H^2

- Health Help -

Stefano Bernagozzi	stefano.bernagozzi@studio.unibo.it
Manuel Bottazzi	manuel.bottazzi@studio.unibo.it
Giulia Lucchi	giulia.lucchi7@studio.unibo.it
Margherita Pecorelli	margherita.pecorelli@studio.unibo.it

Report of the final project for the courses
"Pervasive Computing" and "Laboratorio di sistemi software"

A.Y. 2017/2018

Alma Mater Studiorum - Università di Bologna
via Sacchi,3 – 47521 Cesena, Italia

Index

1 Vision

Our initial vision is a sanitary world where each individual has associated his clinical history from his birth, accessible by all medical personnel from all over the world, independently from his geographical position, through an universal identification system.
A system like this gave advantages to both medical staff and patient: in this way doctors are able to intervene readily having conscience of the clinical history of the patient and support from the system. Consequently the patient could be treated in the best way, even if the patient isn't able to provide autonomously his essential information.
Another fundamental feature of this system is the gathering and monitoring of people's parameters continuously, automatically and in a non invasive manner. In this way it is possible to obtain a system able to self learn, in order to identify urgency or abnormalities, potentially calling help autonomously, or helping the doctor in its initial diagnosis.
In current sanitary world there aren't assumptions to create an unified system of this magnitude. Because of that we decided to focus our attention on a specific area of interest of this huge vision: self-care.
The challenge that we have set ourselves is made up of creating a system both pervasive and non-invasive, so that it can be used by anyone, even people with few technological skills. Moreover thanks to this system we want to give a daily support to pathological patients, giving advantages to both hospital and patient, allowing the last one to cure and control itself autonomously and to delegate the system the communication with the doctor.

2 Goals

We want to realize a distributed system capable to connect chronic ill patients to their own doctor, in a simple way. The main idea is to provide sensors to patients for acquiring and monitoring disease-specific parameters. Those sensors, after the acquisition of the necessary data, will send them to a specific analysis module and to the assisted's clinical folder, which could be accessible to both doctor and patient. The analysis module is designated to collect all patients' data in order to possibly analyse them for discovering anomalies or emergencies. The system has the capability to perform a preliminary analysis of the parameters and, when it detects anomalous situations, it can have different behaviours depending on the seriousness level of the circumstances. In case that there are no high risks the system will only notify the patient with some advices about his habits trying to stabilize his parameters, otherwise it will also alert the doctor or the hospital.

3 User stories

3.1 Patient (P):

1. I want to record one or more of my vital parameters at home and I want them to be saved in my clinical folder at hospital;
2. I want to monitor one or more of my vital parameters at home;
3. I want that a doctor can be able to remotely see my clinical folder;
4. if necessary, I want that the system sends alerts and emergency alerts to the doctor, everywhere I am;
5. I don't want to have many configurations to do;
6. I want the system easy to use;
7. I want my sensors wireless connected, I want to be free from cables;
8. I want sensors as small as possible;
9. I want my sensors with long battery life;
10. I want to be notified in case of parameters out of bound;
11. I want to see my clinical folder;
12. I want that nobody excepts me and the doctors can access my data.

3.2 Doctor (D):

1. I want to remotely visualise my patients' data in a simple way, that is I want the clearest and most intuitive graphics, even in emergency and stressful situation;
2. I want the system able to manage different kind of illness, in other words I want the system and the analyser to be configurable with a specific illness;
3. I want the possibility that the patient can use many sensors simultaneously;
4. I want to be able to use different sensors depending on the specific illness;
5. I want to be alerted if one of my patients has some parameters out of bound;
6. I want to be emergency alerted as soon as possible in critical situations;
7. I want to see their clinical history;
8. I want to visualize the data very quickly, in other words, I want to retrieve relevant data as fast as possible;

9. I want to see the data with immediate impact;
10. I want to visualize patient's parameters with different scope (last hour, last day, last month, ...);
11. I want the system reliable, so that it won't lost any data;
12. I want the system easy to use for each patient;
13. I want the system to do a preliminary data analysis;
14. I want the system to give autonomously simple medical advice to the patient without the supervision of the doctor;
15. I want to have the possibility to send simple medical advice and drugs to the patient;
16. I want to remotely visualise my patients' clinical folder.

3.3 User Stories Analysis

Sensor: device able to measure a specific parameter in the least invasive way and when possible, it should be portable.

Record: the patient wants to save his parameters in a persistent way, in order to visualize them whenever he wants. Further, the patient doesn't want to save parameters by himself, but he wants the record to be automatic.

Vital Parameters: values of physical quantities measured and emitted by specific sensors.

Home: every place where the patient can measure his own parameters (e.g. home, office...).

Monitoring: the patient wants to see real-time the values measured by sensors.

Doctor: anyone belonging to the national sanitary service interested to patient's situation or anyone that wants to visualize patients' parameters. (e.g. general practitioner, nurse, surgeon, ambulance personal, ...)

Remotely: the user doesn't have to stay physically in the same place of the device which is communicating with.

System: the entire system that we are building.

Alert: a communication to both doctor and patient that needs to be seen, but the patient has not a life-threatening issue. It corresponds to white and green codes in Italian emergency code [3].

Emergency Alert: a call to rescuers and a communication to both doctor and patient that needs to be seen immediately because the patient is in life danger or critical situation. It corresponds to yellow, blue and red codes in Italian emergency code [3].

Easy configurations/easy to use: the patient wants to connect sensor to the system in the simplest, fastest and most intuitive way. The patient doesn't want many buttons or cables and he doesn't want to make many operations.

Clinical folder: all about my clinical past, including my recorded parameters, taken medicines and previous contracted illness.

Patient: person with a specific chronic illness (permanent or not), whose parameters must be monitored with a certain frequency.

Analyser: component capable to make a preliminary parameters' analysis, configurable according to the specific illness and that provides a basic support to the doctor.

Immediate impact: the doctor wants the most significant data remarked and he wants a graphical support(e.g. evolution chart, different colours, ...).

Simple Medical Advice: advices on life style and medicine dosage in order to stabilise parameters out of bound.

4 Requirements

4.1 Glossary

Access Device: a device that allows the user to access and display data.

Advice: a basic suggestion to the patient about his lifestyle.

Alert: a communication to both doctor and patient that needs to be seen as soon as possible, but the patient has not a life-threatening issue. (level 2)

Business Logic: a entity that capsulizes the system's logic.

Control Unit: a physical entity that collects all patient's data from the sensors, it does minimal analysis over them and it sends those data to patient's folder.

Doctors: a medical staff involved in patient's treatment.

Emergency Alert: a communication to both doctor and patient that needs to be seen immediately because the patient is in life danger. (level 3)

Feedback: a message that reports the actual situation (measured values) and possibly gives an interpretation of that data and a related advice.

Folder: a collection of historical data related to a single patient.

Patient: an individual that must be monitored.

Sensor: a device related to a physiological parameters that retrieves data from the patient and sends them to system's business logic.

Sensor Data: data acquired from sensors to be sent to system's business logic.

4.2 Requirements Introduction

From user story analysis we have derived our requirements, that subsequently we have divided them into Business, Functional, Non-Functional and Implementation requirements. The division is made following the schema seen in lectures.

- Business Requirements: project's high-level goals, customer's high hope
- Functional Requirements: detailed statements of desired capabilities
- Non-functional Requirements: detailed statements about quality of behaviour or constraints (performance, reliability, security)
- Implementation Requirements: temporary aspects needed in the process (features, training, material)

4.3 Business Requirements

1. Capability of the system to save patient's data permanently in order to have an history; **(P1, P11)**
2. remote access to patient's history from doctors and from the patient itself; **(P2, P3, P11, D2, D10)**
3. capability of the system to analyse data in order to give a first interpretation without medical support;**(D13, D14)**
4. capability of the system to autonomously communicate (alerts and emergency alerts) with both patient and sanitary personnel;**(P4, P10, D5, D6)**
5. measure vital parameters from sensors; **(P1)**
6. communication from doctor to patient; **(D15)**
7. capability to handle multiple patients and doctors;
8. automatic association between patients and their doctors taken from sanitary service; **(P6, D12)**
9. one patient can be related with one or more doctors;
10. only authorized users can access to the clinical folder; **(P12, P11, D7)**
11. ability to separate patients' parameters into their own clinical folders; **(P11 D1, D10)**
12. ability to use many and different sensors. **(D3, D4)**

4.4 Functional Requirements

1. Automatic send of parameters from patient to the data centre; **(P5, P6, D12)**
2. remote visualisation of the patient's clinical folder from both doctors and patient; **(P1, P2, P3, P11, D1, D7)**
3. send a communication to the doctor if parameters are out of bound; **(P4, D5, D6)**
4. send feedback to the patient based on data analysis; **(P10, D13, D14)**
5. doctor can sends advices and drugs to the patient; **D15**
6. connect and disconnect sensors dynamically; **(P5, P6, P7)**
7. use sensors with different kind of output, technology and parameters; **(D2, D4)**

8. the user must be registered in order to login; **(P12)**
9. the user needs to login into the system before accessing the data; **(P12)**
10. handling of multiple doctors and multiple patients;
11. user can access with multiple devices simultaneously;
12. user can access with different type of devices;
13. customized data analysis for each patient; **(D2, D4)**
14. automatic management of the connection to an accessible network; **(P6, D11, D12)**
15. even in absence of connection between patient and data centre the system must provide a first analysis to the patient; **(D13, P10, D14)**
16. even in absence of connection between patient or doctor and data centre the system, in case of emergency (detected by the analyser) must try to call the rescuers; **(P4, D13, P10, D14)**
17. in case of emergency, the patient's data can be visualised by doctors that can be involved in the rescue; **(P4)**
18. the system must avoid to call rescuers when is not absolutely necessary;
19. doctor can see the communications even later if in that moment he's not connected to the application;
20. doctor can visualize the communications and decide to not delete them, in order to visualize them again later.

4.5 Non-functional Requirements

1. scalability on the number of sensors connected to the system: we assume that up to 5 sensors per person are an appropriate number; **(D4, D3)**
2. sensors should be connected to the system without any cable; **(P7, P6, D12)**
3. visualise data from multiple devices simultaneously;
4. visualise data from different type of devices;
5. system reactivity is measured according to those operations:
   (a) remote access to the folder and data saving on it: within 1 minute;
   (b) notification to patient: within 1 minute;
   (c) data analysis from the system: depends on type of analysis, generally within 30 minutes;

(d) dispatch of alerts: within 1 minute after the anomaly is detected;

(e) call for help: within 30 seconds after the decision to call them is taken.

6. The system must be user friendly, suitable also for people with few technological skills (both patients and doctors); **(P6, D12, P5)**;
7. the computer security is not managed;
8. only the hospital can register patients and doctors into the system;
9. only the sanitary system can configure the system based of the illness;
10. local data saving even if the connection between the business logic and the folder goes down; **(D11)**
11. basic analysis should be done also without any type of connection; **(D13, P10)**
12. the Wi-Fi network absence is managed with a GSM network connection located inside the control unit; **(P4, D6)**
13. different kinds of sensors according to the specific illness; **(D2, D4)**
14. from the raise of an emergency, the decision to call help is done between 2 and 3 minutes to avoid false alerts and the user could stop the process in every moment;
15. sensor's parameters can be sent as stream or occasional detections;
16. capability to save data locally in the control unit until a network is available**(D11)**;
17. sensors must be personal;
18. any form of communication will arrive in any logged device;
19. communication will arrive only to involved users; **(P12)**
20. attempt to alert rescuers with all the possible network connections;
21. scalability on the number of patients and doctors connected to the system: in our vision, this system should be used by the national sanitary system (or even from the whole world), but for now our target is a single hospital, so we assume that 1000 users (patients chronically ill and doctors) are an appropriate number.

4.6 Notes

User stories P8 and P9 are not managed by us but we will try to satisfy them anyway.

5 Requirement Analysis

5.1 Domain model

During the requirement analysis we have deduced the main entities and their relationships that we will analyse below.

Beginning from the first requisites, from B1 and F1 we extract 2 different entities: data centre, that saves data persistently in order to have an history, and patient that for now represents the data of the patient. We also knows that these 2 entities interact, particularly the patient sends the data to the folder.

Subsequently, taking in analysis requirements B2 and F2, we obtain a third entity that represents the doctor described in user stories analysis. Going deeper we notice that the data centre is remote regarding patient and doctor and also that there is an interaction between patient or doctor and the data centre, for retrieving clinical history.

During the analysis of B3 and part of F4 we also realise that the data centre is not only a storage but has also a component for analyse data. Afterwards, requirements B4, F3 and a part of F4 lead us to understand that there is another interaction between data centre and doctor or patient, specifically the data centre sends the notifications to both them. Furthermore there is another conclusion that we can retrieve, that is data centre is composed by at least 3 entities: a data collector, an analyser and a notification manager.

From requirements B5, F6 and F7 another entity is revealed: the sensor. Substantially they have to measure the parameters of the patient, they are connected to him because they are patient-specified. It is important to specify that the output of the sensor is variable due to the possibility to have sensors not made by us, therefore is necessary to have a standardisation in the patient entity in order to have a normalised database.

Finally, requirements B6 and F5 show that there is an interaction between doctor and patient, particularly the doctor sends advices to the patient.

From this analysis we have retrieved the following scheme:

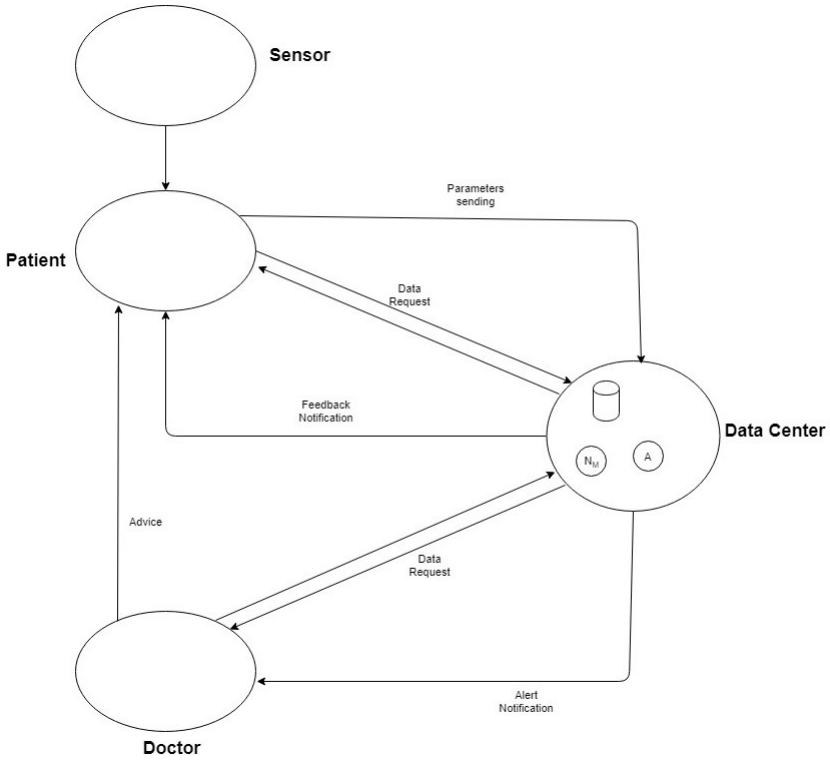

Figure 1: Structure of the domain model

5.2 Formal definition of the model

In order to formalize the model shown above, we have used QActors, a formal meta-modelling language that allows to represent a logical model in an actor-like formalisation. It is capable to express requirements in term of component's structure, to show principal interactions between different components and their specific behaviours. In particular the language is as expressive as UML formalism. Moreover, this language is comprehensible and executable by computers, so it allows to create a first prototype of the system in order to exhibit a demo of the system itself to the stakeholder. We have chosen this particular language thanks to this feature because our principal client (and domain expert) are doctors with a poor knowledge of technical IT formalism and it is very important to us to have something to show them in order to check the resulting model, but also a formal and comprehensible model for IT experts is essential.
An in-depth analysis and description of QActors models and modelling language are presented in [1].
Now, we are going to illustrate the main components, represented in our model with four different logical contexts.

5.2.1 Contexts

Sensor

- **Structure**: it is an entity composed by a data sender.
- **Interaction**: it sends parameters to the patient.
- **Behaviour**: it measures patient's parameters and sends to him. Parameters are composed by value and type (i.e. temperature or glycemia)

Patient

- **Structure**: the patient is an entity composed by a data sender, a data retriever and a notification receiver.
- **Interaction**: it sends parameters to data centre, it sends a history request to data centre, it receives from it all the parameters and it receives notifications from both data centre and doctor.
- **Behaviour**: the patient sends data to the data centre and can request its parameters' history to it. Moreover it receives notifications from both doctor and data centre and displays them.

Data Centre

- **Structure**: the data centre is composed by a storage, an analyser and a notification handler.
- **Interaction**: it receives parameters from patient, sends data history when patient and doctor require them and sends notifications to both patient and doctor following a certain standard.
- **Behaviour**: it saves data into the storage after receiving them from the patient, it is able to analyse parameters and, after the analysis, it can send notifications to both patient and doctor. In addiction, when a user requires data, it sends to him the parameters required.

Doctor

- **Structure**: the doctor is composed by a data retriever, a notification sender and a notification retriever.
- **Interaction**: it receives parameters and notifications from the data centre. It can send advices to the patient. It can send a history request to data centre and receives from it all the parameters of a specific patient.
- **Behaviour**: it can request all patient's data to the data centre and it can receive notification from it when patient's parameters are not regular. It can also send advices to the patient.

```
System h2

Dispatch sending_patient_parameter : sending_patient_parameter(PARAMETER,
    TIMESTAMP)
Dispatch patient_parameters_request : patient_parameters_request
Dispatch patient_parameters_response : patient_parameters_response(LIST)
Dispatch notification : notification(DESCRIPTION)

Event    patient_data_request : patient_data_request
Event    doctor_data_request : doctor_data_request
Event    doctor_notification_sender : doctor_notification_sender(NOTIFICATION
    )


//requirements: business 1,2,3,4 e functional 1,2,3,4,5
Context ctx_data_centre ip[host="localhost" port=8070] -g cyan
Context ctx_patient   ip[host="localhost" port=1234] //-g green
Context ctx_doctor    ip[host="localhost" port=3456] //-g white

EventHandler evh for patient_data_request, doctor_data_request,
    doctor_notification_sender -print;

QActor qpatient_sender context ctx_patient { ... }

QActor qpatient_data_retriever context ctx_patient { ... }

QActor qpatient_notification_receiver context ctx_patient { ... }

QActor qdc_data_receiver context ctx_data_centre { ... }

QActor qdc_analyser context ctx_data_centre { ... }

QActor qdc_notification_manager context ctx_data_centre { ... }

QActor qdoctor_data_retriever context ctx_doctor { ... }

QActor qdoctor_notification_receiver context ctx_doctor { ... }

QActor qdoctor_notification_sender context ctx_doctor { ... }
```

5.2.2 QActors

Going deeper, we will analyse each qactor.

