Internet of Things: Report

MIo Bike

an OM2M-based Bike-sharing service

Ciardi Roberto, Degiovanni Alessandro, Falzone Giovanni

jointly M.Sc Embedded Computing Systems

University of Pisa

Sant'Anna School of Advanced Studies

November 28, 2018

Contents

1 Introduction

In this document we are going to explain the implementation of **MIoBike** system: firstly an overview of sensors and actuators for each bike is given, then the web application and its interaction with server is explained, together with the implementation of CoAP and HTTP communications; finally the Infrastructure Node and Middle Node implementations are explained.

MIoBike is an M2M-based Bike-sharing service. It has some bicycles that are equipped with some sensors that can be useful to manage the bikes or to have information about city pollution or traffic. The interaction with bikes is provided by oM2M platform, whereas a web interface is provided for user, that can unlock bikes and take a look to their utilization statistics, and for admin to check and manage the system.

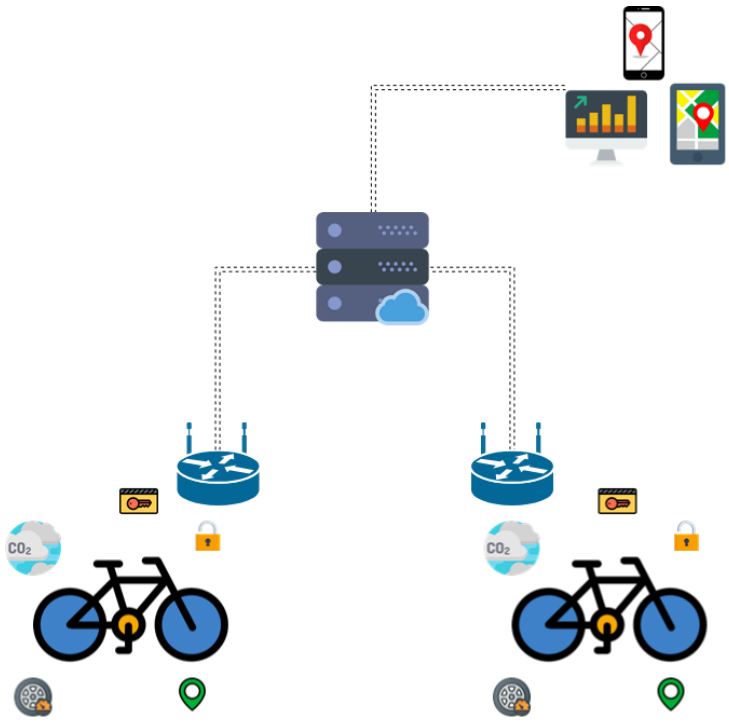


Figure 1: Architecture

The system that will be shown has two bikes (but more could be added or removed dinamycally), simulated one on a PC and the other one on a Raspberry Pi 3, each one with its simulated sensors. The web service is provided by a local server running on another PC, running the whole interface and the servlets to communicate with oM2M and the local database, also present on the server. A ZyXEL router was used to extablish a connection between the simulated bikes, the Infrastructure Node and the web server.

1.1 Technologies

Here are listed the main technologies, with a brief description, involved in system implementation:

- **OneM2M**, the platform explained during IoT course;
- **CoAP**, Constrained Application Protocol used to implement communications in the system, except for user-web server communications that use HTTP protocol;
- **Apache Tomcat Server**, a webserver used as servlet container. It was used to create web pages and Java servlets with which we could communicate by using HTTP get and post methods;
- **Java Servlet**, as said Java Servlets were implemented to have HTTP communications;
- **SSE**, Server Sent Event technology was used to have dynamic updates for webpage without a polling by client;
- **JSON**, to exchange complex and structured data we used this lightweight data-interchange format;
- **Contiki**, a lightweight open source OS, that connects tiny low-cost, low-power microcontrollers to the Internet. It is especially thought for IoT, supporting fully standard IPv6 and IPv4, along with the recent low-power wireless standards: 6lowpan, RPL, CoAP;
- **COOJA**, that allows us to implement and simulate sensor networks, running on virtual microcontrollers Zolertia Z1.

Further details will be explained later.

2 Sensors and actuators

2.1 Purpose

On a bike there are these physical sensors and actuators:

- **GPS** sensor: it returns the GPS coordinates (latitude and longitude) of the bike (for example latitude: 43.708894, longitude: 10.398309); through this sensor the administrator knows the position of the bike;
- **Odometer** sensor: it returns the total distance covered by the bike, measured through an integer in km; it can be useful for the administrator to check how much a bike has been utilized and possibly to selectively substitute some bikes;
- **Speed** sensor: it returns the actual speed of the bike, measured through an integer in km/h; also this sensor is useful for the administrator to have information about the bike usage and in certain cases also information about the traffic;
- **Air Quality** sensor: it returns an integer, representing the concentration of particulates (PM10) in the air, measured in $mg*10^{-3}/m^3$; the returned value can be useful for different purposes, since it allows to know if some areas if the city are more polluted than others, and if at a certain hour there is a peak of the pollution;
- **Temperature** sensor: it returns an integer, representing the actual temperature (measured in °C); as before, it can be used to have information about the temperature of different areas of the city, at different hours;
- **Humidity** sensor: it returns an integer, representing the actual relative humidity (measured in percentage); as before, it can be used to have information about the humidity of different areas of the city, at different hours;
- **Tyres pressure** sensor: it returns two floats, representing the front tyre pressure and the rear tyre pressure, measured in bar. The values can be used to verify if some bikes have a completely deflated tyre, or some tyres need to be inflated;
- **Locker** sensor: it is a sporadic sensor, that receives the command of a user that has finished his bike trip and wants to release the bike;
- **NFC** sensor: it is a sporadic sensor, that reads the user information when he wants to take the bike (like a smartcard reader);
- **Lock** actuator: it is an actuator, that receives a signal locking or unlocking the bike.

2.2 Sensors in Cooja

These sensors have been deployed in Cooja; each sensor is associated to a REST event resource, in this way:

- **GPS** sensor: associated to GPS resource, that handles GET requests to provide data and POST requests to receive data from a Java simulator (as explained after);
- **Odometer** sensor: associated to Odometer resource, that handles GET requests to provide data;
- **Speed** sensor: associated to Speed resource, that handles GET requests to provide data;
- **Air Quality** sensor: associated to AirQuality resource, that handles GET requests to provide data;
- **Temperature** sensor: associated to Temperature resource, that handles GET requests to provide data;
- **Humidity** sensor: associated to Humidity resource, that handles GET requests to provide data;
- **Tyres pressure** sensor: associated to TyrePressure resource, that handles GET requests to provide data;
- **Lock** actuator: associated to Lock resource, handles GET requests that specify to lock or to unlock in the url;
- **Locker** sensor and **NFC** sensor: they are not implemented as sensors: in fact, they are sporadic sensors depending on the user, and user interactions are managed through the web page, since we want to interact with the application.

Moreover, each resource is associated to a CoAP server (each server is associated to a specific mote), in this way:

- **GPS** resource, **Odometer** resource and **Speed** resource: associated to the same CoAP server, with global IPv6 address **aaaa::c30c:0:0:3**;
- **AirQuality** resource: associated to a CoAP server, with global IPv6 address **aaaa::c30c:0:0:4**;
- **Temperature** resource and **Humidity** resource: associated to the same CoAP server, with global IPv6 address **aaaa::c30c:0:0:5**;
- **TyrePressure** resource: associated to a CoAP server, with global IPv6 address **aaaa::c30c:0:0:6**;
- **Lock** resource: associated to a CoAP server, with global IPv6 address **aaaa::c30c:0:0:2**.

Of course, the mote with global address **aaaa::c30c:0:0:1** is the border router.

GPS resource, Odometer resource and Speed resource are in the same mote, because they basically work on the same data (odometer and speed values are updated knowing the GPS position), so it is convenient to have them on the same mote.

