

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 D2 Template

Date	13/11/2017
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

Challenges/Risk Analysis	3
List of Assumptions	3
State of the art.....	5
Informal Description of your system and its Software/System Architecture	6
User Stories.....	8
Functional requirements	8
Extra-Functional requirements	8
User stories.....	9
Views and Viewpoints.....	11
UML static architectural view	12
UML dynamic architectural view	14
Design Decisions	15
From Architecture to Code	23
Implementation - Storing service	23
Performance Analysis	29
Future Improvements.....	29
Summary.....	30

- Code can be found in the “*implementation*” directory.
- All the diagrams/models are provided in full resolution in the “UML” directory.
- A demo video that shows the execution of the implemented service can be found at this link:
<https://youtu.be/RT7HMrQBuil>

Challenges/Risk Analysis

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 plan on 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 plan on 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 plan on using 2 databases at the same time in order to store data: the relational one will store structural information and the NoSQL will store raw data (<u>as the two are not communicating and they contain different kind of data we do not have synchronization problems</u>).

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.

The following architecture is designed to be adaptable to various situations. We are planning to instantiate the problem to the monitoring of the UnivAQ’s **existing** buildings.

The following are the main services we want to provide, based on similar systems we analyzed[1]:

- **Access control** (only certain users are allowed to enter certain areas)
- **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
- **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.).

So, in our instance:

Governance	→	UnivAQ’s rector
Areas	→	Buildings, floors, rooms
Citizens	→	Students

State of the art

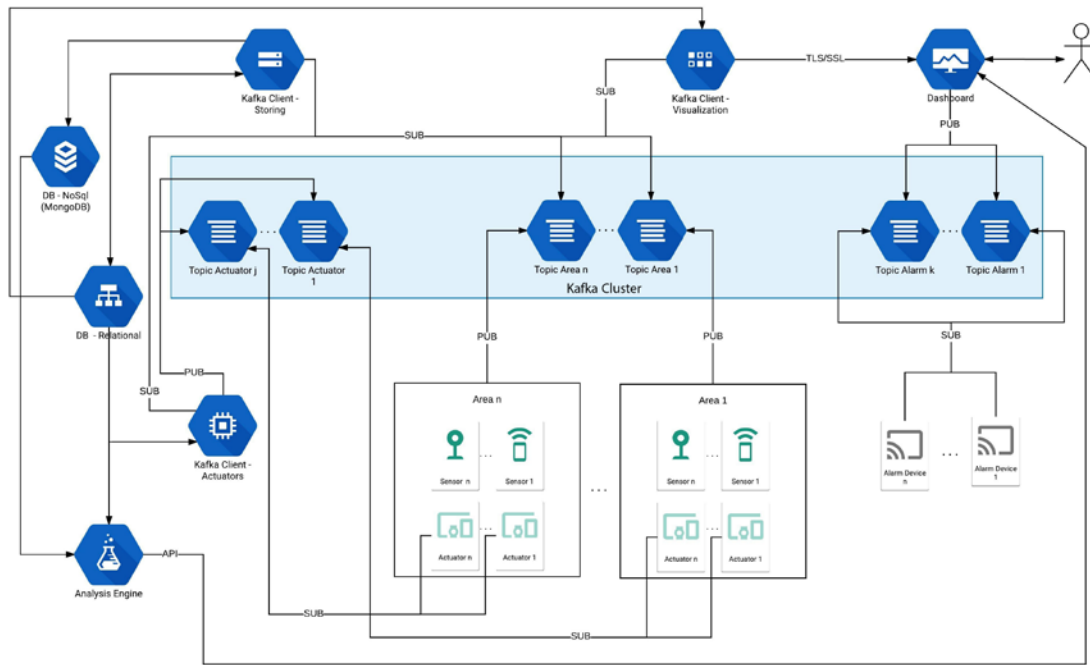
In the field of building monitoring and access control (our instance) the main provided services are [1] [2]:

- **Access control:** installation of card readers at every critical entrance point, which are connected to a central access control panel. You can assign cards or key fobs that grant access via a user interface or software.
- **Web-based Dashboard:** browser-based systems are easy-to-use and allow you to access your system at any time from any location with internet connectivity, including your mobile device. This means that you don't have to be tied to your security room to terminate access, lock down doors, or change a schedule.
- **Notifications:** you'll receive a textual notification for events like unauthorized entry and doors forced or left open.
- **Video recording:** linking your access control system with video surveillance allows you to see exactly who is entering and exiting your building.
- **People counters:** all buildings are required by law to have an emergency evacuation procedure complete with muster point locations. Access control systems can be designed to provide you with an accurate count of how many people are currently in the building. This allows you to quickly login and check the names and photos of everyone that was in the building before the evacuation.
- **Air quality monitoring:** air sensors provide novel ways to assess and characterize environments qualitatively and quantitatively in terms of pollution, and human exposure. More specifically, air sensors offer a rare opportunity to assess air quality of indoor environments in real-time. Most IAQ (Indoor Air Quality) sensors, with installed communication protocols, are able to detect and transmit data in real-time to digital platforms, e.g., to a server, PC or smartphone, which in turn broadcast the data to a designated web portal for real-time analysis and visualization.

[1] - <http://www.spottersecurity.com/services-access-control-systems/>

[2] - <http://www.sciencedirect.com/science/article/pii/S0048969716307124>

Informal Description of your system and its Software/System Architecture



The system is composed by a **Sensor Network**, an **Actuators Network**, a **Kafka Cluster**, two **DB** (NoSQL and Relational), an **Administration Dashboard** and three **Kafka Clients** (Storing, Visualize and Actuator).

The **Sensor Network** represent the whole sensing subsystem: it senses the environment and communicates the gathered data to the rest of the system publishing on **Kafka Topics**. There will be a Topic for each area (an area may be a building, square, quarter, etc.).

The **Kafka Cluster** is an aggregation of **Kafka Brokers**. A broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers (clients), responding to fetch requests for partitions and responding with the messages that have been committed to disk.

The **Kafka Client - Storing** deals with the asynchronous storage of all the raw data collect by the system, by subscribing to the various Area Topics.

The **Kafka Client - Visualize** is responsible to provide refined data to the dashboard and to highlight sensors readings that are out of certain security bounds (those bounds are retrieved from the relational db).

The **Kafka Client - Actuators** reads the data from the sensor network by subscribing to Area Topics and publishes messages on Actuators Topic when certain sensor readings are received, activating the respective actuator(s).

The system stores the raw data on a **NoSQL DB** (MongoDB), this allows for a high flow of data, every reading from a sensor is saved as a *json* file containing the identifier of the sensor, the identifier of the area the sensor belong and the actual reading.

The **Relational DB** is used to store the structure and organization/disposition of the various devices (sensors and actuators) across the areas. It also stores the data regarding the user’s authorizations to access certain areas and the various services provided by the system.