qpatient_ sender

- **Structure**: *qpatient_ sender* is composed by a data sender.
- **Interaction**: it sends messages with parameters to *qdc_ data_ receiver*.
- **Behaviour**: when there is a parameter to save or analyse, it sends it to the data centre and that it returns waiting other parameters to send.

```
QActor qpatient_sender context ctx_patient {

  Plan init normal [
    println(" --- [PS] initializing --- ")
  ] switchTo sendData

  Plan sendData [
    println(" --- [PS] sending data --- ");
    forward qdc_data_receiver -m sending_patient_parameter :
     sending_patient_parameter(37, "12/12/12");
    delay 2500;
    forward qdc_data_receiver -m sending_patient_parameter :
     sending_patient_parameter(38, "12/12/12");
    delay 2500;
    forward qdc_data_receiver -m sending_patient_parameter :
     sending_patient_parameter(36, "12/12/12");
    delay 2500
  ] finally repeatPlan

}
```

qpatient_ data_ retriever

- **Structure**: *qpatient_ data_ retriever* is composed by a request sender, a response receiver and a visualizer.
- **Interaction**: it sends messages to *qdc_ data_ receiver* and receives messages back from it.
- **Behaviour**: when the patient wants to visualize its parameters history, it sends through a panel a request to the data centre and it waits for a response. When the response arrives, it displays the data and then the patient is able to make another request.

```
QActor qpatient_data_retriever context ctx_patient {

  Plan init normal [
    println(" --- [PR] initializing --- ");
    actorOp createGUI
  ] switchTo waitUserRequest

  Plan waitUserRequest [
    println(" --- [PR] waiting for user request --- ")
  ] transition stopAfter 86400000
    whenEvent patient_data_request -> requestData
    finally repeatPlan

  Plan requestData [
    println(" --- [PR] requesting data --- ");
    forward qdc_data_receiver -m patient_parameters_request :
     patient_parameters_request
  ] switchTo waitResponse

  Plan waitResponse resumeLastPlan [
    println(" --- [PR] waiting for a response --- ")
  ] transition stopAfter 600000
    whenMsg patient_parameters_response -> visualiseData

  Plan visualiseData [
    println(" --- [PR] visualizing data --- ");
    onMsg patient_parameters_response : patient_parameters_response(LIST) ->
     actorOp print(LIST)
  ] switchTo waitUserRequest
}
```

qpatient_notification_receiver

- **Structure**: *qpatient_notification_receiver* is composed by a notification receiver and a visualizer.
- **Interaction**: it receives messages from *qdc_notification_manager* and *qdoctor_notification_sender*.
- **Behaviour**: it waits for a notification and displays it when received. After that, it returns again available to process another notification.

```
QActor qpatient_notification_receiver context ctx_patient {

  Plan init normal [
    println(" --- [PNR] initializing --- ")
  ] switchTo waitNotification

  Plan waitNotification [
    println(" --- [PNR] waiting notification --- ")
  ] transition stopAfter 86400000
    whenMsg notification -> visualizeNotification
    finally repeatPlan

  Plan visualizeNotification resumeLastPlan [
    println(" --- [PNR] receive notification --- ");
    onMsg notification : notification(DESCRIPTION) -> println(DESCRIPTION)
  ]
}
```

qdc_ data_ receiver

- **Structure**: *qdc_ data_ receiver* is composed by a messages receiver, a parameter saver, a parameter sender and a request data handler.
- **Interaction**: it sends messages to *qdc_ analyser*, it receives messages from *qpatient_ sender*, from *qpatient_ data_ retriever* and from *qdoctor_ data_ retriever* and it replies to *qpatient_ data_ retriever* and *qdoctor_ data_ retriever*.
- **Behaviour**: it waits a message for saving data or for visualize data history. When it receives the first one, it stores the data in its knowledge base and forwards it to the analyser, otherwise, when it receives the second one, it retrieves all data from its knowledge base and sends them back to the requester.

```
QActor qdc_data_receiver context ctx_data_centre {

  Rules {
    request_parameters(LIST) :- findall(patient_parameter(X,Y),
     patient_parameter(X, Y),LIST).
  }

  Plan init normal [
    println(" --- [DC] initializing --- ")
  ] switchTo waitMessages

  Plan waitMessages [
    println(" --- [DC] waiting messages --- ")
  ] transition stopAfter 86400000
    whenMsg sending_patient_parameter -> saveParameter,
    whenMsg patient_parameters_request -> handleRequest
    finally repeatPlan

  Plan saveParameter [
    println(" --- [DC] saving parameters --- ");
    onMsg sending_patient_parameter : sending_patient_parameter(PARAMETER,
     TIMESTAMP) -> demo asserta(patient_parameter(PARAMETER, TIMESTAMP))
  ] switchTo sendToAnalyser

  Plan sendToAnalyser [
    println(" --- [DC] sending to analyzer --- ");
    [!? patient_parameter(PARAMETER, TIMESTAMP)] forward qdc_analyser -m
     sending_patient_parameter : sending_patient_parameter(PARAMETER,
     TIMESTAMP)
  ] switchTo waitMessages

  Plan handleRequest resumeLastPlan [
    println(" --- [DC] handling parameters request --- ");
    onMsg patient_parameters_request : patient_parameters_request -> actorOp
     memoCurrentCaller;
    [!? request_parameters(LIST)] replyToCaller -m
     patient_parameters_response : patient_parameters_response(LIST)
  ]
}
```

qdc_ analyser

- **Structure**: *qdc_ analyser* is composed by a message receiver, an analyser and a message sender.
- **Interaction**: it receives messages from *qdc_ data_ receiver* and sends messages to *qdc_ notification_ manager*.
- **Behaviour**: it waits for a message with patient's data in order to analyse them. When it receives the parameters, this component analyses those data and sends the result to *qdc_ notification_ manager*.

```
QActor qdc_analyser context ctx_data_centre {

  Rules {
    analyser(VAL, febbre) :- eval(gt, VAL, 37),!.
    analyser(VAL, non_febbre) :- eval(gt, 37, VAL), !.
    analyser(37, non_febbre).

    analyse(RESP) :-
      parameter(PARAMETER),
      analyser(PARAMETER, RESP).
  }

  Plan init normal [
    println(" --- [A] initializing --- ")
  ] switchTo waitParameter

  Plan waitParameter [
    println(" --- [A] waiting parameter--- ")
  ] transition stopAfter 86400000
    whenMsg sending_patient_parameter -> analyse
    finally repeatPlan

  Plan analyse resumeLastPlan [
    println(" --- [A] analyzing parameter--- ");
    onMsg sending_patient_parameter : sending_patient_parameter(PARAMETER,
     TIMESTAMP) -> addRule parameter(PARAMETER);
    [!? analyse(RESP)] println(RESP);
    [!? analyse(RESP)] forward qdc_notification_manager -m notification :
     notification(RESP);
    [!? analyse(RESP)] removeRule parameter(PARAMETER)
  ]
}
```

qdc_ notification_ manager

- **Structure**: *qdc_ notification_ manager* is composed by a message receiver and a message sender.
- **Interaction**: it receives messages from *qdc_ analyser* and sends messages to *qdoctor_ notification_ receiver* and to *qpatient_ notification_ receiver*.
- **Behaviour**: it handles notifications so, when it receives a message, it sends it to the patient and check if it has to forward it to the doctor too.

```
QActor qdc_notification_manager context ctx_data_centre {
  Rules {
    send_to_doctor(febbre) :- notification(febbre).
  }

  Plan init normal [
    println(" --- [NM] initializing --- ")
  ] switchTo waitNotification

  Plan waitNotification [
    println(" --- [NM] waiting notification --- ")
  ] transition stopAfter 86400000
    whenMsg notification -> sendNotification
    finally repeatPlan

  Plan sendNotification resumeLastPlan [
    println(" --- [NM] sending notification --- ");
    onMsg notification : notification(DESCRIPTION) -> addRule notification(
     DESCRIPTION);
    [!? send_to_doctor(DESCRIPTION)] forward qdoctor_notification_receiver -m
      notification : notification(DESCRIPTION);
    [?? notification(DESCRIPTION)] forward qpatient_notification_receiver -m
     notification : notification(DESCRIPTION)
  ]
}
```

qdoctor_data_retriever

- **Structure**: *qdoctor_data_retriever* is composed by a request sender, a response receiver and a visualizer.
- **Interaction**: it sends messages to *qdc_data_receiver* and receives messages from it.
- **Behaviour**: when the doctor wants to visualize its patient's parameters history, it sends through a panel a request to the data centre and it waits for a response. When the response arrives, it displays the data and then it is able to make another request.

```
QActor qdoctor_data_retriever context ctx_doctor {

  Plan init normal [
    println(" --- [D] initializing --- ") ;
    actorOp createGUI
  ] switchTo waitUserRequest

  Plan waitUserRequest [
    println(" --- [D] waiting for user request --- ")
  ] transition stopAfter 86400000
    whenEvent doctor_data_request -> requestData
    finally repeatPlan

  Plan requestData [
    println(" --- [D] requesting data --- ");
    forward qdc_data_receiver -m patient_parameters_request :
     patient_parameters_request
  ] switchTo waitResponse

  Plan waitResponse resumeLastPlan [
    println(" --- [D] waiting for a response --- ")
  ] transition stopAfter 60000
    whenMsg patient_parameters_response -> visualiseData

  Plan visualiseData [
    println(" --- [D] visualizing data --- ");
    onMsg patient_parameters_response : patient_parameters_response(LIST) ->
     actorOp print(LIST)
  ] switchTo waitUserRequest

}
```

qdoctor_notification_receiver

- **Structure**: *qdoctor_notification_receiver* is composed by a notification receiver and a visualizer.
- **Interaction**: it receives messages from *qdc_notification_manager*.
- **Behaviour**: it waits for notification and displays it when received. After that, it returns again available to process another notification.

```
QActor qdoctor_notification_receiver context ctx_doctor {

  Plan init normal [
    println(" --- [DNR] initializing --- ")
  ] switchTo waitNotification

  Plan waitNotification [
    println(" --- [DNR] waiting notification --- ")
  ] transition stopAfter 86400000
    whenMsg notification -> receiveNotification
    finally repeatPlan

  Plan receiveNotification resumeLastPlan [
    println(" --- [DNR] receive notification --- ");
    onMsg notification : notification(DESCRIPTION) -> println(DESCRIPTION)
  ]
}
```

qdoctor_ notification_ sender

- **Structure**: *qdoctor_ notification_ sender* is composed by a message sender.
- **Interaction**: it sends messages to *qpatient_ notification_ receiver*.
- **Behaviour**: when the doctor wants to send an advice to the patient, he inserts the advice in the input box, and the *qdoctor_ notification_ sender* sends the notification to the patient.

```
QActor qdoctor_notification_sender context ctx_doctor {

  Plan init normal [
    println(" --- [DNS] initializing --- ");
    actorOp createGUI
  ] switchTo waitInput

  Plan waitInput [
    println(" --- [DNS] waiting input --- ")
  ] transition stopAfter 86400000
    whenEvent doctor_notification_sender -> sendNotification
    finally repeatPlan

  Plan sendNotification [
    println(" --- [DNS] sending notification --- ");
    onEvent doctor_notification_sender : doctor_notification_sender(
     NOTIFICATION) -> forward qpatient_notification_receiver -m notification
     : notification(NOTIFICATION);
    actorOp print("advice send to patient")
  ] switchTo waitInput
}
```

5.2.3 Other edited files

In order to allow the user to interact with the system, we have created a custom GUI for requesting data history and sending notification.
Because of that, we have edited classes *Qdoctor_ data_ retriever*, *Qpatient_ data_ retriever*, *Qdoctor_ notification_ sender* and we created classes *CustomGUISupportPatient*, *CustomGUISupportDoctor*, *CustomGUISupportAdvice*.
An example is shown below.

```
public class CustomGUISupportAdvice extends SituatedPlainObject {

  private IActivityBase cmdHandler;
  private IBasicEnvAwt envAwt;
  private QActorContext ctx;

  public CustomGUISupportAdvice(IBasicEnvAwt env, QActorContext myCtx) {
    super(env);
    envAwt = env;
    init();
    this.ctx = myCtx;
  }

  protected void init(){
    cmdHandler = new CmdHandler(envAwt);
    setCommandUI();
    setInputUI();
  }

  protected void setCommandUI(){
    envAwt.addCmdPanel("commandPanel", new String[]{"SEND ADVICE"},
     cmdHandler);
  }

  protected void setInputUI(){
    envAwt.addInputPanel(50);
  }

  public void printOnGUI(String str) {
    println(str);
  }

  private class CmdHandler extends SituatedPlainObject implements
     IActivityBase {

    public CmdHandler(IBasicEnvAwt env) {
      super(env);
    }

    @Override
    public void execAction(String cmd) {
      String input = env.readln();
      println("CmdHandler -> " + cmd + " input= " + input);
      try {
        QActorUtils.raiseEvent(ctx, "input", "doctor_notification_sender", "
     doctor_notification_sender(" + input + ")");
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

```
public class Qdoctor_notification_sender extends
     AbstractQdoctor_notification_sender {

  private CustomGUISupportAdvice gui;

  public Qdoctor_notification_sender(String actorId, QActorContext myCtx,
     IOutputEnvView outEnvView )  throws Exception{
    super(actorId, myCtx, outEnvView);
  }