Moreover, we chose to put Temperature resource and Humidity resource on the same mote, even if they are not strictly correlated, because often temperature sensors and humidity sensors are packaged in the same physical device.

2.3 Simulation of sensor data in Cooja

Since we do not have physical sensors, we simulate data in the way now described.

2.3.1 Simulation of Air Quality, Temperature, Humidity and Tyres pressure data

Air Quality, Temperature, Humidity and Tyres pressure data are obtained choosing a mean value (statically defined) and applying a noise in a statically defined range, so that the difference between the minimum feasible value and the maximum feasible value is fixed. The noise is applied in a pseudo random like way, employing a couple of prime numbers: these number were chosen quite small (with four digits), and the sequence is periodic, but the aim of this procedure, of course, is only to simulate the behaviour of the sensors.

For all the previous sensors, a new value is generated every 10 seconds.

In particular, for each sensor we have these values:

- **Air Quality** sensor: mean value = $15*10^{-3}$ mg/m^3; minimum value = $13*10^{-3}$ mg/m^3; maximum vale = $17*10^{-3}$ mg/m^3;
- **Temperature** sensor: mean value = 20°C; minimum value = 15°C; maximum vale = 25°C;
- **Humidity** sensor: mean value = 70 percent; minimum value = 50 percent; maximum vale = 90 percent;
- **Tyres pressure** sensor: mean value = 2.5 bar; minimum value = 1 bar; maximum vale = 4 bar.

2.3.2 Simulation of GPS data

To simulate the data from GPS sensor, we do not simply choose in a random like way the latitude and the longitude; in fact, we want that the motion of the bikes can be visibile on the map by the system administrator, and that it has a reasonable evolution.

To do this, a Java class with a motion algorithm has been deployed: given an initial position, every second a function is triggered and the bike moves choosing in a random way one direction among nine (none direction, north, south, east, west, and the four intermediate direction); once chosen the direction a random step length in a statically defined range is chosen, so that the speed obtained is in the typical bike speed range. Once a direction has been chosen, the bike tends to mantain this direction up to a certain limit: the aim is that the bike does not change direction immediately, but also that it changes direction after a while.

Moreover, it is guaranteed that if the bike goes too far from the city (approaching to code-defined borders), it tends to go toward the ciy.

To notify the GPS coordinates to the mote, a POST request with the coordinates is sent to the GPS resource.

2.3.3 Simulation of Odometer and Speed data

Odometer and Speed data are obtained inside the previous Java class simply knowing the old and the new GPS coordinates; then, these data are sent in the same POST request explained before, that is sent to the GPS resource, but updates also data of Odometer and Speed resource, that are in the same mote together with GPS resource.

2.4 Simulation of sensor data in Java

If the sensor do not run on Cooja and are implemented directly in Java, the process of simulation of GPS, Odometer and Speed data is like before, while simulation of Air Quality, Temperature, Humidity and Tyres pressure data is slightly more sophisticated.

In particular, once fixed a mean value, a minimum value and a maximum value (the same of the respective Cooja sensors), the perturbation around the mean value is obtain through a sinusoidal function, with a different period depending on the sensor, and through a random function. This allows to better simulate some oscillations.

For example, for the Temperature sensor, the value strongly depends on the hour, since usually the minimum temperatures are around 4 a.m., while the maximum temperatures around 4 p.m.; thus, the sine function is designed so that the value depends on the hour of execution, and it reaches the minimum at 4 a.m., the coldest hour, and the maximum at 4 p.m., the hottest hour.

3 Web Service

3.1 Technologies

The web service was provided implementing a web project on eclipse, running on a local Tomcat server: it implements the frontend part, a user-friendly interface, and the backend part, that provides HTTP and CoAP communications.

3.1.1 Frontend

The frontend exploit some wellknown technologies as **jQuery** and **bootstrap**, to have responsive and dynamic webpages, and some others as **openLayer**, a javascript plugin to show and use maps from the opensource wiki *OpenStreetMap*, and **Chart** plugin, used to plot charts with sensors' data.

3.1.2 Backend

The backend manages the interaction with **oM2M** and with a local **SQL** database through some **Java Servlets**: some of them respond sporadically to HTTP GET or POST action, directly sent by user or by the system with **ajax**, whereas some others use SSE (server sent events) technology. Two different Java classes, *BikeManager* and *DbManager*, are used to manage CoAP request and SQL queries, to interact with databases.

3.2 Frontend

In this part of the document the functionalities of the interface will be explained. Some screenshots are shown, whereas use cases will be deepen during the exam.

The home page of the application is very simple and allows user to log in the system, as a User, if he wants to access to his data and look for a locked bike, or as Admin, if he wants to have an overview of the system and its bikes.

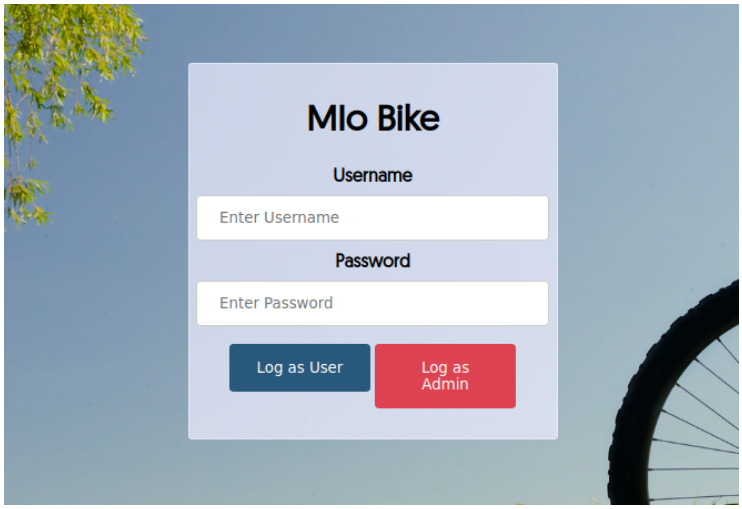


Figure 2: Home screenshot

3.2.1 User Dashboard

The User Dashboard has a sidebar where user main info are shown, and a main part where user can choose to show three different sections: Overview, Unlock Bicycle and Settings. The *Overview* section is the first one seen by user: on the sidebar are shown its profile picture, username and subscription ID, retrieved from SQL database; if he is actually using a bike the name of the bike is shown.

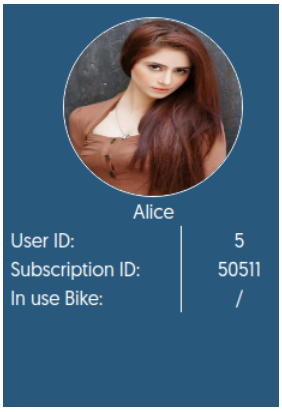


Figure 3: User sidebar

In the main part the user can see when does its subscription expires and, if he wants, he can renew its subscription and its balance would be decreased. Obviously these mean that subscriptions'info will change on SQL database. In the part below the user can see its trips statistics as, how many trips he has made, how many kilometers, its average speed and the calories he has burnt calculated based on its speed and kilometers.

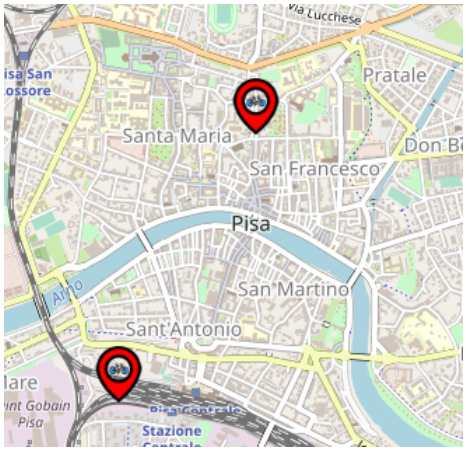


Figure 4: User map with two locked bikes

In the *Unlock Bicycle* section the User can unlock a chosen bike or release the bike he is currently using: he has a map in the main section, where locked

bikes are shown with red markers, as seen in figure 4, whereas in the sidebar he can press **unlock** to take a bike or **release** to release the bike he is using.