The **Administration Dashboard** receives data from the **Kafka Client - Visualize** through a secure connection, then it displays the received data to the user. It also shows a warning when there are sensor’s readings that are not within certain “safe” bounds. In those cases, the systems ask the user for a check before emitting an alarm message that, in case of confirmation is published on Alarm Topics that will trigger the respective Alarm Devices.

The **Actuator Network** represent the whole actuation subsystem: the actuator waits for actuation messages to be published on its Actuation Topic, when such message is published the actuator activates and publishes on the Area Topic that it has performed a certain action.

The **Analysis Engine** is responsible to perform batch/background data analysis on the previously stored data, and to make available an interface to allow the dashboard to display such data.

Architectural Pattern

The main architectural pattern used in the system is the Publish/Subscribe pattern. Publish/subscribe messaging is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/Sub systems often have a broker, a central point where messages are published, to facilitate this.

Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. This pattern provides greater network scalability and a more dynamic network topology.

To implement this pattern we chose to use **Apache Kafka**. Apache Kafka is often described as a “distributed commit log” or more recently as a “distributing streaming platform.” A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

User Stories

Functional requirements

FR1: the system has to monitor several areas.

FR1.1: the system exploits a **sensor** network in order to read data.

FR1.2: the system has to **collect** raw data from sensors.

FR1.3: the system has to **store** the sensed data.

FR1.4: the system has to **organize** the collected data and has to provide a way of checking users accesses on restricted areas.

FR1.5: the system has to **analyse** and produce statistics about the collected data.

FR2: the system has to provide a set of visualization tools in order to show the current state of the monitored areas and highlight crucial situations.

FR3: the system has to provide a set of actuators that support the human user actions in case of emergencies.

Extra-Functional requirements

EF1: dependability

EF1.1: availability - the system has to correctly and continuously operate under the regular hardware state/condition.

EF1.2: safety - the system has to avoid false alarms and other unwanted behaviors that may be dangerous for user's safety.

EF1.3: security - the system must ensure that the data and the services are accessed only by the authorized users.

EF1.4: fault-tolerance - the system has to be operative even in case of disasters (with an eventual degraded mode) and has to guarantee that no critical messages are lost and delivered in at most 5 seconds.

EF2: performance - the system has to handle 40.000 messages per hour coming from (up to) 2.000 sensors, the sensed data are collected within different time ranges (from a few seconds to few minutes, with an average of one message every 180 seconds).

EF3: scalability - the system should be easily upgradable (in terms of new areas and sensors added to the network).

EF4: usability - the system has to be easy to use for the end user.

User stories

- Citizen

1. *<citizen>, <FR2, FR3, EF1.2, EF1.1, EF4>*

As a citizen, I want to feel safe: in case there's an emergency I would like to receive a notification of how to behave and possibly which areas of the city it is best to avoid.

2. *<citizen>, <FR3, EF1.3, EF1.1, EF4>*

As an authorized user, I want to be able to access the areas for which I have permission.

3. *<citizen>, <FR2, EF1.2, EF4>*

As a citizen, I want to be informed about the crowding of the structures: I would like to avoid crowded areas because I'd prefer to avoid crowded rooms.

- Emergency Operator

1. *<emergency operator>, <FR2, FR3, EF1.1, EF1.4, EF2, EF4>*

As a safety operator, when there's an emergency, I need to be alerted by a notification that indicates me where the emergency is, so I can go to help people as soon as possible.

2. *<emergency operator>, <FR2, EF1.1, EF1.4, EF4>*

As a safety operator, I want to be informed about the current situation of the place where I am, to monitor it and avoid critical situation.

- Security Manager

1. *<security manager>, <FR3, EF1.1, EF1.3, EF1.4>*

As a security manager, I must ensure that only the authorized people access the restricted areas.

2. *<security manager>, <EF1.3>*

As a security manager, I have to make sure that the collected data is not accessed by unauthorized people.

3. *<security manager>, <EF1.3>*

As a security manager, I have to guarantee the privacy for the people and keep their sensible data safe.

- Sensor Network Administrator

1. *<sensor network administrator>, <FR1.1, FR1.2, EF1.1, EF1.2, EF1.4, EF2>*

As sensor network administrator, I have to ensure that all the sensors operate correctly.

2. *<sensor network administrator>, <EF3>*

As sensor network administrator, I want to be sure that if the number of sensors increases, the system still works as intended.

- Database Administrator

1. *<database administrator>, <FR1.3, FR1.4, EF1.1, EF1.4, EF2>*

As a database administrator, I have to ensure that the databases are correctly working at any time.

2. *<database administrator>, <FR1.3, FR1.4, EF1.3>*

As a database administrator, I have to be sure that the data is not accessed by unauthorized people.

- Governance

1. *<governance>, <FR1.5, EF1.1, EF1.2, EF1.3>*

As the government, I'm concerned about citizens safety, security and information retrieved by data analysis.

- System Administrator

1. *<system administrator>, <FR1, FR2, FR3, EF1, EF2, EF3, EF4>*

As a system administrator, I'm concerned about every aspect of the system.

Views and Viewpoints

Stakeholders:

- **Citizen**
- **Emergency Operator**: is the user's class that is responsive into the safety of the citizen.
- **Security Manager**: is the user responsible of the cyber-security of the system.
- **Sensor Network Administrator**: is the user responsible of the proper functioning of the sensors and the network where the MASACCIO system is deployed.
- **Database Administrator**
- **Governance**: is the customer who requested and paid for the deployment of the system.
- **System Administrator**: is the user responsible of the whole system.

	Citizen	Emergency Operator	Security Manager	Sensor Network Administrator	Database Administrator	Governance	System Administrator
Security	X		X		X	X	X
Privacy	X		X		X	X	X
Sensing				X			X
Emergency Response		X					X
Energy Consumption				X			X
Networking & Communication				X			X
Usability	X	X					X
Dependability	X	X		X			X
Performance	X	X		X			X
Costs				X		X	X
Citizen Engagement						X	X
Data Analysis					X	X	X

Sensing Subsystem: every sensor is managed by a microcontroller capable of running Java code. Those microcontrollers are referred as *Sensor Managers*. Those components are responsible to fetch the sensors readings and publish messages containing such data to the respective *Area Topic*.

Actuation Subsystem: the Actuation Subsystem is composed by 2 parts:

- **Kafka Client – Actuator:** this component is a kafka client who listens (is subscribed) to all the *Area Topics*, when it reads a message who contains a sensor reading which is supposed to trigger an actuator (this information is stored on the relational database) it published a message on the respective *Actuator Topic*.
- **Actuator Managers:** it follows the same structure as *Sensor Managers*. They are responsible of listening, by subscribing to the respective *Actuation Topic*, for actuation messages. Those messages are generated by the *Kafka Client – Actuator*, when an actuation message is published on *Actuator Topic X*, it is read by the *Actuator Manager Y* who triggers the *Actuator* (physical device) *Z*.