/*
 * ADDED BY THE APPLICATION DESIGNER
 */

  protected void addInputPanel(int size) {}

  protected void addCmdPanel() {}

  public void print(String str) {
    gui.printOnGUI(str);
  }

  public void createGUI() {
    IBasicEnvAwt env = outEnvView.getEnv();
    if(env == null) {
      env = new EnvFrame("H2-advice", Color.white, Color.black);
      env.init();
      ((EnvFrame)env).setSize(800,430);
    }
    env.writeOnStatusBar("H2-advice" + " | working ... ",14);
    gui = new CustomGUISupportAdvice(env,myCtx);
  }
}
```

6 Problem Analysis

Starting from the previous requirement analysis, we are going to face the problems that arise taking in consideration other requirements.

The first problem appeared is the automatic association between patients and doctors. It comes from the analysis of requirements B8 and B9. In order to solve the problem we understood that another entity has to be added, in particular it is association manager that has the work of retrieving the association between patient and doctor from an external storage. For us is important to use the external storage because it can be already made by the customer or it can change during the time. Another consequence of this analysis is that the doctor advice has to pass from the data centre to verify the association between him and the patient.

From requirements B10 and B11 descend the problem that the folders must be visible only to the owner and the connected doctors. This problem has been already solved by the analysis done right above.

Due to requirements F8 and F9 derives the problem to manage the registration and the login of the user. It is solved by adding a database to the data centre, where registered user are saved. Logins are managed by creating another entity, the sanitary service, that registers the users with their credentials already used to identify them in the sanitary service. We suppose that this entity already exists, so it is only simulated by us.

Through the exploration of requirements F15 and F16 raise the problem that the analysis of the parameters should be made also without connection. This leads to the creation of two different analysers: one locally in the patient and another one in the data centre for analyse all patients' data. The last one is not implemented by us.

Finally, after analysing requirement F15 and F12 we discovered that the patient must be split into 2 parts, one local in the patient that manages the side that must work even in absence of connection and the other one that can be accessed anywhere.

Other problems raised from the analysis of the requirements are:

- **B7, F10**: the system has to manage a large amount of patients and doctors, it should manage an hospital with about 100 doctors and 1000 patients;
- **B12, F6, F7**: the user must be able to connect many and different type of sensors, and so the analyser has to manage them;
- **F11**: the system has to handle accesses from more than one device simultaneously connected to the same account;
- **F12**: the account can be seen from different type of devices, like tablets, smart phones, laptops, etc...;
- **F13**: the analyser should be specific, it has to consider gender, age, illnesses and other important factors (e.g. the user is running and its b.p.m.

value is above the maximum);

- **F14**: sensors has to handle their connection to the system automatically with the minimum interaction of the patient;
- **F17**: in case of emergency patient's history can be visualised by all the doctors possibly involved in the rescue;
- **NF2**: sensors should be wireless connected, so they need a minimum device for handling wireless communications;
- **NF6**: the application must be user friendly for both doctors and patients;
- **NF7**: the system should be secure in the way that nobody can steal or edit users' data.

The problems cited above doesn't affects the logical architecture, but only on technological aspects and implementation.

6.1 Logical Architecture

After the analysis of the problems seen before, we have edited the domain model in order to solve them. After this process the result is the following logical architecture:

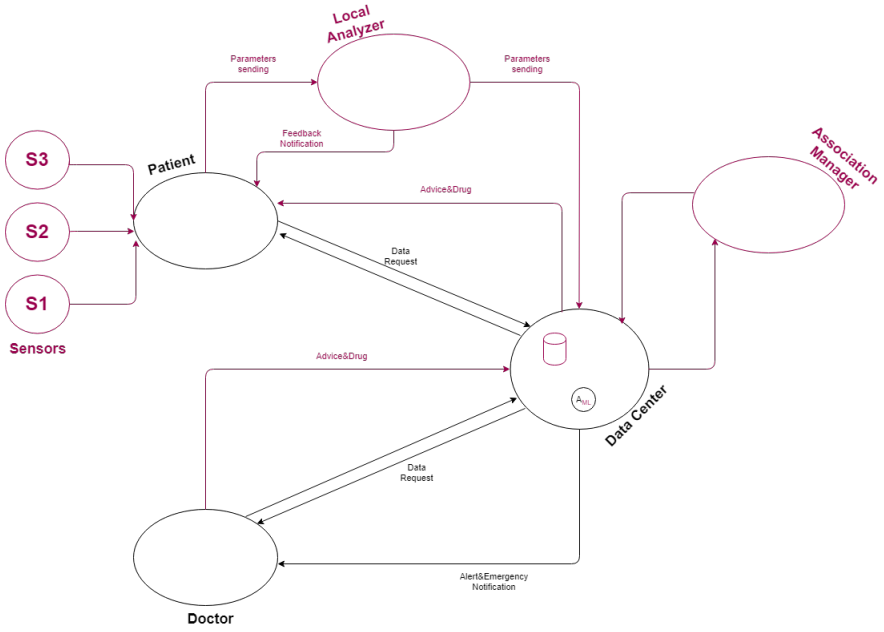

Figure 2: Structure of logical architecture

6.2 Formal definition of the logical architecture

As done previously in the domain model, we have formalised also the logical architecture with QActors meta-language.

6.2.1 Contexts

External association Knowledge Base: actually, it isn't a part of our project, but we need it to implement something that replicates its behaviour.

- **Structure**: it is an entity composed by a data sender and data retriever.
- **Interaction**: it receives requests from data centre and sends answers to it.
- **Behaviour**: the data centre sends requests to the External association Knowledge Base to know if specific doctor and patient are associated or to know all doctors associated to a specific patient.

User Data Knowledge Base

- **Structure**: it is composed by a data sender and a data retriever.
- **Interaction**: it receives requests from the data centre and answers to it.
- **Behaviour**: it stores all the data related to the patients (parameters, registration, login...). All this data are sent by the data centre. It is the knowledge base of the data centre.

Analyser

- **Structure**: it is an entity composed by a data retriever, a data sender and an analyser.
- **Interaction**: it receives requests from the patient and replies to it. Moreover, it can send messages to patient's doctors.
- **Behaviour**: it makes the first analysis of all the parameter that the patient sends. Moreover, it sends the results to the patient (with a very small description) and when it detects some parameter out of normal range, it sends an emergency/alert message to all patient's doctors.

Data centre The data centre is slightly edited since the previous version, in particular it has the following changes:

- the analyser is not anymore in this context,
- it receives advices and drugs from doctors and it forwards them to the specific, patients
- it manages the analyser notifications and forwards them to patients and doctors.

Other Contexts

We have also edited two other contexts explained in the previous section, in particular patient and doctor, by dividing them into 3 contexts each:

- **ctx_patient and ctx_doctor** that manages the login of the different patients and doctor and contains the QActors at runtime.
- **ctx_patient_init and ctx_doctor_init** that has the creation of the actors
- **ctx_patient_model and ctx_doctor_model** that contains all the code of the QActors. They are in another context because they must be initialized at runtime.

Sensor context is the same as before.

6.2.2 QActors

Going deeper we will analyse each QActor:

qassociation

- **Structure**: it is an entity composed by a data sender and data retriever.
- **Interaction**: it receives requests from data centre and sends answers to it.
- **Behaviour**: the data centre sends requests to the External association Knowledge Base to know if specific doctor and patient are associated or to know all doctors associated to a specific patient.

qassociation_manager

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it interacts with *qassociation* to verify the association between patients and doctors and it interacts with other part of the data centre (i.e. *qdc_data_collector* and *qdc_notification_manager*) that need these associations.
- **Behaviour**: it is the part of the data centre that manages *qassociation*. It requests to it the associations between patients and doctors that are needed by other parts of the data centre (i.e. the part that sends emergency/alert to all patient's doctors).

quser_ data

- **Structure**: it is composed by a data sender and a data retriever.
- **Interaction**: it receives requests from the data centre and answers to it.
- **Behaviour**: it stores all the data related to the patients (parameters, registration, login...). All this data are sent by the data centre. It is the knowledge base of the data centre.

qdc_ register

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it sends messages to the *quser_ data* (data centre's knowledge base) and it receives messages from *qpatient_ login* and *qdoctor_ login*.
- **Behaviour**: it handles users registration and login. It requests some information about registration and login to *quser_ data*, it receives messages from *qpatient_ login* and *qdoctor_ login* to verify if users are already registered and it responds to them.

qdc_ nickname_ id_ handler

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it receives requests from *qpatient_ login*, *qdoctor_ login* and *qdc_ notification_ manager* and sends requests to *quser_ data* and *qdc_ notification_ manager*
- **Behaviour**: it receives requests to make associations between user's nickname and user's actor ID from *qpatient_ login* and *qdoctor_ login* and sends those requests to the *quser_ data* to save the association. It is useful because *qdc_ notification_ manager* can request to this actor the actor ID of a specific user to send to it a notification.

qdc_ data_ collector

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it receives messages from *qpatient_ sensor_ manager*, *qpatient_ data_ retriever*, *qdoctor_ data_ retriever* and sends messages to *quser_ data* and *qassociation_ manager*.
- **Behaviour**: it receives user's parameters from *qpatient_ sensor_ manager* and sends them to *quser_ data* in order to save them. It can receive a request for the clinical history of a specific patient from *qpatient_ data_ retriever* or *qdoctor_ data_ retriever*. Then, it requests to *quser_ data* the clinical history of the specific patient and forwards it to the patient or doctor that wants to know it. If the history was asked from the doctor, it requests to *qassociation_ manager* if the doctor is associated to the patient before sending the message.

qdc_ data_ analyser

- **Structure**: it is composed by a data sender, a data retriever and some rules to analyse data.
- **Interaction**: it receives messages from *qpatient_ sensor_ manager* and sends messages to *qdc_ notification_ manager* and *qdoctor_ emergency_ manager*.
- **Behaviour**: it receives the parameters to analyse from *qpatient_ sensor_ manager*, it forwards the analysis result to *qdc_ notification_ manager* and, only in case of an emergency, also to *qdoctor_ emergency_ manager*.

qdc_ notification_ manager

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it receives messages from *qdc_ data_ analyser* and *qdoctor_ advice_ sender*, and it sends messages to *qassociation_ manager*, *qdc_ nickname_ id_ handler*, patient and doctors.
- **Behaviour**: when it receives a result of an analysis from *qdc_ data_ analyser*, it sends it to the specific patient and, in case of emergency or alert, to *qassociation_ manager* to check which doctors are related to the patient and then it sends the result of the analysis to those doctors. Moreover, when it receives an advice from *qdoctor_ advice_ sender*, it forwards it to the patient, only if the doctor is associated to that patient (it asks to *qassociation_ manager* and to *qdc_ nickname_ id_ handler* to find patient's actor).

qpatient_ login

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it sends messages to *qdc_ register*, *qpatient_ initializing* and *qdc_ nickname_ id_ handler*.
- **Behaviour**: when it receives a request of login, it sends the request to *qdc_ register* to check credentials. If they are right, it creates all patient's actors, it sends to *qdc_ nickname_ id_ handler* patient's nickname and actor's ID to associate them, and it sends them (nickname and ID) to *qpatient_ initializing*.

qpatient_ initializing

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it receives messages by *qpatient_ login* and sends messages to *qpatient_ sensor_ manager*, *qpatient_ data_ retriever*, *qpatient_ notification_ receiver* and *qpatient_ gui_ manager*.
- **Behaviour**: it initialises all the patient's actors created by *qpatient_ login*.

qpatient_ sensor_ manager

- **Structure**: it is composed by a data sender and data retriever.
- **Interaction**: it sends messages to *qdc_ data_ collector* and *qdc_ data_ analyser*, and receives messages from *qsensor_ temperature* and *qsensor_ glycemia*.
- **Behaviour**: it creates and removes sensors related to a specific patient. As now, the only sensors that can be handled are temperature and glycemia. It also receives parameters by those sensors (in particular, from *qsensor_ temperature* and *qsensor_ glycemia*) and sends them to *qdc_ data_ collector* to store them and to *qdc_ data_ analyser* for the analysis.

qsensor_ temperature

- **Structure**: it is composed by a data sender.
- **Interaction**: it sends parameters to *qpatient_ sensor_ manager* via sensor_data('temperature' ,VALUE) message.
- **Behaviour**: it generates normal/alert/emergency parameters and sends them to *qpatient_ sensor_ manager*

qsensor_ glycemia

- **Structure**: it is composed by a data sender.
- **Interaction**: it sends parameters to *qpatient_ sensor_ manager* via sensor_data('glycemia' ,VALUE) message.
- **Behaviour**: it generates normal/alert/emergency parameters and sends them to *qpatient_ sensor_ manager*

qpatient_ data_ retriever

- **Structure**: it is an entity composed by a initial state and another state for setting the nickname. After that it has another state where it receives request from the GUI, a state for requesting his clinical history, a state for waiting the clinical history and another state for handling the clinical history.
- **Interaction**: It must receive at the beginning the id of the patient associated with a my_nickname(NICKNAME) message, after that it waits constantly for a patient_data_request_gui message from the GUI. Subsequently it sends a clinical_history_request(PATIENTNICKNAME,ACTORID) to the qdc_data_collector and waits for a clinical_history_response(ANSWER) message. After that it returns waiting for the patient_data_request_gui message and the following repeatedly.

- **Behaviour**: Initially it retrieves the id of the QActor and it saves it into his own base knowledge, after that it waits for a my_nickname(NICKNAME) message where it retrieves the patient nickname. Once the nickname is received it saves it into his own base knowledge and it goes to another state where it waits for the request from the GUI. After that message is received it retrieves its own nickname and his actor id and forwards a clinical_history_request(PATIENTNICKNAME,ACTORID) to the qdc_data_collector. Once the message is sent it waits for a clinical_history_response(ANSWER) with the response inside and when it is received it is printed on the GUI and it returns waiting for the message from the GUI.

qpatient_notification_receiver

- **Structure**: it is an entity composed by a initial state and another state for setting the nickname. after that it has a state where it waits for the notification and a state for handling the notification.
- **Interaction**: It must receive at the beginning the id of the actor associated with a my_nickname(NICKNAME) message, after that it waits constantly for an analysis_notification(ANALYSIS) message where it receives the notification of its own analysis.
- **Behaviour**: Initially it retrieves the id of the QActor and it saves it into his own base knowledge, after that it waits for a my_nickname(NICKNAME) message where it retrieves the patient nickname. Once the nickname is received it saves it into his own base knowledge and it goes to another state where it waits for an analysis_notification(ANALYSIS) message. Once the message is retrieved it goes into another state where it prints on the GUI that message, then it returns to the state where it waits for another analysis_notification(ANALYSIS) message.

qpatient_gui_manager

- **Structure**: it is an entity composed by a initial state, another state for setting the nickname and the last 2 states where it waits for a message and then it prints it on the GUI
- **Interaction**: it must receive an initial message my_nickname(NICKNAME) where it receives the nickname of the patient and after that it waits for a print_patient_gui(X) message where X is the message to print
- **Behaviour**: Initially it retrieves the id of the QActor, after that it goes into another state where it waits for the nickname of the patient. Once the nickname is received it saves it into his own base knowledge and then it goes into another state where it waits for print_patient_gui(X) message. When this message is received it prints on the GUI that message and then it returns to the state where it waits for print_patient_gui(X) message.

qdoctor_ emergency_ manager

- **Structure**: it is an entity composed by a initial state, another state for waiting the emergency and a state for printing the emergency to the GUI
- **Interaction**: It waits constantly for an emergency_alert(ALERT) message.
- **Behaviour**: When it receives an emergency alert message it prints it out on the GUI and it returns waiting for another message.

qdoctor_ login

- **Structure**: it is an entity composed by a initial state, another state for waiting the nickname from the GUI and a state for checking the login. After that it has a state for waiting the check result, a state for handling that result and another state for creating the actors.
- **Interaction**: It waits initially for a login_doctor_gui(NICKNAME) message, then it sends a check_doctor_login(NICKNAME,"d") to the qdc_register. After that it waits for a login_result(RESULT) message, then it sends a nickname_id_association(NICKNAME,ID) message to the qdc_nickname_id_handler and a nickname_and_id(NICKNAME,ID) message to the qdoctor_initializing. After that it waits again for a login_doctor_gui(NICKNAME) message.
- **Behaviour**: At the beginning it waits for a login_doctor_gui(NICKNAME) message, when that message is received it checks if the nickname is a doctor by saving it and by sending the nickname via check_doctor_login(NICKNAME,"d") message to the qdc_register. After that it waits for a login_result(RESULT) response message and if the response is equal to "Can not login. You are not register" then it returns waiting to another login message. Otherwise if the response equals to "Login ok" then it creates the actors needed for the doctor, it sends the association between nickname and id to the qdc_nickname_id_handler, the nickname and id to the qdoctor_initializing and it returns waiting another login.

qdoctor_ initializing

- **Structure**: it is an entity composed by a initial state and another state for setting the nickname. After that it has a state for creating the QActors needed by the doctor for the normal management
- **Interaction**: it waits constantly for a nickname_and_id(NICKNAME,ID) message where it receives the nickname of the doctor and the id that identifies the session. Then it sends a my_nickname(NICKNAME); message to the created qdoctor_data_retriever, qdoctor_notification_receiver, qdoctor_advice_sender and qdoctor_gui_manager that are related to his id and nickname.

- **Behaviour**: At the beginning it waits for a nickname_and_id(NICKNAME,ID) message where it retrieves the id and the nickname of the doctor. Once the message is retrieved it saves it into his own base knowledge and then it forwards the nickname to qdoctor_data_retriever, qdoctor_notification_receiver, qdoctor_advice_sender and qdoctor_gui_manager that are related to his id and nickname. After that it deletes the nickname_and_id(NICKNAME,ID) message and waits again for another message.

qdoctor_data_retriever

- **Structure**: it is an entity composed by a initial state, another state for setting the nickname. After that it has another state where it receives the nickname of the patient, then a state for waiting the clinical history and another state for handling the clinical history.
- **Interaction**: It must receive at the beginning the id of the actor associated with a my_nickname(NICKNAME) message, after that it waits constantly for a doctor_data_request_gui(PATIENTNICKNAME) from the GUI. Then it sends a clinical_history_request_from_doctor(DOCTORNICKNAME,PATIENTNICKNAME,ACTORID) to the qdc_data_collector and waits for a clinical_history_response(ANSWER) message. Subsequently it returns waiting for the doctor_data_request_gui(PATIENTNICKNAME) message and the following repeatedly.
- **Behaviour**: Initially it retrieves the id of the QActor and it saves it into his own base knowledge, after that it waits for a my_nickname(NICKNAME) message where it retrieves the doctor nickname. Once the nickname is received it saves it into his own base knowledge and it goes to another state where it waits for the patient nickname whose history must be retrieved. Once that message is received it saves it into his own base knowledge, then it retrieves his own doctor nickname, his actor id and the patient nickname and it forwards a clinical_history_request_from_doctor(DOCTORNICKNAME,PATIENTNICKNAME,ACTORID) to the qdc_data_collector and it deletes the patient id previously saved. Once the message is sent it waits for a clinical_history_response(ANSWER) with the response inside. When the response is received it prints it on the GUI and it returns waiting for the patient id.

qdoctor_notification_receiver

- **Structure**: it is an entity composed by a initial state and another state for setting the nickname. after that it has a state where it waits for the notification and a state for handling the notification.
- **Interaction**: It must receive at the beginning the id of the actor associated with a my_nickname(NICKNAME) message, after that it waits constantly for an analysis_notification_of_patient(RESULT,PATIENTNICKNAME) message where it receives the notification of a patient.

- **Behaviour**: Initially it retrieves the id of the QActor and it saves it into his own base knowledge, after that it waits for a my_nickname(NICKNAME) message where it retrieves the doctor nickname. Once the nickname is received it saves it into his own base knowledge and it goes to another state where it waits for an analysis_notification_of_patient(RESULT,PATIENTNICKNAME) message. Once the message is retrieved it goes into another state where it prints on the GUI that message, then it returns to the state where it waits for another analysis_notification_of_patient(RESULT,PATIENTNICKNAME) message.

qdoctor_advice_sender

- **Structure**: it is an entity composed by a initial state, another state for setting the nickname. then it has a state for setting the patient nickname, a state for setting the advice, another state where it sends the advice and a state where it receives the result of the advice.

- **Interaction**: It must receive at the beginning the id of the actor associated with a my_nickname(NICKNAME) message, after that it waits for an advice_to_send_gui(PATIENTNICKNAME) message where it receives the patient nickname and another advice_to_send_gui(ADVICE) message where this time it receives the advice to send. After that it sends an advice_to_send(PATIENTNICKNAME,ADVICE,DOCTORNICKNAME,DOCTORACTORID) to the qdc_notification_manager and waits for a advice_to_send_result(RESULT) message where it receives if the advice is sent correctly.

- **Behaviour**: Initially it retrieves the id of the QActor and it saves it into his own base knowledge, after that it waits for a my_nickname(NICKNAME) message where it retrieves the doctor nickname. Once the nickname is received it saves it into his own base knowledge and it goes to another state where it waits for the patient nickname. After the nickname is received it saves it into his own base knowledge and it goes to the state where it waits for the advice to send. At the reception of the advice it saves it and then it goes to the state for sending the advice. Once in this state it retrieves the patient nickname, the doctor nickname, the advice to send and his own actor id and it sends the message to the qdc_notification_manager and deletes the patient nickname and the advice previously saved. Then it goes to another state where it waits the result of the dispatch of the message. Once the result is retrieved it returns to the state where it waits for the patient nickname.

qdoctor_gui_manager

- **Structure**: it is an entity composed by a initial state, another state for setting the nickname and the last 2 states where it waits for a message and then it prints it on the GUI

- **Interaction**: it must receive an initial message my_nickname(NICKNAME) where it receives the nickname of the doctor and after that it waits for a print_doctor_gui(X) message where X is the message to print
- **Behaviour**: Initially it retrieves the id of the QActor, after that it goes into another state where it waits for the nickname of the doctor. Once the nickname is received it saves it into his own base knowledge and then it goes into another state where it waits for print_doctor_gui(X) message. When this message is received it prints on the GUI that message and then it returns to the state where it waits for print_doctor_gui(X) message.

6.3 abstraction gap

The abstraction gaps that we have between the QActor implementation and the real implementation are:

- The sensors can send data not clean, we have to stabilise them.
- The system must scale with many devices, more than 1000.
- The system must have more than one web interface, that can't be done with QActors.
- The history retrieve should be parametrised.
- The analyser has to send notifications also to the patient.

6.4 risk analysis

The main risks emerging from the analysis are :

- it is possible to use more time than expected due to our inexperience in technologies used and the overlap between lectures and the project,
- It is possible that the implementation of a machine learning analyser could take more time than expected,
- It is possible that the infrastructure and the protocols used for the sensor is not the best and is not supported by the sensors.
- it is possible to use more time than expected because we are not in the same city and for the meeting we use many time for transports.

Evaluating the risks that can arise in this project we have chosen to not develop the Machine Learning analyser due to its high complexity and to manage only the types of sensor where there is only a single event and with a stream, excluding the ones where you have to make a request for retrieving the data, due to the risk of high time usage.

7 Work Plan

The development process chosen for our application is an hybrid approach principally based on Agile process because we decide to do sprints.

Initially, we drafted together the system's requirements described above and the architectural design of our application. Hereafter, we set up the base structure of control unit and application's business logic with web interface mock-up. The workload has been divided in two parts: first one is related to sensors and to the analyser with connected problems; the second one is related to patient and doctor application and to the H2 application itself, concerning all the functionalities and connections with the application's database. The first section has been assigned to Stefano Bernagozzi and Manuel Bottazzi and the other one has been assigned to Giulia Lucchi and Margherita Pecorelli.

We decided to use "Trello", a on-line, complete and intuitive free software of project management. It has made it possible for us to manage the sprints and the task organization. The sprints and the respective tasks are represent in following table.

	H2 application - Sprint	
	Task ID	Story / Description
Sprint 1	1	sending data to data centre from control unit
	2	receiving data from control unit
	3	configuration of Gradle and Travis
	4	Structure of data in server and message
Sprint 2	5	connection sensors to control unit
	6	retreive data from aduino sensors
	7	Swagger interface
	8	Association database RESTful API
Sprint 3	9	H2 application RESTful API
	10	sending data to control unit from sensors
	11	refactoring code and error management
	12	access point raspberry PI
	13	Node-documentation and refacoring RESTful API
Sprint 4	15	Data communication blu arduino and wi-fi android
	16	Creating mapping interface RESTful API in data centre (java)
	17	Creating RabbitMq server
	18	Creation a basic stucture RabbitMq in the data centre
Sprint 5	19	file management instead of mongoDB in control unit
	20	fullfillment sub/pub pattern in data centre
	21	integration sensors' code
	22	developement web server raspberry
	23	fullfillment sub/pub pattern in node (web servers)
	24	modify android app
Sprint 6	25	management alert and emenrgency for wrong vital prameters
	26	connection and disconnection web client
	27	web socket to do pupop (sending to multiple web socket)
	28	simulator
	29	check documentation and refactoring
	30	testing of all application's functionalities

Figure 3: Sprint planner

8 Design Process

8.1 Architectural Design

Concerning the logical architecture obtained in the problem analysis, we are going to expand it in order to create the final architecture.

8.1.1 Basic Structure

The basic structural entities are:

Patient: a distributed part of the entire application used by the patient.

Doctor: a distributed part of the entire application used by the doctor.

Analyser: a part of the system that we decide to locally place with the patient as explained in the problem analysis.

Data Centre: a distributed part of entire application representing the system's core handler.

Sensors: a part of the system that we decided to locally place with the patient because they belong to patient himself.

Association knowledge base: the knowledge base of the sanitary service where the associations between patients and doctors are saved. We decided to keep it external from the other entities because we assume that it already exists.

Application knowledge base: the knowledge base of the data centre that we decided to keep external from the other entities in order to avoid useless dependencies.

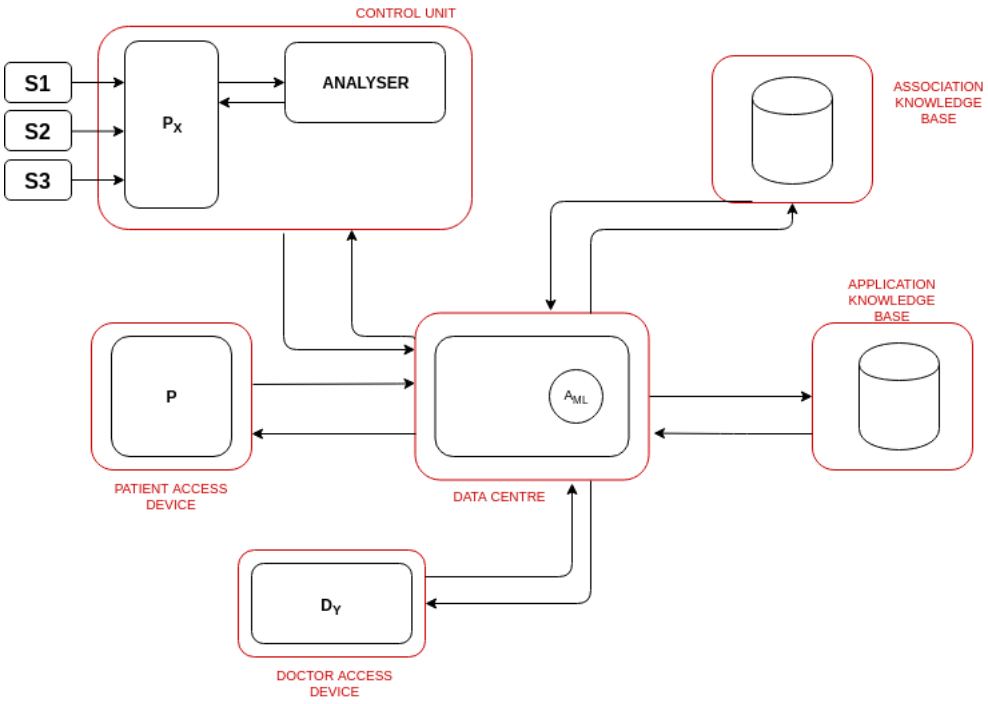

Figure 4: Structure of general architecture

8.1.2 Architectural Pattern

In this section we are going to exhibit all the architectural pattern that we use in this project, those choices arises from requirement analysis and problem analysis.

- **RESTful API** We have chosen the RESTful API pattern because it allows to uncouple the 2 end points of the communication (listed below), it is easy to use, it allows communication through the web and it is consolidated.

  The RESTful APIs are used for:

  - **data centre - association knowledge base** asynchronous communication;
  - **data centre - application knowledge base** asynchronous communication;
  - **sensor - control unit** (web-based sensor).

- **Publish-subscribe** We have chosen the publish subscribe pattern because it is well known in our team, it allows to receive real-time notifications without asking for them and it maintains messages in the server

when the user is not connected. The publish-subscribe pattern is used for handling all communications between the distributed part of the application, that is:

  - **patient - data centre**
  - **doctor - data centre**
  - **control unit - data centre**

- **Event communication** We have chosen event communication because it allows to emit events without knowing who is receiving them and it allows to uncouple the sensors from the control unit. The event communication is used for:

  - **sensor - control unit** (direct connection sensors)

8.1.3 Sub-structure

Sensor
We have two categories of sensors:

- Sensors capable to connect through the internet. They will communicate with the system using the RestAPI offered by the control unit.
- Sensors that require a specific protocol to create connections.

Control Unit

- **Sensor manager** that handles data measured by sensors;
- **User interaction manager** that handles the communication with the physical patient;
- **Communication manager** that handles the communication with analyser and data centre and if needed locally stores data;
- **Analyser**

  - **Analyser** that does the first local analysis.
  - **Communication manager** that handles the communication of the analyser

Patient
It's the web server that interact with the patient and it handles all the application functionalities for patient. The cited functionalities are: request of personal data, advices, drugs stored and delivery of notifications from doctors.

Doctor
It's the web server that interact with the doctor and it handles all the application functionalities for doctor. The cited functionalities are: request patients'

data, send prescribed drugs and advices to a specific patient and delivery of alert/emergency notifications.

Data centre
It is the core of communications within the application that links all the others distributed application's parts and manages data.

Application knowledge base
It stores all the data needed by the application: sensor values, patient's prescribed drugs, doctor's advices to patients and patient and doctor informations.

Association knowledge base
It stores the associations between doctors and patients. We wanted to separate this DB from the other one because, in our thinking, it's handle by national sanitary service.

8.2 Technologies and Models

The technologies use in our application are:

- **stack MEAN**: this stack is used in most of the application. This choice is derived from same feature: this stack is supported by all the operation system, use a unique programming language as Javascript and use JSON that is a standard for the transfer of information on the network between servers and the web application. Furthermore, to store data, it uses MongoDB, that approach a OO language and queries take place through API request. Finally this stack allows to keep separated the view and the model, like in MVC pattern.
- **Jersey**: is a RESTful Web Services framework, used client side by us. In particular, we decide to use this framework, because it supports better than other propose more testing tools, including JUnit. Moreover it is very easy to use.
- **RabbitMQ**: is a message-oriented middleware, that implements the Advanced Message Queuing Protocol (AMQP). It implements publish-subscribe (pattern that we use in this project), it is easy to configure and it is free.

The models used are:

- **Actor Model**: is used to manage in concurrent manner all the interaction between the entity within data centre. It has been used as support for the publish-subscribe pattern to manage the request and create a response. Moreover it's very suitable to be distributed, if, in the future, we will decide to distribute the data centre.
- **Event model**: is used to manage the control unit, the databases and the web server used by both client and doctor. It is asynchronous and it is suitable for handling a large number of connections.

The programming languages are:

- **Arduino Language**: for programming the Wifi Kit 8
- **Javascript**: is the language that totally support the stack MEAN.
- **Java**: is the language used for all the data centre. This choice was made because it's a native language of Jersey framework and it's the best known language by developers.

8.3 Structure Models

8.3.1 Control Unit

The Control Unit represent the distributed part of the system that is used by the patient. Every patient / family would have a Control Unit in their home and use this to automatically communicate to the system. In particular the Control Unit has multiple roles:

- it works as a gateway for the sensors to communicate with the system, allowing different type of connection and different kind of sensors
- it performs a simple analysis on the measured parameters to decide if there is an emergency or an alert situation
- it stores data until they are sent to the remote system (aka Data Centre) avoiding data loss
- it offers a web service that allows the user to easily connect to the Control Unit itself in order to configure it.
- it offers a REST API that allows the Control Unit to be used also through the web (for example by a web-capable sensor).

The logical structure of the Control Unit are described in the following picture:

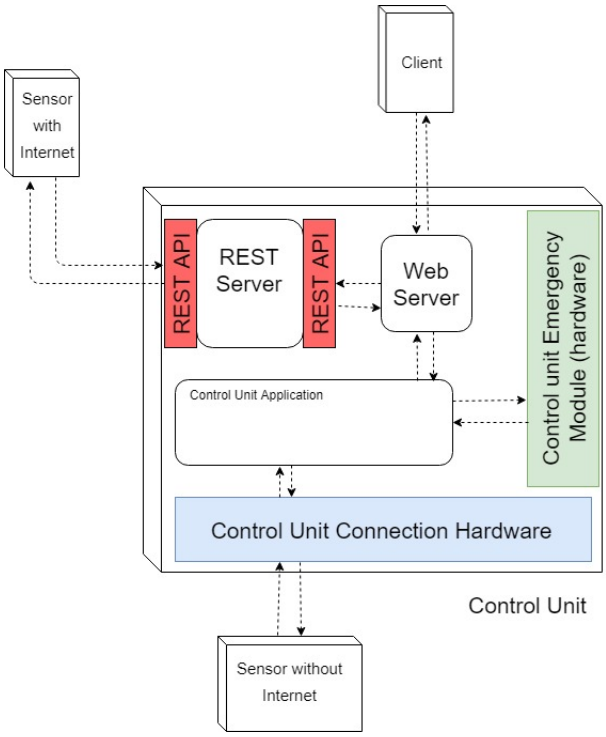

Figure 5: Control Unit Software Modules

8.3.2 Data Centre

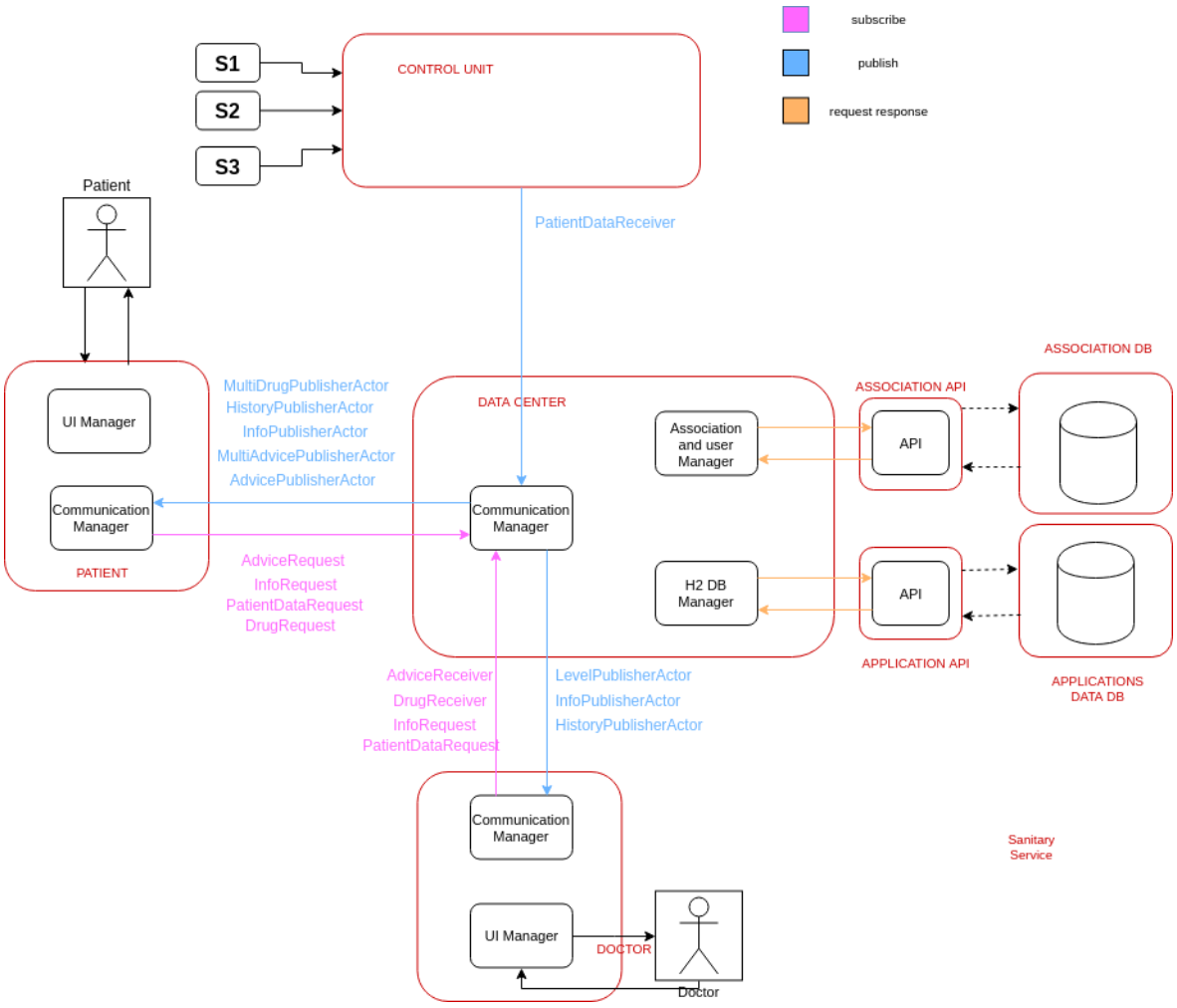

Figure 6: Structure of data centre's general architecture

Publish-Subscribe

The subscribers are:

- **Advice Receiver**: receives advices written by doctors for patients, store it in application Knowledge base and sends them to *advice publisher*.
- **Advice Requester**: receive request to visualized personal advices by patient and sends them to *multi advice publisher*

- **Drug Receiver**: receives prescribed drugs written by doctors for patients and store in application Knowledge base.
- **Drug Requester**: receive request to visualized personal drugs by patient and sends them to *multi drug publisher*
- **Info Requester** : receive request to visualized personal information by patient and sends them to *info publisher*
- **Patient Data Receiver**: receives the sensor values from specific patient and store this values. If a sensor values has level 2 or 3 as attribute, the sensor value is send to *level publisher* that manages the notification to the doctor.
- **Patient Data Requester**: receive request to visualized personal sensor values by patient and sends them to *history publisher*

The publisher are:

- **Advice Publisher**: send the advice notification written by doctor to the patient
- **Level Publisher**: send the emergency (level 3) or alert (level 2) to the patient's doctors
- **Info Publisher**: send the personal information to the requester
- **History Publisher**: send the patient's clinical history to the requester
- **Multi Advices Publisher**: send to the requester all advices relative to a specific patient on a time range
- **Multi Drugs Publisher**: send to the requester all drugs relative to a specific patient on a time range

Knowledge base interfaces

For a better structure of our code, we decided to use Java interfaces to map the RESTful API requests and responses to Application and Association database. In this way, we manages the request and response in higher level.

8.4 Doctor and Patient Web Server

Publish-Subscribe

With respect to what we described in the data centre e what is represent in the previous figure, the patient or doctor web server has complementary publish-subscribe: each publisher in the data centre has a subscriber in the patient/doctor web server and vice versa. Some of the user web server pub-sub

components are shared by both doctor and patient, so we are going to describe them all specifying the user that use them.

The subscribers are:

- **Multi Advices Receiver**: is used by the patient to get all the advices related to him and already received in the range of time specified
- **Multi Drugs Receiver**: is used by the patient to get all the drugs prescribed to him in the range of time specified
- **History Receiver**: is used by both patient and doctor to receive the clinical history of a specific patient in the range of time specified
- **Info Receiver**: is used by both patient and doctor to receive their personal informations
- **Advice Notification Receiver**: Is used by the patient, it subscribes to its own advice queue and its work is to retrieve real time notification and send them via web socket to all the web clients connected with the specific user name.
- **Level Notification Receiver**: Is used by the doctor, it subscribes to its own notification queue and its work is to retrieve real time notification and send them via web socket to all the web clients connected with the specific user name.

The publisher are:

- **Multi Advices Requester**: is used by the patient (and in future it will be use also by the doctor) to request all the advices related to him and already received in the range of time specified
- **New Advice Publisher**: is used by the doctor to send a new advice to a specific patient
- **Multi Drugs Requester**: is used by the patient (and in future it will be use also by the doctor) to request all the drugs prescribed to him in the range of time specified
- **New Drug Publisher**: is used by the doctor to send a new prescription of a drug to a specific patient
- **History Requester**: is used by both patient and doctor to request the clinical history of a specific patient in the range of time specified
- **Info Requester**: is used by both patient and doctor to request their personal informations

Server Side Web Socket This module has the task to manage all the web socket connected and when there is a notification it takes all the clients that must receive it and sends it to them.

Client Side Web Socket The client side part of the web socket has the task to retrieve the messages from them and display a modal window with the new notification and, if present, old notifications.

8.5 Knowledge base

We decided to use a document-base Data Base, driven by the stack MEAN's choice. The knowledge base considered are the following.

Association Knowledge base

The association's database is independent from the application's database because we thought that the national health system already have some type of association knowledge base and related API. Nevertheless, we needed a replacement database of associations to use in our application. So, the related schema of this database is very simple, essential and linear, as represented in the following image.

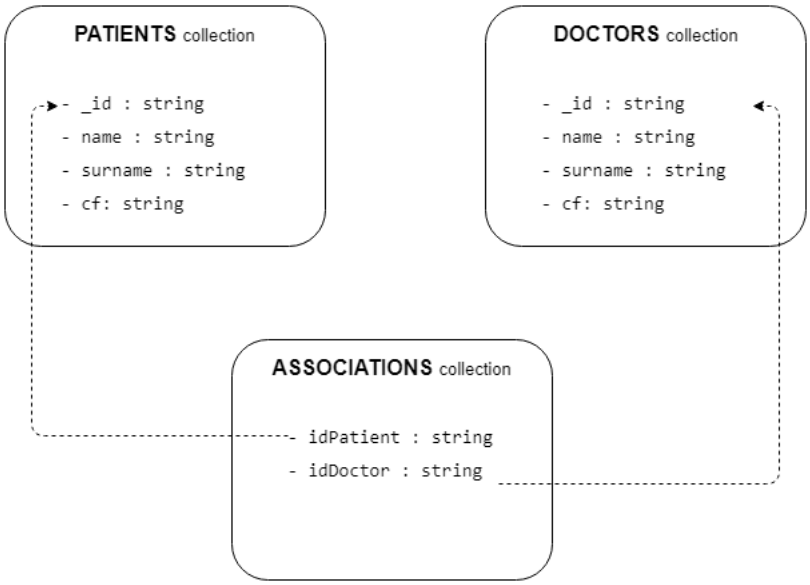

Figure 7: Schema of association database

Application Knowledge base

This is the main application database. It stores:

- all values acquired by all patient's sensors

- all drugs prescribed to the patient by any doctors
- all advices send to the patient by any doctors
- all patients' and doctors' informations (like name, surname, CF, etc...)

Since all this data could be a lot, we decided to scale horizontally instead of vertically. In order to do that, the application creates a new MongoDB collection for each sensor related to a specific patient (i.e. mario.rossi.temperature) and another one for patient's drugs (i.e. mario.rossi.drugs). Instead, for patients' advices there is just one collection for all patients because advices are much less. There are another two collections (patients and doctors) to storage all patients' an doctors' informations (like name, surname, CF, etc...)

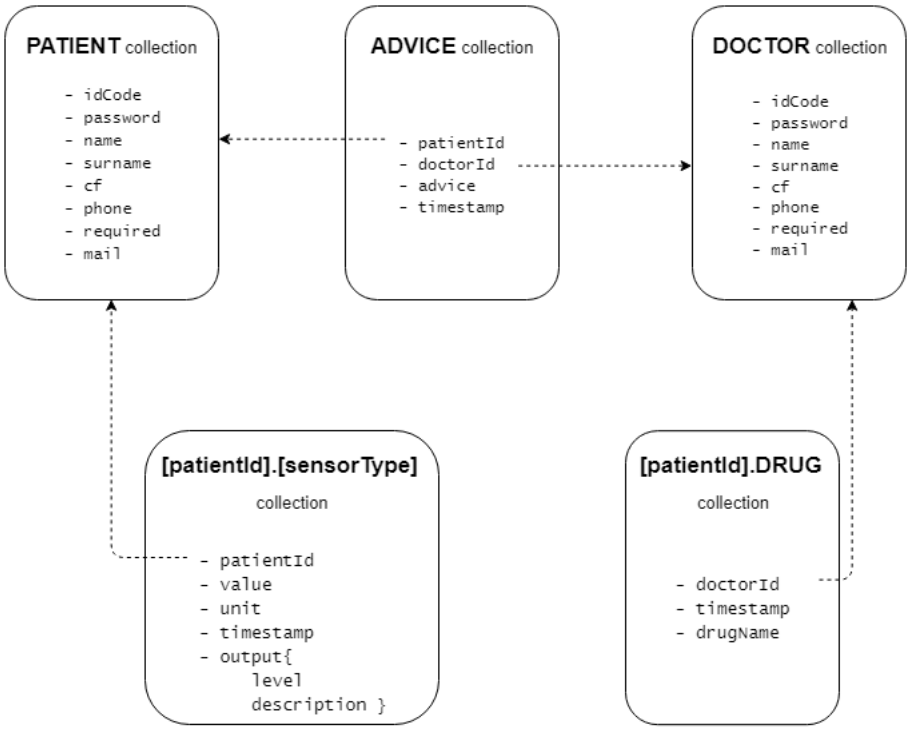

Figure 8: Schema of application database

8.6 RESTful API databases

There are two RESTful API, one for each database, to communicate with them across the web. We decides to use RESTful API because it is a standard, is a scalable web service and it has uniform interfaces to access his resource. The interfaces are defined below:

Association RESTful API

It allows to:

- get patients' and doctors' informations
- insert a new patient or doctor
- delete a patient or doctor
- get a specific relationship between a patient and a doctor
- get all relationships between a specific patient and all his doctors
- get all relationships between a specific doctor and all his patients
- insert a new relationship between a patient and a doctor
- delete a specific relationship

The API's are described via swagger, their definitions can be found at [9]

Application RESTful API

It allows to:

- register a new patient or doctor inserting him to the application's database
- log in a patient or doctor
- get all sensor types related to a specific patient
- add a new sensor type to a specific patient
- add a new sensor's value to a specific patient collection (i.e. a new temperature's value of patient Mario Rossi: added in the collection mario.rossi.temperature)
- delete all values of a specific patient's sensor
- get all values of a specific patient's sensor
- add a new advice related to a specific patient
- return all advices related to a specific patient
- add a new prescribed drug to a specific patient (i.e. in the collection mario.rossi.drugs)
- return all prescribed drugs related to a specific patient
- get patients' and doctors' informations
- delete a patient or doctor

The API's are described via swagger, their definitions can be found at [10]

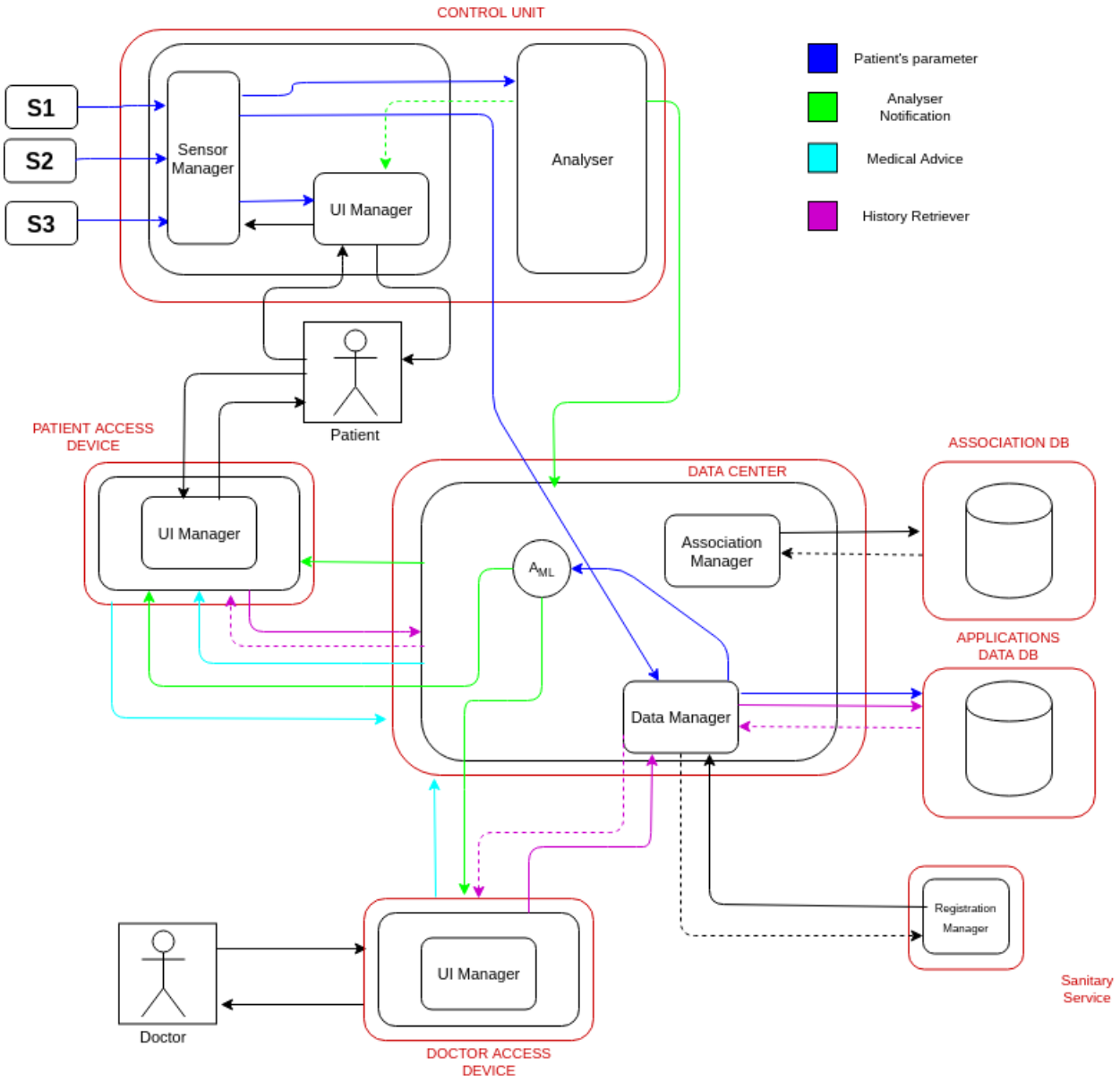

Figure 9: Structure of general architecture

9 Coding

9.1 Control Unit

9.1.1 Hardware configuration

The control unit prototype is realized with a Raspberry Pi model 3 B [7] with a Raspbian OS. The board is provided with some basic hardware to handle interaction with the human users of the system. In particular the control unit has:

- A 16x2 LCD screen in order to visualize some simple message,
- A Buzzer that emit sound when an emergency is detected,
- A Red Led that turns on when an emergency is detected,
- A Blue Led that simulate the phone call to emergency service when required,
- A Button used to stop emergency protocol.

The hardware architectural scheme and the connections are shown in the image below:

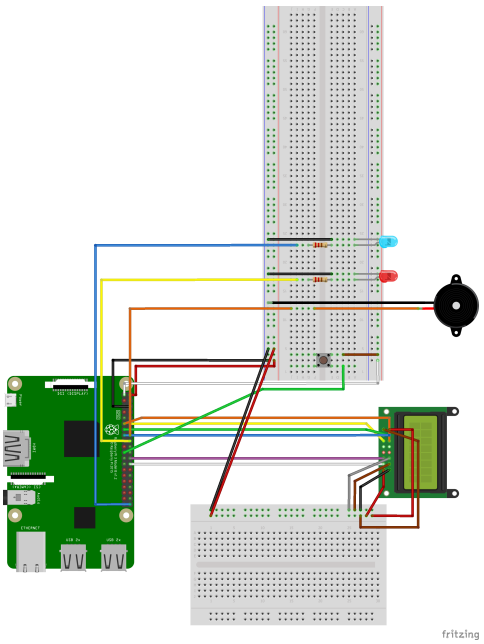

Figure 1: Hardware Connections of control Unit

the following pin are used :

- pin 1 -> 3.3V to breadboard
- pin 2 -> 5V to breadboard
- pin 6 -> GND to breadboard
- pin 11, 12, 13, 15 -> LCD data
- pin 16 -> out to Buzzer
- pin 18 -> out to Emergency Led
- pin 22 -> in from button
- pin 24 -> LCD E
- pin 26 -> LCD RS
- pin 40 -> out to Phone Led

9.1.2 Software configuration

The control unit application is composed by different modules:

- A RESTful Service that has to manage request from the web to the control unit data structure. The Sensor capable of HTTP connections send their data directly to this module.
- A web server that give the user a Web-based GUI to interact easily with the control unit itself. This module use heavily the REST service to read and update data.
- A core application (control unit application in the picture) that realize the business logic of the unit itself, handling the analysis of data and the publish-subscribe logic. This also care about the persistence of the data on the system and the emergency-related protocol logic. Also the connection hardware, used to handle sensor that cannot communicate through the web and need a different type of connection, are managed by this module.

The whole code of control unit application is in *controlUnit* folder.

The control unit control application is totally realized using javascript language, with different technology in every module.

9.2 Implementation

9.2.1 REST Service

The REST service is realized as a part of a web service using parts of the MEAN Stack. In particular for this module we have used NodeJs and Express for the request handling. For managing the data storage the optimal choice should have been MongoDB, but on our version of raspberry pi this wasn't available, so we

have decided to use the MEAN Stack anyway and implement from scratch our data storage function working directly with the file system. In this part also the Angular part of the stack was not necessary.
This module can be found in package *h2api*. Here we have a file (*jsonUtilities.js*) with utilities for handling JSON data and two folders:

- the *controllers/* folder that contains all the function that handle the data storage and retrieval of both sensors and patients' data. In particular there are a specific function for every functionality offered by the REST API Interface shown above in paragraph 8.2.1.

  In order to work properly need to read and write from the file system, using the function defined in package *controlUnitApp/dataManager*

- *routes/index.js* is the router file that specify how to route every request received from the client to the specific controllers function to be executed.

