**“Software Engineering”**

**Course**

**a.a. 2018-2019**

**Template version 1.0**

**Deliverable #3**

**Lecturer: Prof. Henry Muccini (henry.muccini@univaq.it)**

**Dashboard Monitoraggio Ambientale**

|  |  |
| --- | --- |
| **Date** | 20/01/2019 |
| **Deliverable** | 3 |
| **Team (Name)** | 5 Curly Brackets |

|  |  |  |
| --- | --- | --- |
| **Team Members** | | |
| **Name & Surname** | **Matriculation Number** | **E-mail address** |
| Marco Ovidi | *247986* | *marco.ovidi@student.univaq.it* |
| Matteo Di Paolo | *243581* | [*Matteo.dipaolo@student.univaq.it*](mailto:Matteo.dipaolo@student.univaq.it) |
| Carlo Antonio Venditti | *248254* | *Carloantonio.venditti@student.univaq.it* |
| Dima Mhrez | *247052* | [*Dima.mhrez@student.univaq.it*](mailto:Dima.mhrez@student.univaq.it) |
| Andrea D’Angelo | *248817* | *Andrea.dangelo6@student.univaq.it* |

**Project Guidelines**[do not remove this page]*This page provides the Guidelines to be followed when preparing the report for the Software Engineering course. You have to submit the following information:*

* *This Report*
* *Diagrams (Use Case, Component Diagrams, Sequence Diagrams, Entity Relationships Diagrams)*
* *Effort Recording (Excel file)*

***Important:***

* ***document risky/difficult/complex/highly discussed*** *requirements*
* *document decisions taken by the team*
* ***iterate****: do not spend more than 1-2 full days for each iteration*
* ***prioritize*** *requirements, scenarios, users, etc. etc.*

Project Rules and Evaluation Criteria

***General information:***

* *This homework will cover the 80% of your final grade (20% will come from the oral examination).*
* *The complete and final version of this document shall be* ***not longer than 40 pages*** *(excluding this page and the Appendix).*
* *Groups composed of five students (preferably).*

*I expect the groups to submit their work through GitHub*

***Use the same file to document the various deliverable.   
Document in this file how Deliverable “i+1" improves over Deliverable “i".***

**Project evaluation:**

*Evaluation is not based on “quantity” but on “quality” where quality means:*

* *Completeness of delivered Diagrams*
* *(Semantic and syntactic) Correctness of the delivered Diagrams*
* *Quality of the design decisions taken*
* *Quality of the produced code*

Table of Contents of this deliverable

Sommario

**Requirement Collection 5**

Detailed Scenarios 5

Functional Requirements 7

UML Use Case Diagrams 8

Tabular Descriptions 10

GUI/DB/Business Logic Requirements 15

Non-functional Requirements 16

Excluded Requirements 18

Assumptions 18

Prioritized Requirements 18

**Software Architecture 19**

Component Diagram 19

Sequence Diagrams 20

State Machine Diagram 23

**ER Design 24**

Class Diagram 25

Overview 25

Class Diagram 26

Object Diagram 27

**Design Decisions 28**

Mapping 30

Effort Recording 33

PERT Diagram 33

**Appendix Code 34**

List of Challenging/Risky Requirements or Tasks

|  |  |  |  |
| --- | --- | --- | --- |
| **Challenging Task** | **Date the task is identified** | **Date the challenge is resolved** | **Explanation on how the challenge has been managed** |
| Identifying the right technologies | 19/11 | Still open | We tested performances on both our own computers and test servers. NetData was extremely efficient and allowed us to easily respect our performance requirements, but that option was discarded, so we rented a server and started testing on it until we reached satisfying results. |
| Learning how to implement NetData efficiently | 19/11 | 11/12 | After working several hours on a node.js plugin for NetData and asking questions to its Github forums, we realized it would be too time and effort consuming for it to be a viable choice. We managed to make NetData recognize our plugin, but no input was taken. Therefore, we decided to implement that subsystem our own way, without relying on an external software. |
| Using efficient Database technologies. | 19/11 | 21/12 | We split responsibilities and managed to implement a new DB technology we were not familiar with (Time series Databases) while still being able to respect our time constraints. The Influx – java library is also poorly documented, but necessary for data storage. |
| Reinventing the system respecting our time constraints | 11/12 | 23/12 | It was a group effort as we had to spend way more time on this deliverable than we did on the first one. We did not have to change the system as a whole, just the particular subsystem that checked and managed data. |
| Managing parallel threads | 10/01 | 18/01 | We knew making multiple threads and managing them efficiently would result in a significant boost of our server’s performances. That is why we started testing and implementing semaphores on writing locks to ensure no data would be lost or corrupted. Eventually, we studied the possible deadlocks that could occur and modified our code accordingly (more on that on the appendix). |
| Having High performances on our system | 05/01 | 19/01 | Our system must be high in performance so clients are able to visualize all the data, show alert/warning immediately and select the visualizing of any areas at any moment specifically. Since we are handling huge amounts of data that have to be visualized in a very short unit of time(minute), it should be fast. Therefore we have used two http server ports; one sends data to relational database only for the first registration of the sensor with all the related information and another http server port only to send it for time series database and a cache which is a hashmap structure, its key represents sensor's id and its value represents sensor's value. Thus, the dashboard will visualize data faster. |

A. Requirements Collection

A.0 Detailed Scenarios

***Scenario 1***

The sensor (we'll call it Sensor A) in the Hospital building keeps blindly sending short messages to the server, composed of its unique ID and the value it's sensing, to a certain port that is instructed to only accept signals of that kind [ID+Value]. The server keeps receiving these messages and checking if any of the values are above the threshold. In this scenario, it finds that none is: the situation is stable near Sensor A and thus no Warnings are generated.

The server then stores the values in the Time Series Database, and the Values Cache. The clients will get those values from there and process them to show them as simply as possible to the actual human Managers. [Continues with Scenario 2].

***Scenario 2***

Paul, the building manager, logs in the system using the credentials the admin Nora gave him. He was just hired, so he had no chance to change his password yet: he has to use his email and the password that were generated by the system, based on his data. As soon as he logs in, the dashboard appears on his screen and each sensor he has control over is assigned to a certain value.

Paul only has to work three hours today (lucky him!), so he keeps the dashboard running and experiences no warnings or alerts during the entire period. Values keep getting updated real time but none of them is above the threshold, so all of them are green.

Eventually, Paul logs off the system and goes home.

***Scenario 3***

Another sensor (Sensor B) is near the hospital right where a wildfire is located. It sends anomalous data to the server, that finds out something is wrong when checking the thresholds for temperature. Sensor B then increases its frequency and the server stores the Warning in a cache, while still sending data to the Time Series DB and the values cache. Sensor B starts sending data more frequently, and the server keeps the warning up for the client for as long as the values are above the threshold.

[Continues with Scenario 4]

***Scenario 4***

Erika begins her shift right after Paul. She’s not just a building manager, but a Urban Manager, so she has a bigger area to inspect; when she launches the system, she notices a big warning saying that many values are above the threshold near the building and a wildfire might be starting.

Considering the values and their updates, Erika decides to call the firefighters to extinguish the fire (as emergency management is an excluded requirement of the system). When done, Erika notices that the sensor has been flagged as inactive by the system, as it probably was damaged during the wildfire. She decides to contact the admin to understand what’s next: the sensor will probably need to be replaced by an expert technician, because it needs to be a special sensor that is preconfigured to work with our system.

***Scenario 5***

The company has just hired a new manager, Paul. Nora, the admin, must set up a new account for him, so she logs in the system as the administrator, and select the button “New User”. Eventually, another password input is required to make sure she is the admin and she is currently active on the screen. Nora inserts the new manager’s data into the system, and they are stored in the Relational DB. Nora receives a computer-generated password to give to Paul, and he is now able to log in the system with these new credentials. Nora is done for the day, so she logs off the system.

A.1 Functional Requirements

* Unusual values must be shown with appropriate colors based on their priority.
* Values over the defined threshold must be shown explicitly.
* There are 3 types of managers, related to the zones they have to monitor (building, area, city).
* Dashboard must show all sensors.
* Admin must be able to change the defined threshold values.
* The System must be able to detect sensors’ failures and display a warning.

If there are backup sensors for a measured property, the warning will have a lower priority.

* Different managers are given information with different levels of detail, based on a hierarchical relationship.
* The user must be able to select a zone to restrict the sensors displayed to only those in that specific zone.

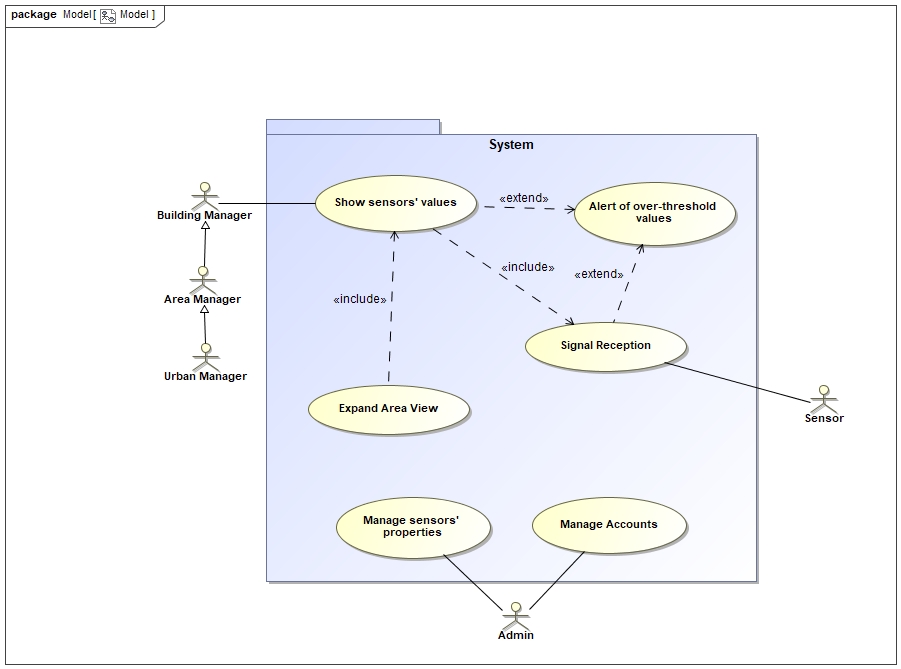


Figure 1: Use Case Diagram

Figure 1 represents the use case diagram of our model with a very high level of abstraction. We identified two types of users: the managers (Urban, Area, and Building Managers where there are generalization relations among them) and the admins.

