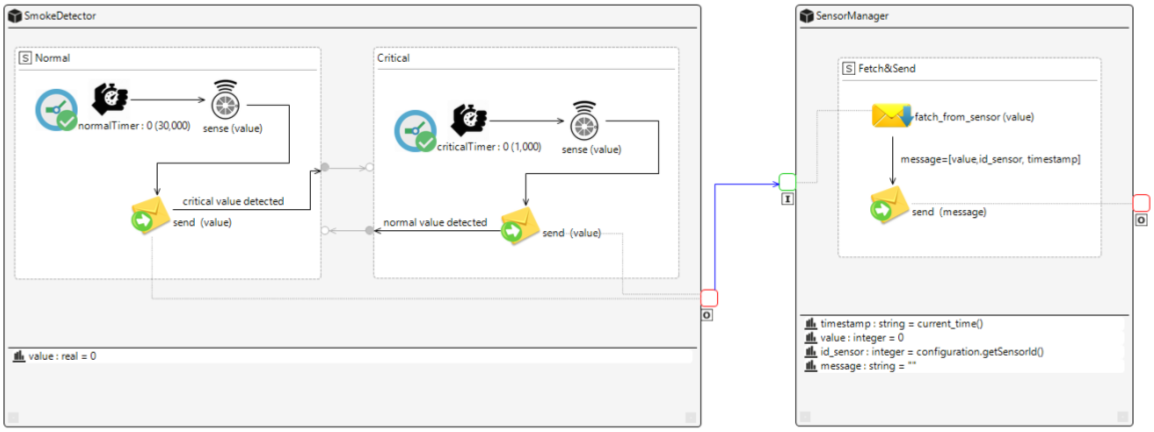

PeopleCounter
S Sense
sense (n)
send (n)
n : integer = null
O
I
SensorManager
S Fetch&Send
fetch_from_sensor (value)
message = [value, id_sensor, timestamp]
send (message)
O
timestamp : string = current_time()
value : integer = null
id_sensor : integer = configuration.getSensorId()
message : string = ""

The **PeopleCounter** component represent the actual device.
When a person enters/leaves the room/area the device computes the current room/area occupation and send this information to its **SensorManager**.

The **SensorManager** build the final message by adding the sensor id and the timestamp in which the lecture has been made and publishes the message to the respective Kafka topic (as shown in the sequence diagrams for both the developed services).
NOTE: the out-message port at the right side of the **SensorManager** component is connected to the Kafka cluster (described in the component diagrams/sequence diagrams), that’s why it has not represented in the CAPS notation.

The **SensorManager** component is repeated for all the other devices so its description will not be reproposed.

Smoke Detector

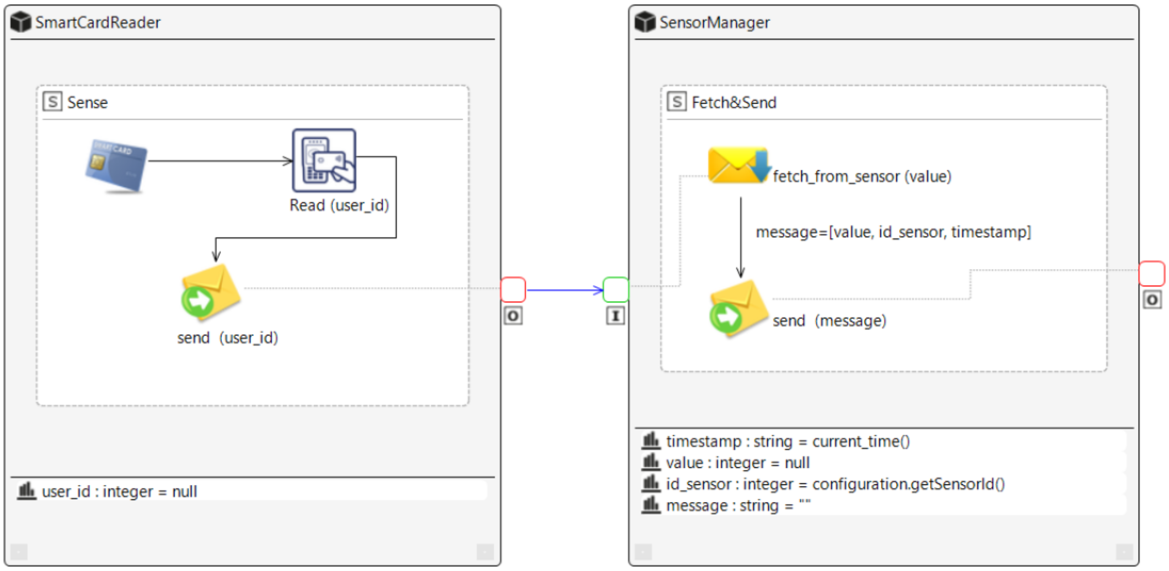


The **SmokeDetector** component is composed by two operative modes:

- **Normal:** it’s the mode in which the sensor operates in case of the sensed values are defined as “normal” and so they not imply a dangerous situation. In this mode, the readings are made every 30 seconds.
- **Critical:** It’s the operative mode in which the sensor enters in case of abnormal readings, in this mode the readings are made every second. If the sensor sees a “normal” value it enters again in **Normal Mode**.

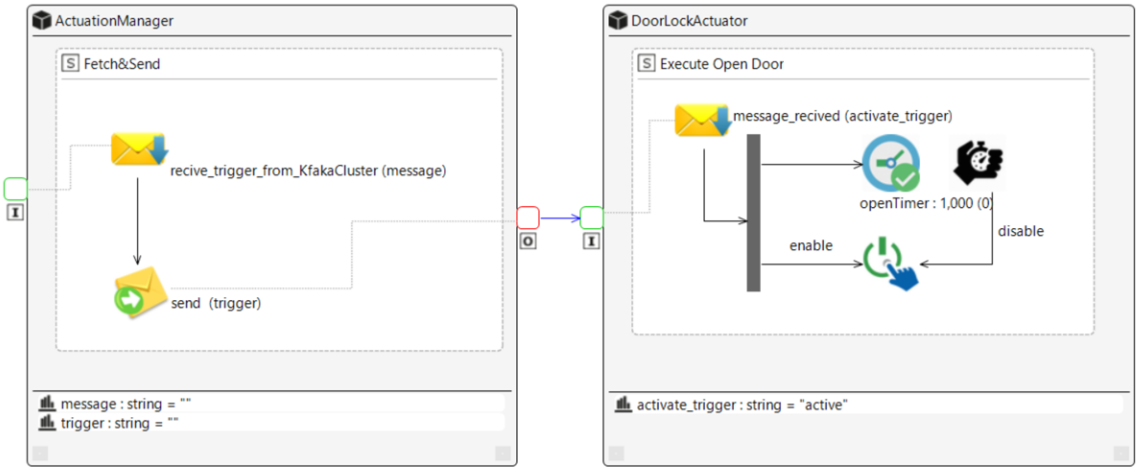
The component is equipped with a periodic timer with no delay, in **Normal Mode** the timer expires every 30 seconds, in **Critical Mode** in 1 second.

Smart Card Reader



The **SmartCardReader** component represent the physical device that is installed at the side of the door/area entrance, to filter the accesses.
It reads the used id from the card when the user passes it on the reader, and sends that information to the **SensorManager**.
<u>We only use the user id (without its access level or other information) because we want the system to interactively check for the user’s permissions</u>.