```
1  // Control Unit REST API
2  //// Sensors-related Request
3  // Request for the whole sensors list.
4  router.get('/sensors', sensorsController.getSensorsList);
5
6  // Request for adding a sensor to the list of connected ones.
7  router.post('/sensors', sensorsController.createSensors);
8
9  // Request for details for a specific sensor.
10 router.get('/sensors/:sensorID', sensorsController.
       getSensorDetails);
11
12 // Request to delete a specific sensor
13 router.delete('/sensors/:sensorID', sensorsController.
       deleteSensor);
14
15 // Request to add measured values from a specific sensor.
16 router.post('/sensors/:sensorID/data', sensorsController.
       addData);
17
18 //// Patients-related Request
19 // Request for the whole patients list.
20 router.get('/patients', patientsController.getPatientsList);
21
22 // Request for adding a patient to the list of users.
23 router.post('/patients', patientsController.createPatient);
24
25 // Request for details for a specific patient.
26 router.get('/patients/:patientID/', patientsController.
       getPatientDetails);
27
28 // Request to delete a specific patient.
29 router.delete('/patients/:patientID/', patientsController.
       deletePatient);
```

9.2.2 Web Server

The Web Server is realized using parts of the MEAN Stack. In particular for this module we have used NodeJS and Express for the request handling. No AngularJS is used at the time of writing due to a lack of time, but the dynamic pages are a future todo for the project. We have achieved a certain grade of adaptability of screen type and dimension using *Bootstrap* framework for graphic rendering.

This also offer a REST API with the main feature, that would be described below :