Managers access the dashboard for environmental monitoring while admins can change threshold values of sensors and add new managers.

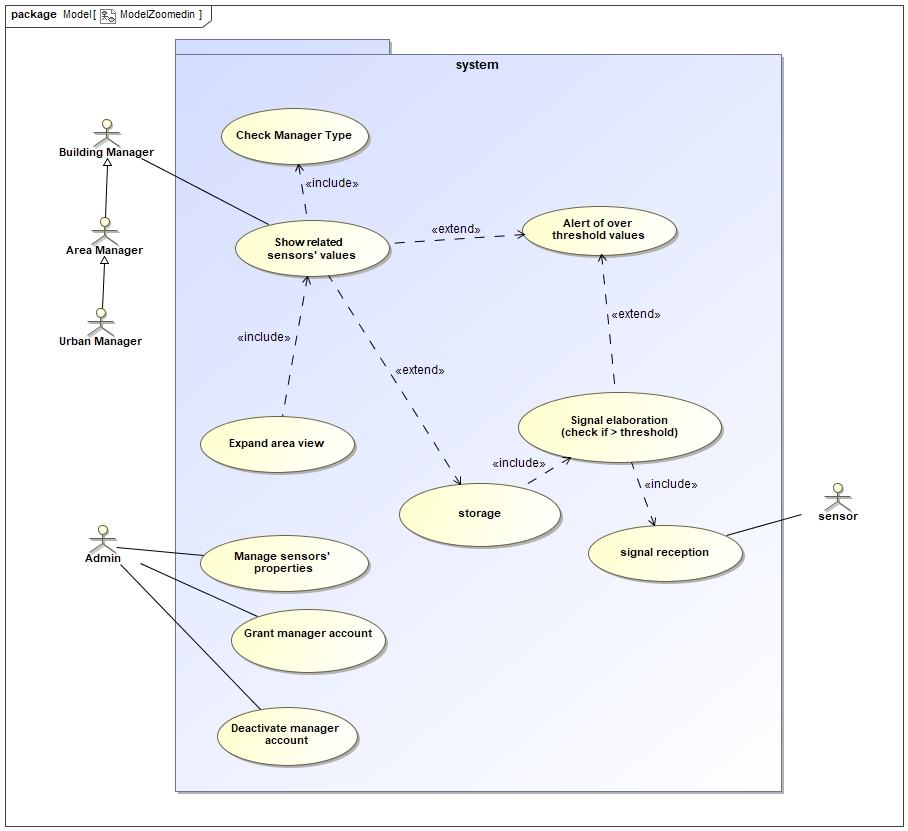


Figure 2: More detailed Use Case Diagram

Figure 2, is the second use case diagram. It shows in greater details the most complex use cases we found. Specifically, *“Signal Reception”* was split in *Signal Reception*, *Storage*, and *Signal Elaboration*, while a new Use Case called *Check Manager Type* was added as an include for *Show sensors’ values.*

The tabular descriptions below are in reference to the Figure 2 diagram. They’re saved as images for format purposes. The editable version of these tables can be found in the Diagrams directory.

















A1.1 GUI Requirements (da riempire a partire dalla Versione 2)

The functional requirements that are related to the GUI are:

* *Unusual values must be shown with appropriate colors based on their priority*
* *Values over the threshold must be shown explicitly*
* *Dashboard must show all sensors*
* *The user must be able to select a zone to restrict the sensors displayed only to those in that specific zone.*

*Our GUI must be crystal clear and as immediate, evident and user-friendly as possible. For this reason, we added Clarity and Straightforwardness* in our non-functional Requirements*.*

In particular, our main requirement for the GUI is to show a list of all of the registered zones, which can be exploded into lists of sensors with their specific values (constantly updated). However, when a warning is issued, the user must be alerted immediately no matter which lists are open. It also must be clear to the active user which zones he has the right to control, based on their role in the system (Building – Area – Urban Manager).

When only one sensor of a certain type has detected values that are above the threshold, the value should be shown yellow. At this stage, it is still possible that the sensor has malfunctioned but the situation should indeed be checked.

If two or more sensors detect values that are above the threshold, the value must be colored red as the situation must be handled immediately.

A1.2 Business Logic Requirements (da riempire a partire dalla Versione 2)

The functional requirements that are related to the Business Logic are:

* *Admin must be able to change the defined threshold values*
* *The system must be able to detect sensors’ failures and warnings. If there are backup sensors for a measured property, the warning will have a lower priority.*
* *Different managers are given information with different levels of detail, based on a hierarchic relationship.*
* *There are 3 types of managers, related to the zones they have to monitor (building, area, city).*

Our main concerns here are security and efficiency. They’re always inversely proportional, and it is cardinal for us to find the right balance. They were both added to our non-functional requirements and explained in greater detail in the related section.

Updating the sensor’s frequency is not the server’s task, as they are configured to do it on their own. Our server should correctly find that the values are above the threshold and generate a warning, but no message is sent to the sensor.

We must also recognize which zone each sensor belongs to, and that is done by building a tree when the server first fires up. This way we can easily propagate errors and communicate them to whoever is in charge of controlling that area.

That means our system must:

* Check if the detected values are above the threshold
* Store the data in the Values Cache or Warning Cache
* Store the data in the Time Series DB
* Be able to insert new sensors in the Relational DB
* Send the correct data to the clients
* Allow to insert or remove Managers

A1.3 DB Requirements (da riempire a partire dalla Versione 2)

There are no explicit functional requirements that relate to the DB.

**Efficiency:** The main (non-functional) requirement we have for our Databases is that they have to allow our system to handle at least 150000 signals per minute. We know the DBs are the slowest part of the system and we have to be especially careful in their management.

**Consistency:** It is also cardinal for us that the two DBs we use are consistent with each other. The system must also keep running in case of hardware fault, so we must keep backup DBs running.

Our Databases must store:

* Zones, including Areas, Buildings, Floors and Rooms.
* All sensors, with their ID.
* All signals received (Time Series)
* All managers account, with their name, informations, and role.

A.2 Non Functional Requirements

1. **Efficiency**

We must handle *at least* 150000 signals per minute. Because of this, we chose to use two different DBs: a relational DB and a Time Series one. This choice was made because Time series Databases are extremely efficient for our purpose.

It is important we query the relational Database as little as possible, because those accesses can decrease performances rather heavily if not optimized. In order to reduce unneeded accesses to the databases the system shall keep a cache in memory for nearly everything, this also makes the system more resilient and fault tolerant.

But that isn’t enough, we also had to optimize our server and communication methods: packets should be as small as possible in order to be processed faster and better. That is why, when a sensor is activated for the first time, it communicates with a different port sending all the initial information we need (ID, Threshold, Type) and storing them in the relational Database. From that moment onward, the sensor starts communicating with another port that only accepts simple packets containing the Sensor ID and the Value detected.

This way we don't have to check if the packet is from a new sensor or an "old" sensor each time we receive one.

1. **Usability**

The GUI must be crystal clear and as immediate as possible. In our system, that translates to the manager being ale to identify issues the moment they come up. Warnings must be explicit and unmissable.

Values must be updated real time and show clear, bright colors (green, orange and red) depending on the level of danger.

As this system is not meant for commercial use, **clarity must have priority over visual appeal**. Menus should be clear, not cluttered, and even a non-tech

savvy manager must be able to use the system at its fullest potential with ease. We don't know whether the managers are engineers or just simple workers - therefore we have to keep it simple, fast, and efficient.

1. **Security**

Ill-intentioned people might break into our system and send false data to the server: this can cause erroneous or missed warnings and possibly greater risks. We must keep the connection between the sensors and the server as secure as possible, while also being able to guarantee the system’s efficiency and fulfilling its requirements. Connection between Server and Client, or server and the two DBs, should also be secure, but assuring the correctness of our data takes priority.

This comes down to choosing the right balance for efficiency and security: tests are necessary to understand which protocol is the fittest for our project.

1. **Scalability**

We assume the system is constantly growing, therefore new sensors could be installed rather frequently. Our system should be able to grow while maintaining the same efficiency. This is done by separating our Model (stored in the relational DB) from the signal handling, maintained in a Time Series Database. This non-functional requirement is also heavily tied to *Efficiency.* Having an efficient and scalable system, also means having a cheaper system both to set up and to operate.

A.3 Excluded Requirements

1) **Emergency Management.** The system does not provide any functionality to interact with emergencies (e.g., calling firefighters). It is only used to check the data the sensors are sending.

2) **Portability.** Our system is very specific and must work with a particular kind of sensors that are preconfigured for a certain behavior. It would be hard to code modules that are generic for any data-collection software without lowing performances, so it won’t be our goal for now.

3) **Legal Requirements.** We assume sensors are placed according to the law and no invasion of privacy is perpetuated, so we will ignore this for now.

A.4 Assumptions

*1) We assume the sensors are already in place and their positioning is reasonable. A sensor placed near a fireplace would probably show unusual values when it is not needed. We also assume they are preconfigured with an ID already known to the system, and it is placed in the correct area.*

*2) “Alert” and “Warning” have very different meanings in our system. We have an alert when one type of parameter (e.g.: temperature) has a value that is over the threshold. A warning is issued when two or more types of parameters are above the threshold, and the manager must be alerted immediately.*

*3) We assume we have no budget limit, as it was not explicitly expressed in the requirements. Of course, we still tried to plan smart and without waste.*

*4) We assume sensors are configured to have a unique ID that does not already exist in our system, and that can be communicated to our server to a certain port. We also assume they are pre-configured to send more signals if they find that the detected value is above the threshold.*

*5) We assume sensors are placed according to the law and no invasion of privacy is perpetuated.*

***A.5 Prioritization***

Here is the list of the functional requirements in priority order:

1. Unusual values must be shown with appropriate colors based on their priority.
2. Values over the defined threshold must be shown explicitly.
3. There are three types of managers, related to the zones they have to monitor.
4. Dashboard must show all sensors.
5. Admin must be able to change the defined thresholds.
6. The system must be able to detect sensors’ failures and warnings. If there are backup sensors for a measured property, the waring will have a lower priority.
7. Different managers are given information with different levels of detail, based on a hierarchical relationship.
8. The user must be able to select a zone to restrict the sensors displayed only to those in that specific zone.