Door Lock Actuator

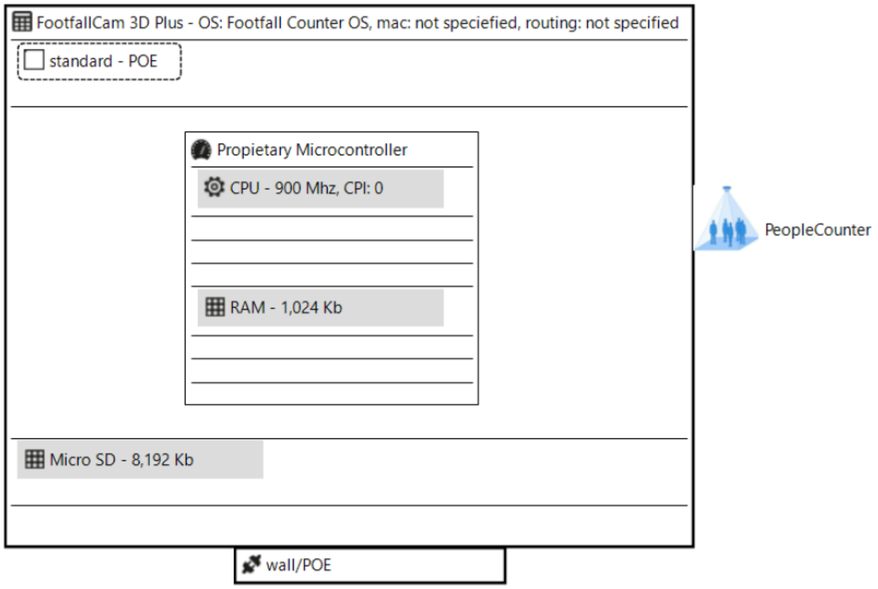


The **DoorLockActuator** receives a triggering message from its **ActuationManager** (the **ActuationManager** receives it by subscribing to the respective **Actuator Topic**).
When a trigger is received a timer is fired, the timer is not periodic but has a delay of 1 seconds before it expires. The same message opens the door lock, when the timer expires the lock is closed again.

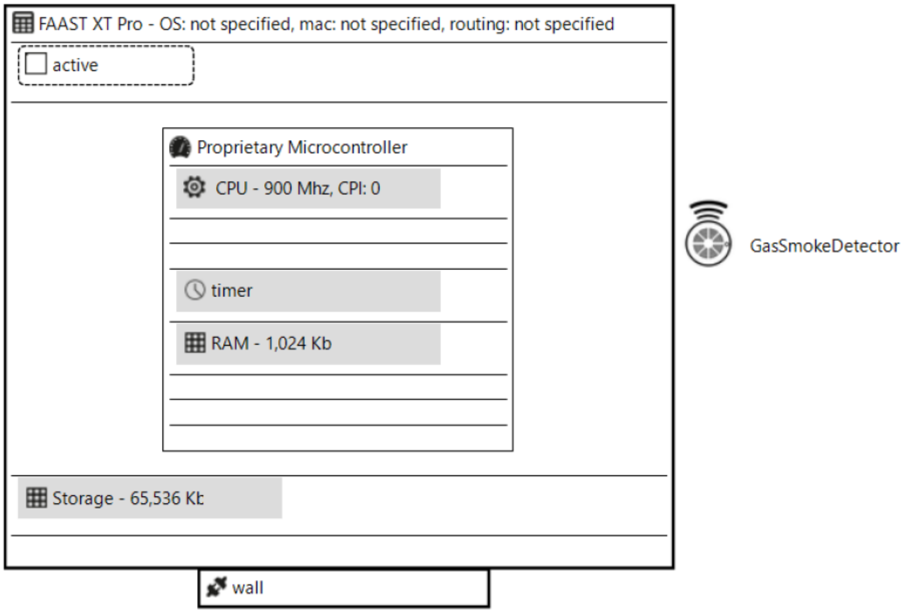
CAPS HWML

In this section, we provide the hardware specification of the sensors exposed in the SAML representations.
The provided HWML models are purely indicative of the hardware that can be used to implement the system’s services. As previously stated the sensors can and should be extremely simple as they do not perform (or do perform very little) computation.
However, we wanted to provide real examples of devices with their real specifications anyway.

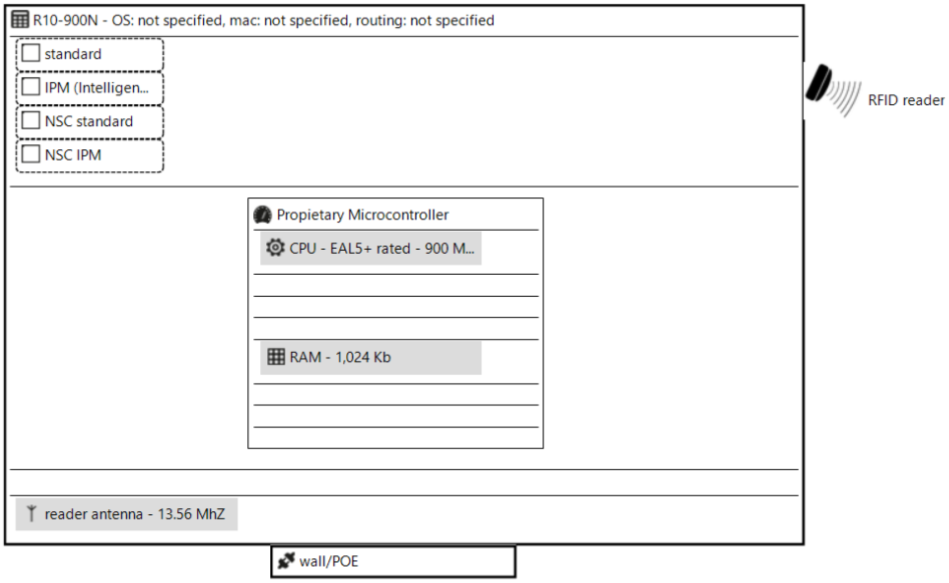
People Counter [1]