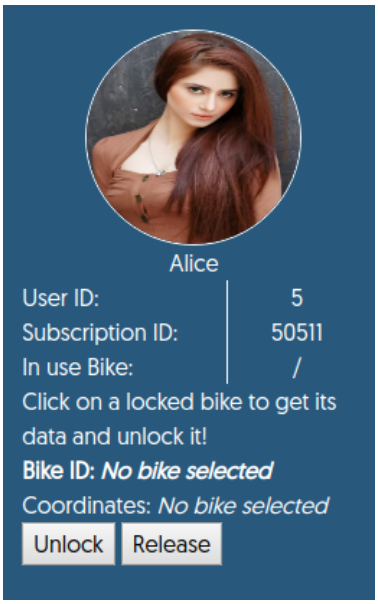


Figure 5: Sidebar before the unlock

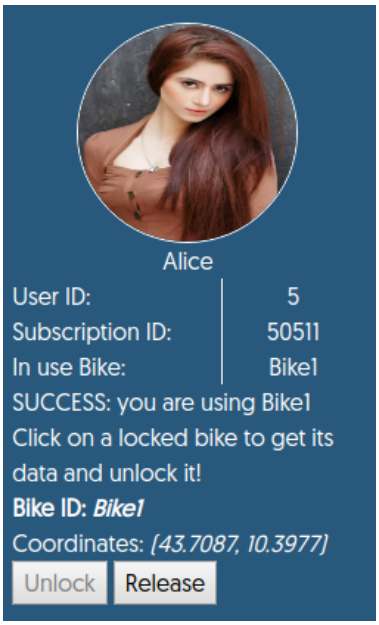


Figure 6: Sidebar after the unlock of Bike1

On the left figure we can see what the sidebar looks like before the user decides to unlock one of the bike. After he has clicked on the bike (on the map) and on the button unlock, the sidebar is shown as in the right figure, showing the user the ID of the bike he is using and disabling unlock button. On the map the user can now see his bike as a red marker, whereas the others, that are disabled, so not unlockable for the user, as gray marker.

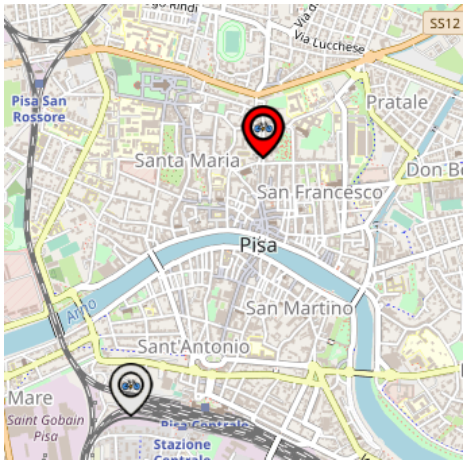


Figure 7: User map with its unlocked bike and a disabled one

In the *Settings* section the user can change some of its data with simple forms that execute SQL queries to database: he can update its weight, its email or the balance of its subscription.

3.2.2 Admin Dashboard

The Admin Dashboard is more complex than the User one and is the core of the web application: it has three sections that are Overview, Data and Users. The *Overview* section is, as before, the first one seen by admin and it has two main parts: on the left the bikes section and on the right the map. In the bike section the Admin can see the bikes of the system and their status (locked or unlocked) and the last read value of all its sensors. This data are retrieved from oM2M and requested every 10 seconds by a periodic servlet, that update this section if needed. On the map the Admin can see Locked (Red) and Unlocked (Green) bikes. Clicking on the marker of one of these bikes its information ar given on the top-left part of the window, to highlight them. Obviously the Admin can see the movements of the Unlocked bikes, that change their marker position. Further description about technologies and database interactions are given in the next chapter.

The *Data* section is a section made to have an overview of the data sensed with the bikes. Here the admin can see the average value of temperature, humidity and air quality, given by bikes and the plots of some data: in particular he can see the last value of **Humidity**, **Air Quality** and **Temperature**, retrieved by each bike, in a bar graph, this could be helpful to make statistic about city pollution and climate changes or to understand climate differences between different zones. Then the admin can see some graph about bikes'use: there is a pie chart that represents the total **distance** traveled by each bike, useful to understand if a bike needs mainteinance or to be changed with a newer one, and two bar graphs that show the tyre pressure of the front and rear tyres of each bike, giving an error message if one of the tyre has a too low or too high value of pressure. Another graph can be used by the admin to plot the last 100 values sensed by a specific sensor of one of the bikes.

The *User* section is the last section, where Admin can see all the users and their data, useful to manage the systems.

3.3 Backend

In this part of the document we'll explain the implementation of the web project and how the system can actually interact with oM2M and SQL database. In particular Servlets role, BikeManager and DbManager functions will be explained.

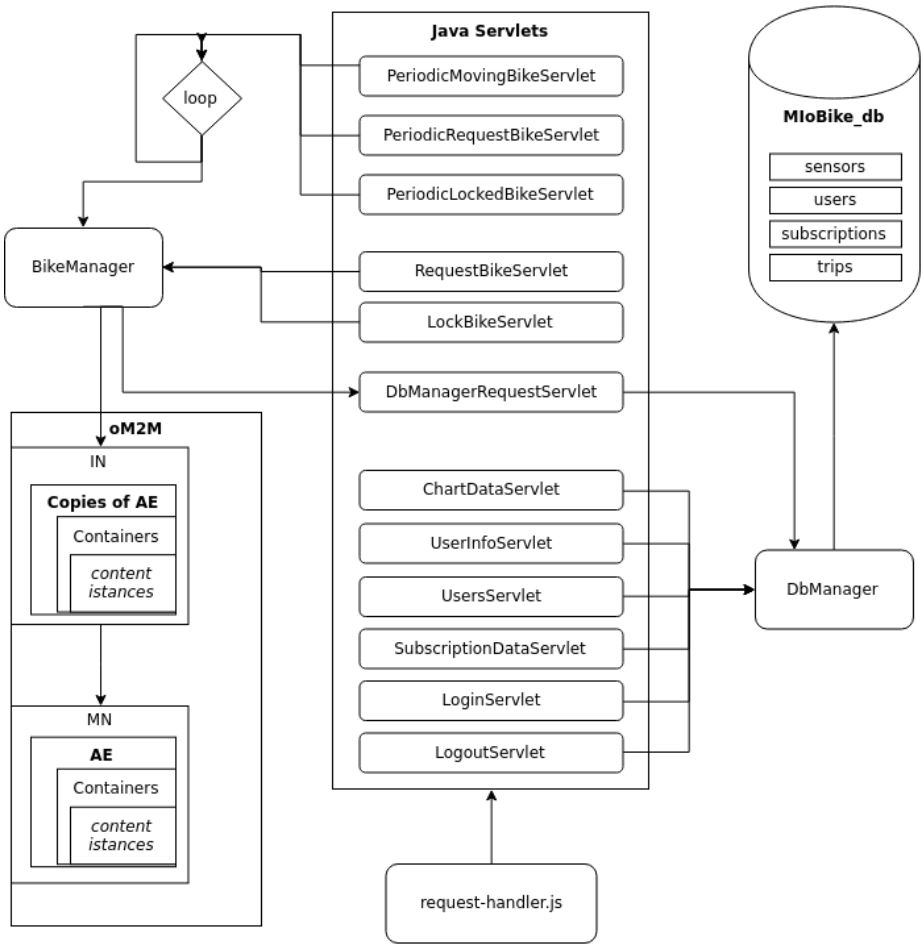


Figure 8: Backend architecture

In this figure 8 we can take a look at the architecture of the system where from requests-handler.js is responsible of most of the HTTP requests forwarded to the servlets that can interact with SQL database, through DbManager, and with oM2M through BikeManager.