B.1The static view of the system: Component Diagram

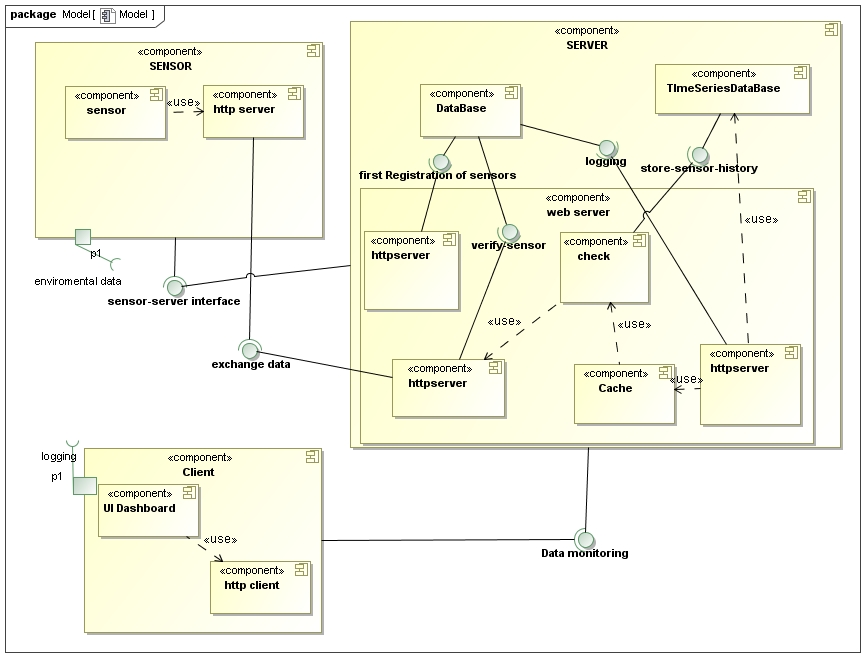


Figure 3: Component Diagram

The updated Component Diagram replaces NetData with an architecture of our own. Just like last time, the interactions are simplified to show the logical subdivision without too many technical details. The three main components are the Sensor, the Server and the Client. The Server manages and checks all of the signals before making them available for the Clients.

B.2 The dynamic view of the software architecture: Sequence Diagram

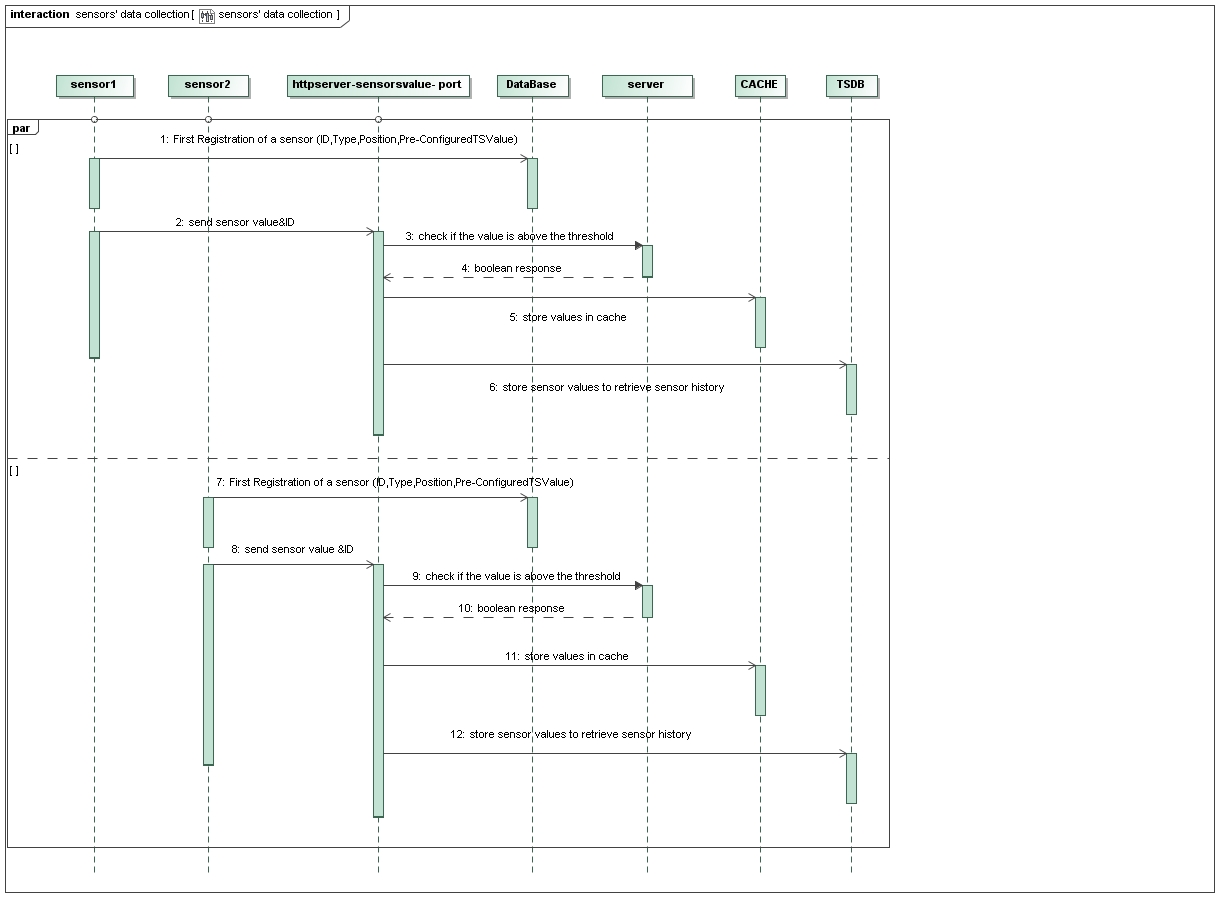
**

Figure 4: Sensors’ data collection Sequence Diagram

This diagram shows the parallel data collection from the sensors, it states clear the importance in our system of the cache(s) and the other in-memory structures as they allow us to defer writes to the databases while still keep sending updated sensor values and status to the clients. We use different ports for receiving values and for registering new sensors. The use of HTTP allows fast prototyping and should suffice the performance requirements but in case of necessity can be easily replaced with plain TCP communication as it is loosely coupled with the rest of the system.

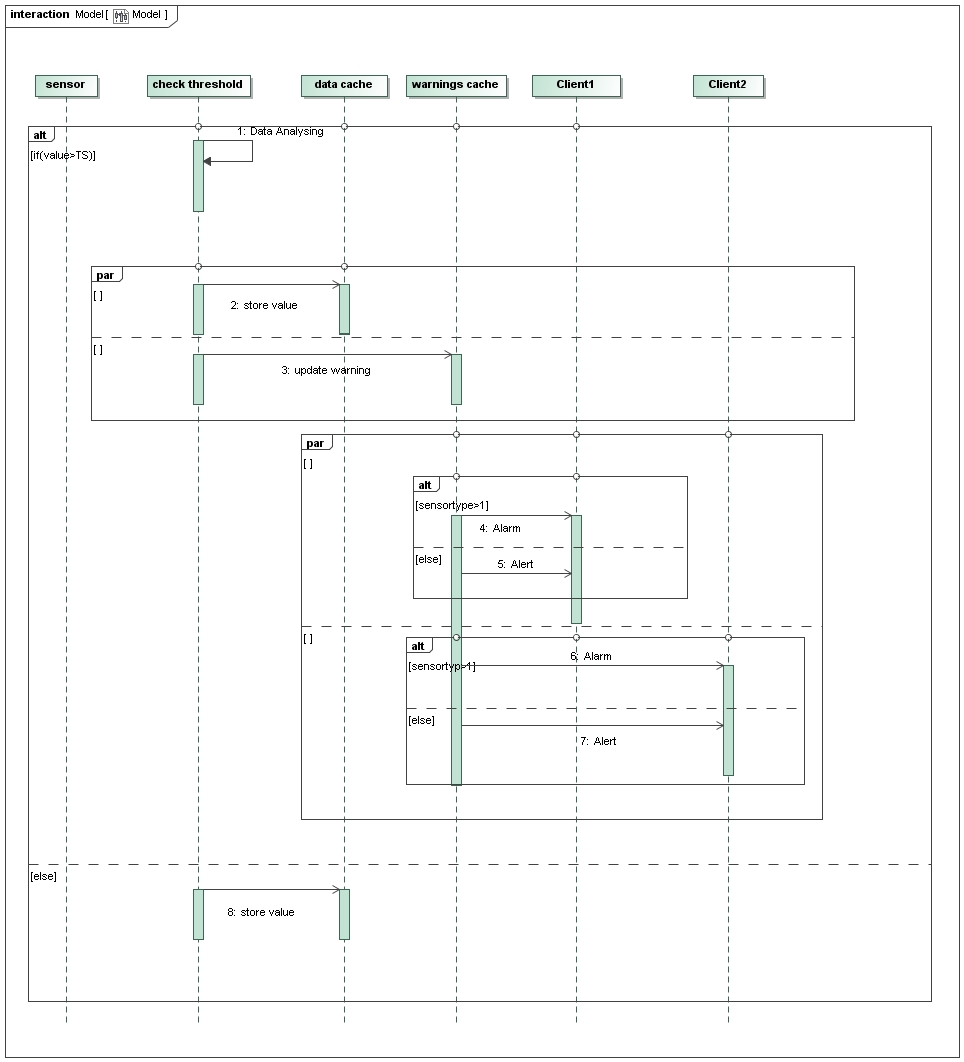


Figure 5: Data Processing Sequence Diagram

Here we show the parallel processing of the data received from the sensors and the streaming (along with eventual warnings and alerts) to the clients of users to whom the sensor is relevant (so the managers of zones containing that sensor).



Figure 6: Client Data Monitoring Sequence Diagram

This diagram shows how the client receives data from the server. Once the manager is logged in, his dashboard displays and continuously updates status and values for all the sensors of his interest. If the manager wants to restrict the zone he’s displaying, the client sends his request to the server, which from this moment on will only send the appropriate zone’s values (until a new zone request is made from the client). The dashboard only shows the last known value for each sensor unless the user requests the values history for a specific sensor. In this case the server retrieves the data from the TSDB and sends it to the client.