Alarm Subsystem: every alarm device is handled by a microcontroller (just like in the *Sensing Subsystem*). Those microcontrollers form the *Alarm Managers* component, each element of this component listens (is subscribed) to the respective *Alarm Topic*, when a triggering message is received on the Topic of interest, the respective alarm device is triggered by its microcontroller.

Analysis: is the subsystem that contain the *Analysis Engine* component. This component contains the logics that is responsible to compute analytics on the gathered data. It can be accessed by a dedicated API by the other component of the system.

Dashboard: contains the component that are dedicated with the visual representation of the data. It is composed by the following components:

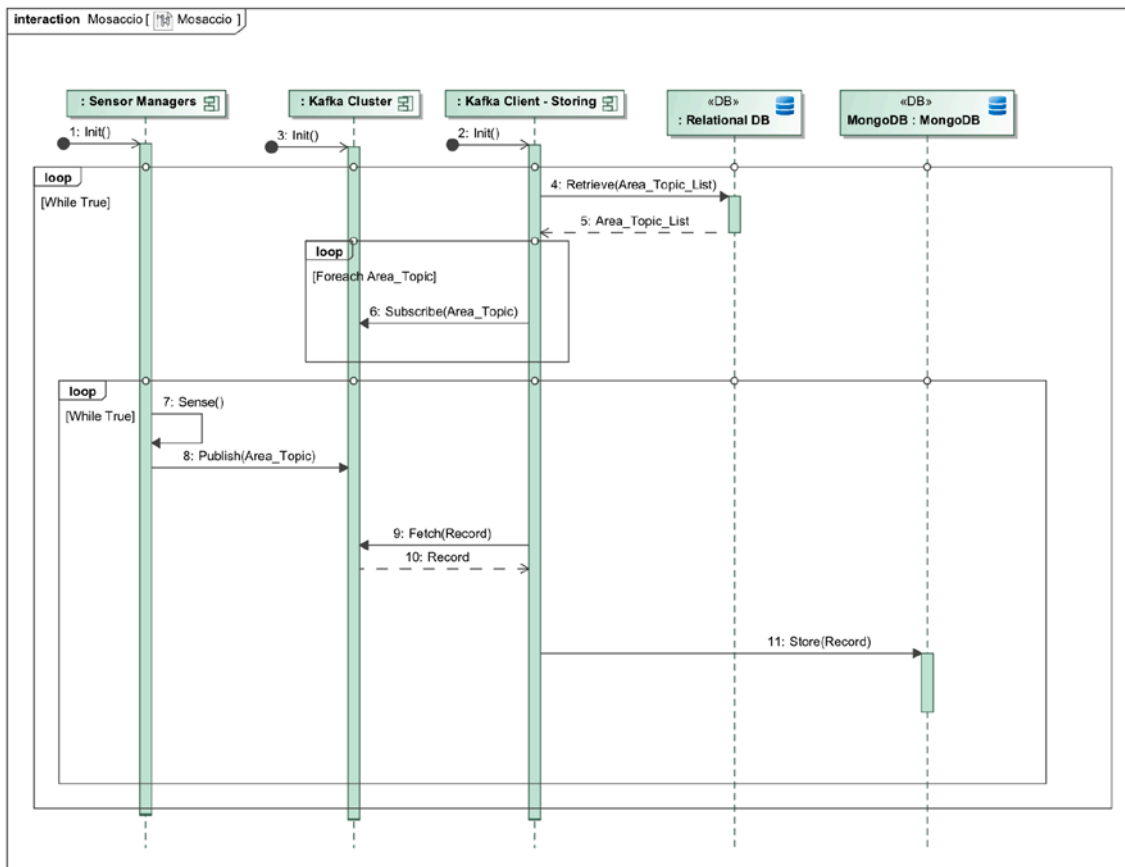
- **Kafka Client - Visualization:** this component is a kafka client who is subscribed to all the *Area Topics*. It analyzes all the incoming sensor's lectures and checks if the sensed data is inside certain defined "safe" bounds (that are retrieved from the relational db). If a reading is out-of-bounds, then a notification is displayed on the dashboard itself.
- **Dashboard:** is the component that allows the operators to visually see the incoming data (in real-time, received by the *Kafka Client – Visualization*) and to receive alarm notifications. When an alarm notification is received, the situation is checked by an operator who confirm/reject the dispatch of the alarm associated to that situation. If the alarm is confirmed the triggering message is published on the respective *Alarm Topic*. The *dashboard* also accesses the analysis engine to allow the users to visualize the result of more complex analyses.

Relational DB: is the database who contains the data that refers to the structure of the system/system's organization. For instance: sensor's positions, user's authorizations, etc.

MongoDB: is the database that contains only and all the sensed data coming from the sensors.

UML dynamic architectural view

The following sequence diagram describes the storing process. That is the procedure who is responsible to the collection of the sensed data and the storing on the databases. We decided to implement this feature because it is the core functionality on which all the other ones are based on. This allows us to test the performance/usability/behavior of the system.



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

During the initialization phase the *Kafka Client – Storing* fetches the list of all the *Area Topics* from the *Relational database*. Then, it subscribes to all the retrieved *Area Topics*, now it is ready to listen for messages.

At this point the sensor is up and running and it starts to sense the environment, and (through its manager) it publishes the message containing the sensed data to its *Area Topic*.

The message is received from the *Kafka Client – Storing* which then asynchronously deserializes it to a *.json* file and stores it in the *MongoDB* database.

Then the cycle repeats, but after a configurable amount of time, the *Kafka Client – Storing* pauses and updates the list of all the *Area Topics* from the *Relational database*, then continue his normal execution.

Obviously, in this representation, we are looking at the execution from the point of view of a single sensor who is sending its lectures, but in the real execution there will be several sensors and several sensors managers which will publish their message to the *Area Topics*.

By “*publishing to the Area Topic*” we are implicitly saying that the device transmits its message to the cluster through its **Sensor Manager**.

Design Decisions

The rationale by which we decided the following strategies/tools are described at the end of the document using Architectural Knowledge design SPace Modeling (AK-SPAM) tables.

- **Storing**

The storage is achieved by utilizing two kinds of databases. The NoSQL database (we are planning to use MongoDB) will be used to store raw data, or more in general all the data coming from the various sensors and actuators. The reading of the sensor is stored as a *.json* file containing the id of the sensor, the id of the area the sensor is operating in and the sensor’s reading. We chose a NoSQL db because we want to be able to store huge amount of data with the smallest amount of latency we can achieve. MongoDB as a documental db allow us to organize the data in a way that is particularly suited for our use. Allowing the definition of time to live for the stored data, decentralization, great scalability and replication (data redundancy and automatic failover).

In particular in our system we have a volume (in average) of 40.000 messages per hour, that means 960.000 per day and 432.000.000 per 15 months, that is the suggested time interval in which data should be kept to maximize the analytic value [3].

Moreover, it is easier for us to store sensed data in an unrelated way using a NoSQL db then storing them in a relational one.