3.3.1 Servlets

As anticipated the system respond to requests using Java servlets that work on Tomcat server. Most of these requests are sent asynchronously using AJAX and jQuery. The file *request-handler.js* is where most of the HTTP requests are sent using the jquery method $.get() in this format $.get("ServletName", eventual parameter function(resp)) where resp contains the response of the servlet, usually a JSON. Some other requests are not asynchronous as GET request to *LoginServlet*, that is forwarded from the form on the home page. Eventually an HTTP get is sent by BikeMonitor (we will talk about this later), to *DbManagerRequestServlet* to update **sensors** table on the database with last read values of sensors. Moreover there are three periodic servlets that actually implement a SSE service, to send periodically data from server. In particular these servlet

are threads that send content of type "event-stream" asking for data to the BikeManager. For example *PeriodicMovingBikeServlet* is a servlet that ask for unlocked bike to BM, in order to send the new position of a moving bike to the client, and allows it to update the bike position on admin map. Finally these are the implemented servlets:

- **ChartDataServlet**: that respond to a GET request, querieng data from sensors table in SQL database to plot in "Data" section of the admin;
- **DbManagerRequestServlet**, that is explained above;
- **LockBikeServlet**, that respond to a POST sent by the user to unlock or release a bike;
- **LoginServlet** and **LogoutServlet**;
- **PeriodicMovingBikeServlet**, explained above;
- **PeriodicLockedBikeServlet**, as the one before is periodic and asks for locked bikes to allow user to unlock one of them (if present);
- **PeriodicRequestBikeServlet**, a periodic servlt that request all the bike of the system, to know if a bike was added or removed;
- **RequestBikeServlet**, that respond to a GET request to have the actual Infrastructure Node configuration, with all the MiddleNodes and their containers' last content instances;
- **SubscriptionDataServlet**, that respond either to a GET or to a POST to get or update info about subscriptions from SQL db;
- **UserInfoServlet**, as the one above get or update info about user or users;
- **UsersServlet**, respond to a GET to have all informations about users.

3.3.2 DbManager

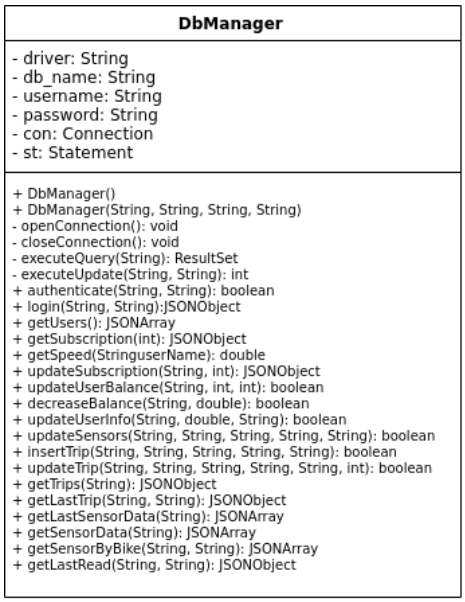


Figure 9: UML for DbManager

Firstly we must talk about the SQL database: it is named MIoBike_db and contains four tables.

- **users** table, that contains users'info as user_ID, username, password, email, avatar, weight ecc.
- **subscription** table, that contains the subscriptions, each one associated to a user_ID, with an activation date, an expiration date and a type (weekly, monthly or annual);
- **sensors** table, that contains sensors'data and it is periodically updated from oM2M. Each entry has a name, for the name of the sensor, a value, that is the value read, a date and a user, if it is associated to a bike used by an user;
- **trips** table, that contains trips' data for each user and it is update when a user unlock or release a bicycle. In particular each entry contains a name that can be 'Start_Trip', when the user unlock the bicycle, than it is updated to 'End_Trip', a value field indicates the kilometers traveled and a time value that indicates (in milliseconds) the duration of that trip. Naturally each trip has a username field.

As anticipated the DbManager manages the connection to SQL database and the execution of query to insert and select data. An istance of this type of object is created by each servlet that need to send request to the database: the DbManager must open a connection with the database, build and execute the queries and then close the connection, after having send the response to the servlet. Some of the functions implemented in DbManager are:

- *login* function, that allow user to login, checking if he has the permission to log as admin and initializing session's data for that user;
- some *get* functions, that retrieve data from database with SELECT queries (as *getUsers(), getSubscription(int id), getSpeed(String userName)* ecc.);
- some *update* functions, that update the tables with INSERT or UPDATE queries (as *updateSubscription(String userId, int balance), updateUserBalance(String user_id, int subs_id, int balance)* ecc.).

Eventual exceptions can be thrown by these function and must be catched by the servlet that executes the function.

3.3.3 BikeManager

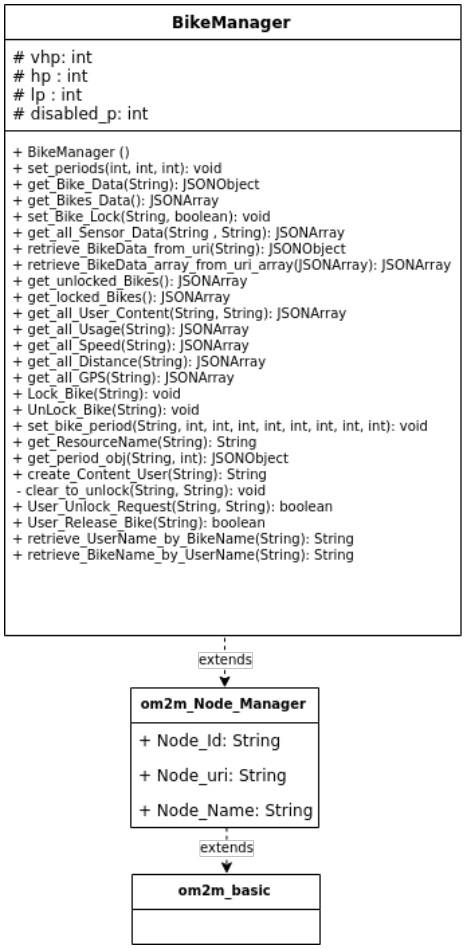


Figure 10: UML for BikeManager

As said before the BikeManager represents an intermediate node between oM2M and the web application. It allows to request data to the Infrastructure Node managing all the coap requests. As for the DbManager, the servlets can

create and use an instance of BikeManager to retrieve the needed informations. Basically all the informations about bikes and sensors could be retrieved from oM2M, because each MiddleNode(bicycle) maintains the content instances of each container (sensor). On the contrary Infrastructure Node maintains the last ten read contentinstances for each container. Obviously retrieving all data from each MiddleNode wasn't a scalable approach, so we decided to save them, as explained, in the local database with GET request to DbManagerRequestServlet. However for the other requests that are meant to retrieve the last read data, the CoAP requests could be sent to the IN, and that is what is done using the BikeManager.

As we saw in figure 10 BikeManager extends the class *om2m_Node_Manager* that extends again *om2m_basic*, and we'll start from BikeManager going down to explain their functionalities. BikeManager functions allow to retrieve the informations from the Infrastructure Node, using the labels associated with containers or application entities, the subclasses will then build the uri to send the CoAP requests. For example to get the last read value of all the sensors of a specific bicycle we can look for the container that have as label "BikeName" the specified name. So BikeManager implements a function **get_Bike_Data(String BikeName)** that forwards a call to the function **get_all_Container(String [] labels)** that is implemented in om2m_Node_Manager and will interact to om2m_basic to send a CoAP GET to request all the containers that has that specified label, identifying that bike containers.