```
// Control Unit REST API
//// Sensors-related Request
// Request for the whole sensors list.
router.get('/sensors', sensorsController.getSensorsList);

// Request for adding a sensor to the list of connected ones.
router.post('/sensors', sensorsController.createSensors);

// Request for details for a specific sensor.
router.get('/sensors/:sensorID', sensorsController.getSensorDetails
    );

// Request to delete a specific sensor
router.delete('/sensors/:sensorID', sensorsController.deleteSensor)
    ;

// Request to add measured values from a specific sensor.
router.post('/sensors/:sensorID/data', sensorsController.addData);

//// Patients-related Request
// Request for the whole patients list.
router.get('/patients', patientsController.getPatientsList);

// Request for adding a patient to the list of users.
router.post('/patients', patientsController.createPatient);

// Request for details for a specific patient.
router.get('/patients/:patientID/', patientsController.
    getPatientDetails);

// Request to delete a specific patient.
router.delete('/patients/:patientID/', patientsController.
    deletePatient);
```

This module can be found in package *h2*. Here we have three folders :

- *controllers*, that contains the controller functions. Every function in this package is called (by the router) when a specific request arrive to the server and and works to create an appropriate response. Some simply render an HTML page, some others interact with the API to modify the state of the control Unit itself, for example adding / removing sensors or user. An example of the controller function is the one that retrieve the sensors main page :

```
/**
* GET main sensors management page
* @param req Is the HTTP Request. Not used by this function.
* @param res Is the HTTP Response, returned to the requesting
    host with the rendered page.
*/

module.exports.sensorsHome = function(req, res){
  // request to the REST Service the list of the previously
    connected sensors.
  request.get('http://localhost:3000/api/sensors', function (
    error, response, body) {
    if (error)
      console.log('error:', error);
    console.log('statusCode:', response && response.statusCode
    );
    if(body){
      var sensorsList = JSON.parse(body);
      renderSensorHomepage(req,res, sensorsList);
    }
  });
};
```

In this function we have a request to the API to have the list of the sensors connected to the control unit and then the rendering of the result page with the provided information.

- *routes*, that contains the router file (*index.js)* that specify how to route every request received from the client to the specific controllers function to be executed.
- *views*, that contains the jade file that generate the HTML file returned to the client after a request is completed. Jade is a template engine used to pre-process CSS and generate HTML pages with some optimized feature. Now we see as example the sensors manager home page code:

```
extends layout

block content

  #banner.page-header
    .row
      .col-lg-6
        h1= title

  .row
    .col-xs-8.col-sm-8
      .panel.panel-primary
        .panel-heading
          h2.panel-title Near Sensors
        .panel-body
          .row.list-group
            each ns in nearSensorsList
              .col-xs-6.list-group-item
                h4
                  span #{ns.id}
```

```
21                      a.btn.btn-default.pull-right(href='/sensors
    /#{ns.id}/connectNew') Connect
22      .col-xs-12.col-sm-4
23        p.lead
24          | Sensor manager help you to connect and manage your
    sensors through
25          | the H2 framework to monitor your vital parameter with
    no effort.
26    .row
27      .col-xs-8.col-sm-8
28        .panel.panel-primary
29          .panel-heading
30            h2.panel-title My Sensors
31          .panel-body
32            .row.list-group
33              each sensor in sensorsList
34                .col-xs-6.list-group-item
35                  h4
36                    a(href="/sensors/#{sensor.sensorId}/details
    ") #{sensor.value.name}
37                  a.btn.btn-default.pull-right(href='/sensors
    /#{sensor.sensorId}/delete') Delete
38                  p.list-item
39                    | Type: #{sensor.value.dataType} <br/>
```

Resulting in the following page :

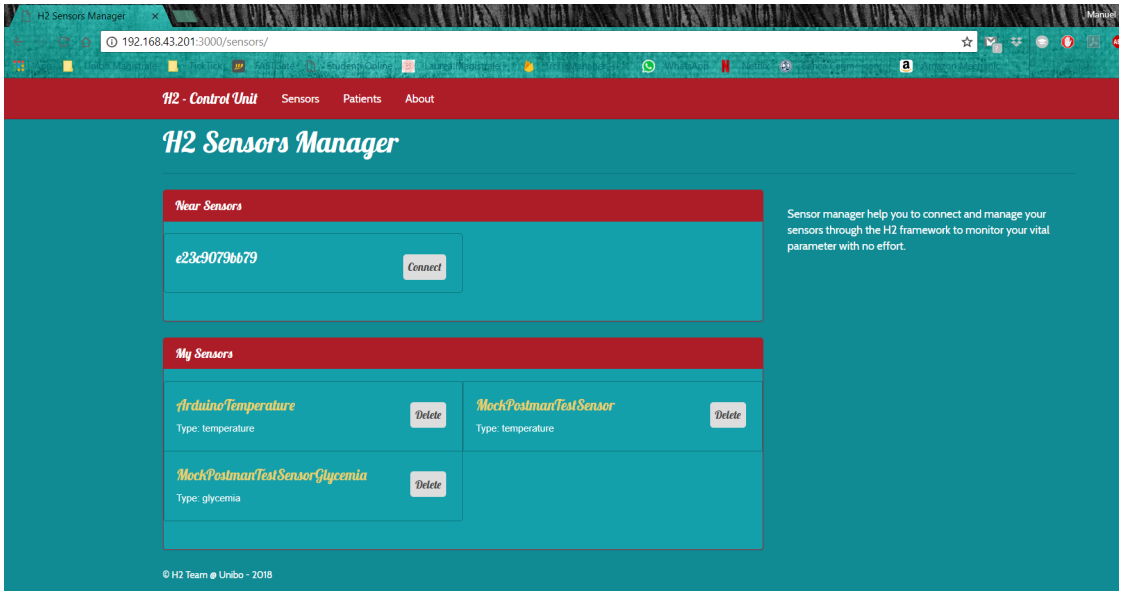

Figure 2: Control Unit Sensors Page

9.2.3 Control Unit Core Application

The core application is composed by different modules realized using Javascript language.

- *BLE Controller :* Module to handle nearby BLE sensors and retrieve parameters from it. This is connected to the GUI so when the user click the connect button for the sensor it automatically starts retrieve data. Also when there is a device already paired and it is nearby it retrieves data without any user interaction.

- *Data Manager:* Module to handle the interaction with the control unit file system to read and write data from/to file. This functionality is used by API server to store information about patient and sensor registered to this control unit and by the analyser when Internet is not available to store unsent message to warrant persistence of the measured vital parameter. Everything is handled in JSON format.

- *Hardware Manager :* Module that contain code for the basic handling of the hardware component and the implementation of the logic of the emergency protocol of the control unit. Node module *'onoff'* is used to work with GPIO pins of the Raspberry Pi.

  Emergency Protocol
  When an emergency is raised both the emergency led and the buzzer turn on for 60 second. If the user in this time did not press the button to stop the emergency an emergency call is performed (we have simulate the call with a led due to the lack of time, but is possible to add to the control unit itself a GSM module that can perform phone call even without Internet connection). We have insert the button to stop the emergency call because as written upward we want to avoid false emergency call due to the lack of technological skill of the end user of the system. e.g. a patient that is monitoring his heart rate can take off the sensor without stop the monitoring, and the sensor could send to control unit a 0 beat rate, that is a very critical emergency situation. The protocol also handle only one emergency at a time, so while the rescuers are arriving no new emergency call are performed even if new emergency-causing value could be received from sensors.

```
1  var Gpio = require('onoff').Gpio;
2
3  var buzzer = require('./buzzer');
4  var emergencyLed = require('./emergencyLed');
5  var phoneLed = require('./phoneLed');
6  // use a button to interact with users.
7  var pushButton = new Gpio(25, 'in', 'rising', {debounceTimeout
       : 10});
8  var lcd = require('./controlUnitLCD');
9  lcd.initializeLcd();
10
```

```
11 var emergency = false;
12 var initialized = false;
13 var lastEmergencyTerminated = true;
14
15 /**
16 * Function that initialize the whole hardware.
17 */
18 function initialize () {
19
20   if (!initialized){
21     console.log("initializing");
22     emergencyLed.initializeLed(24);
23     phoneLed.initializeLed(21);
24     buzzer.initializeBuzzer(23);
25     lcd.initializeLcd();
26
27     pushButton.watch(function (err, state) { //Watch for
       hardware interrupts on pushButton GPIO, specify callback
       function
28       if (err) {
29         console.error('There was an error : ', err);
30         return;
31       }
32       if (emergency) {
33           emergency = false;
34           reset();
35           lcd.write("NO EMERGENCY");
36       }
37     });
38     initialized = true;
39   }
40   console.log("Emergency Manager Modules initialized.");
41 }
42
43 /**
44 * Start protocol and wait for 10 second to check if is a real
     emergency or a false alarm ( the user  * could stop it).
45 */
46 module.exports.startEmergency = function () {
47   if (lastEmergencyTerminated){
48     console.log(" -- Start Emergency ");
49     if (initialized) {
50     emergencyProtocol();
51     setTimeout(checkEmergency, 10000);
52     } else {
53     initialize();
54     emergencyProtocol();
55     setTimeout(checkEmergency, 10000);
56     }
57   } else {
58     console.log(" -- Already handling this emergency ! ");
59   }
60 }
61
62 /**
63 * Function that handle hardware output when an emergency is
     detected.
```

```
*/
function emergencyProtocol() {
  emergency = true;
  lastEmergencyTerminated = false;
  lcd.write("EMERGENCY");
  emergencyLed.blink(250, 10000);
  buzzer.beep();
}

/**
* When Timeout occur check if the emergency was real or not.
*/
function checkEmergency() {
  console.log(" -- End of control time. What to do ? ");
  buzzer.stop();
  if(emergency){
      callRescuers();
  }
  else {
      console.log("NO MORE EMERGENCY");
      emergencyLed.turnOff();
      phoneLed.turnOff();
      lcd.reset();
  }
}

/**
* Function that SIMULATE emergency call to rescuers.
* FUTURE TODO : implement real calling using GSM module
*/
function callRescuers() {
    console.log("CALLING EMERGENCY RESCUERS");
    lcd.write("CALLING RESCUERS");
    emergencyLed.turnOff();
    phoneLed.turnOn();
    emergency = false;
    setTimeout(reset, 10000);
}
```

- *Publish Subscribe Manager :* Module that contain a Javascript version of a RabbitMQ publisher. No Subscriber is required because the control unit did not receive message from the outside different of the sensor data, handled differently. *rabbitmqLibrary* handle the connection to the remote RabbitMQ server and utilities to publish message on a specific channel. As example here the function to send message :

```
/**
* Function to publish a message to a remote RabbitMQ server.
* @param exchangeName is the name of the RabbitMQ exchange in
    which the message has to be published.
* @param exchangeType is the type of the exchange( e.g. topic)
    .
* @param routingKey is the key used to route and identify the
    message inside the exchange.
* @param isDurable set the durability of the messages on the
    queue.
* @param message is the message to be sent.
```

```
*/
function publish (exchangeName, exchangeType, routingKey, isDurable, message) {
  if(amqpConnection) {
    amqpConnection.createChannel(function(error, channel) {
      if(error) {
        console.log('[AMQP] Error creating channel');
        //return setTimeout(start, 1000, serverURL);
        console.log('[AMQP] Storing data locally');
        localDataFileManager.addData(JSON.parse(message));
      } else {
        // console.log(" [AMQP] Channel created")
        channel.assertExchange(exchangeName, exchangeType, { durable: isDurable});
        console.log(" [AMQP] Exchange created : ", exchangeName )
        // Note: on Node 6 Buffer.from(msg) should be used
        channel.publish(exchangeName, routingKey, new Buffer(message));
        console.log(" [x] Sent :" + message);
      }
    });
  }
  else {
    //TODO save data locally
    console.log("[AMQP] No Connection available - storing data locally");
    localDataFileManager.addData(JSON.parse(message));
  }
}
```

When a connection is performed this module also try to sent all the message unsent due to a lack of Internet connection and stored locally.

```
function whenConnected(callback){

  // Try to send locally stored messages never sent to the server.
  var exchangeName = 'patientData';
  var routingKey = 'datacentre.receive.patientdata';
  var exchangeType = 'topic'
  var isDurable = false;

  // Retrive data stored locally when connection is missing.
  localDataFileManager.getDataList( function(error, previousDataList) {
    if (error) {
      console.log(error);
    } else {
      previousDataList.forEach(function(element){
        publish(exchangeName, exchangeType, routingKey, isDurable, element);
      });
      // Once sent data is deleted.
      localDataFileManager.clearList();
    }
  });

```

```
22    callback();
23  }
```

The control unit use two different publisher, that use that library, even if they are very similar:

- *sensorPublisher* that is used when a new sensor is connected to the control unit in order to send his information to the data center.
- *dataPublisher* that is used to send measured data to the data center in the related patient's folder.