FootfallCam 3D Plus - OS: Footfall Counter OS, mac: not speciefied, routing: not specified
standard - POE
Propietary Microcontroller
CPU - 900 Mhz, CPI: 0
RAM - 1,024 Kb
Micro SD - 8,192 Kb
wall/POE
PeopleCounter

Smoke Detector [2]


FAAST XT Pro - OS: not specified, mac: not specified, routing: not specified
active
Proprietary Microcontroller
CPU - 900 Mhz, CPI: 0
timer
RAM - 1,024 Kb
Storage - 65,536 Kb
wall
GasSmokeDetector

Smart Card Reader [3]

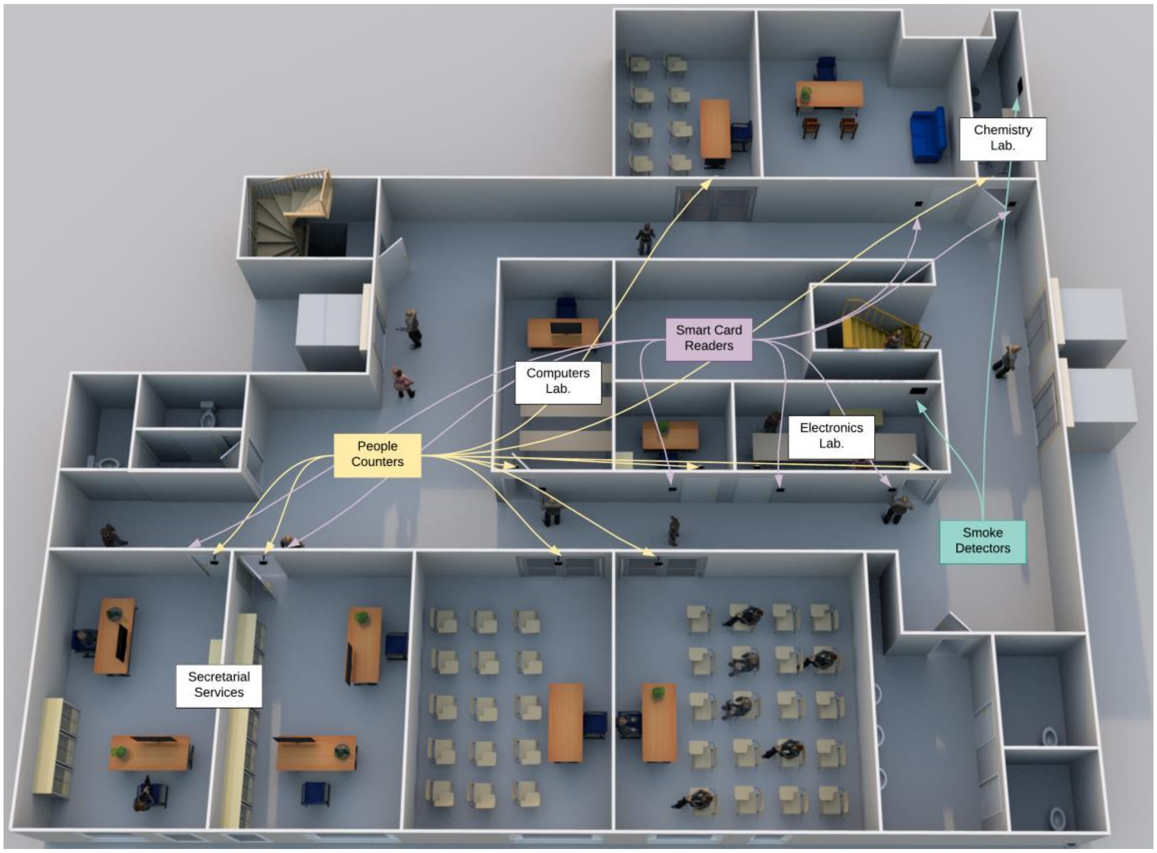

R10-900N - OS: not specified, mac: not specified, routing: not specified
□ standard
□ IPM (Intelligen...
□ NSC standard
□ NSC IPM
RFID reader
Propietary Microcontroller
CPU - EAL5+ rated - 900 M...
RAM - 1,024 Kb
reader antenna - 13.56 MhZ
wall/POE

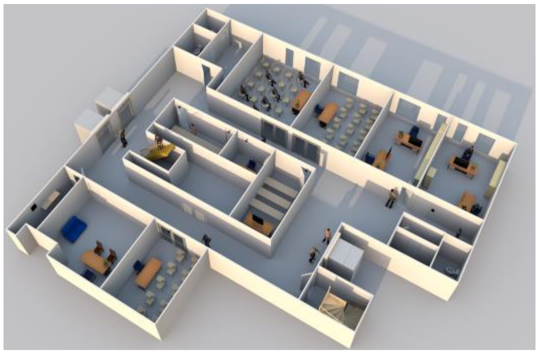
[1] - http://www.footfallcam.com/Product/3DPeopleCounter
[2] - https://www.systemsensor.com/en-us/Pages/FAAST%20XT%20Pro%209440X.aspx
[3] - https://www.rfideas.com/products/door-readers/hid-r10-model-900

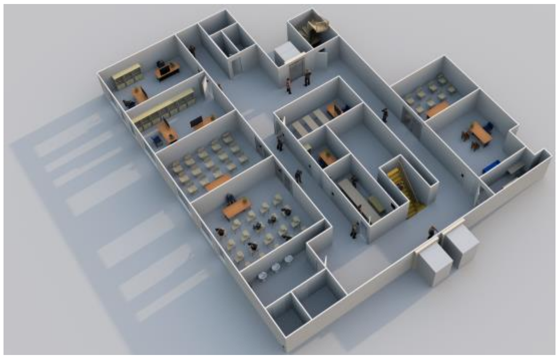
CAPS ENVML

The follow ENVML representations describe how the system is integrated in the UnivAQ’s buildings, in particular we have chosen to represent the first floor of Coppito 2 because we consider it as an interesting area of the building; it contains several lecture rooms, 3 laboratories and the secretarial rooms.

NOTES: full resolution images are available in the *“CAPS/ENVML”* directory.
Please also note that the devices (people counter, smart card readers, smoke detectors) have been represented using models that are not meant to be used for that purpose, so it may be possible that certain items are positioned in a “strange” way or are represented with wrong scale/dimensions.







As showed in the above pictures we have **People Counters** positioned in almost all rooms (on top of the door, as suggested by the device’s manufacturer), in particular in all the lecture rooms, the secretary rooms, and the laboratories.

Smart Card Readers have been positioned in restricted access rooms such as the laboratories and private offices, they are positioned at the side of the door.

Smoke Detectors have been placed in rooms that have a high degree of dangerous substances inside them such as the electronics lab. and the chemistry lab.
They have been placed on the walls, above the working benches even if the position is not really important because (at least for the devices that we have considered) they are able to cover pretty large areas.

CAPS Design Decisions

The developing of the CAPS models confirmed the architecture we had built in the previous deliverables. In particular, as we have concentrated all the logic on the “server side” (or better, in the Kafka Clients) the physical devices we can use are extremely simple because all that they have to do is to sense and forward the lecture to their Sensor Manager.
So, we have not made other decisions regarding the system’s architecture.