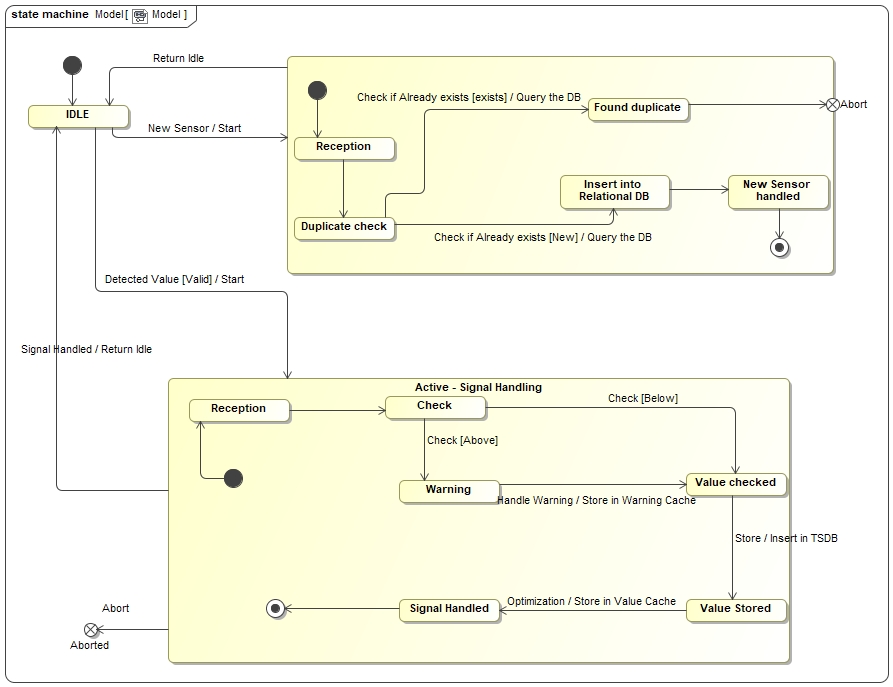
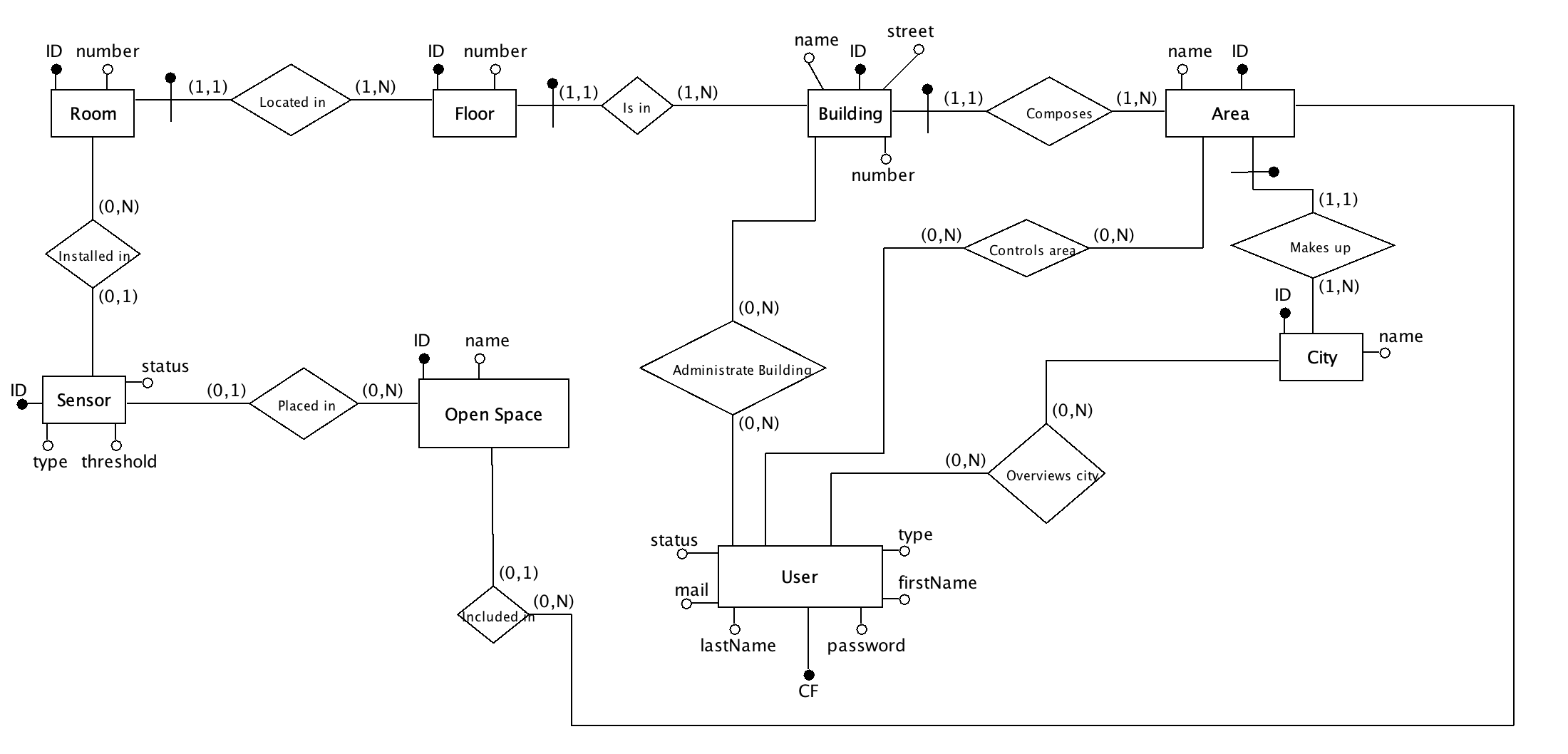


Figure 7: Server State Machine Diagram

This state machine Diagram depicts the server and its basic functions **that are related to the sensor**. First of all, the server behaves differently if he receives a “normal” Value packet or the packet from a new sensor that wants to be recognized. In the former case, the value goes through a check and then gets stored both in the cache and Time Series DB. This is basically a server-focused view of what was illustrated in Figure 4. If the server receives a New Sensor request, it checks if we already have a sensor with that ID in the relational DB and eventually inserts a new one.  
Normally, a State Machine Diagram would also have internal operations that do not cause a transition from one state to another. Our server, however, operates a series of actions that cannot be paused, so it never stays in a certain state for too long.

C. ER Design

<Report here the Entity Relationship Diagram of the system DB>

Figure 8: ER Diagram

The ER diagram has already been restructured and ready to be transformed into the corresponding relational model. The basic entity that constitutes the system is the sensor, while rooms, floors, buildings, areas and cities are elements of aggregations of sensors necessary for the location of the sensors as well as for the attribution of the different types of areas monitored to the various types of managers. For security reasons, user passwords will be stored in encryption to avoid unwanted access as much as possible. To speed up indexes and to make the system more robust, the keys identified in the ER diagram will be used as unique constraints at the implementation level and as IDs will be used auto-increment integer.

Open Space is only connected to “Area” and not “City” because we assume a City is composed of many Areas, that can eventually be Open Spaces such as parks, parking lots, etc…  
Areas can also be a mixture of Buildings and Open Spaces. D. Class Diagram of the implemented System

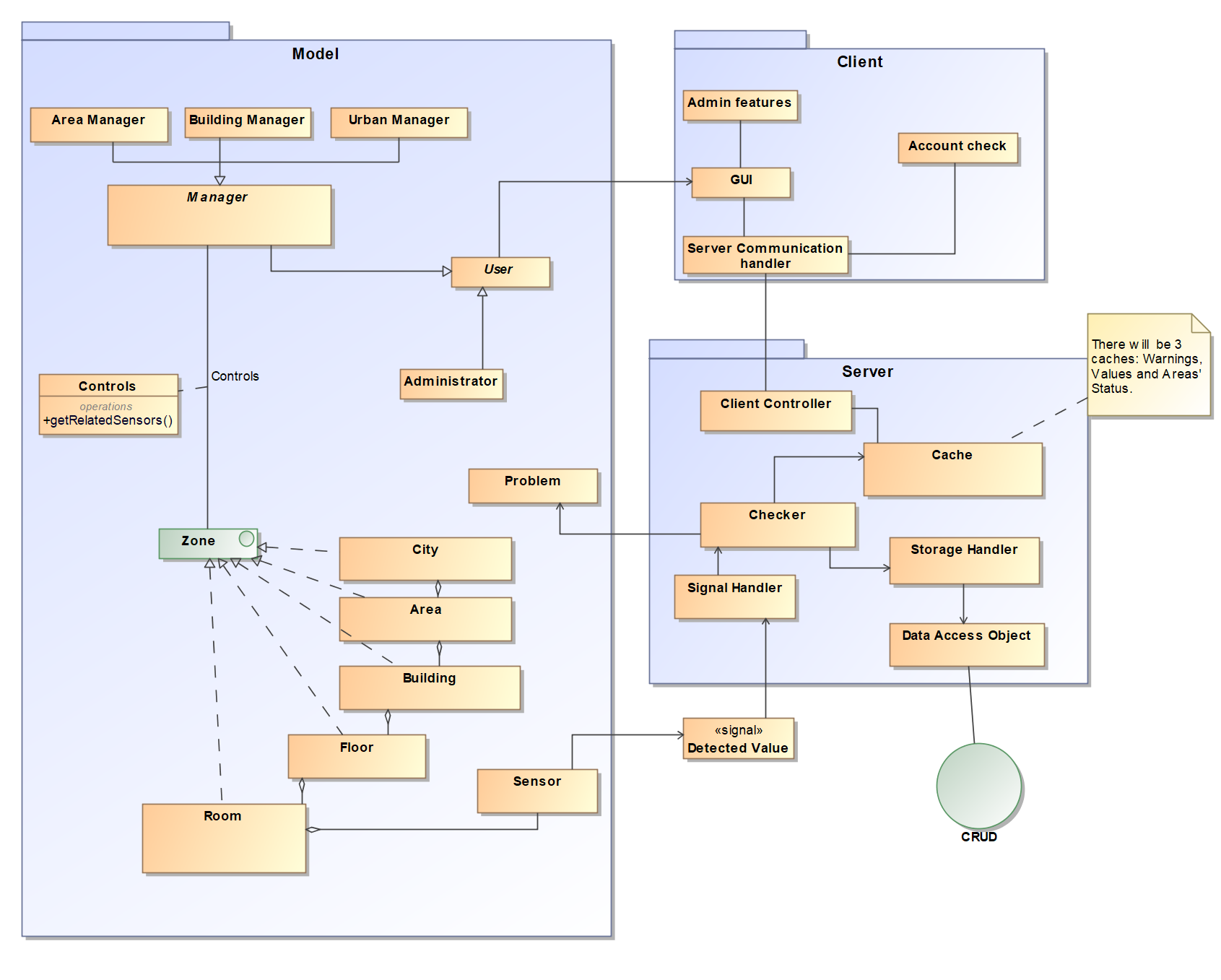


Figure 9: Overview of the Class Diagram

This figure is an overview of our Class Diagram. It does not represent attributes, operations, or other details as it is only meant as an aid to quickly grasp the basics of our system. Figure 12 will go in details about all of the classes presented here. Packages are very similar to the main components of figure 3, the component diagram.