```
1  // RabbitMQ Publisher that send message when a new data is
       received from a sensor.
2
3  var rabbitMQ = require('../pub_sub/rabbitmqLibrary');
4  var exchangeName = 'patientData';
5  var routingKey = 'datacentre.receive.patientdata';
6  var exchangeType = 'topic'
7  var isDurable = false;
8
9  var connected = false;
10
11 /**
12 * Connect to the remote server.
13 */
14 function connect(callback) {
15   rabbitMQ.connectToServer(callback);
16   connected = true;
17 }
18
19 /**
20 * Publish the message to the remote exchange.
21 */
22 module.exports.publishMessage = function (data) {
23   console.log("Connected ? " , connected);
24   if (connected){
25     rabbitMQ.publishToServer(exchangeName, exchangeType,
       routingKey, isDurable, data);
26   } else {
27     connect(function() {rabbitMQ.publishToServer(exchangeName,
        exchangeType, routingKey, isDurable, data);});
28   }
29 }
```

- *Analyser :* Module handle analysis on data received from the sensors and try to send the data and the result to the remote data centre. In particular this module :
  - Check if the data send by sensor is valid. By default every sensor that would communicate with the control unit has to send measured data in Json format like :
    "sensorData" {
    "sensorID" : String ,

"data" : Double

}

– Retrieve information about sensor necessary to the analysis (analyze cannot be done without knowing the datatype of the sensor itself). The module has a local list of sensor information that is checked before retrieve information from the remote API of the control Unit. If sensor cannot be found in those lists the sensor was never connected previously with the current control unit and analysis cannot be performed due to lack of sensor informations. In this case an error is shown.

```
/**
* Function to retrieve sensor's information necessary for the analysis.
*/
function getSensorInfo(id, callback) {
  var found = false;
  var sensorInfo = "{}";

  // check if it's already loaded locally :
  sensorsList.forEach(function(element) {
    json = JSON.parse(element)
    if (json.sensorId === id){
      console.log("Found sensor info locally");
      found = true;
      sensorInfo = json;
      callback(JSON.stringify(sensorInfo));
    }
  });

  if (! found){
  // retrieve sensor informations from data storage using REST service.
    console.log(id);
    request.get('http://localhost:3000/api/sensors/' + id, function (error, response, info) {
      console.log('error:', error); // Print the error if one occurred
      console.log('statusCode:', response && response.statusCode); // Print the response status code if a response was received
      console.log('body : ' + info );
      if ( info != undefined && info != ''){
        sensorsList.push(info);
        console.log("Found sensor info remotelly");
        sensorInfo = info;
        callback(sensorInfo);
      } else {
        callback(sensorInfo)
      }
    });
  }
}
```

– Analyse the data based on sensor datatype and some threshold for each type of vital parameter handled by the system. We have tested the system with a limited support to only **temperature**, **heart beat**

and **glycemia**, but a simple addition can be done to add support to different values. This simple threshold analyser can decide if the measured values are ok or there are some medical issues. This analysis is very simple and limited because it analyse only the single value provided and cannot correlate different values or a stream of data in time. It has two different level of issues: if the anomaly is little an alert is send to the doctor associated to that patient (if the net is available). If the value is considered as a life-threatening issues the control unit also trigger an emergency protocol that is able to call emergency rescuers for the patient.

The protocol use the hardware of the control unit to interact with the user, so the module that handle the hardware described above is required. If a medical emergency is detected a counter is increased until a threshold is reached. This because we want to avoid false positive due to sensor spike and misreading. In this first version this threshold is 10 event. When the threshold is reached an emergency is raised and the hardware protocol start as described above.

```
/**
* Perform simple analysis on the data received from the
    sensors based on information available for this sensor
    .
*/
function analyse(sensorData, sensorInfo) {
  console.log("Analysis started ! ");
  console.log("Analysing ", sensorInfo.value.dataType);

  var value = sensorData.data;
  var level;
  var description;

  // analysis result depend on the vital parameter
    measured by the sensor.
  switch(sensorInfo.value.dataType) {
    case 'heart_rate' :
    case 'heartbeat':
        if (value <= 20 || value >= 250 ){
          level = 3;
          description = 'Emergency: critical heart beat';
          lcd.write("Emergency: heart");
          lcdHasMessage = true;
          emergencyCount += 1;
          if (emergencyCount >= emergencyThreshold){
            emergencyManager.startEmergency() ;
            emergencyCount = 0;
          }
        } else if ( value <= 40 || value >= 180) {
          level = 2;
          description = 'Warning: high heart beat !';
          lcd.write("Alert: heartbeat");
          lcdHasMessage = true;
        } else {
          level = 1;
          description = 'Heart Beat OK !';
      if (lcdHasMessage ) {
        lcd.reset();
        lcdHasMessage = false;
```

```
37        }
38          }
39          break;
40
41      case 'temperature':
42        if (value >= 33 && value < 37 ){
43          level = 1;
44          description = 'Temperature OK !';
45      if (lcdHasMessage ) {
46          lcd.reset();
47          lcdHasMessage = false;
48        }
49        } else if ( value >= 37 && value <= 41) {
50          level = 2;
51          description = 'Warning: high temperature';
52          lcd.write("Alert: high temp");
53      lcdHasMessage = true;
54        } else {
55          level = 3;
56          description = 'Emergency: critical temperature';
57          lcd.write("Emergency: temp");
58          lcdHasMessage = true;
59          emergencyCount += 1;
60          if (emergencyCount >= emergencyThreshold){
61            emergencyManager.startEmergency() ;
62            emergencyCount = 0;
63          }
64        }
65        break;
66
67      case 'glycemia' :
68        if (value >= 60 && value < 110 ){
69          level = 1;
70          description = 'Glycemia OK !';
71      if (lcdHasMessage ) {
72          lcd.reset();
73          lcdHasMessage = false;
74        }
75        } else if ( value >= 100 && value <= 200) {
76          level = 2;
77          description = 'Warning: high glycemia level';
78          lcd.write("Alert: glycemia");
79      lcdHasMessage = true;
80        } else {
81          level = 3;
82          description = 'Emergency: Critical Glycemia level'
    ;
83          lcd.write("Emergency: glyc");
84          lcdHasMessage = true;
85          emergencyCount += 1;
86          if (emergencyCount >= emergencyThreshold){
87            emergencyManager.startEmergency() ;
88            emergencyCount = 0;
89          }
90        }
91        break;
92      default:
```

```
93          level = 2;
94          description = 'Warning: Cannot analyze this
        parameter !';
95          lcd.write("Alert: Error");
96        lcdHasMessage = true;
97        }
98
99        console.log("Analysis completed ! ");
```

– Send the message complete with all the info and the result of the analysis to the data center to store and elaborate it. The communication is performed using RabbitMQ Publish-Subscribe, so the module described above is required. the message has the format :

```
{
    "type" : String (dataType),
    "message" : {
        "patientID" : String,
        "value" : String,
        "unit" : String,
        "timestamp" : String,
        "output" : {
            "level" : String,
            "description" : String
        }
    }
}
```

9.3 Sensors

9.3.1 Hardware Configuration

We have implemented different kind of sensors as described in the previous section, in particular we focused on using:

- **Wifi Kit 8** [6] because it is programmable via Arduino IDE, it has built in display and battery interface, it is small and can have connected various sensors. For this particular sensor we have implemented heart rate and temperature monitor.
- **Samsung Galaxy S7 edge** because it was available and has a built-in sensor for heart rate and oximeter.

Wifi Kit 8 . Both sensors made with this board have a common schema, in particular we used the SH HC 08 [4] BLE module for the communication and the main board for each of them. SH HC 08 is connected to the board via a software serial port, with TX on pin D7 (corresponding to RX in bluetooth) and RX on pin D8 (corresponding on TX in bluetooth) . Wifi Kit 8 has also built-in WiFi but for less power consumption we have chosen to use the BLE module. Another important thing is that Wifi kit 8 has also a connector for

a small battery, so that it can be transported easy and used everywhere. The software part is explained deeply in the next section.

Temperature Sensor. For this sensor we used a Dallas temperature sensor DS18B20 [5] connected via 1-wire to pin D6, as shown below. Unfortunately we don't have any temperature sensor that measure human temperature so for testing it we have used a environmental temperature sensor.

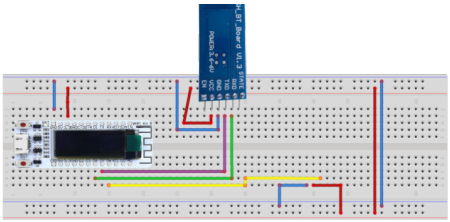

Figure 3: Hardware connection of Temperature Sensor

Pulse Sensor. For the pulse sensor we have used a pulse sensor bought on Ebay like the one that can be found here [8]. It is connected to the board via analog input on pin A0.

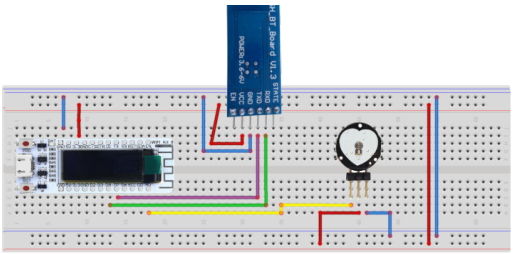

Figure 4: Hardware connection of Pulse Sensor

Samsung Galaxy S7 edge We have decided to use this device to represent a type of sensors capable of internet connection, so we can test the most wide range of possible sensors family with our system. The device has a built-in sensor that spoil two different LED (a red one and an infrared) to measure heart rate, and other useful information about the health of the user, like oxygen saturation, from the finger of the user. Unfortunately the built-in application that handle that sensor and perform analysis on them to obtain information cannot permit to share this data to other application easily so we have built a specific application for sending data to our system, but limited to the heart rate. This application, named *H2 - HeartBeat Android Monitor*, simply allow user to measure heart rate using the built-in sensor and send it to a near control unit using the internet connection. More details on the development will follow in the next section.

9.3.2 Software Component

Android Heart Beat sensor

The Android application is designed to read heart beat data from the user and send it using a REST API (so over the web) to a near control unit. The android application use the built-in sensor to read heart-beat parameter from the user. In order to do that it was necessary to use a non-standard android library given by the manufacturer of the device. So the code will work only on a Samsung device. Note that this is a prototype application, useful to test system with real sensor, not designed for a generic user, so the user interface is really simple and application could still have some buggy behaviours.

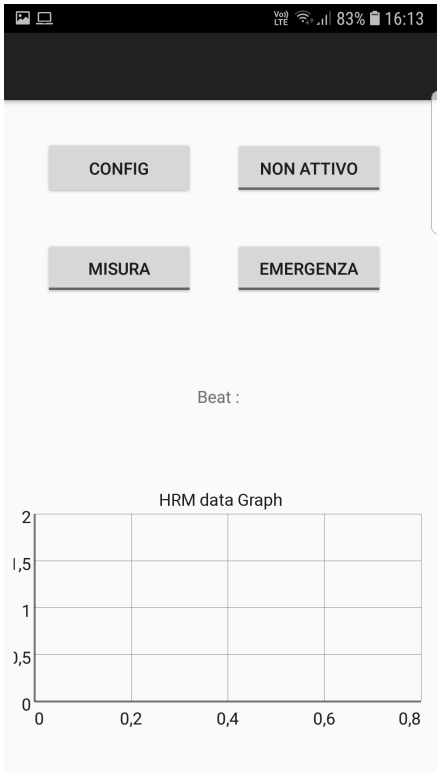

Figure 5: Android Sensor App Homepage

The application has a single Activity that allow the user to :

- Configure the information about the user of that sensor (Sensor Name, PatientID, data type and unit of measurement).
  This feature is represented by a Dialog that appear on click on the button 'config' and allow the user to insert all the information about the sensor. Those informations are saved in the *SharedPreferences*, so the user has to

insert it only at the first usage of the app. Those information are useful to automatically add a new sensor to a control unit at the first connection.

- Set the Host address to connect to and to which messages will be sent. This should be the address of the control unit REST API server.
- Start / Stop the measurement of the heart beat. The click on the button turn on/off the sensor and start the analysis over the data read. The sensor return a raw data that need to be elaborated in order to have a correct reading of the heart beat. The approach used was found in [11]. Shortly the procedure is :
  - Gather 128 sample measure.
  - Applying to this values a simple Hamming window function.
  - Applying of the Fast Fourier Transform algorithm in order to obtain their Discrete Fourier Transform.
  - Applying a band-pass Butterworth filter on the result.
  - Calculate the Magnitude of the filtered result.
  - Obtain the beat analysing the peak value of the result vector and applying a corrective constant to normalize the result.

```
1  public class SignalAnalyser {
2
3      private static final int SEGMENT_LENGTH = 128;
4      /**
5       * Calculate heart beat from a list of data read from a
       sensor.
6       * @param dataList a list of data measured by sensor. it
       should be at least 128 value.
7       * @return the value of the heart beat
8       */
9      public static float analyzeData( List<Float> dataList) {
10         double[] sample = new double[SEGMENT_LENGTH];
11         for (int i = 0; i < SEGMENT_LENGTH; i++) {
12             sample[i] = dataList.get(i);
13         }
14
15         double[] hamming = new double[SEGMENT_LENGTH];
16
17         for (int i = 0; i < SEGMENT_LENGTH; i++) {
18             hamming[i] = (float) ((0.54 - (0.46 * Math.cos(
       Math.PI * 2 * (double) i / (double) (SEGMENT_LENGTH - 1)))
       ));
19         }
20
21         double[] im = new double[SEGMENT_LENGTH];
22         double[] re = new double[SEGMENT_LENGTH];
23
24         for (int i = 0; i < SEGMENT_LENGTH; i++) {
25             re[i] = sample[i] * hamming[i];
26         }
27
```

```
28          FFT fft = new FFT(re.length);
29          fft.fft(re, im);
30
31          Butterworth filter = new Butterworth();
32          filter.bandPass(2, 125, 2, 0.5);
33
34          for (int i = 0; i < SEGMENT_LENGTH; i++) {
35              re[i] = filter.filter(re[i]);
36              im[i] = filter.filter(im[i]);
37          }
38
39          double[] fftMagnitude = fftMagnitude(re, im);
40
41          double peak = Double.MIN_VALUE;
42          for (int i = 0; i < 128; i++) {
43              if (fftMagnitude[i] > peak) {
44                  peak = fftMagnitude[i];
45              }
46          }
47
48          double beat = getBeat(peak);
49      return (float) beat;
50      }
51
52      private static double getBeat(double peak) {
53          double s = 57.14;
54          return (peak / 100000) * s;
55      }
56
57      private static double[] fftMagnitude(double[] re, double[]
        im) {
58          if (re.length != im.length)
59              return null;
60          double[] fftMag = new double[re.length];
61          for (int i = 0; i < re.length; i++) {
62              fftMag[i] = Math.sqrt(Math.pow(re[i], 2) + Math.
        pow(im[i], 2));
63          }
64          return fftMag;
65      }
66  }
```

The Main Activity also has a label that display the measured beat and a graph that plot the data measured by the 2 different sensors, as shown in the picture below :

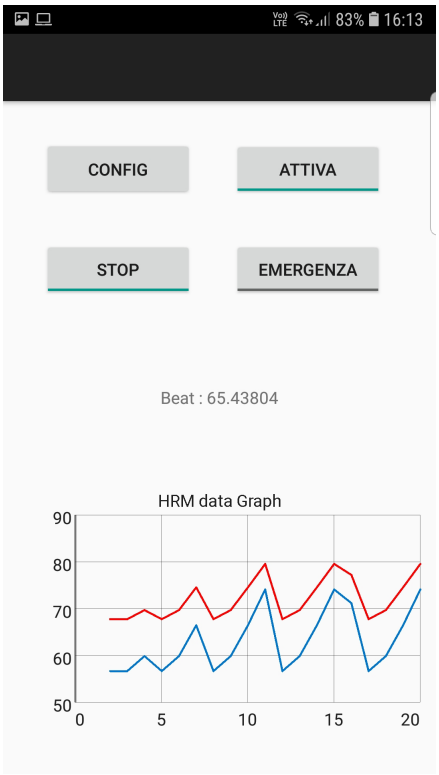

Figure 6: Android Sensor App in use

9.3.3 Wifi Kit 8 Sensors

Wifi kit 8 is designed to be as simple as possible and also to use little power, that is why we have used BLE instead of WiFi. It is totally implemented in the Arduino language. It is composed by 3 common modules that should be used if we want to change the type of sensor and other modules specific for the sensors.

Common Modules The 3 common modules are:

- **ble-device-manager** This module has 2 functions and it has the task to initialize the BLE device and send the data over it. It requires the SoftwareSerial Arduino library to be installed and its functions are:
  - *setupBluetooth(int baudRate)* That takes as input the baudRate for begin the serial port (usually is 9600) and it starts the serial over pins 13 for RX and 15 for TX. It must be called in the setup function.
  - *sendDataOverBLE(const char *text)* That takes as input a String and writes it to the software serial created with the function above.

- **data-display** This module has the task to display on the built-in screen in the the data retrieved and sent via Bluetooth. It uses the u8g2 Arduino library and has 2 functions:
  - *setupDisplay()* That initializes the communications with the built-in display.
  - *drawTextCentered (const char *text)* That takes as input a String and displays it centered on the built-in screen. In order to do this it sets the font and calculates the pixel width of the text. After that it takes the width of the display and starts writing the text from pixel $\left[\frac{displayWidth - Textwidth}{2}, \quad 0\right]$.

Temperature Sensor The temperature sensor has 1 main class, another class for retrieving the temperature and on for stabilizing it:

- **temperatureRead** That requires the OneWire and DallasTemperature Arduino libraries. It has 2 functions:
  - *readTemperature()* that reads the temperature from the sensor and returns a float with the read temperature.
  - *processAndSendTemperature()* That reads the value from the sensor each time, then compares it with the previous read value and if the difference is less than 0.5 degrees then it retrieves the second value and so on, otherwise it takes the latest as base value and then it starts again. When 10 values are retrieved it stops and calculates the average with the function mediumValue(), it displays the value on the screen and sends it over BLE.
  - **temperature-sensor-dallas** This is the main module, it has the setup function that sets up all the connections (bluetooth, temperature sensor and display) and the loop functions that calls the function processAndSendTemperature() with 1 second delay each time for avoiding control unit overload.
- **stabilization** This module has the task to take 10 retrieved values and calculate the average of them taking the 6 items in the middle. In particular it has 4 functions:
  - *setValueForAverage(int valuePosition, float value)* That takes as input an index from 0 to 9 and sets the corresponding value to the value passed.
  - *getNumValuesForAverage()* That returns the number of values to be stored before calling the mediumValue() function.
  - *int compare (const void* num1,const void* num2)* That compares the 2 numbers pointed by the 2 pointers, it is required by the quickSort. it returns 1 if num1 is greater than num2, 0 if they are equal, -1 otherwise.

– *mediumValue()* That is called when the array is full, it sorts the array with the built-in Arduino function qsort, then it discards the first 2 values and the last 2 values and returns the average of the left values.

Heart Rate Sensor The heart rate sensor is slightly different from the temperature sensor since it has an interrupt that triggers each 2 milliseconds and after that it retrieves the value of the heart rate. When is retrieved in the main class is checked that the difference between the value retrieved and the past is less than 3 and after 6 times that is not so different than it is sent and the sensor is stopped.

9.4 Data Centre

It is totally implemented in Java language and it is divided in two parts that use different technology: (i) interfaces to map RESTful API's functionality and (ii) an actor model to mange the publish subscribe patter. We will now go to explain them in more detail.

9.4.1 DB Manager

This is the part that manages the interactions with both Association and Application RESTful API. In order to do that we decided to use the framework Jersey client side, because this allows us to make remote calls to API in a higher way, without using pure HTTP request. The code below represents how to create a client that make this request.