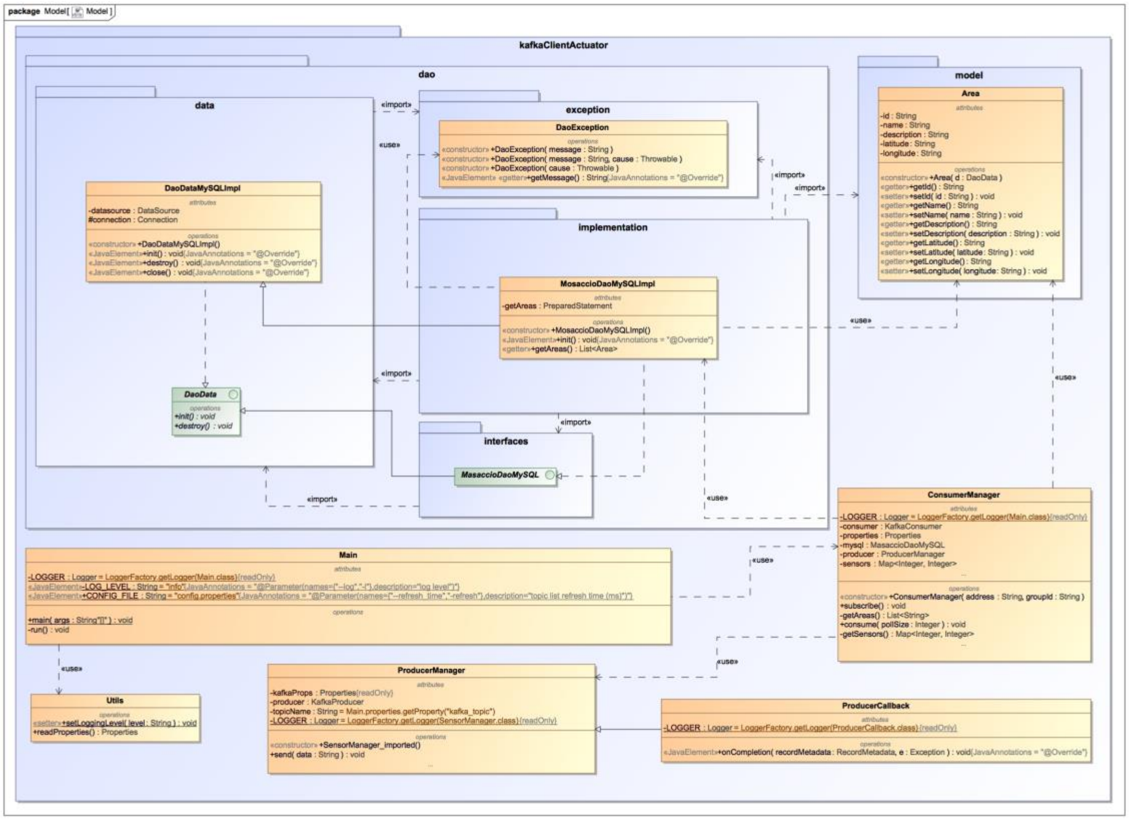
The only thing we would prefer is to deploy the devices in a wired environment, both with respect to the power supply and the network connection, even if this is not strongly required as the system is designed to tolerate network disconnections without data losses (for reference, the demo videos have been recorded in a wireless environment).

From Architecture to Code (v2)

Implementation - Actuation service

The video demo of the implemented service can be found here: https://youtu.be/YfUbq6JpVns
As described in the previous sections in this deliverable we have implemented the Actuation service.
The service is realized by the components that have been introduced in the sequence diagram.
We now describe how these components have been realized.