The NoSQL db is coupled with a relational db that is used to store the data regarding the authorizations, sensor-area organization, and other general information about the system that needs to have a strict structure.

A dedicated Kafka Client is appointed to the storing of the incoming data. It is subscribed to all the area topics and it is only responsible of inserting all the incoming readings into the NoSQL db, it uses the relational one to verify sensors dispositions etc.

As the client is dedicated we are removing load and overhead from the other component of the system allowing for even more performance.

As requested, a REST interface is used to wrap the system for external (by external we mean “outside” of the system) querying.

- **Sensors organization**

We chose to organize the sensors in areas, each area describes a geographical zone around the city and contains different sensors and actuators. This allow the system to have **great scalability** as if we need to add more sensors/actuators we just need to install them and let them publish their messages to the dedicated area topic.

- **Data acquisition**

We discarded the possibility to use a **pure event driven** approach because, after analysing the pattern we identified several criticalities such as: resilience is more difficult to achieve in an pure event-driven system due to the short-lived nature of event consumption chains, when processing the event, the listeners have to immediately react to and transform the result, these listeners typically handle success or failure directly and in the sense of reporting back to the original client [6]; the flexibility of event driven architecture raises complexity when the application grows. The reason is that one event triggers a range of routines, and with more events it becomes unpredictable. The type and amount of routines is not specified causing some components to behave in an unpredictable manner.

An event driven architecture is easy to develop but hard to control [8].

In **Pub/Sub** we have strong advantages that strictly concerns our system, such as [7]:

- dynamic targeting: instead of maintaining a roster of peers that an application can send messages to, a publisher will simply post messages to a topic.
- decoupling and scalability: publishers and subscribers are decoupled and work independently from each other, which allows you to develop and scale them independently. You can decide to handle orders one way this month, then a different way next month.
- simplified communication: the Publish Subscribe model reduces complexity by removing all the point-to-point connections with a single connection to a message topic.

One of the strong disadvantages of the Pub/Sub architectural pattern is that there could be difficulties when it is decided to modify the message structure, but this does not apply to our instance as every sensor microcontroller will always publish the same message in the same format.

Plus, event driven techniques can be adopted on top of the Pub/Sub skeleton if ever needed.

We chose to follow the PubSub architectural pattern and to implement it using **Apache Kafka**, because as a framework, adds important features to the classic Pub/Sub pattern.

Kafka allows to implement secure real-time streaming data pipelines that reliably get data between systems or applications and secure real-time streaming applications that transform or react to the streams of data [4].

Kafka can be used to stream data, as a message dispatcher and as a storage system.

Security is achieved by encrypting the data transferred between brokers and clients, between brokers, or between brokers and tools using SSL/TLS.

Kafka also allow for a throughput of millions of records/second [5], replication, redundancy, scalability, decentralization and fault tolerance that are the crucial points of our system.

Moreover, Kafka is extremely immediate to use even if it requires a certain degree of expertise to exploit all of its features.

The sensors are organized in areas, each area has its dedicated topic, so the sensor belonging to the area x, publish its reading to the topic of the area x, the published message consists of the id of the sensor itself and the actual reading.

- **Visualization and analysis**

A dedicated Kafka Client is responsible to read the incoming data from the area topics (just as the client dedicated to the storing) and transmit them with a secure connection to a dashboard.

The data can be subjected to analysis and refactoring before being sent to the dashboard. The data is displayed in form of graphics, having a dedicated client that directly (i.e. without necessarily going through the database(s)) allow us to eliminate the overhead that would have been present otherwise.

Sensors readings that are out of certain “safe” bounds (retrieved from the relational db) will trigger a warning notification to be displayed in the dashboard.

An operator that sees the warning can activate an alarm. Triggering an alarm is achieved by publishing on the topic dedicated to that alarm (or alarms) device(s), these devices will be activated by the published messages.

Data could be analysed even in an “offline” way by accessing the data through the REST APIs.

- **Actuation**

Another dedicated Kafka Client is subscribed to the area topics. When it reads a message that implies the triggering of an actuator, it publishes to the topic related to that actuator (also more actuators could be subscribed to that topic) activating it (or them).

As for the sensors organization, this pattern allows for a high factor of scalability, as for introducing more actuators is enough to install them, allow them to publish on their topic and insert the record *sensors reading* -> *actuation* into the relational db.

[3] - <https://www.datadoghq.com/blog/monitoring-101-collecting-data/>

[4] - <https://kafka.apache.org/intro>

[5] - <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>

[6] - <https://www.reactivemanifesto.org/glossary#Message-Driven>

[7] - <https://aws.amazon.com/pub-sub-messaging/benefits/>

[8] - <https://www.linkedin.com/pulse/event-driven-architecture-michel-herszak/>

[9] - <https://benchmarksgame.alieth.debian.org/u64q/python.html>

Concern (Identifier: Description)		Con#1: How can we gather the data from the sensors?
Ranking criteria (Identifier: Name)		Cr#1: Reliability - all messages must be received Cr#2: Fault-tolerance - the system must be resilient to failures and critical conditions Cr#3: Performance - the messages must be received in real-time Cr#4: Scalability - the system must be easily scalable (by means of adding sensors/areas)
Options	Identifier: Name	Con#1 - Opt#1: PubSub (Apache Kafka)
	Description	The data is collected by allowing the sensors to publish messages containing their readings to specific topics, several clients (dedicated to different aspects of the system) can subscribe to these topics in order to retrieve the data.
	Status	decided
	Relationship(s)	-
	Evaluation	<p>Cr#1: Kafka guarantees that every message that is published is then received by the subscriber (sooner or later).</p> <p>Cr#2: Kafka implements several fault-tolerance techniques such as:</p> <ul style="list-style-type: none"> • in case of a cluster fails another one takes its place, with all the updated data, every member of the cluster has all the replicated/up-to-date data; • data can be saved for an arbitrary amount of time on the brokers; • for a topic with replication factor N, it will tolerate up to N-1 server failures without losing any records committed to the log; • Kafka replicates the log for each topic's partitions across a configurable number of servers. This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures. • ... <p>Cr#3: Kafka allows to transmit order of millions of messages per second (that is much more than the required 40.000/hour).</p> <p>Cr#4: Kafka allows our system to have great scalability: if we need to add more sensors/actuators we just need to install them and let them publish their messages to the dedicated topic.</p>
	Rationale of decision	We chose this option as it satisfies all the criteria and it provides out-of-the-box dependability features that would have been manually implemented otherwise.
	Identifier: Name	Con#1 - Opt#2: PubSub (plain MQTT)
	Description	The data is collected by allowing the sensors to publish messages containing their readings to specific topics, several clients (dedicated to different aspects of the system) can subscribe to these topics in order to retrieve the data.
	Status	rejected
	Relationship(s)	-
	Evaluation	<p>Cr#1: MQTT guarantees that every message that is published is then received by the subscriber (sooner or later).</p> <p>Cr#2: MQTT as a machine to machine protocol <u>lacks</u> of dedicated fault-tolerance techniques (it requires wrapping).</p> <p>Cr#3: MQTT allows to transmit order of millions of messages per second (that is much more than the required 40.000/hour).</p> <p>Cr#4: MQTT allows our system to have a good enough scalability: if we need to add more sensors/actuators we just need to install them and let them publish their messages to the dedicated topic.</p>
	Rationale of decision	We rejected this option because it requires wrapping to deliver fault-tolerance.