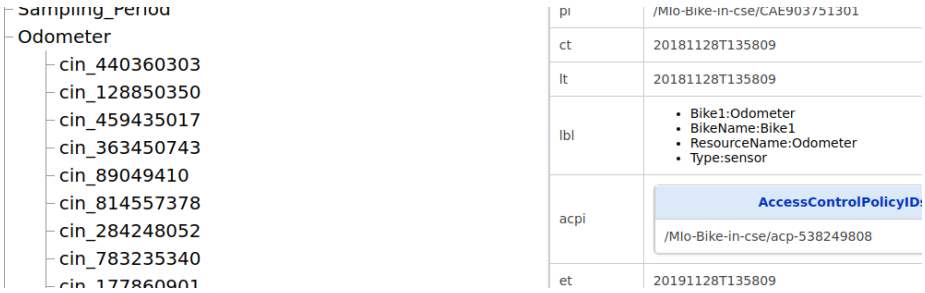


Figure 11: Container labels

In this figure 11 we can see the labels of each container, so that BikeManager can filter the containers by the value of one of this label. Some labels are used also for Application Entities as *BikeStatus* that can be set to locked or unlocked and allows to retrieve all the locked or unlocked bikes.

Finally some of BikeManager functions that will call the subfunctions to send the requests are:

- *get_Bike_Data(String BikeName)* explained before and *get_Bikes_Data()*;
- *get_unlocked_Bikes()* and *get_locked_Bikes()* that, as anticipated, allows to filter the containers by label *BikeStatus*;
- *User_Unlock_Request(String UserId, String BikeName)* function, that check if the bike BikeName is unlockable by user UserId and, if it is possible, it updates the labels and set the lock value to true;

- *User_Release_Bike(String BikeName)* that is basically the alter-ego of the one before.

Going down to the *om2m_Node_Manager* it implements the functions that forward the BikeManager request to create the CoAP requests and implements some functions to update Application Entities and containers label.

Finally we must talk about the class *om2m_basic* that implements the functions for CoAP requests:

- *get_request(String uri, String access_credentials, String UriQueries)*, used to send a CoAP GET to a coap client whose uri is specified, as to send a Discovery Request;
- *post_request(String uri, JSONObject json_payload, int opt_num, int opt_val, String access_credentials)*, used to send a CoAP POST as to create Application Entities, Containers, content instances and subscriptions;
- *put_request(String uri, JSONObject json_payload, int opt_num, int opt_val, String access_credentials)*, user do send CoAP PUT requests.

4 Bike Manager Monitor

The Bike Manager Monitor extends the functionalities of the standard Bike Manager, his goal is to manage directly the requests and the data coming from the bottom part of the architecture, the requests that can come from each bike are:

- **Unlock** Request using the NFC Badge
- **Release** Bike Request coming from the Locker

In order to do that, there is a Notification Manager running a CoAP Monitor, that is subscribed to each bike and to the relative container from which we are interested to receive notifications.

Another useful functionality of this entity is to forward the informations directly to the Web Service, in order to do that the Bike Manager creates a thread worker which has the goal to send an http GET request for each specific event:

- **Unlock** request using the NFC Badge is forwarded to the Web Service to keep track of each trip for each user.
- **Release** request using the locker on the bike, the request is forwarded to the Web Service to keep track for the end of a trip for a specified user in order to calculate useful information on the distance and the cost of the trip
- **Sensors' data** coming from each bike, useful for bike's maintenance and other informations

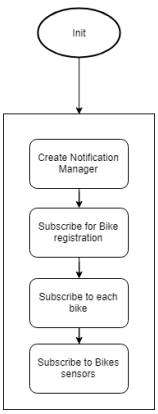
4.1 Bike Manager Monitor Initialization

During the initialization, the Notification Manager is created and then subscribed to the root of the IN, in order to receive a proper notification when a new AE Bike is created. When it receives a notification relative to the creation of a new Bike AE, the Bike Manager is subscribed to the AE relative to that Bike in order to receive the notification for the creation of each container.

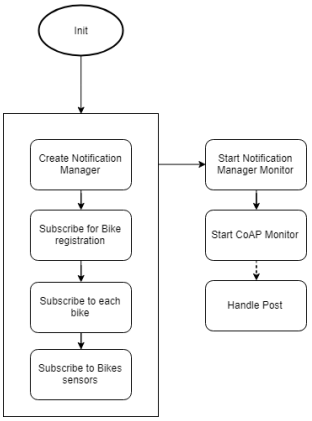
OM2M CSE Resource Tree

http://127.0.0.1:8080/~/MIo-Bike-in-cse/cnt-221416479

- MIo-Bike-in-name
  - acp_admin
  - System
  - Bike1
    - Sampling_Period
    - User
    - NFC
      - BikeManager_NM
    - Locker
      - BikeManager_NM
    - AirQuality
      - BikeManager_NM
    - GPS
    - Temperature
      - BikeManager_NM
    - Humidity
      - BikeManager_NM
    - TyrePressure
    - Odometer
      - BikeManager_NM
    - Speed
      - BikeManager_NM
    - Lock
    - BikeManager_NM
  - Bike1-mn-name
  - BikeManager_NM
  - MIo-Bike_NM



(a) Bike Manager Monitor Initialization phase



(b) Bike Manager Monitor Notification Manager creation

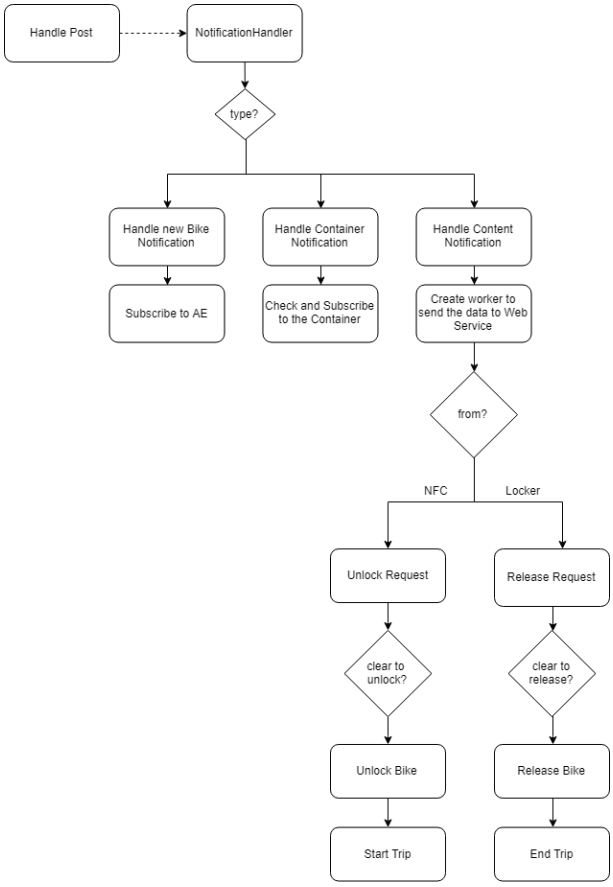


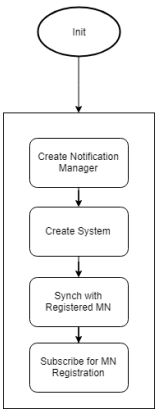
Figure 13: BM Notification handler

5 Infrastructure Node

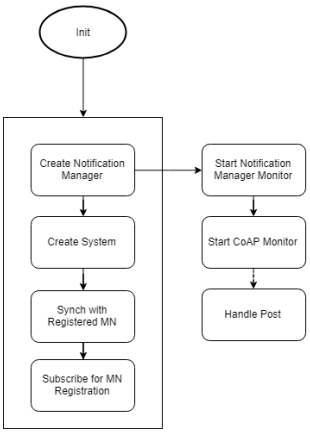
The goal of the IN is to realize the main structure of the Bike Sharing service, managing the information's relative to the actuators and the management of the system with the the nodes at the bottom of the architecture.