Kafka Client - Actuator


package Model[Model]
kafkaClientActuator
dao
data
exception
DaoException
operations
«constructor» +DaoException(message : String)
«constructor» +DaoException(message : String, cause : Throwable)
«constructor» +DaoException(cause : Throwable)
«JavaElement» «getter»+getMessage() : String{JavaAnnotations = "@Override"}
DaoDataMySQLImpl
attributes
-datasource : DataSource
#connection : Connection
operations
«constructor» +DaoDataMySQLImpl()
«JavaElement»+init() : void{JavaAnnotations = "@Override"}
«JavaElement»+destroy() : void{JavaAnnotations = "@Override"}
«JavaElement»+close() : void{JavaAnnotations = "@Override"}
implementation
MosaccioDaoMySQLImpl
attributes
-getAreas : PreparedStatement
operations
«constructor» +MosaccioDaoMySQLImpl()
«JavaElement»+init() : void{JavaAnnotations = "@Override"}
«getter»+getAreas() : List<Area>
DaoData
operations
+init() : void
+destroy() : void
interfaces
MasaccioDaoMySQL
model
Area
attributes
-id : String
-name : String
-description : String
-latitude : String
-longitude : String
operations
«constructor» +Area(d : DaoData)
«getter»+getId() : String
«setter»+setId(id : String) : void
«getter»+getName() : String
«setter»+setName(name : String) : void
«getter»+getDescription() : String
«setter»+setDescription(description : String) : void
«getter»+getLatitude() : String
«setter»+setLatitude(latitude: String) : void
«getter»+getLongitude() : String
«setter»+setLongitude(longitude: String) : void
«import»
«use»
ConsumerManager
attributes
-LOGGER : Logger = LoggerFactory.getLogger(Main.class){readOnly}
-consumer : KafkaConsumer
-properties : Properties
-mysql : MasaccioDaoMySQL
-producer : ProducerManager
-sensors : Map<Integer, Integer>
...
operations
«constructor» +ConsumerManager(address : String, groupId : String)
+subscribe() : void
-getAreas() : List<String>
+consume(pollSize : Integer) : void
-getSensors() : Map<Integer, Integer>
...
Main
attributes
-LOGGER : Logger = LoggerFactory.getLogger(Main.class){readOnly}
«JavaElement»-LOG_LEVEL : String = "info"{JavaAnnotations = "@Parameter(names={"--log","-l"},description="log level")"}
«JavaElement»+CONFIG_FILE : String = "config.properties"{JavaAnnotations = "@Parameter(names={"--refresh_time","-refresh"},description="topic list refresh time (ms)")"}
operations
+main(args : String"[]") : void
-run() : void
Utils
operations
«setter»+setLoggingLevel(level : String) : void
+readProperties() : Properties
ProducerManager
attributes
-kafkaProps : Properties{readOnly}
-producer : KafkaProducer
-topicName : String = Main.properties.getProperty("kafka_topic")
-LOGGER : Logger = LoggerFactory.getLogger(SensorManager.class){readOnly}
operations
«constructor» +SensorManager_imported()
+send(data : String) : void
...
ProducerCallback
attributes
-LOGGER : Logger = LoggerFactory.getLogger(ProducerCallback.class){readOnly}
operations
«JavaElement»+onCompletion(recordMetadata : RecordMetadata, e : Exception) : void{JavaAnnotations = "@Override"}

The component is composed by the following sub-packages such as:

- **dao**: that is equal to the **Kafka Client - Storing** class diagram (lacks of only the MongoDB part that isn’t needed for this component).
- **model**: that is equal to the **Kafka Client - Storing** class diagram.

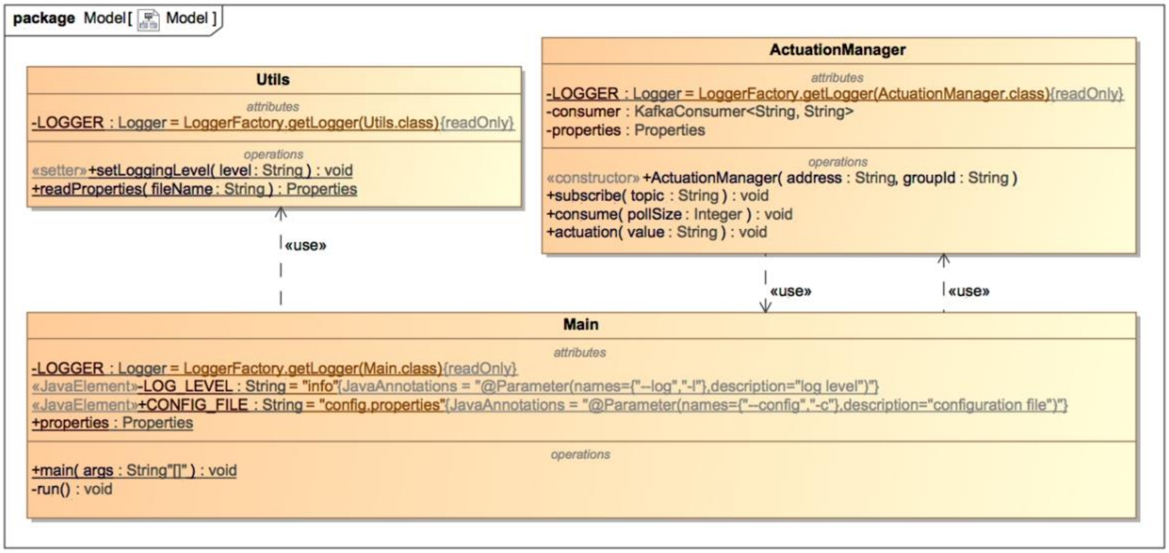
The class **ConsumerManager** contains the methods that allow to consume the messages published on the **Area Topics** that are retrieved (using the DAO infrastructure) from the relational database (following the same scheme described in the **Kafka Client - Storing**).
The difference is in that this component (**Kafka Client - Actuator**) instead of storing the data as the **Kafka Client - Storing**, it produces, through the **ProducerManager** class, a triggering message for the actuator that is related to the sensor that initially sent the message.