The Class Diagram was made starting from the Use Case Diagrams and modeling each Use Case with the necessary classes to implement that feature.

We also added the DAO (Data Access Object) and CRUD design patterns we’re already familiar with due to previous projects. The checker can now spawn a “Problem” entity which is basically how we modelled warnings and alerts.

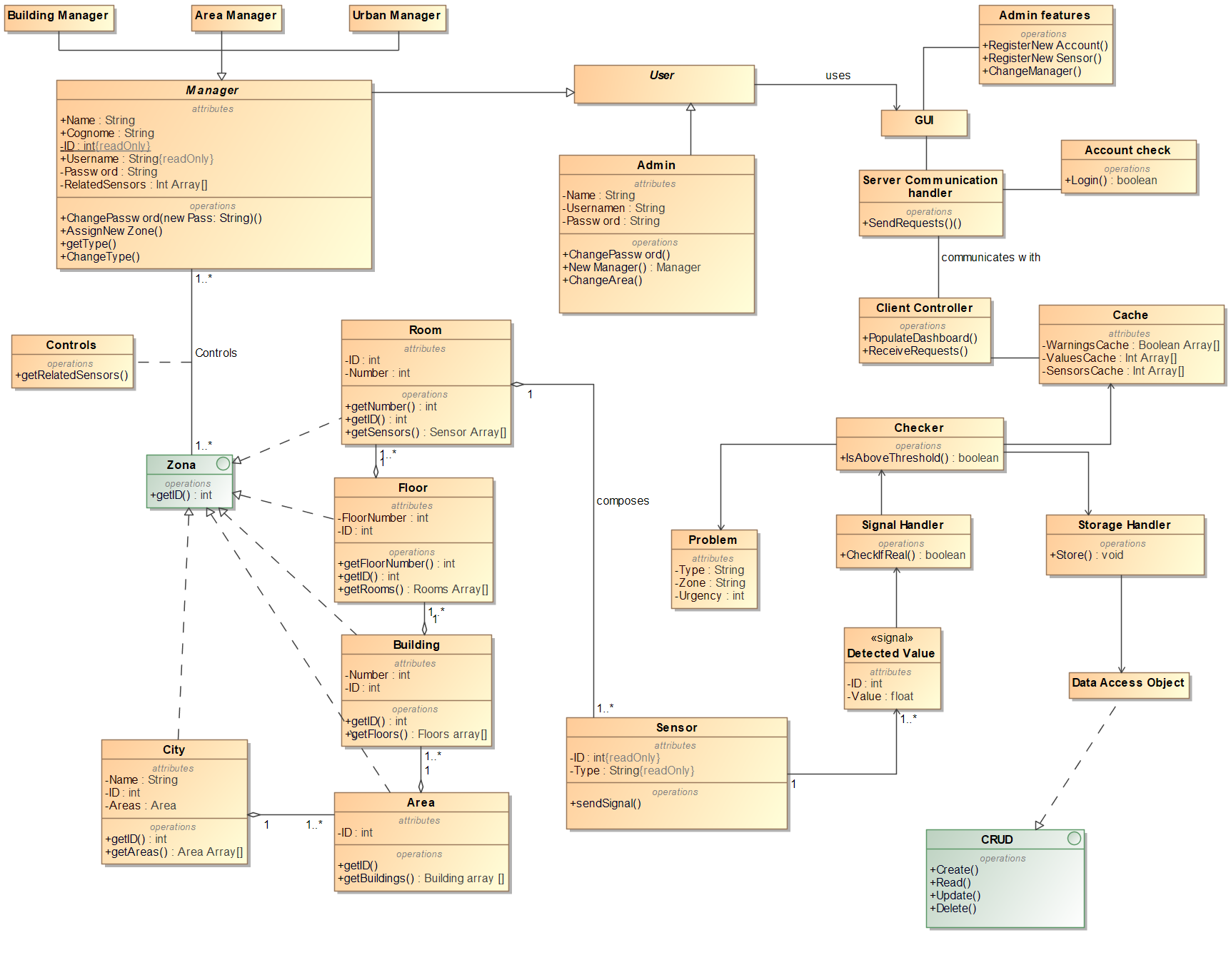


Figure 10: Class Diagram

This is an in-depth view of our Class Diagram. It does not only show the Model part of our system, but the main components of the Control and View (GUI) segments as well. The model is clearly very similar to the one illustrated in the ER Design.

It is also important to note that both Manager and User are abstract classes, thus cannot be instantiated. The class relationship “Controls” uses both the data from Manager and Zone to understand which sensors a certain manager is able to control. Sensor and Room have an Aggregation relationship because in our system, rooms only exist if at least one sensor is in them. As there are no empty rooms, we can say a room is composed of 1 or many sensors.

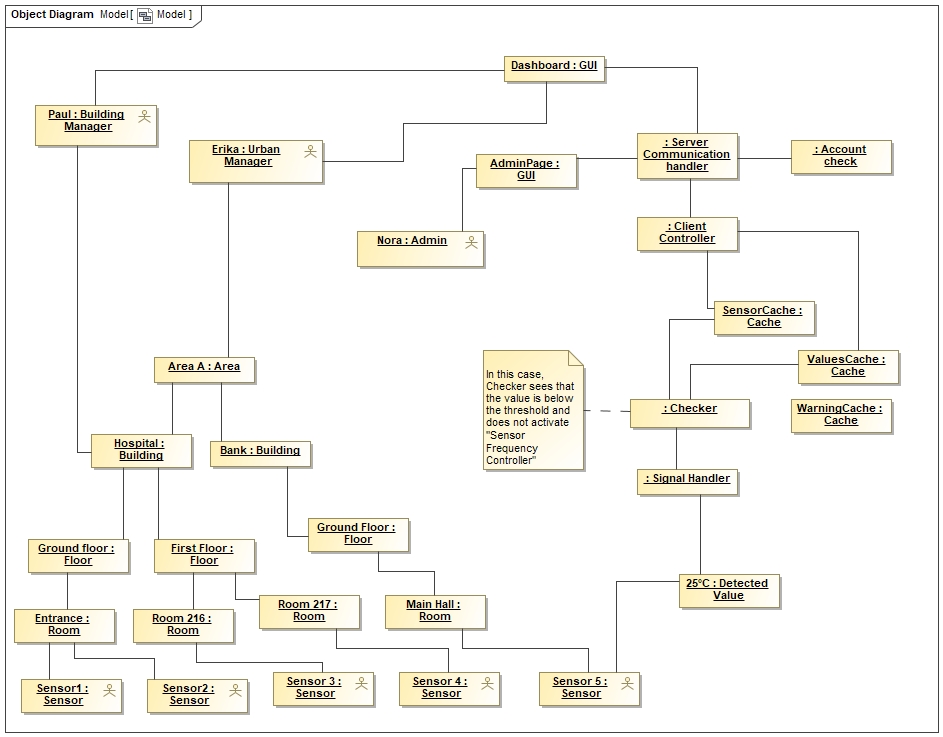


Figure 11: Object Diagram

Figure 13 represents an Object diagram, an instance of the Class diagram of figure 12. For coherence, we used the same names from the scenarios illustrated at the beginning of this documentation. So Paul is the Building Manager that has to control the hospital, Erika is still the Urban Manager, and Nora is the admin. As we can see the building is composed of many floors, which are composed of many rooms, and a room only “exists” in our system if at least one sensor is in it. For example, sensor 5 detects a temperature value of 25 degrees, and sent the signal to the Control part of the system that will handle it and eventually show it to the clients.

E. Design Decisions

1. NOT using NetData

We were perfectly conscious of the risks related with our previous decision of using netdata, so we started as earlier as possible to test the needed technologies. While we had good results at first, the effort revealed to be unsustainable (at least with the time constrains we had). The workaround to the intrinsically synchronous nature (given we cannot query sensors for values) of netdata we thought could be done in a few hours may instead have required a lot of time due to the way netdata handles node.js plugins by default (we omit further discussion about a topic no more related to the project). After all the time spent on this was not entirely wasted as we learned a lot about the architecture and we perfected part of the design thank to the knowledge acquired.

1. Administrator User

As the managers are meant to just control the sensors’ values, we also chose to insert an Administrator as a distinct user beside the managers. One or more administrator users take care of tuning sensors threshold values, add new manager and grant permissions (e.g. assign the zones to the proper managers). This also simplifies the implementation of the GUI, moving all unneeded functions outside the dashboards of the manager users.

1. Time Series Database

To maximize the efficiency of the system we decided to store the values of the sensors in a Time Series Database (TSDB), in the specific InfluxDB. This type of databases is specifically designed for the task of storing a great number of values sequentially (like the values received from a sensor). It is extremely fast for this kind of task and the management is easier than using a SQL database or many more common NoSQL databases. Of course, we also have a separate relational database where the systems stores all the structured information it needs to work properly (e.g. information and about sensors, zones, users and relationships between those).

1. Assuming sensors are preconfigured

Considering we cannot query the sensors no matter what, how could we tell them to increase their frequency when a value is above the threshold? One feasible option was to increase the rate at which our server fetches for results. However, that wasn’t really compatible with how we designed the system, and would probably result in a massive overhaul of what we had done. We also felt it wasn’t the most elegant solution, so we ended up assuming that a sensor is preconfigured to do that. We already had imagined that a sensor must register with the server first, sending its ID and other information. That means preconfiguring the sensors was already a thing in our system, so we felt this solution was more in line with what we had done before.