5.1 IN Application in a nutshell

The code used to implement the Application is composed of two parts, the initialization phase, and the runtime event based work-flow. During the initialization phase, first of all a Notification Manager is created in order to create the subscription and receive notifications, the Notification Manager is implemented as a CoAP Server.



(a) IN Application Initialization phase



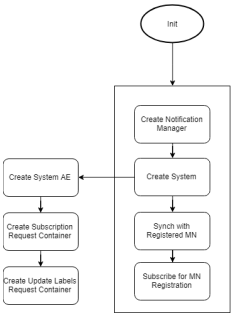
(b) IN Application Notification Manager

Then an Application Entity is created in order to receive additional requests relative to te initialization of the other nodes or relative to an update request coming from other nodes.

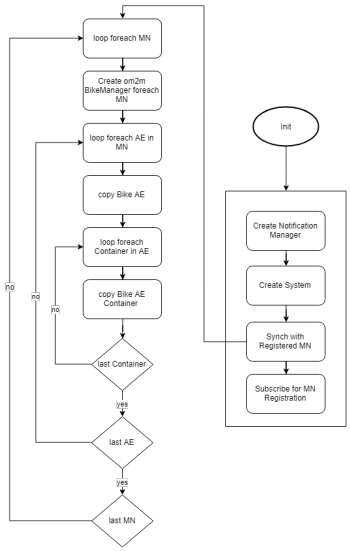
Once the IN is initialized will react to the incoming notifications with a proper handler depending on the notification received, as shown in the 16. In case of a new MN registered the IN will create a Bike Manager associated to that node, this object is a **om2m_Node_Manager** class and is used each time the IN needs to operate on that node, then the IN is subscribed on the registered MN's root to receive a notification when the Application Entity is created on the MN.

As soon as the MN creates the Application Entity relative to the Bike, the IN will receive a notification for this event and reacts creating a copy of that AE and will subscribe to the AE on the Bike node to receive a notification for the Containers' creation on that node.

Once the Bike application starts creating the containers, the IN receives the notification and reacts to that event, subscribing itself on that container if this container is associated to a sensor; in the case of an actuator is the Bike that is subscribed to the container on the IN in order to automatically send



(a) IN System AE



(b) IN Application Synch phase

the notification when a new action is required for that node. This mechanism allows the IN to receive the incoming data from the sensors and have a fresh copy of the Bike.

Starting from this situation the IN starts to receive a notification for each content created in the containers in which is subscribed, and to recap, they are:

- Root on the IN itself
- Root on the Bike node
- AE on the Bike node
- Container associated to a Sensor
- Container in the System AE

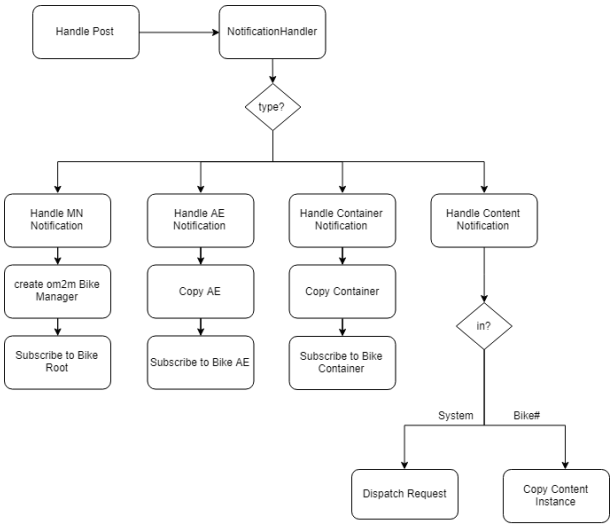


Figure 16: IN Notification handler

5.1.1 Resource Tree Evolution during initialization

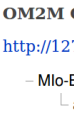
The figure 17a shows the initial state of the Resource Tree in the IN, without MN node registered.

Running the IN Application a subscription is registered to the root of the IN, as shown in 17b, this is done in order to receive a proper notification when a Bike is turned on and the relative MN is registered on the IN.

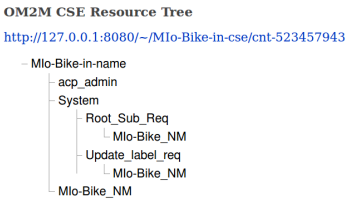
As can be seen from 17b, the IN Tree contains also an Application Entity *System* containing two Containers, this particular AE is used to notify the system for particular events, as:

- **Root_Sub_req** when a MN crashes and reboot the IN cannot know the new registration because is already there, so the MN can create a new content in the in order to notify the IN to subscribe to the rebooted MN, in this way the MN App can continue with the initialization.
- **Update_label_req** when a Bike is unlocked, the IN must now to update the labels because these are used to filter the status of the Bike and also the User that is using the bike, those labels are updated by the IN App also in the MN node.

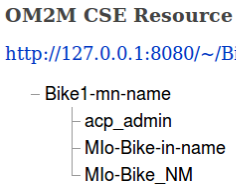
After the registration of one MN the IN Application running on the IN Node automatically receives a notification for this event and will subscribe to the new MN to receive notifications on the root of the MN, as shown in in 17c. In the same way the IN will subscribe for each Bike and then for each container relative to the sensors in order to receive the data coming from the bottom part of the architecture, as shown in 5.1.1.



(a) Resource Tree at the beginning

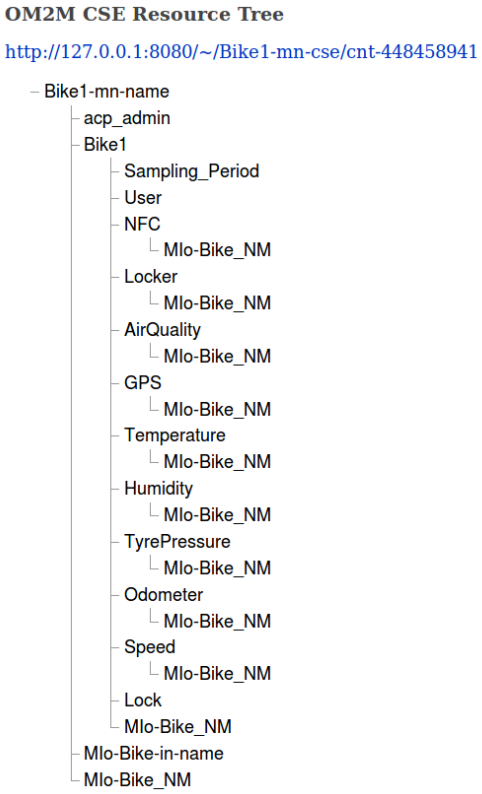


(b) Resource Tree after the initialization of IN App



(c) IN subscribe to each MN root

As can be seen in the figures, the IN creates the AE System with Containers and then is subscribed to the container inside, once the MN is turned on and the registration on IN is completed, the IN is subscribed to the root of MN and then to the AE relative to the Bike and then the Containers as explained before. At this point the MN finished the initialization phase and starts to create the contents relative to the data coming from the sensors. The IN completes the *handshake* needed to start the Bike and can start to operate on the data, coming from the bike and acting on the actuator.