The **ProducerManager** class contains the methods that allow to produce messages in order to trigger the correspondent Actuator. The **ProducerCallback** is a **sub-class** of the **ProducerManager** and allow us to produce messages in asynchronous way.
The **ProducerManager** class follows the same scheme as the **SensorManager** class (described in the previous deliverable).

The **Utils** class contains general purpose static methods.

The **Main** class contains the initialization of the **properties** object that allows to parse properties files that can be used to set configuration variables such as the address of the Kafka Cluster, the credential to access the databases etc.
It also contains the instantiation of the **ConsumerManager** object and the main loop.

Actuation Manager


package Model[Model]
Utils
attributes
-LOGGER : Logger = LoggerFactory.getLogger(Utils.class){readOnly}
operations
«setter»+setLoggingLevel(level : String) : void
+readProperties(fileName : String) : Properties
ActuationManager
attributes
-LOGGER : Logger = LoggerFactory.getLogger(ActuationManager.class){readOnly}
-consumer : KafkaConsumer<String, String>
-properties : Properties
operations
«constructor»+ActuationManager(address : String, groupId : String)
+subscribe(topic : String) : void
+consume(pollSize : Integer) : void
+actuation(value : String) : void
«use»
«use»
«use»
Main
attributes
-LOGGER : Logger = LoggerFactory.getLogger(Main.class){readOnly}
«JavaElement»-LOG_LEVEL : String = "info"{JavaAnnotations = "@Parameter(names={"--log","-l"},description="log level")"}
«JavaElement»+CONFIG_FILE : String = "config.properties"{JavaAnnotations = "@Parameter(names={"--config","-c"},description="configuration file")"}
+properties : Properties
operations
+main(args : String"[]") : void
-run() : void

It follows the same configuration as the **SensorManager** described in the previous deliverable but instead of producing messages, it consumes messages and triggers the related actuator (physical device). It is composed by the following classes:

- **Utils**: contains general purpose static methods.
- **ActuationManager**: It contains the methods **subscribe()** and **consume(),** (that are the same that are used by the **Kafka Client - Storing** and **Kafka Client - Actuator**) used to subscribe and consume messages published in the **Kafka Cluster**. In this component, there is no need to retrieve data from the relational database because the topic is only one (the one related to the physical actuator) and it is provided as a parameter in the component’s configuration file.
  It also contains the method **actuation()** that simulates the actuation of the physical device.
- **Main**: is the class that initialize the component and reads the properties from the configuration file, it is also responsible for the call of the **ActuationManager** methods.

Improvements proposed in deliverable 2

Asynchronous production: we modified the way in which we publish the data on the cluster from a fire-and-forget way (default in Kafka) to an asynchronous way. It improves the throughput and if the message doesn’t arrive an exception will be thrown and subsequently handled to avoid that situation.

Asynchronous storage: we imported the asynchronous storage driver for MongoDB in order to not wait the DB when we call the method to store (insertOne()), this allow even greater throughput.

Commit management: we modified the Kafka commit management from an automatically management to a manual asynchronous management. By doing this we are sure that if something happens and the data is not successfully stored on MongoDB database, the whole data set will still be available on the Kafka Cluster. In fact, we commit if and only if we successfully store the data.

Parallelization: after some tests, we discovered that for the actual throughput we don’t need to parallelize the **Kafka Client - Storing** by manually spawning threads in a single process. It is way more convenient to simply run more instances of the client (with all the benefits exposed in the previous deliverable).

Presentation

The presentation has been committed in a separate file *“Presentation.pptx”.*