	Identifier: Name	<i>Con#1 - Opt#3: asynchronous task queue/event based (Celery)</i>
	Description	<i>Celery it's a task queue with focus on real-time processing, while also supporting task scheduling. It is based on MQTT protocol it communicates via messages, usually using a broker to mediate between clients and workers. The data are sent to the workers, which triggers tasks that will store/make computations of the received data.</i>
	Status	rejected
	Relationship(s)	-
	Evaluation	<i>Cr#1: The tasks are guaranteed to be executed sooner or later after the message is sent</i> <i>Cr#2: Celery <u>does not</u> implement fault tolerance techniques, so it would be necessary to manually develop them from scratch.</i> <i>Cr#3: Celery allows for millions of tasks per second (that is much more than the required 40.000/hour).</i> <i>Cr#4: Celery allows to add workers as needed so to achieve good enough scalability.</i>
	Rationale of decision	<i>We rejected this option because it not satisfies Cr#2, even if it provides greater performances with respect the other solutions, but in our instance performance is not an important requirement as fault tolerance and dependability in general.</i>

Concern (Identifier: Description)		Con#2: How can we store and organize the data?
Ranking criteria (Identifier: Name)		Cr#1: Data organization - the data must be easily organisable Cr#2: Scalability - the system must be easily scalable (by means of amount of incoming data) Cr#3: Performance - amount of data that can be stored/how fast can we retrieve-insert them
Options	Identifier: Name	Con#2 - Opt#1: NoSQL DB + relational DB
	Description	The sensors readings are stored in the NoSQL DB, the data regarding the organization of the system and the authorizations are stored in the relational DB.
	Status	decided
	Relationship(s)	-
	Evaluation	Cr#1: The use of a relational DB allows to easily to store the data regarding the system structure. Cr#2: The use of a NoSQL DB allows us to store billions of records without performance impacts. Cr#3: The NoSQL DB delivers noticeable performance at massive scale: millions of ops/sec, 100s of billions of records, huge amounts of data. Also, splitting the load to two different DBs allows us to avoid further overhead that would derive by using only a DB (we can query the relational DB without impacting on the NoSQL one).
	Rationale of decision	We chose this option as it optimally satisfies all the criteria.
	Identifier: Name	Con#2 - Opt#2: NoSQL DB
	Description	Both sensor's readings and system structure data is stored on the NoSQL DB.
	Status	rejected
	Relationship(s)	-
	Evaluation	Cr#1: Storing strongly related data in a NoSQL environment is not always easy. Cr#2: The use of a NoSQL DB allows us to store billions of records without performance impacts. Cr#3: The NoSQL DB delivers noticeable performance at massive scale: millions of ops/sec, 100s of billions of records, huge amounts of data; but all the load is assigned to the NoSQL DB.
	Rationale of decision	We rejected this option because it not optimally satisfies the Cr#1.
	Identifier: Name	Con#2 - Opt#3: Relational DB
	Description	Both sensor's readings and system structure data is stored on the Relational DB.
	Status	rejected
	Relationship(s)	-
	Evaluation	Cr#1: The use of a Relational DB allows to easily to store the data regarding the system structure. Cr#2: The use of the Relational DB does not allow us to easily store huge amount of data, at least not without performance impact. In our system, there is an expected amount of 423 millions of records that would impact the performances if stored on the relational DB, plus we plan to perform analysis on this data and relational DB are not suited for this. Cr#3: The Relational DB usually does not perform well with high amount/volume of data/traffic.
	Rationale of decision	We rejected this option because it not satisfies the Cr#2 and Cr#3.

Concern (Identifier: Description)		Con#3: Where can we store the data? (in our instance)
Ranking criteria (Identifier: Name)		Cr#1: Privacy/Security Cr#2: Dependability - the data must be easily organisable Cr#3: Costs
Options	Identifier: Name	Con#3 - Opt#1: Everything in local storage
	Description	All the data are stored in the UnivAQ's servers.
	Status	decided
	Relationship(s)	-
	Evaluation	Cr#1: The data is stored in a private and closed environment. Cr#2: The dependability depends on the dependability of the UnivAQ's infrastructure. Cr#3: The costs are near to 0 as we are using an existing infrastructure.
	Rationale of decision	We chose this option as it optimally satisfies all the criteria: we have low costs, we are certain that the data is only accessed by us and we have a good enough level of dependability.
	Identifier: Name	Con#3 - Opt#2: Everything in the cloud
	Description	All the data are stored in the cloud.
	Status	rejected
	Relationship(s)	-
	Evaluation	Cr#1: Data is located in a not directly controlled environment and it is managed by third parties. Cr#2: Maximum level of dependability assured by the cloud provider but it is strongly dependant to the availability of an active internet connection (in case of disasters we could not have an active internet connection). Cr#3: The costs could be high with respect to the amount of data we want to store.
	Rationale of decision	We rejected this option because it does not optimally satisfy Cr#1, Cr#2 and Cr#3.
	Identifier: Name	Con#3 - Opt#3: Part of the data in local storage and part on the cloud
	Description	Some of the data is stored in the UnivAQ's infrastructure and some is stored in the cloud.
	Status	rejected
	Relationship(s)	-
	Evaluation	Inherits the problems from the evaluation of Con#3 - Opt#2
	Rationale of decision	We rejected this option because it not optimally satisfies Cr#1, Cr#2 and Cr#3.

Concern (Identifier: Description)		<i>Con#4: Which programming language should we use?</i>
Ranking criteria (Identifier: Name)		<i>Cr#1: Performance</i> <i>Cr#2: Support</i> <i>Cr#3: Speed of development</i>
Options	Identifier: Name	<i>Con#4 - Opt#1: Java</i>
	Description	<i>The components are developed using Java</i>
	Status	<i>decided</i>
	Relationship(s)	<i>Con#1</i>
	Evaluation	<i>Cr#1: Java is arguably one of the most performant programming languages available.</i> <i>Cr#2: Kafka is developed in Java and provides official APIs and documentation for Java.</i> <i>Cr#3: Java is a pretty verbose language and development can be tedious and slow, even if several frameworks (such as Spring - for REST APIs) are able to significantly increase the speed of development.</i>
	Rationale of decision	<i>We chose this option mostly because we have official APIs and plenty of documentation, plus Java is totally portable and in an IOT settings it's a relevant advantage.</i>
	Identifier: Name	<i>Con#4 - Opt#2: Python</i>
	Description	<i>The components are developed using Python</i>
	Status	<i>rejected</i>
	Relationship(s)	<i>Con#1</i>
	Evaluation	<i>Cr#1: Python is recognized to be one of the slowest programming languages [9].</i> <i>Cr#2: There are not official APIs for Kafka, only community developed ones.</i> <i>Cr#3: Development in Python is incredibly fast, what is done with tens of lines of code in other programming languages can be done with a few lines in Python.</i>
	Rationale of decision	<i>We rejected this option because even if we are able to produce prototypes in a faster way we have not the performance of other languages and we cannot use official APIs.</i>

From Architecture to Code

Implementation - Storing service

The storing process is realized by the components that have been exposed in the previous sequence diagram.