1. Only the administrator can add new users.

We envisioned our system to be used in a private company, public services, or generally any workspace with an employer and employees, so not for public use. That is why we decided that only the admin can use the Register feature to add a new manager to the system, supposedly when a new employee is hired.

The passwords are generated by the system. We think that is the best option as it would be awkward and clunky for the new employee to choose and communicate their desired password to the admin. Therefore, it is generated automatically (and paired with the employee’s chosen email), and can be modified later.

F. Explain how the FRs and the NFRs are satisfied by design

**A1.1 GUI requirements**

GUI was created using HTML5, CSS and JavaScript.

The GUI has been implemented in a way that it can be accessed via the web, thanks to the implementation of a server created through Java and Java Servlet that can be reached remotely. This allows managers to view the dashboard at any time by knowing the server's IP address.

***• Unusual values must be shown with appropriate colours based on their priority***

For unusual values classified “Alert” (see assumption #2) we add a “orange” background color to the box(intenteded as a room box if on dashboard the user is logged as a building manager , intenteded as a building box if on dashboard the user is logged as an area manager and etc.) that contains the sensors that is over threshold . For unusual values classified “Warning” (see the same assumption) we do the same operation after have done risks evaluation but colouring the box with a “red” color. An example of this is given in the Appendix section below.

***• Dashboard must show all sensors***

Every manager depending on his role have a different dashboard,operations performed by managers like login and data request are made by using an http client that sends/receives data from/to server using a GET request.

***• The user must be able to select an area***

Managers’ permissions are stored in a relational Database (MariaDB) so that every manager only sees the areas assigned to him and with the right level of detail. A building manager will be able to choose between the various buildings assigned to him, an area manager among the various areas assigned to him and in the same way an urban administrator can select one of the cities assigned to him will specifically see a division for floors and rooms. Focusing on a room, he will be able to see specifically all the sensors available in the single room. The same reasoning is applied to all types of managers who, however, will receive information grouped together with the growth of the vastness of the monitored infrastructure with the possibility of expanding to go into details.

**A1.2 Business Logic Requirements**

***• Admin must be able to change the defined threshold values***

The admin, by connecting to the dashboard using the credentials, can check the thresholds of the sensors through an http call to the server and change the value if necessary, for each sensor the threshold level is stored in the relational DB.

***• The system must be able to detect sensors’ failures and warnings. If there are***

***backup sensors for a measured property, the warning will have a lower priority***.

The server will receive messages from the various sensors through an HTTP call. For each message received, it will verify the authenticity of the sensor by checking the relational DB in which the sensor identities are recorded. A "check" control module which resides on the server will check if it receives from each sensor (at least once a minute under normal conditions) a message and will compare the measured value that the sensor will send through the message with the maximum threshold value preloaded in a cache inside the server to be as efficient as possible in the comparison. The failure of a type of sensor will be shown in orange (considering an alert situation) if in the same environment there are other sensors that detect the same type of data otherwise in red (considering a warning situation)

***• Different managers are given information with different levels of detail, based on***

***a hierarchic relationship.***

A building manager will specifically see a division for floors and rooms. Focusing on a room, he will be able to see specifically all the sensors available in the single room. The same reasoning is applied to all types of managers who, however, will receive information grouped together with the growth of the vastness of the monitored infrastructure with the possibility of expanding to go into details.

***• There are three types of managers.***

In the relational DB each manager is assigned according to their role buildings or areas or cities to be monitored.

**• A1.3 DB Requirements**

It was decided to use a DB time series for storing the data collected by the sensors, which by its nature is suitable for storing this type of data and is optimized for handling time series data, that are arrays of numbers indexed by time (at datetime). For the management of roles, permissions and storage of active sensors in the system we decided instead to use a relational database (MariaDB). Although slower than a NOSQL database since it is not used to a maximum if not in the login phase of the managers to recover access data and areas monitored by him, we have decided that it is the best technology for this purpose. To guarantee data preservation, we will certainly adopt backup systems for both databases in particular, the relational one because while the loss of old measurements present in the database time series (if not for analysis) does not represent a serious problem, the loss of the roles of the various managers can be serious and would prevent the access of the various managers. For security and efficiency, the latest detection measured by the individual sensors is stored in the server cache as well as in the DB to provide managers with less waiting time.

**• Non-functional Requirements**

1. ***Efficiency***

The need to store many measurements per minute is guaranteed by a type of NOSQL server specifically designed for measurements called DBs time series. The server chosen in question is INFLUXDB which as reported in the official documentation *"Our influx-stress benchmark running on an AWS c4.4xlarge results in approximately 900,000 values per second on average."*

source: https://www.influxdata.com/blog/influxdb-1-1-released-with-up-to-60-performance-increase-and-new-query-functionality/

***2. Security***

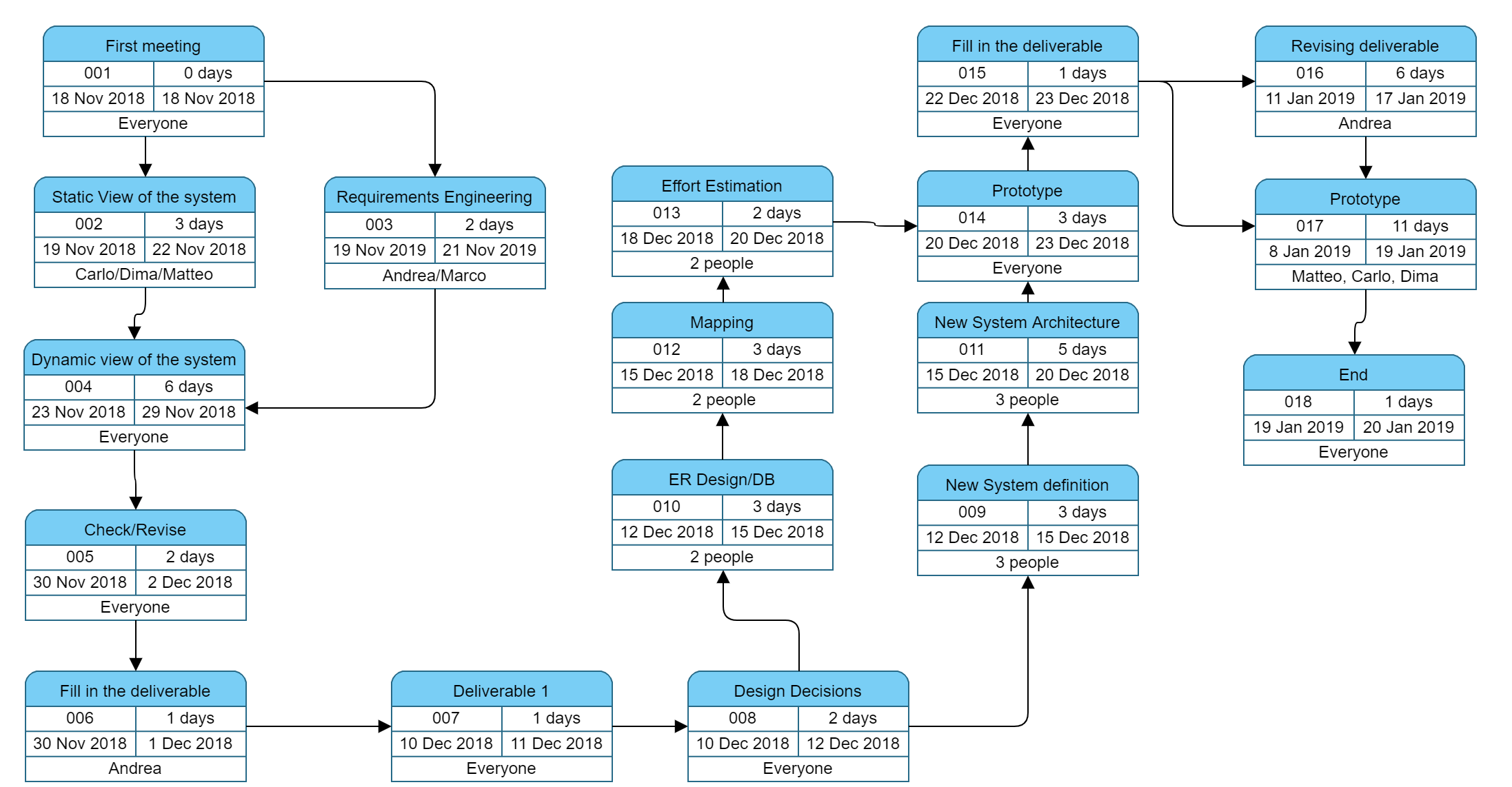
We will use the https protocol to manage the communication between servers and users, in particular the access of an admin is very critical because criminals logging in as admin could lead to exorbitant levels sensors’ thresholds so that catastrophic events can occur without the server being able to detect them and consequently without the managers being aware of it.

We also implemented a password generation feature that randomly generates a password for each new user, with a random length from 7 to 10 characters. This should shield us from potential intruders that try to guess a manager’s password, or bruteforce attacks. We also store the passwords in the DB with the md5 cryptographic hash function.

***3. Scalability***

The MariaDB will only store the identity of each individual sensor so we will have a tuple for each sensor that can be easily managed if the number of monitored sensors is increased. For storing the measurements detected by the sensors, up to 54,000,000 measurements per minute can be written to the NOSQL db.

G. Effort Recording

***PERT*****

***Time spent for the first deliverable: 55 hours***

***Time spent for DOING (first deliverable): 36.5 hours***

***Time spent for LEARNING (first deliverable): 18.5 hours***

***Time spent for the Second Deliverable: 110.5 hours***

***Time spent for DOING (Second Deliverable): 91.5 hours***

***Time spent for LEARNING (Second Deliverable): 19 hours***

***Note: Data is updated to 19/01/2019. Hours spent on the last day of the deliverable were not counted.***

***Time spent for the Third Deliverable: 144.5 hours***

***Time spent TOTAL: 310 hours***

***Time Spent for DOING (TOTAL): 243 hours***

***Time spent for LEARNING (TOTAL): 67 hours***

Appendix. Code

Introduction

Our main concern was to improve concurrency and parallelization on our system as much as we could. We strongly believe those are cardinal to the efficiency and scalability (two important non-functional requirements) of the system, and that’s why they are the things we spent most of the time on.