5.2 Full IN Application Work-flow

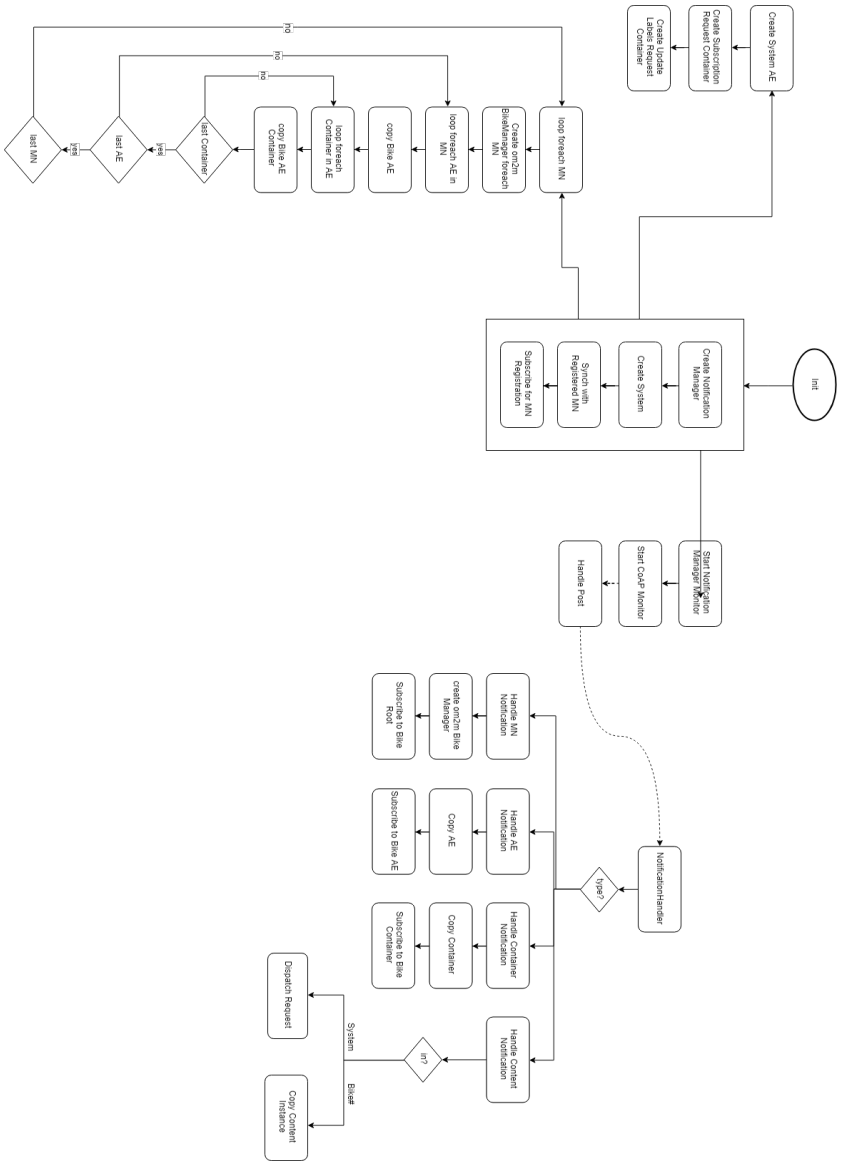


Figure 18: IN application work-flow description

6 Middle Node

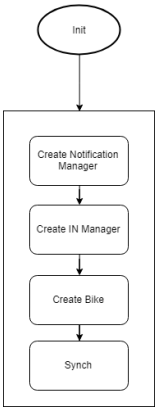
Each Middle Node is located on one single bike, the main goals of the MN Application running on the bike are:

- creation of the main structure relative to the bike representation
- retrieving of data from the bike sensors and create a content on the bike structure
- changes of bikes' status with the data coming from the top of the architecture

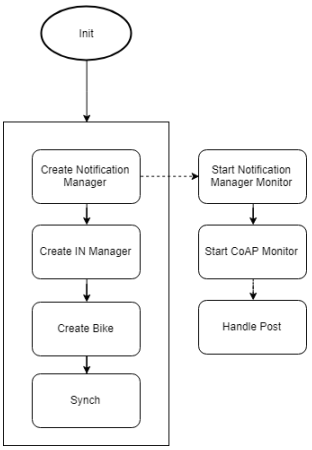
6.1 MN Application in a nutshell

The code used to implement the Application is composed of three parts, the **initialization** phase, the **runtime event** based Notification Manager and a set of **Threads** acting on the sensors and actuators.

During the initialization phase, first of all a Notification Manager is created in order to create the subscription and receive notifications coming from the top part of the architecture, the Notification Manager is implemented as a CoAP Monitor.



(a) Initialization code representation



(b) Creation of the Notification Manager

When a notification due to a subscription comes, an appropriate handler is executed in order to react to that event as seen in 20. The MN is subscribed to containers related to the actuator and some virtual resources as we will see soon, so it will receive just notifications for content instances published on that containers.

Then an active wait for the IN starts, waiting for the subscription to the root on the MN, when the IN is subscribed the initialization continue creating the structure that represents the bike's resources: first of all an Application Entity related to the Bike is created and an active wait starts in order to know if the IN received the notification and is subscribed to that Resource.

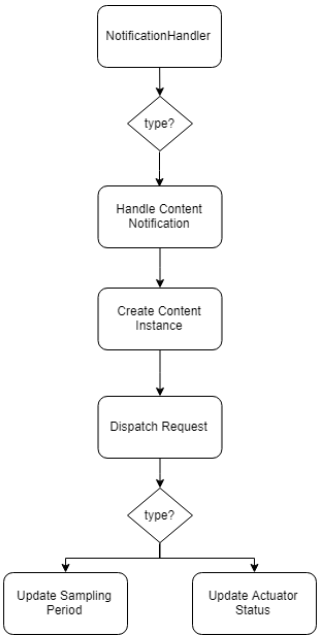


Figure 20: Bike Notification handler

Then the initialization continue creating a Resource manager and a Container under the Application Entity; for each Container related to a sensor an active wait starts in order to know if the IN is subscribed to that container.

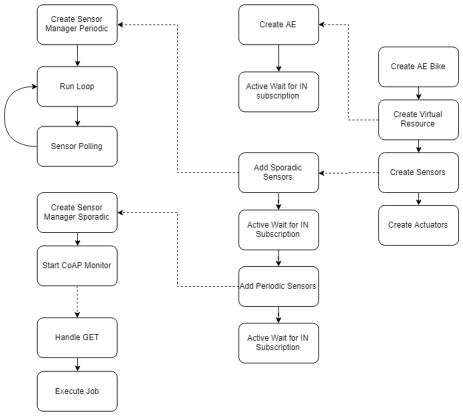


Figure 21: Bike creation of Resources

The resources on the bike can be divided in three different categories:

- Resources related to sensors
- Resource related to the actuator
- Resources related to virtual resources

Once all the containers are created a **synch** is done in order to update the status of the Bike Structure with the same on the IN, this is done to be sure

that in case of a reboot of the Bike node, the Node can recover the old status and continue working in the same way as before.

In order to manage the change of frequency and the actual user that is using the bike, two different containers associated with that information are created in the resource tree, each time a content is created under that container the bike receives a notification and react in the proper way.

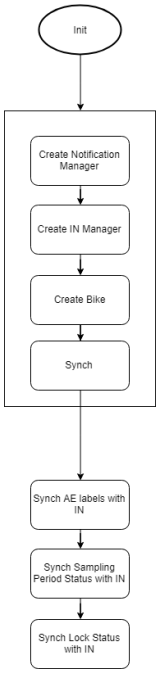


Figure 22: Bike Synch with IN

6.2 Resource Manager

As seen we have three different types of resource, for each of them there is a specialized Resource Manager:

- Periodic Sensor Manager
- Sporadic Sensor Manager
- Periodic Actuator Manager

For each of them there is a Thread, in the case of periodic sensors or the actuator it is a periodic thread running at a specified frequency, chosen by default at the beginning and then modified by the Bike Manager, for a specified situation (e.g during the motion we are not interested on the Temperature and Humidity but we are more interested to receive the information related to the motion with higher frequency).

The main task of these workers is to retrieve data from the sensors at the specified frequency, encapsulate it in a field of a specified JSON format and push it on the related Container in the MN Resource Tree.