A demo video that shows the execution can be found at this link: <https://youtu.be/RT7HMrQBuil>

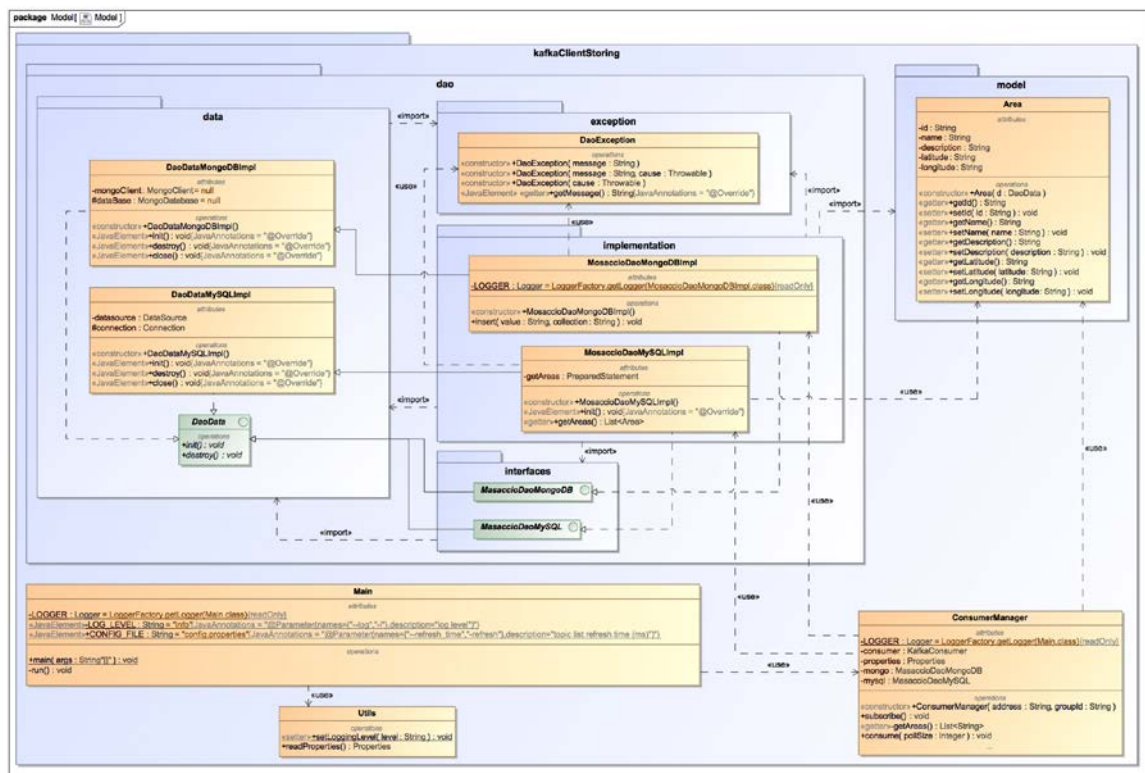
In the showed execution there are 3 machines:

- Machine 1 hosts the mysql relational database and the NoSQL MongoDB database
- Machine 2 hosts the Kafka Client – Storing and the Kafka Cluster
- Machine 3 simulates 4 sensors with their Sensor Managers

A more detailed description can be found directly in the video annotations/description and in the sequence diagram provided in the previous sections.

We now show the structure of the implemented components:

the following class diagram describes the composition of the **Kafka Client - Storing** component:



The component is composed by several sub-packages such as:

- **dao**: to implement the communication with both the databases it has been used the DAO (Data Access Object) design pattern. It makes available interfaces that allow to map the business logic calls to the persistent data model, it is composed by:

- **dao.data:** contains the classes who implements the connection to the dbs using connection pooling techniques, when possible (mongodb, for instance, do not allow to use connection pooling as it uses its own protocol).
 - **dao.exception:** contains the implementation of personalized exceptions types.
 - **dao.interfaces:** contains the interfaces that will be implemented by the classes inside the **dao.implementation** sub-package, they allow to define certain fixed behaviors that these classes must follow.
 - **dao.implementation:** contains the classes who implement the interfaces inside the **dao.interfaces** sub-package. These classes are used to physically interact with the databases (they perform the queries and return the results).
- **model:** this sub-package contains the representation of the database entities, this component (Kafka Client - Storing) only need to interact with one entity: **Areas** (please, check the ER schema for more details). So, in this sub-package we have the class **Area** which represent the corresponding entity on the relational database.

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.

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.

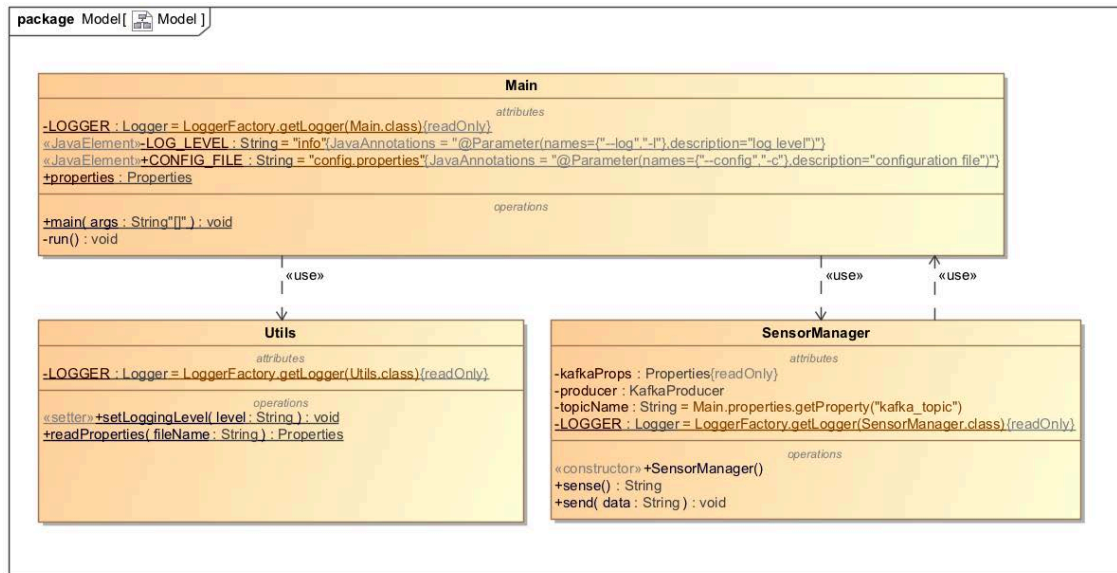
The flow of the computation follows what has been described on the sequence diagram.