```
1 private static Client CLIENT =ClientBuilder.newClient();
2 private static WebTarget TARGET = CLIENT.target(URIrequest.H2_ROUTE
      .getPath());
```

Once we have the client instance, we can create a WebTarget using the root URI of the targeted web resource. The URI specified (*URIrequest.H2_ROUTE-.getPath()*) belongs to the enum reported below:

```
1  public enum URIrequest {
2      PORT_ASSOCIATION("5225/"),
3      PORT_H2_ROUTE("3000/"),
4      BASIC_ROUTE("http://localhost:"),
5      H2_ROUTE(BASIC_ROUTE.getPath()+PORT_H2_ROUTE.getPath()+"
       database/application"),
6      ASSOCIATIONS_ROUTE(BASIC_ROUTE.getPath()+PORT_ASSOCIATION.
       getPath()+"database/associations");
7
8      private String path;
9
10     URIrequest(final String path){
11         this.path = path;
12     }
13
14     public String getPath() {
15         return path;
16     }
17
18 }
```

To make a specific request, you have to use a specific URI that can be create from the WebTarget using the method *path* as shown below:

```
1 private static WebTarget H2_REGISTRATION = TARGET.path("/
    registration");
```

Now we will go to describe all DB Manager components:

- **AssociationsManager**: this is the interface that manages the request related to the associations. It can:
  - crate a new association between a specific patient and a specific doctor
  - request if a specific association really exist
  - delete an association

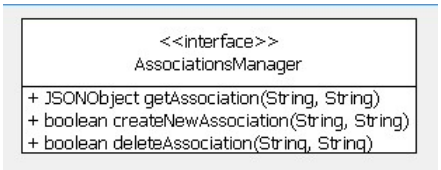

Figure 7: UML Association Manager

- **UserManager**: this is the interface that manages the request related to users. It can:
  - create a new user
  - delete a specific user
  - get all user's associations
  - get user's data

  There are two other interfaces that extends from this one:
  - **PatientManager**: is specific for a user that have the "patient" role
  - **DoctorManager**: is specific for a user that have the "doctor" role

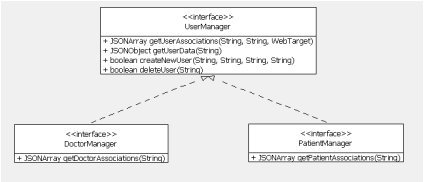

Figure 8: UML Association Manager

- **H2dbManager**: this is the package that includes all the Java interfaces related to all application's functionalities. It handles:
  - new user's registration
  - new user's login
  - user's informations
  - user's sensors and values recorded
  - user's advices sent by doctors
  - user's drugs prescribed by doctors

  There are four interfaces, that respect this functionalities. This choice was made to logically separate the operational scope. Besides to encapsulate at best, we decided to keep shared and common operations in one *H2dbManagerUtils* file.
  The following image shows a UML diagram of this.

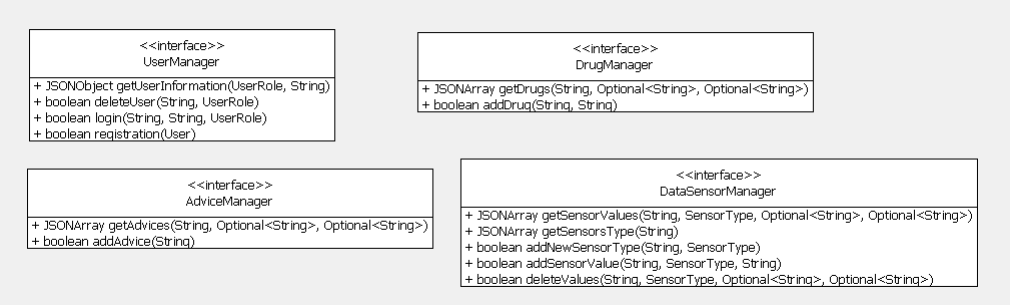

Figure 9: UML Application Manager

Moreover there are classes to shape the Builder Pattern we chose to create a large object. This object represents the users in our application. We applied an advance way(supported by Java 8) to build the Builder Pattern.

```
public UserBuilder with(
            Consumer<UserBuilder> builderFunction) {
        builderFunction.accept(this);
        return this;
    }
```

The with method shown below takes a function of type Consumer. Consumer is a functional interface provided by Java 8, which takes single argument and returns no result. In this case it accepts an object of type person builder which is passed to accept method. Which means that the instance of the builder would be accessible in the lambda expression and we can do whatever we want.

```
User patientUser = new UserBuilder()
        .with(userBuilder -> {
                    userBuilder.idCode = "giulia.lucchi";
```

```
                    userBuilder.name = "Giulia";
                    userBuilder.surname = "Lucchi";
                    userBuilder.cf = "LCCGLI45C64S579I";
                    userBuilder.password = "ciao";
                    userBuilder.mail="lucchigiulia@gmail.com";
                    userBuilder.phones="05476143 3246543123";
                    userBuilder.role = UserRole.PATIENT.getRole();
        })
        .createUser();
}
```

9.4.2 Publish Subscribe

This is the part that manages all communications of the Data Centre. It was decide to use Actors for communications between publishers and subscribers inner the data centre because this they are reactive and in this way it is easier to distribute the Data Centre (if you want to do so, in future). To handle the publish-subscribe pattern, we use RabbitMQ, a message-oriented middleware, that implements the Advanced Message Queuing Protocol.

Now we will go to describe the basic components for publish-subscribe used by Actors:

- **TopicPublisher**: this is the interface that manages Data Centre's publisher. In the implementation of this interface, **TopicPublisherImpl**, we have adapted the documentary code to our needs
- **TopicSubscriber**: this is the interface that manages Data Centre's subscriber. In the implementation of this interface, **TopicSubscriberImpl**, we have adapted the documentary code to our needs

Actors

- **AdvicePublisherActor**: it is called when a new advice is received and it requests to add the advice to the database and publishes it, through the **TopicPublisher**, to the specific patient.

```
return receiveBuilder()
  .match(AdviceMessage.class, message -> {
      String messageToInsert =utils.
            convertToFormatApi(message.getMessage());
    h2dbManage.addAdvice(messageToInsert);
      this.publisher.publishMessage(message.getMessage(),
            "patient."+message.getPatientId()+".receive.advice
    ");
}).build();
```

- **HistoryPublisherActor**: it is called when a clinical history request is received and it requests all values to the database and publishes them, through the **TopicPublisher**, to the user who made the request.

```
return receiveBuilder()
```

```
    .match(AdviceMessage.class, message -> {
        String messageToInsert =utils.
                convertToFormatApi(message.getMessage());
      h2dbManage.addAdvice(messageToInsert);
        this.publisher.publishMessage(message.getMessage(),
                "patient."+message.getPatientId()+".receive.advice
      ");
}).build();
```

- **InfoPublisherActor**: it is called when a info request is received and it requests user's informations to the database and publishes them, through the **TopicPublisher**, to the user who made the request.

```
return receiveBuilder().match(UserMessage.class, userMessage
    -> {
  publisher.publishMessage(userMessage.toJson(), ROUTING_KEY);
}).build();
```

- **LevelPublisherActor**: it is called when the critical level of a value, is not the normal one (alert or emergency). It publishes message, through the **TopicPublisher**, in two different queues, the alert one and the emergency one, to send the critical value to user's doctors, as shown below

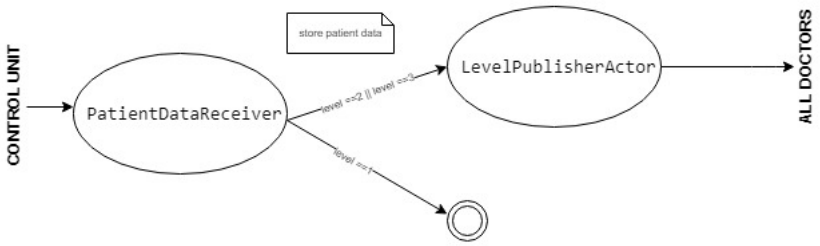

Figure 10: Sequence diagram of alerts and emergency

```
return receiveBuilder().match(ValueMessage.class, message -> {
  JSONArray doctors = patientManager
      .getPatientAssociations(message.getPatientId());
  IntStream.range(0, doctors.length()).forEach(x -> {
      try {
          if (message.getLevel() == 2) {
              this.publisher.publishMessage(message.getValue()
    ,
              "doctor." + doctors.get(x) + ".receive.alert");
          } else {
              this.publisher.publishMessage(message.getValue()
    ,
                  "doctor."+doctors.get(x) + ".receive.
    emergency");
          }
      } catch (Exception e) {
          e.printStackTrace();
```

```
        }
    });
}).build();
```

- **MultiAdvicePublisherActor**: it is called when a request to get the advices of a specific user is received and it requests all advices to the database and publishes them, through the **TopicPublisher**, to the patient who made the request

```
return receiveBuilder().match(AdviceRequestMessage.class,
    message -> {
  JSONArray values = h2dbManage.getAdvices(message.
    getPatientId(),
      Optional.of(message.getStart()),
            Optional.of(message.getEnd()));
    publisher.publishMessage(values.toString(),
      "patient."+message.getPatientId()+".receive.advice");
}).build();
```

- **MultiDrugPublisherActor**: it is called when a request to get the drugs of a specific user is received and it requests all drugs to the database and publishes them, through the **TopicPublisher**, to the patient who made the request

```
return receiveBuilder().match(DrugRequestMessage.class,
    message -> {
  JSONArray values = h2dbManage.getDrugs(message.getPatientId
    (),
    Optional.of(message.getStart()), Optional.of(message.
    getEnd()));
    publisher.publishMessage(values.toString(),
      "patient."+message.getPatientId()+".receive.drug");
}).build();
```

- **AdviceReceiverActor**: it is called by the **TopicSubscriber** when it receives a new advice for a specific patient from a doctor. It creates a new message containing the advice and sends it to **AdvicePublisherActor**, that does what we described before

```
JSONObject json;
try {
  json = new JSONObject(message);
    String patientId = json.getString("patientId");
    String doctorId = json.getString("doctorId");
    String advice = json.getString("advice");
    String timestamp =  json.getString("timestamp");

    AdviceMessage adviceMessage = new AdviceMessage
          (patientId, doctorId, advice, timestamp);
  getContext().actorSelection("/user/app/advicePublisherActor"
    )
          .tell(adviceMessage, ActorRef.noSender());
} catch (JSONException e) {
  e.printStackTrace();
}
```

- **AdviceRequestActor**: it is called by the **TopicSubscriber** when it receives a request to get all the advices related to a specific patient. It creates a new message containing the request and sends it to **MultiAdvicePublisherActor**, that does what we described before

```
JSONObject json;
try {
  json = new JSONObject(message);
  String patientId = json.getString("patientId");
  String start = json.getString("start");
  String end =  json.getString("end");

  AdviceRequestMessage adviceRequest = new
    AdviceRequestMessage
            (patientId, start, end);

  getContext().actorSelection("/user/app/
    multiAdvicePublisherActor")
        .tell(adviceRequest, ActorRef.noSender() );
} catch (JSONException e) {
  e.printStackTrace();
}
```

- **DrugReceiverActor**: it is called by the **TopicSubscriber** when it receives a new drug for a specific patient from a doctor. It requests to add the drug to the database

```
JSONObject json;
try {
  json = new JSONObject(message);
  String patientId = json.getString("patientId");
    String prescribedDrug = json.getString("message");
    String messageToInsert = utils.convertToFormatApi(
    prescribedDrug);

     H2dbmanager.addDrug(patientId, messageToInsert);
} catch (JSONException e) {
  e.printStackTrace();
}
```

- **DrugRequestActor**: it is called by the **TopicSubscriber** when it receives a request to get all the drugs related to a specific patient. It creates a new message containing the request and sends it to **MultiDrugPublisherActor**, that does what we described before

```
JSONObject json;
try {
  json = new JSONObject(message);
    String patientId = json.getString("patientId");
    String start = json.getString("start");
    String end =  json.getString("end");

    DrugRequestMessage drugRequest = new DrugRequestMessage
        (patientId, start, end);
    getContext().actorSelection("/user/app/
    multiDrugPublisherActor")                    .tell(
    drugRequest, ActorRef.noSender() );
```

```
} catch (JSONException e) {
  e.printStackTrace();
}
```

- **PatientDataReceiverActor**: it is called by the **TopicSubscriber** when it receives a new sensor to add for a specific patient or when it receives a new sensor's value for a specific patient. It adds the sensor to the database or it adds the value to the database and creates a new message containing the value and sends it to **LevelPublisherActor**, that does what we described before

```
JSONObject json;
try{
  if(envelope.getRoutingKey().equals(ROUTING_KEY_DATA)){
    json = new JSONObject(message);

    String type = (String) json.get("type");
        JSONObject value = (JSONObject) json.get("message");
    JSONObject output = (JSONObject) value.get("output");
        String idPatient = (String) value.get("patientId");
        int level = output.getInt("level");

        if(level == 2 || level == 3){
          getContext().actorSelection("/user/app/
    levelPublisherActor")
            .tell(new ValueMessage(level,json, idPatient),
                    ActorRef.noSender());
    }


      String messageToInsert =
          utils.convertToFormatApi(value.toString());
      H2manager.addSensorValue(idPatient,
          SensorType.valueOf(type.toUpperCase()),
    messageToInsert);
  }else if(envelope.getRoutingKey().equals(ROUTING_KEY_SENSOR)
    ){
      json = new JSONObject(message);
      String patientId = (String) json.get("patientId");
      String type = (String) json.get("type");
      H2manager.addNewSensorType(patientId,SensorType
                  .valueOf(type.toUpperCase()));
     }
} catch (JSONException e) {
  e.printStackTrace();
}
```

- **PatientDataRequestActor**: it is called by the **TopicSubscriber** when it receives a clinical history request for a specific patient. It creates a new message containing the request and sends it to **HistoryPublisherActor**, that does what we described before

```
JSONObject json;
try {
  json = new JSONObject(message);
    String patientId = json.getString("patientId");
    String type = json.getString("type");
```

```
6       String start = json.getString("start");
7       String end =  json.getString("end");
8       String requesterRole =  json.getString("requesterRole");
9       String requesterId = json.getString("requesterId");
10
11      HistoryMessage historyMessage = new HistoryMessage
12          (patientId, type, start, end, requesterRole,
        requesterId);
13     getContext().actorSelection("/user/app/
        historyPublisherActor")
14          .tell(historyMessage, ActorRef.noSender());
15  } catch (JSONException e) {
16    e.printStackTrace();
17  }
```

9.4.3 RESTful API

The RESTful APIs are realized as a part of a web service using parts of the MEAN Stack. Particularly, to develop this module we used NodeJs and Express for the request handling. For managing the data storage the optimal choice should have been MongoDB, because it results very scalable. The Angular part of the stack was also not necessary.
There are the two RESTful APIs, independent one another, as already shown in above chapter:

- *Application API*
- *Association API*

For these APIs, we used a MVC pattern in the deployment of this modules, though the view part has not been developed, because it is not necessary in RESTful API. Because of that the folder structure is the same. In both cases, the RESTful API contain a main file js, named *app.js* and also a folder named *app_server* that reflects the MVC pattern and it includes some folder:

- */routes*: contains the router file that specify every request received from the data centre.
- */controller*: contains all the callbacks of the request that handles the storage and retrieval of data from application database.
- */models*: contains all the database schemes mentioned in the previous chapter.
- */view*: it's not necessary, but it used to get a error page in error case, during the development.

The callbacks have the aim of storing data in the application database and retrieving them from it. The connection to this database was made in *database.js* file in the following way, as written by mongoose documentation.

```
var mongoose = require('mongoose');
var debug = typeof v8debug === 'object';

mongoose.connect('mongodb://h2db:h2db@ds111420.mlab.com:11420/h2db'
    , function(err) {
    if (err) throw err;
});

var con = mongoose.connection;
con.on('error', function (err){
    console.log('errore di connessione', err);
});

con.once('open', function (){
   console.log('connessione riuscita!');
});
```

Application API

This module was named *H2-db RESTful API*. Hereafter we'll specify the content of folders, mentioned above. Here we have:

- *routes*: contains the router file called *H2routing.js*, that specify every request received from the data centre.

```
router.post('/registration', userAuthentication.register);

router.get('/login', userAuthentication.login );

router.get('/sensors', sensorTypeOperations.getSensorTypes);

router.put('/sensors', sensorTypeOperations.putSensorType );

router.post('/sensors/values', sensorDataOperation.addValue);

router.delete('/sensors/values', sensorDataOperation.
    deleteValue);

router.get('/sensors/values', sensorDataOperation.getValues);

router.post('/advices', adviceOperations.addAdvice);

router.get('/advices', adviceOperations.getAdvices);

router.post('/drugs', drugsOperations.addDrugs);

router.get('/drugs', drugsOperations.getAllDrugs);

router.get('/patients', userOperations.getPatient);

router.get('/doctors', userOperations.getDoctor);

router.delete('/patients', userOperations.deletePatient);

router.delete('/doctors', userOperations.deleteDoctor);
```

- *controllers*: It includes the file .js divided by callback's scope, as in the *H2dbManager* package in the data centre. These file.js are:
  - *userAuthentication.js*
  - *UserOperations.js*
  - *sensorDataOperations.js*
  - *sensorTypeOperations.js*
  - *adviceOperations.js*
  - *drugsOperations.js*

  Each file is well structured and named to result readable to anyone.

Association API This module was named *Association RESTful API*. Hereafter we'll specify the content of folders, mentioned above. Here we have:

- *routes*: contains the router file called *associationRouting.js*, that specify every request received from the data centre.

```
1  /** GET request to return patient's informations */
2  router.get('/patients', patientOperation.findPatient);
3
4  /** POST request to insert a new patientv */
5  router.post('/patients', patientOperation.insertPatient);
6
7  /** DELETE request to remove a patient and all his
       associations with doctors */
8  router.delete('/patients', patientOperation.removePatient);
9
10 /** GET request to return doctors's informations */
11 router.get('/doctors', doctorOperation.findDoctor);
12
13 /** POST request to insert a new doctor */
14 router.post('/doctors', doctorOperation.insertDoctor);
15
16 /** DELETE request to remove a doctor and all his associations
        with patients */
17 router.delete('/doctors', doctorOperation.removeDoctor);
18
19
20 /** POST request to insert a new medical association of doctor
        and patient */
21 router.post('/relationship', associationOperation.
       insertAssociation);
22
23 /** GET request to find the association between specific
       doctor and patient
24  *  or to find all doctor's or patient's associations */
25 router.get('/relationship', associationOperation.
       getAssociations);
26
27 /** DELETE request to remove an association between specific
       doctor and patient */
28 router.delete('/relationship', associationOperation.
       removeAssociation);
```

- *controllers*: It includes the file .js divided by callback's scope. These file.js are:
  - *associationOperation.js*
  - *doctorOperation.js*
  - *patientOperation.js*
  - *userOperation.js*

  Each file is well structured and named to result readable to anyone.

9.4.4 Patient and Doctor Web Server

The patient and doctor Web Server is realized using parts of the MEAN Stack. In particular for this module we have used NodeJS and Express for the request handling. No AngularJS is used at the time of writing due to a lack of time, but the dynamic pages are a future todo for the project. We have achieved a certain grade of adaptability of screen type and dimension using *Bootstrap* framework for graphic rendering.
This module can be found in package *H2_ Client*. Here we have same main folders :

- *routes*: contains all the folder, that is routing file, pub/sub files and socket management files.
- *views*:that contains the jade file that generate the HTML file returned to the client after a request is completed. The structure of jade files consist in a*layout.jade* file, that is extended to other .jade files.