We coded everything in Java and rented a server where we installed Apache Tomcat, that creates a thread for every HTTP request that is received. We could also substitute this with sockets to improve performances (and we have done so, with good results), but in the end we decided to stick with the HTTP packets.

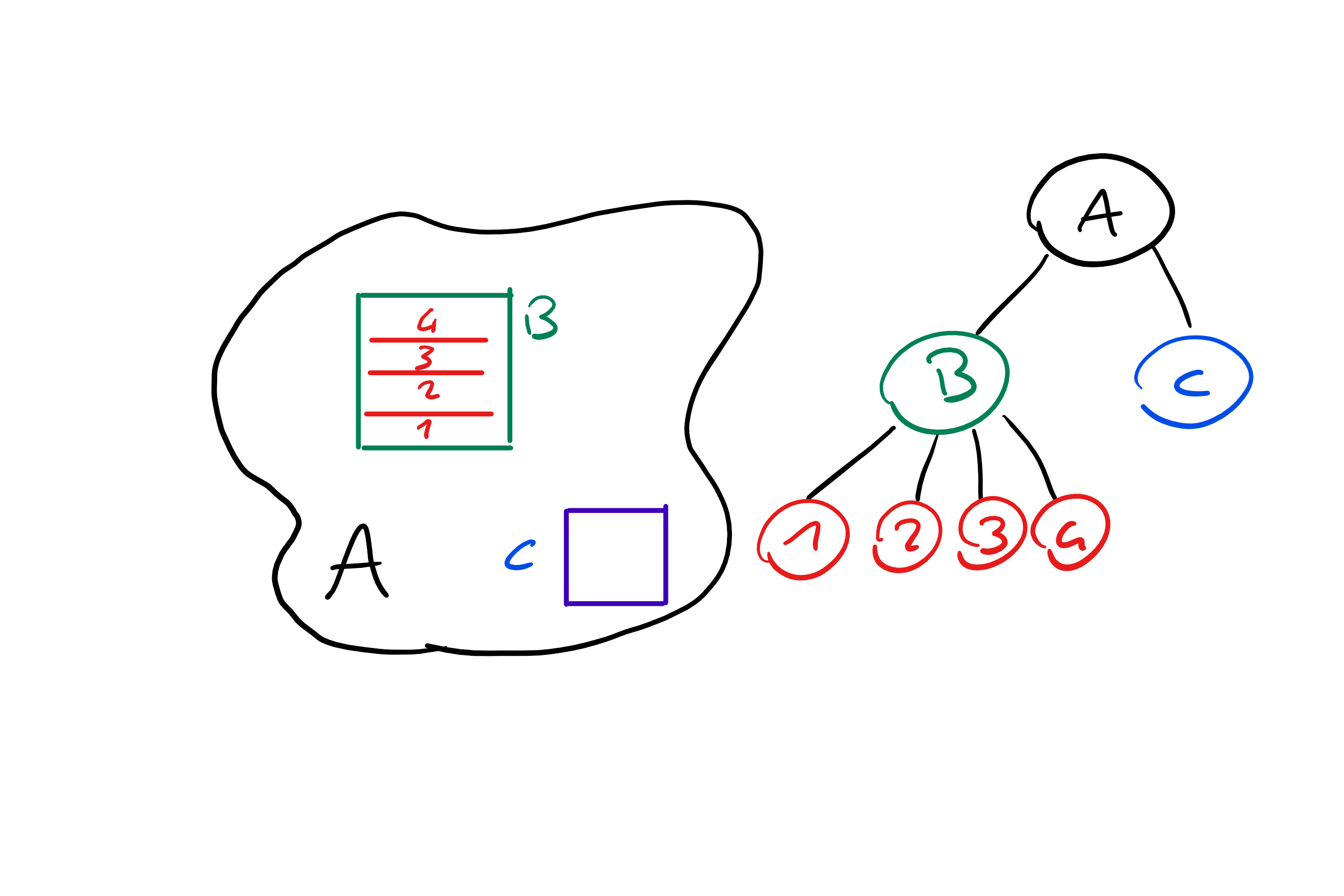
First things first, we’re going to look at the architecture of the data structures our system implements. We’re then going to talk about concurrency, threads, and how we handled them. We’re also going to have a deeper look at how the modules we described in previous diagrams work together. Lastly, we show a brief study on what could be done do improve and build on this system.

Data Structures

The most important data structure we implemented in our system is a N-ary tree that basically models our city. The root node is the city itself, while each parent node is connected to its children if the zone it represents contains them.

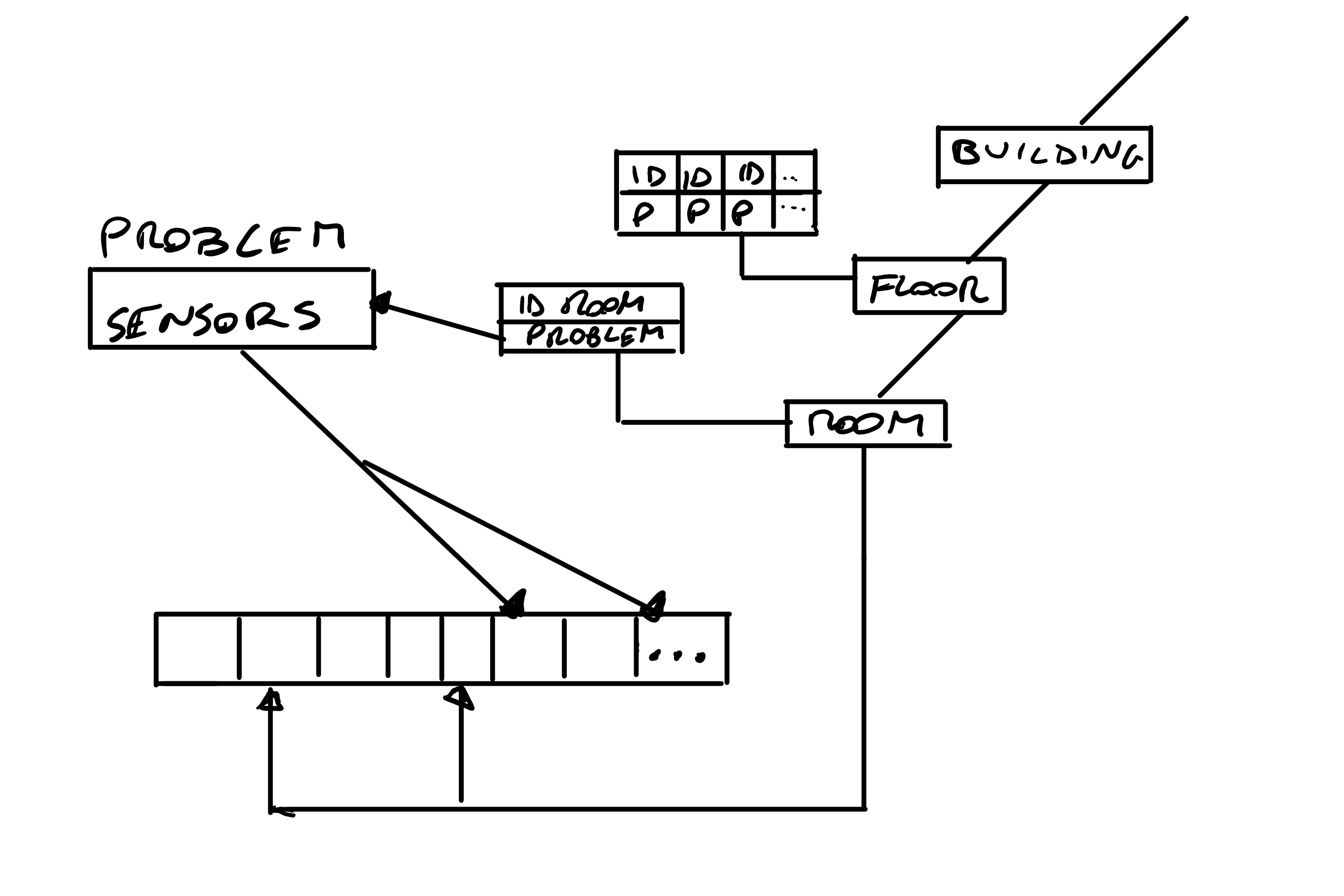
Imagine we have a city called A (its founders were not very creative), that contains two main buildings called B and C. Building B has also 4 different floors.

When the server first fires up, we build a 4-ary tree (in this case) that correctly models our entire city, from the largest areas down to the rooms:



Sensors are then referenced by individual rooms. They are stored in an array in order to speed up accesses,

We modelled alerts and warnings with a new class we called **Problem**. The goal of this class is to correctly propagate warnings to parent nodes of the tree. This way, a Building manager, for instance, can see the building have a warning message even if it is only related to one of the rooms.

The way we do that is thanks to a map that links together Rooms and Problems. A handy visualization of how our data structures work combined is shown on the left. 

Linked to Building, Floor and Room we have what we call **Problem Maps.**

A room is only linked to a certain problem, but a floor, containing many rooms, is linked to a map of [ID, Problem] where ID is the ID of the room that contains the sensor that detected something wrong.

Combining what we know, we can say our city is modelled by a n-ary tree in which each node represents a zone, be it an Area, Building, Floor, or Room. Each node is also linked to a Problem Map that propagates to upper nodes, so we can always see which area contains an alert or a warning.

Even if a certain sensor returns to normal values after triggering an alert, said alert stays until a manager manually confirms that they have seen it.

We also implemented a particular module that further studies the situation certain sensors have been monitoring. For instance, let’s say three different sensors of three different kinds (temperature, pressure, humidity) record strange values. Our module sees that and triggers a particular problem called **Danger**. This is the problem with the Highest priority that is shown in an explicit message to the clients that are connected.

Now that we explained how our Data structures work and interact, we can show how we handled concurrency.

Threads and Locks

As we already mentioned, we really focused on parallel threads. We use read/write locks to make sure that accesses to our memory don’t cause any problems.

Apache Tomcat spawns a thread for each HTTP request it receives. Given that each sensor sends an HTTP request when it detects a value, and that we need to manage at least 150000 signals per minute, it was obvious we also needed some way to handle threads.