Now, we better specify the behavior of the **Kafka Client - Storing** component (:

- first, the instantiation of the **ConsumerManager** object is followed by the fetch of the area topics names from the relational database (using the DAO infrastructure)
- subsequently the client subscribes to all of them (this happens inside the method *subscribe()*).
- now, the client is put in listen mode with the method *consume()*, this allow the client to listen for new messages on **all** the available topics.
- when a message is received its payload is stored on the Mongo database (using the DAO infrastructure).
- this cycle is repeated every X seconds (the amount can be changed): *this allow the system to adapt itself to newly added area topics*, in fact, when the cycle restarts, the client fetches the list of the topics and it subscribes to them again, so there is no need of a manual restart; plus, thanks to kafka every message that has been not received because of any kind of unexpected downtime is safely stored in the cluster and it will be retrieved as soon as the client (Kafka Client - Storing) comes back online.

The most important aspect of this implementation is that *it is enough to run multiple instances of the application to have an automatic load-balance among the instances with respect to the topics/partition assignment*, plus every instance can be executed on a totally different machine in a totally different geographic area. It is enough to provide the address of the cluster and the databases through command line arguments.

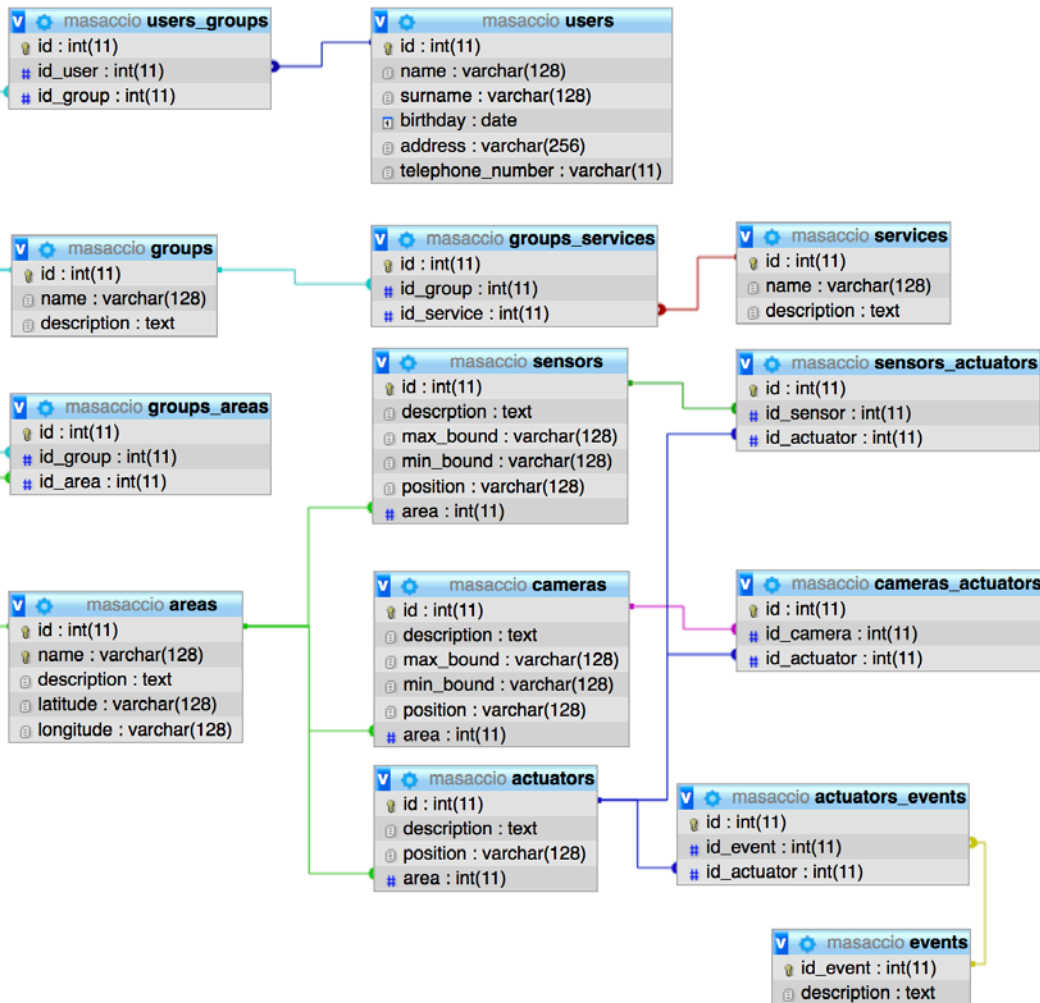
The following class diagram describes the composition of the **Sensor Manager** component.



The component is composed by the following classes:

- **Main**: it is responsible to initialize the **SensorManager** object and to run it.
- **Utils**: it contains some useful functions used by the Main class (such as the function to change the logging level and the function to read the configuration file).
- **SensorManager**: it contains the function to **sense the world** (this function will be the function that will communicate with the sensor: now the reading of the sensor is simulated by generating a random alphanumerical string) and the function that is responsible to the publishing of the messages to the respective Kafka topic; each message is a string representing a .json file containing the id of the sensor, the reading and the timestamp.

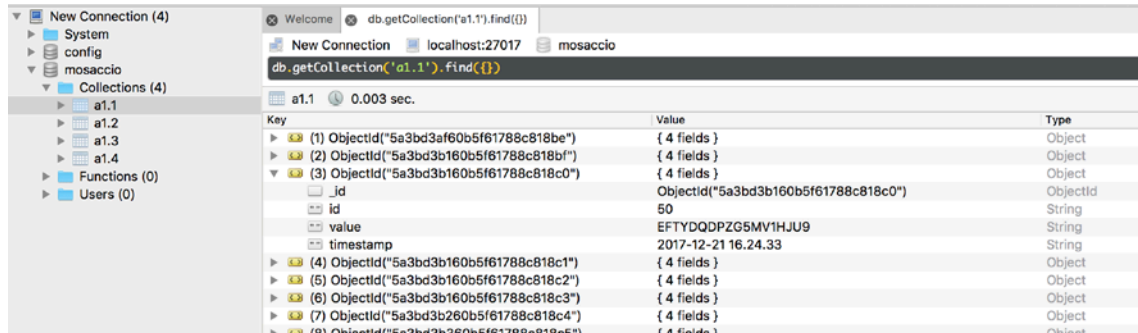
The **relational database** is structured as follows (we used mysql):



- The **users** table contains all the data about people that have a profile on the system (we are talking about authorized people). We store the name, surname, birthday, address, telephone's number and we distinguish every registered person with a unique id.
- The **groups** table contains all the info about the group of peoples (or categories) that have responsibility and special permission in some kind of area.
- The **services** table contains all the services that can be accessed by authorized groups.
- The **users_groups** table contains different pairs of user's id and group's id and identifies which person belong to a determinate group.
- The **groups_services** table contains different pairs of group's id and service's id and represent which services are available for a specific group.
- The **groups_areas** table is used to store the ids of groups that have some kind of authorization in a specific area, identified by the respective id.
- The **areas** table is used to store all the data about the different areas that the system monitors.
- The **cameras** table contains all the data about every camera that is deployed in the system. Indeed, in this table there is a field that contains the id of the area in which the camera has been positioned.