```
1  doctype html
2  html
3    head
4      meta(name='viewport', content='width=device-width, initial
       -scale=1.0')
5      title= title
6      link(rel='stylesheet', href='/stylesheets/solar.css')
7      link(rel='stylesheet', href='/stylesheets/style.css')
8      script(src='/jquery/jquery.min.js')
9      script(src='/bootstrap/js/bootstrap.min.js')
10
11
12
13   body
14     block scriptWebSocket
15     block notifications
16     .navbar.navbar-default.navbar-fixed-top
17       .container
18         block nav
19
20
21   .container
22     block content
23
```

```
24      footer
25        .row
26          .col-xs-12
27            small &copy; H2 Team @ Unibo - 2018
```

This module focuses on the publish subscribe pattern, implemented with RabbitMQ. To encapsulate the publish and subscribe method, we have been created two file:

- *constants.js*: collects all the constants to configure the RabbitMQ (ampq address, queue, exchange, routing key ...)
- *RabbitMQ Library*: includes the RabbitMQ Publish-Subscribe utility functions:
  - **start function**: it tries to connect to the RabbitMQ server and initialize the module

```
1  function start(callback) {
2      amqp.connect(constants.amqpAddress, function(error,
       connection) {
3        if (error) {
4          console.log(error)
5        } else {
6          connection.on("error", function(error) {
7            console.log(error)
8            if (error.message !== "Connection closing") {
9            }
10         });
11
12         connection.on("close", function() {
13           console.error("[AMQP] connection closed");
14         });
15         if (error) {
16           console.error("[AMQP connection]", error.message);
17         } else {
18           console.log("[AMQP] connected");
19           callback(connection);
20         }
21       }
22     });
23   }
```

- **publish function**: to publish a message to a remote RabbitMQ server, specifying exchange name, exchange type, routing key and message.

```
1  function publish (exchangeName, exchangeType, routingKey,
       isDurable, message, callback, amqpConnection) {
2    amqpConnection.createChannel(function(error, channel) {
3      if(error) {
4        console.log('[AMQP] Error creating channel');
5      } else {
6        // console.log(" [AMQP] Channel created")
7        channel.assertExchange(exchangeName, exchangeType, {
       durable: isDurable});
```

```
8         channel.publish(exchangeName, routingKey, new Buffer(
      message));
9         callback();
10      }
11    });
12  }
```

- **subscribes function**: to subscriber to a remote RabbitMQ server, specifying exchange name, exchange type, queue, routing key and callback.

```
1  function subscribe(exchangeName,exchangeType, queue,
       routingKey, isDurable, callback, amqpConnection) {
2
3    console.log()
4    amqpConnection.createChannel(function(error, ch) {
5      if(error) {
6        console.log('[AMQP] Error creating channel');
7      } else {
8        ch.assertExchange(exchangeName, exchangeType, {durable:
       isDurable});
9        ch.assertQueue(queue, {exclusive: false}, function(err,
       q) {
10         if(!err) {
11           ch.bindQueue(q.queue, exchangeName, routingKey);
12           ch.consume(q.queue, function(message) {
13             callback(message);
14           }, {noAck: true});
15         }
16         else {
17           console.log(err)
18           console.log(exchangeName)
19           console.log(queue)
20           console.log(routingKey)
21         }
22       });
23     }
24   });
25 }
```

- **subscribeToServer** e **publishToServer**: use the start function taking the subscribe or publish function described above as the callback. These functions are used in the other file .js to communicate with data centre trough the apposite RabbitMQ channels.

Web socket

The web socket module has the task to manage all the connected web sockets and to send over them the proper data. Since Javascript is Asynchronous all the concurrency is done by callbacks. With this method the webSocket server is initialized at the beginning of the program and then it sets up a callback. This callback is called when a new client is connected and stores the id of the connection with the user id into a list, so that if the user connected with multiple devices all the web sockets are related to him. When this user connects it also starts a new subscriber, if it is a doctor to the emergency and alert

queues, otherwise to the advice queue. With this function it passes the control to the subscriber that calls the function *sendMessageToUsers(user, message, roleSender)* when it receives a new message over the queue.

Client Side

The client Side is very simple, it has a modal in both client and doctor homes and when a message is retrieved via web socket it creates a new div into this modal and put all the relevant information. This div has also a delete button so that if the user accidentally closes the modal it can see all the notification by reopen it and it can delete them when they are not useful anymore. The delete function has inside the id of the div and it calls a remove for deleting it. In particular in the *htmlManagement* module there are all the functions for editing the DOM, while in *doctorWebSocket* and *patientWebSocket* there are all the functions for managing the web socket and the modal.

10 Testing

We tested both the correct functioning of data centre's operations and the general behavior, also under pressure.

10.1 JUnit

We tested the behavior of each data centre's functionalities, except the actor model (tested by simulator). This testing are in *src/test/java/core/dbmanager*, that includes the following testing file:

- *AssociationAndUserManagerTest*
- *H2dbManagerTest*

10.2 Simulator

In order to test the system with multiple patients and doctors and to execute stress test on the various part of the system itself a simulator was developed. This simulator is used in particular to test the scalability of the system and the API with :

- Massive Users (Patient and Doctor) creation on the database.
- Massive associations between users.
- Massive associations between Patient and sensors.
- Massive flows of sensors' data to the system.
- Massive deletion of Users.

10.2.1 Architecture

The simulator is written using Java language and has a very simple structure. It simulates the behaviour of different Control Unit with multiple patients each that send data to the system. In order to do that two entity is defined:

- *Patient* entity that hold information about the patient ID and the ID of every doctor associated with it.
- *ControlUnit* entity which is defined as a thread, so every different unit can work independently form each others. When a new *ControlUnit* is created a certain amount of *Patient*s are created too and they are:
  - added to the system database.
  - associated with a temperature sensor (only temperature are tested, but different kind of sensor can be use here, we have chosen to use only one type to keep the code as simple as possible)

– associated with some doctors. In particular every *Patient* is randomly associated with 1 to 3 doctors presents in the system. In this way also multiple patient-doctor association can be tested.

For the sake of simplicity every *ControlUnit* has the same amount of patients because it did not affect a lot the result of the simulation. All the associations and the DB interactions are created using the API given by the DataCenter module, imported as a jar in *libs* folder.

```
1     /**
2      * Create a single simulated control unit with a specific
      amount of patients.
3      * Every patient is created and associated with the doctor
       in H2 environment.
4      *
5      * @param patients - The amount of patients in this
      control unit.
6      * @param countFrom - A progressive value used for having
      unique ID for the patient also in different control unit.
7      * @param lastDoctorIndex - The index of the last doctors
      created that can be associated with patient.
8      */
9     public ControlUnit(int patients, int countFrom, int
      lastDoctorIndex){
10
11        this.patientList = new ArrayList<>();
12        this.publisherList = new ArrayList<>();
13        this.patientManager = new PatientManagerImpl();
14        this.associationsManager = new AssociationsManagerImpl
      ();
15        this.h2dbManager = new UserManagerImpl();
16        this.h2SensorManager = new DataSensorManagerImpl();
17
18        for (int i = countFrom; i < countFrom + patients; i
      ++){
19            String patientId = "patient.test." + i;
20            Patient patient = new Patient(patientId);
21            PatientDataPublisher publisher = new
      PatientDataPublisher(patientId);
22
23            publisherList.add(publisher);
24            // add patient to application db
25            h2dbManager.registration(new User(patientId,"test"
      , "test", "patient" + i, "TESTTESTTEST", "123456789", "
      test@test.com", UserRole.PATIENT.getRole()));
26            // add patient to association db
27            patientManager.createNewUser(patientId, "patient"
      , "test" + i , "TSTTST12A34B345C");
28            try {
29                // associate patient with some random doctors.
30                // N.B.: If attempt to associate with a doctor
       already associated to him nothing happened, so this can
31                // generate 1 to 3 different association
      randomly.
32                for( int j = 0; j < ASSOCIATION_AMOUNT ; j++ )
       {
```

```
33                      int id = (int) (Math.random() *
    lastDoctorIndex);
34                      String doctorID;
35                      if ( id != 0) {
36                          doctorID = "test.doctor" + id;
37                      }
38                      else {
39                          doctorID = "test.doctor";
40                      }
41                      System.out.println("Associating with
    doctor " + doctorID);
42                      patient.addDoctor(doctorID);
43                      associationsManager.createNewAssociation(
    patientId, doctorID);
44                  }
45              } catch (Exception e) {
46                  e.printStackTrace();
47              }
48
49              patientList.add(patient);
50              System.out.println("[CONTROL UNIT] Patient created
     : " + patientId);
51          }
52
53          for (int i = countFrom; i < countFrom + patients; i
    ++) {
54              String patientId = "patient.test." + i;
55              System.out.println("Adding Temperature Sensor to "
     + patientId + " !");
56              boolean isAssociated = h2SensorManager.
    addNewSensorType(patientId, SensorType.TEMPERATURE);
57              System.out.println("Associated ? " + isAssociated)
    ;
58          }
59          run = true;
60      }
```

10.2.2 Simulation

When started every *ControlUnit* send a fake temperature message with random values and related analysis result to the Data Center, using the RabbitMQ publish-subscribe server thanks to a simple *TopicPublisher*, already described elsewhere in this report. Every *Patient* send a single message and then the *ControlUnit* wait from 0 to 3000 ms for sending the next one. When all the patient in the *ControlUnit* has sent a message the unit waits 20 seconds before restart the sending of messages. This simulate a stream of sensor's data from fake patients to the Data Center (and then to the whole system).

10.2.3 The Simulator GUI

The simulator GUI is created and handled by the main class (*ControlUnitSimulator.java*) of the simulator, that also :

- Instantiate the different *ControlUnit*.

- Create the fake doctors.

Using the GUI the user can setup the size of the simulation, specifying 2 different parameters :

- *N of Control Unit* - That is the amount of the *ControlUnit* generated. A range from 1 to 10000 is possible. This limit was fixed because every *ControlUnit* is a different Thread, so a very high number can slow down the host computer.
- *N of patient per Control Unit* - That is the amount of patient for **every** *ControlUnit*. A range from 1 to 100 is possible.

So the simulator **can simulate from 1 to 1 million patients simultaneously**.

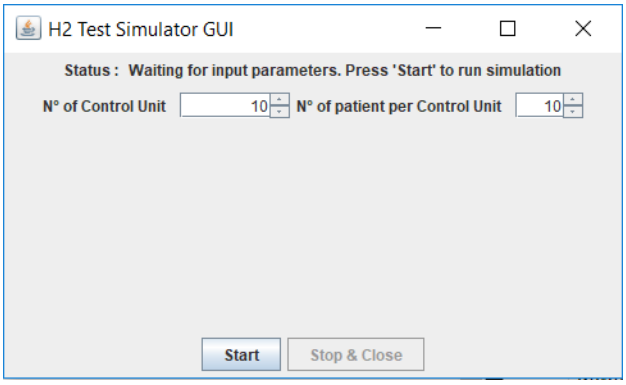

Figure 11: GUI of the simulator

Simulation can be paused and restarted from the simulator GUI, described below.

When the simulation are stopped the whole users and associations are deleted from the system database, so no fake data are left behind.

10.2.4 Test Results

Note that our deployment is a test one so we use only the hardware resources available, that is not enough for the workload that the system will face at full load (various millions of users all over Italy or the world). In particular the RabbitMQ Publish-Subcribe server is host on a Raspberry Pi, so the load of connection to that server is limited by the hardware specification of that device. We tried some tests to check the data centre behaviour with multiple patients, doctors and control units. We used to perform this tests:

- DELL XPS15 PC with: processor Intel Core i7-6700HQ; CPU 2.6 GHz; RAM 16GB

- RASPBERRY PI model B with 700 MHz single-core ARM11 processor, 512 MB of SDRAM, Ethernet 10/100 Mbit/s with a Raspbian OS for the rabbitmq server.

	n control unit	n patients	n doctors numbers	n of mex
TEST 1	5	50	5	50
TEST 2	200	600	200	3600
TEST 3	1000	6000	1000	?

TEST 1: application's behaviour was similar to the basic conditions (one control unit, one patient and one doctor).

TEST 2: application's behaviour was correct but message notifications was delayed in the client web interface (in order of minutes, less than 5).

TEST 3: application has saturated the available connection for the RabbitMQ server after 800 patients. For this reason, with the technology previously described, in our opinion the most significant TEST is the second one.

10.3 Tested requirements

Let's go check now one requirement at a time to see if and how it was tested:

- BUSINNESS:
  1. Tested with unit test with JUnit (test.java.core.dbmanager.H2dbManagerTest) and directly using the system for testing purpose (when a patient/-doctor make an history request, he will receive the list of parameters previously measured and saved)
  2. Tested directly using the system (requests from different devices)
  3. Tested directly using the system (At every out of bound parameters a notification is generated)
  4. Tested directly using the system (At every out of bound parameters a notification is generated and both patient and related doctor are notified)
  5. Tested directly using the system (Using real sensors that are able to measure our real parameters)
  6. Tested with unit test with JUnit (test.java.core.dbmanager.H2dbManagerTest) and directly using the system (When a doctor send advices to a patient it will appear in his device)
  7. Tested with the Simulator (N-N association between patients and doctors)
  8. We suppose that the association are provided directly from the sanitary service and our system is capable to retrieve all the associations automatically.

9. Tested using the simulator (like 7)
10. Tested directly using the system (If a patient or a doctor try to access to not authorized data the system will return an error)
11. Tested directly using the system (Every patients parameters send to the system are saved on a specific document for each patient, so only his data are accessible.)
12. Tested directly using the system (tested with different kind of sensors : heart beat and temperature (real sensors) + glycemia (simulated))

- FUNCTIONAL:

1. Tested directly using the system (Measuring some of our parameter using the real sensor and checking that the value are received from the system without any type of interaction with the user)
2. Tested directly using the system (Request from different device and account)
3. Tested directly using the system (A notification is shown in doctors device every time an out-of-bound parameter is received)
4. Tested directly using the system (A short description is shown every time a parameter is received, e.g. if is too high or too low or simply normal)
5. Tested with JUnit (test.java.core.dbmanager.H2dbManagerTest) and directly using the system (When a doctor send an advice / drugs suggestion the message is received by the related patient)
6. Tested directly using the system (Tested connecting and disconnecting a sensor multiple time without restart the system)
7. Tested directly using the system (We have developed (and tested) two type of sensors : the first one capable of connecting through the web, the latter that need a specific protocol in order to connect (we have used BLE). Morover they measure different vital parameters)
8. Tested directly using the system (Only registered doctors and patients can login to the system, otherwise an error is returned)
9. Tested directly using the system (The application can be accessed only through login pages)
10. Tested using the Simulator (N-N association between patients and doctors)
11. Tested directly using the system (trying to connect from multiple device at the same time)
12. Tested directly using the system (trying to connect from different kind of devices (laptop and mobile))
13. This requirement was not implemented.
14. This requirement was not implemented.

15. Tested directly using the system (The analysis is performed before the system try to connect to the publish-subscribe service, so it is always performed (and shown in the control unit) even if this connection fail.
16. Tested directly using the system (The real phone call is not implemented due to the lack of specific hardware, but a simulation module is contained in the system and it is tested during the normal usage.)
17. This requirement was not implemented.
18. Tested directly using the system (Sending some isolated out-of-bound parameters and checking that the emergency is not called without a certain amount of subsequent out-of-bound parameters)
19. Tested directly using the system (Sending notification to a doctor that is offline and only after that login to the system with that doctors credential. He will receive all the pending notification)
20. Tested directly using the system (if notifications are not deleted, the doctor can retrieve them from the specific section of his account)

- NOT FUNCTIONAL

1. Not completely tested: we have connected the 2 sensor available to us
2. Tested using the system (we have tried to connect different sensor simulators through bluetooth and wifi)
3. Tested using the system (we have tried to use multiple devices simultaneously)
4. Tested using the system (we have tried to use different devices: laptops, smartphones, ...)
5. Tested using the system
6. We cannot test it, we tried to keep all as simple as possible and the interaction method has been designed to make the interaction simple.
7. Security not implemented.
8. This requirement was not implemented.
9. This requirement was not implemented.
10. Tested using the system (the control unit has been disconnected while it was sending data and later reconnected and all the data was received)
11. Tested using the system (the system has been designed to have a local analyser without network connection)
12. This requirement was not implemented.
13. Tested using the system (we have tried with heart rate and temperature)

14. Tested using the system (in an emergency situation it simulates a call to rescuers. We have tried to send isolated out of bound parameters and in this case it doesnt call the rescuers, they are called only after some parameters out of range)
15. Tested using the system (tested with real sensors that sends a flow and with postMan that sends sporadic simulated parameters)
16. Tested using the system (the control unit has been disconnected while it was sending data and later reconnected and all the data was received)
17. Not testable: is made by common sense because when data are retrieved from a sensor they are sent to the clinical folder of a specific patient, so they cannot be used from other people.
18. Tested using the system (we have tried to log into the system from different devices with the same account and all the devices receives the notifications)
19. Tested using the system (we have registered multiple patients and doctors and the doctors received only the notifications related to their patients)
20. This requirement was not implemented.
21. Testing using the Simulator (Trying to create 200 fake doctors and 600 fake patients; The second test was performed with more users (1000 doctors and 6000 patients) and it fail. We have noticed that the limit with our hardware is around 800 patients and 200 doctors => the limit of 1000 is acceptable. More details are presented in the description of this test above.

11 Deployment

To start our application, is necessary to:

- run a single instance of the **data centre** (*datacentre/src/main/java/-core/Main.java*)
- run a single instance of both **association RESTful API**, **application RESTful API** and **H2 client**. In order to do that, it is necessary to run the following instructions:
  - *npm install*
  - *node bin/www*
- run the patient's control unit:
  - remotly connect to the **control unit**:
  - enter in the H2 folder
  - run the server with command *node bin/www*

11.1 Note

The RESTful APIs have to be instantiate because now they aren't hosted in a hosting service.

12 Conclusion

Our application have just basic functionalities to resolve the goal. To be a usable application in reality, it would take some changes that we haven't been fully developed due to a lack of time and resources. For this reason we describe what could be done to complete our application. Regarding the initial requirements, there are some aspect, that we have not developed. These are:

- **P8** and **P9**: this requirements are not our responsibility, because we have dealt with the development of the application, not the build of sensors.
- **D2** and **I2**: in our application, it isn't possible to configure the system with relative specific illness, but the patient can connect all the sensor that he wants. Despite this, it is simple to develop this requirement.
- **D8** and **F13**: we have not developed a data aggregation algorithm and display of this data through graphic for each patient.
- **D14**: It isn't possible to give autonomously simple medical advice to the patient without the supervision of the doctor, because we haven't developed a machine learning algorithm with an automatic training.
- **F17**: in our application, it isn't managed that in case of emergency, the patient's data can be visualised by doctors that can be involved in the rescue
- **I5** and **I13**: in our application, it isn't managed the Wi-Fi network absence in deep way, in fact it's not possible to alert rescuers with all the possible network connections except wi-fi.

12.1 Future Feature

The possible feature to add in our application or to modify are:

- The sensors could in future be used by more users (sharing sensor) and have a profile for everyone
- improvement and adding a real computer security with a respect privacy concept
- make the sensor system and the analyzer wearable as much as possible so as to have continuous and h24 monitoring
- improvement automatic management of the drugs with relative warning
- adding the machine learning through trained algorithms
- sensor management in which you are requesting the data, but we do not do it for time saving

13 Other Possible Applications

This system is applicable to other situations different from the one described this report.
Some of that could be:

- monitoring patient in case of accident or after a sudden illness, sending all his parameters to the ambulance technicians before and during the operation. In that way the intervention could be more efficient, more effective and, at the arrival to the hospital, the medical staff already knows the situation so they can intervene in time.
  To do that, is necessary to place around the town (in public places) specific devices equipped with the main medical sensors integrated with our control unit;

- monitoring healthy people using sensors already present in common devices, such as a smartphone, to collect data for patient's medical history, to find out undiscovered illnesses and to handle emergency situations.

References

[1] Antonio Natali.
Introduction to QActors and QRobots
Available in the 'Laboratorio di sistemi software' lecture materials (https://137.204.107.21), 2017

[2] Getting MEAN With Mongo, Express, Angular, and Node

[3] http://118italia.it/codici-di-priorita-118/

[4] http://www.dsdtech-global.com/2017/08/sh-hc-08.html

[5] https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

[6] http://www.heltec.cn/project/wifi-kit-8/?lang=en

[7] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[8] https://pulsesensor.com/products/pulse-sensor-amped

[9] https://app.swaggerhub.com/apis/ste93/H2_PROJECT/1.0.0

[10] https://app.swaggerhub.com/apis/MargheritaPecorelli/H2-db_RESTful_API/1.0.0

[11] Massimo Conti, Natividad Martínez Madrid, Ralf Seepold, Simone Orcioni.
Mobile Networks for Biometric Data Analysis
Lecture Notes in Electrical Engineering 392 - Springer International [pp. 79-93]