Each sensor in the array has an individual r/w lock. Clearly, the write lock only activates if we’re modifying something, while the read lock works when we access data.

We modeled this sensor with the **LockedSensor** class, which is basically a sensor with a Read/write lock.

We also have a global lock on the n-ary tree we described earlier.

This led to a situation where a **Deadlock** could possibly happen. To solve this situation, it was important to activate sensors’ locks and tree lock in the right order, and make sure that the locks for each sensor were completely independent. As of now, we are sure that a deadlock cannot happen in our system.

All the studies we made on Data Structures and parallel threads were in order to improve the non-functional requirements of Efficiency and Scalability.

Now we’re going to show how we implemented the modules we talked about in this documentation.

Implementation of the modules

1. Checker Module

The first module we’re going to study is the **Checker** module. Its main function is to *check* if the detected value is above or below the threshold, but it also does other things. In order:

1. It inserts the value in the sensor’s cache.
2. It sends the value to the Time Series Database.
3. It checks if the value is above the threshold.

If the value is actually above the threshold, it creates a new instance of the class **Problem** (as shown in the Class Diagram) with the right attributes, and assigns it to that specific room, so it can be propagated, with the tree structure we already described.

The checker module basically implements what we described as *Business Logic Requirements*, or most of its requirements. Of course, other modules implement the other missing requirements.

1. Other Business Logic Features

When a sensor doesn’t send signals for two minutes, it is marked as Inactive. This is also an instance of Problem; even if the sensor starts working again, the Problem remains until it is manually removed by one of the managers. That is to ensure that the problem is known, and the sensor is eventually checked.

In order to improve efficiency (non-functional requirement), we only access the relational DB for the first construction of the tree and to manage users (but that happens rather rarely compared to signal handling).

1. GUI

When showing values in the **GUI**, it is important that they are colored based on the detected values. This is also done thanks to the **Problem** class: its *type* attribute shows whether we have an alert, a warning or a danger, and the values can be colored accordingly. In case we have both an alert and a warning in the same, room, the value is colored based on the gravest problem (so, in this case, it will be colored red).

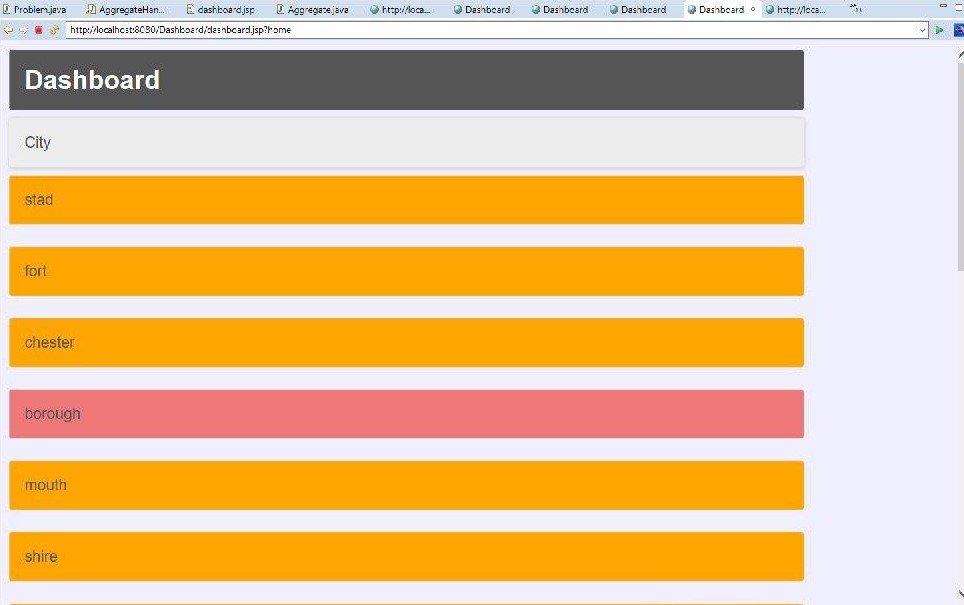


Figure: Example of the dashboard as seen on the Eclipse IDE

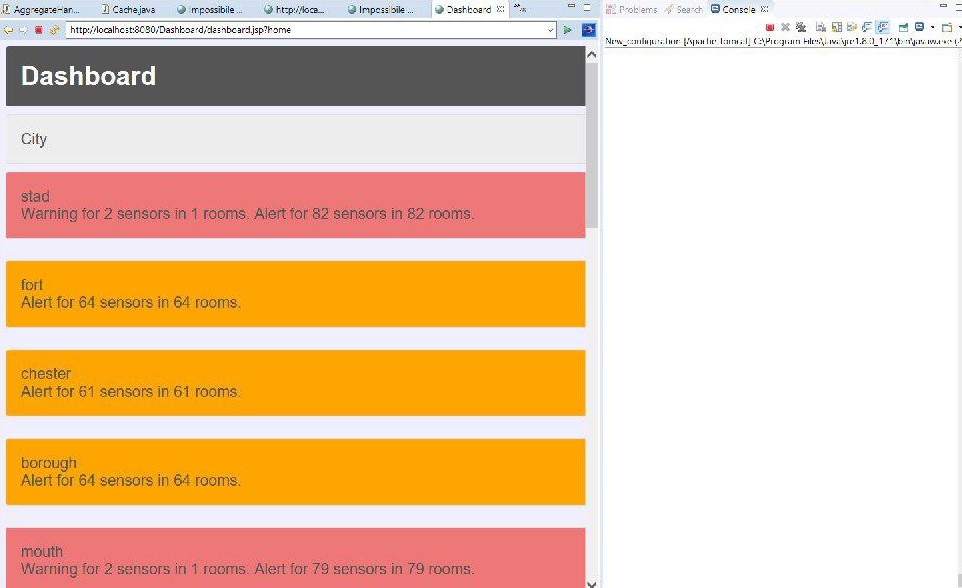
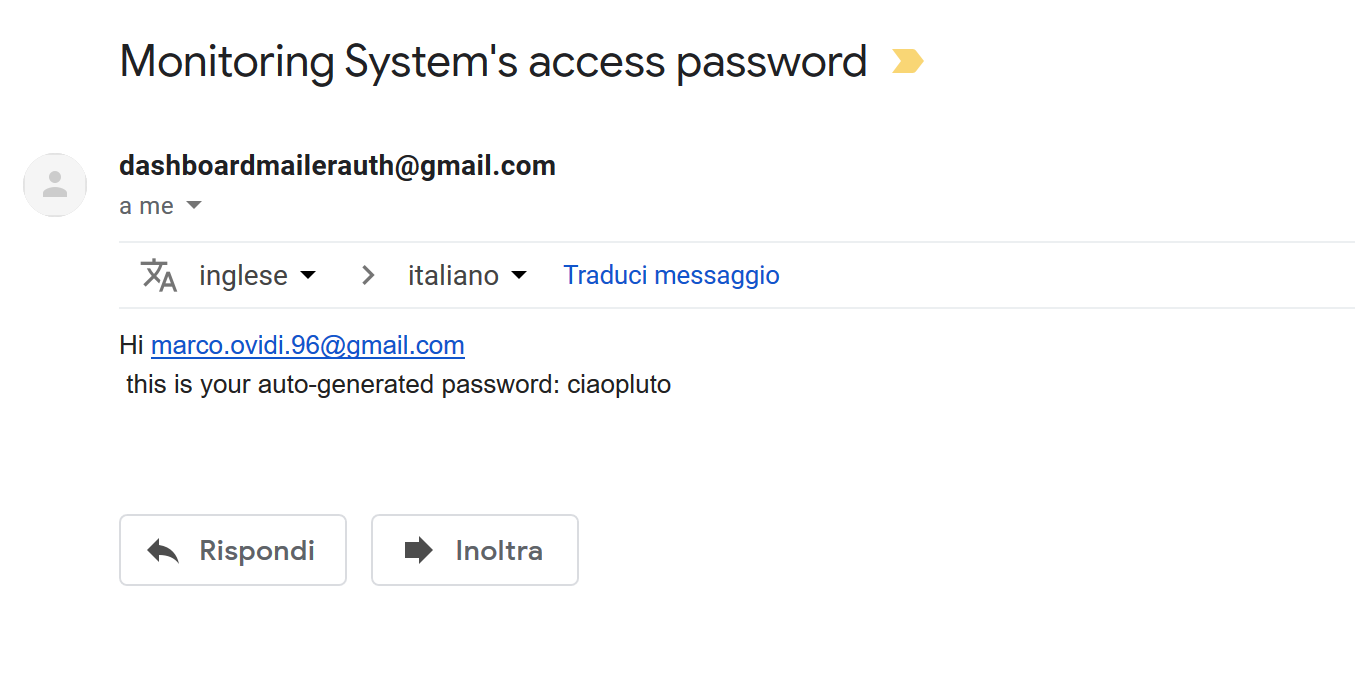


Figure: Example of what happens if we have both a warning and an alert in the same area. The area is colored red, the Warning color, because it is more urgent.

When a user logs in (so right after the Login screen), they will be able to see a list of the zones that they are allowed to control. This is how we implemented all the functional and non-functional requirements that are related to the GUI.

1. Password Generation

When we register a user, we use their email as username, and a password is randomly generated from the system, with a randomized length from 7 to 10 characters. Only the admin can register a user, so their password is sent to the new manager by email.



Here on the left there is an example of what that email looks like. Of course, we created a new email address for this purpose.

The password is also stored to the MariaDB with a md5 cryptographic hash function. This is done because of the **Security** non-functional requirement. This way, we ensure that intruders or brute-force attacks are not very effective.

Future Implementations

We’ve seen what our system can do and how it works. There are many improvements that can be made in a relatively short amount of time to build on what we have done and move it even further.

Sensors and Server use the HTTP protocol to communicate (as we mentioned in the introduction, this could also be changed with a socket implementation, that is, however, less intuitive with Apache Tomcat). In order to satisfy the non-functional requirement of **Security**, we envisioned the client and the server connecting with the *HTTPS* protocol instead. This, paired with the random password generator that is already implemented, would grant acceptable security performances.

Multi-levels managers are not possible in our system. That means, a Building Manager cannot be an Area Manager too. We think this would not really be useful, but if the potential client envisions a system where this is necessary, it would not be too hard to implement, so we’re leaving it in the “Future Implementations” section.