- The **sensors** table contains all the data about every sensor that we have used for each area (this table contains also a field with the id of the area in which the sensor has been positioned).
- The **actuators** table contains all the data about every actuator that we have used for each area (also this table have a field that contains the id of the area in which the actuators has been positioned).
- The **events** table contains all the information about the events that are generated in the system's areas.
- The **sensors_actuators** table contains the information about the relation between sensors and actuators, the single row of the table says which sensor triggers a certain actuator.
- The **cameras_actuators** table contains the information about the relation between cameras and actuators, the single row of the table says which camera triggers a certain actuator.
- The **actuators_events** table contains the data concerning the relation between events and actuators, in particular we want to keep a log in which we store which actuator has been activated for a certain event.

The NoSQL database (**MongoDB**) is structured as follows:



Key	Value	Type
(1) Objectid("5a3bd3af60b5f61788c818be")	{ 4 fields }	Object
(2) Objectid("5a3bd3b160b5f61788c818bf")	{ 4 fields }	Object
(3) Objectid("5a3bd3b160b5f61788c818c0")	{ 4 fields }	Object
_id	Objectid("5a3bd3b160b5f61788c818c0")	Objectid
value	50	String
timestamp	EFTYDQDPZG5MV1HJU9	String
(4) Objectid("5a3bd3b160b5f61788c818c1")	{ 4 fields }	Object
(5) Objectid("5a3bd3b160b5f61788c818c2")	{ 4 fields }	Object
(6) Objectid("5a3bd3b160b5f61788c818c3")	{ 4 fields }	Object
(7) Objectid("5a3bd3b260b5f61788c818c4")	{ 4 fields }	Object

The MongoDB database works with databases and collections, each database is composed by different collections.

We are using a collection-per-topic approach, so we store the data of the sensors belonging to the area X to the collection named X.

In our instance, there's a database called "**masaccio**". In this database, there are three main folders.

Collections, that contains all the collections that the system generates (a collection for each area/topic).

Every collection contains a set of .json documents that are composed by the data that the system receives from sensors and cameras plus their ids and timestamps (it also contains the internal id of the document itself, used by Mongo and called "**_id**").

For example, in the figure we show the set of documents that characterize the area **a1.1**. We show also the third document of this set, that contain all the pair "key-value" that corresponds to a reading performed by the sensor with id=50, the actual reading and the timestamp.

Functions and **Users** are default collections that are generated when the database is created.

Performance Analysis

To validate our architecture, we had to check the performance of the system. After some tests (on some cheap laptops) we can assert that our system can easily:

kind of operation	# of topics	# of messages per topic	# of total messages	time (seconds)	kind of message
Publish (Produce)	6	40000	240000	13.8170	String
Subscribe (Consume)	6	40000	240000	15.3411	String
Store (from string to .json)	-	-	240000	real-time	.json

This test was made on a network composed by three laptops: this implies some small network delay. The storage on the NoSQL DB of this prototype is synchronous: the *Kafka Client - Storing* subscribes to the topics, consumes data and then it stores the records on the database.

We can assert that the numbers we got are much higher with respect to the customer's required numbers (40.000/hour <<< 240.000/13s).

So, we can conclude that the system is fully capable of handling the requested amount of data traffic.

Future Improvements

- **Parallelization:** in the next deliverable, we plan to improve our *clients* by parallelizing the consumption of the messages (more thread we have, more we can parallelize) assigning each *client* to the same *group* of consumers.
Assigning clients to the same group will assure us that all the messages will be delivered to the group and that a message will be read only by a single client belonging to that group, but not from the other members: this implies an improvement in the throughput of the consumption of the messages (in the current implementation this can be done by running more instances of the client).
- **Asynchronous storage:** in the next deliverable, we plan to improve our *clients* by making the storage procedure asynchronous, this will even more improve the performance as the client does not need to wait to the database to send the acknowledgement before proceeding with the consumption of the next message.
- **Commit management:** in the next deliverable, we plan to improve our *clients* by committing the whole data topic offset only after we successfully stored the fetched data on the database. If something goes wrong the whole dataset will still be available on the Kafka topic, ready to be read again and sequentially stored successfully.
In the provided prototype the offsets are committed after the record is received by the consumer, not after the consumer effectively stored it in the database. This opens a very, very small window in which there could be a data loss (in the sense that the data is considered as committed from the point of view of the Kafka cluster, and so it is removed from the queue even if it is not actually saved on the database, so if the client crashes after committing the message but before storing it on the database the message could be lost).

Summary

In our instance, we want to monitor and analyze UnivAQ’s buildings. Our architecture is based on a PubSub pattern, implemented using the Kafka framework. The data concerning the system’s organization is stored in a relational database while the sensed data is stored in a NoSQL MongoDB database.

This approach, along with features provided by Kafka, allow the system to have a great level of dependability such as: if a producer loses the connection to the cluster it will wait for the connection to come back online without crashing or losing messages. If a consuming client loses the connection it will simply wait for the cluster to come back online and it will resume the consumption from where it left.

The fault-tolerance of the cluster can be increased by adding replicas: a leader will be elected among the participant and when a component goes offline the others will proceed its work with all the up to date data.

It also provides good performance (as exposed on the previous pages we are able to sense-and-store 240000 values in 13 seconds when the requirements were referring to 40000/hour), with the clients running on a common laptop (we expect greater performance on dedicated hardware).

All the clients can run on different machines and on different geographic areas (as demonstrated on the video-demo).

We can have an automatic load balance by simply running more instances of a client. For instance, if we run the Storing client 3 times, as each of the instances belong to the same group, the cluster will automatically assign a portion of the messages to each client, balancing the load (as each instance can run on a different machine).

This approach also provides a great level of decoupling as all the components are independent from each other.

For instance, the Storing client is not related in any way to the Visualization client or the Analysis Engine or the Actuators client.

We are also able to add new sensors and new areas simply by installing the physical device, let him publish on its topic and register the topic on the relational database: the clients will periodically check for changes and will subscribe automatically to the new topics without any kind of manual action, plus each client receives its configuration by a configuration file, this avoid changing the code if, for instance, the cluster address changes.