For the sporadic resources as the NFC reader and the locker, due to their nature, there is a thread implementing a CoAP Server that is listening at a specified port for the incoming sporadic requests coming from the associated sensors.

The format used to encapsulate the data is the following, with Date format that is in GMT+0 and the data on the field "ResourceData" that comes directly from the sensor:

```
{
"ResourceData":{
"format":"mge-3/m^3",
"value":"16"
},
"ResourceName":"AirQuality",
"BikeName":"Bike1",
"Date":"2018.11.28_21:41:58"
}
```

6.3 Full MN Application Work-flow

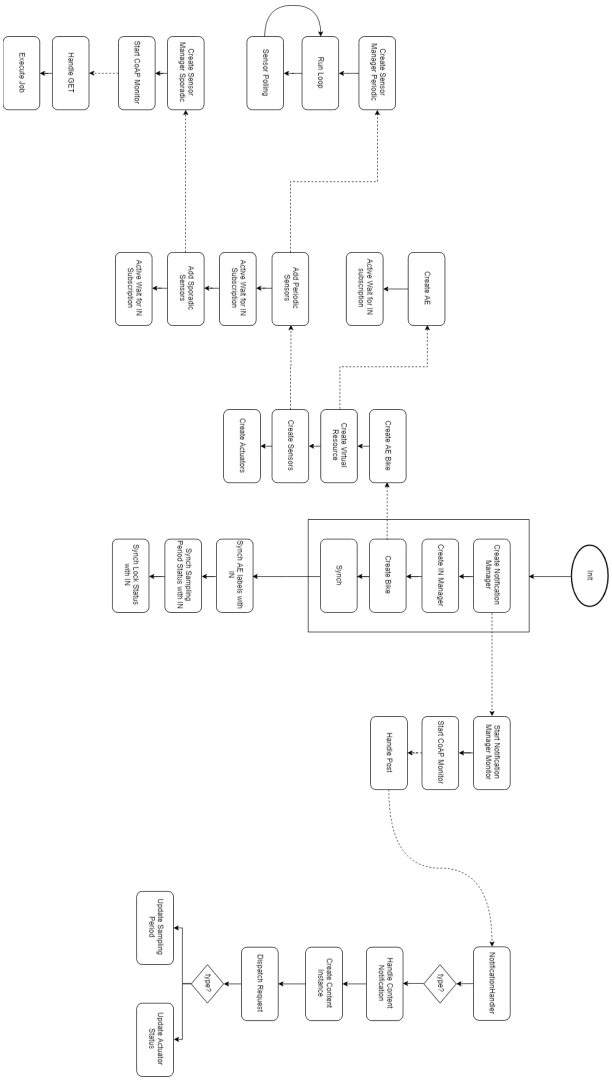


Figure 23: MN application work-flow description

7 Conclusions

In this document we've seen the architecture and the implementation of the IoT bike-sharing system **MIoBike**. We've seen the implementation of bikes sensors and actuators, simulated thanks to COOJA and that would allow to receive data about bicycle usage and environment: this allow to use this Bike Sharing service, not only for users' movements across a city, but also to implement useful services for traffic and pollution management. A simple web application was shown, provided to allow simple users' interactions and have a user-friendly service. Then particular details were given about IN and MN implementations, that explained the work made to build these system, highlighting its powers and allowing to have a complete overview of the architecture and the implementation.

At the end of the day, we've shown the first version of this user-friendly bike sharing useful for people and for smart cities.

Appendices

A User Manual

The entire project can be found on the GitHub page.

A.1 MIoBike_common

contains the Constants and common libraries used by the other projects as the **om2m_Node_Manager.java**, **om2m_basic.java**. There are also two different files for the system's configuration: Constants.java where can be chosen the address of each resource, in order to run the system simulating the resources on cooja the variables must be set in this way:

```
public static final boolean SENSORTAG = false;
public static final boolean COOJA = true;
public static final String NFC_URI = NFC_URI_LOCAL;
public static final String LOCKER_URI = LOCKER_URI_LOCAL;
public static final String LOCK_URI = LOCK_URI_LOCAL;

public static final String AIRQUALITY_URI = AIRQUALITY_URI_COOJA;
public static final String GPS_URI = GPS_URI_COOJA;
public static final String ODOMETER_URI = ODOMETER_URI_COOJA;
public static final String SPEEDOMETER_URI = SPEEDOMETER_URI_COOJA;
public static final String TEMPERATURE_URI = TEMPERATURE_URI_COOJA;
public static final String HUMIDITY_URI = HUMIDITY_URI_COOJA;
public static final String TYRE_PRESSURE_URI = TYRE_PRESSURE_URI_COOJA;

public static final boolean PHY_SIMULATOR_MOTION = true;
public static boolean PHY_SIMULATOR_LOCKER = true;
public static boolean PHY_SIMULATOR_NFC = true;
public static boolean PHY_SIMULATOR_TEMPERATURE = false;
```

```
public static boolean PHY_SIMULATOR_HUMIDITY = false;
public static final boolean PHY_SIMULATOR_AIRQUALITY = false;
public static final boolean PHY_SIMULATOR_TYRE_PRESSURE = false;
```

otherwise to execute the java version of the resource in java:

```
public static final boolean SENSORTAG = false;
public static final boolean COOJA = false;

public static final String NFC_URI = NFC_URI_LOCAL;
public static final String LOCKER_URI = LOCKER_URI_LOCAL;
public static final String LOCK_URI = LOCK_URI_LOCAL;
public static final String AIRQUALITY_URI = AIRQUALITY_URI_LOCAL;
public static final String GPS_URI = GPS_URI_LOCAL;
public static final String ODOMETER_URI = ODOMETER_URI_LOCAL;
public static final String SPEEDOMETER_URI = SPEEDOMETER_URI_LOCAL;
public static final String TEMPERATURE_URI = TEMPERATURE_URI_LOCAL;
public static final String HUMIDITY_URI = HUMIDITY_URI_LOCAL;
public static final String TYRE_PRESSURE_URI = TYRE_PRESSURE_URI_LOCAL;

public static final boolean PHY_SIMULATOR_MOTION = true;
public static boolean PHY_SIMULATOR_LOCKER = true;
public static boolean PHY_SIMULATOR_NFC = true;
public static boolean PHY_SIMULATOR_TEMPERATURE = true;
public static boolean PHY_SIMULATOR_HUMIDITY = true;
public static final boolean PHY_SIMULATOR_AIRQUALITY = true;
public static final boolean PHY_SIMULATOR_TYRE_PRESSURE = true;
```

of course also each address and port must be adapted in the proper way.

A.2 MIoBike_BikeManager

Contains the implementation of the Bike Manager and the Bike Manager Monitor, it is sufficient to compile and execute the jar file in the target directory.

A.3 MIoBike_IN_App

Contains the Application running in the IN, first of all the IN node must be configured and started, in the github there is also the configuration for the IN Node, in order to execute the IN Application it is sufficient to compile and run the jar in the target directory.

A.4 MIoBike_MN_App

Contains the Application running in the MN, first of all the MN node must be configured and started, in the github there are also the configuration for the MN node, in order to execute the MN Application it is sufficient to compile and run the jar in the target directory, must be specified the address and Bike Name as parameters.

A.5 MIoBike_Bike_Simulator

Contain the main applicattion running the simulator for the bike, once fixed the constants in **Constants.java** in the **MIoBike_common project**, the Simulator must be compiled and executed with the Bike Name as parameter

A.6 MIoBike_Phy_Simulators

contains the Java implementation of the motion generator, the code is used to generate the physical motion data of the bike, in the Constants.java in **MIoBike_common** it is possible to configure the system in order to send that data to the cooja implementation or the java implementation of the motion's related resources.

A.7 MIoBike_WebUI

contains the implementation of the web project: the web interface under directory WebContent, the Java implementation of servlets (subpackage .web.servlets) and the Java implementation of DbManager.