(FP7 614100)

D3.4 Network Management

29.10. 2015

Published by the IMPReSS Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme and the Conselho Nacional de Desenvolvimento Científico e Tecnológico
Objective ICT-2013.10.2 EU-Brazil research and development Cooperation

Document control page

Document file: IMPRESS D3.4_for_internal_review_ISMB_prefinal.docx
Document version: 2
Document owner: Matti Eteläperä (VTT)

Work package: WP3 Resource Abstraction and IoT Communication Infrastructure
Task: T3.4 Network Management
Deliverable type: P

Document status: ☒ approved by the document owner for internal review
☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Matti Eteläperä (VTT)	07/08/2015	First draft
0.2	Matti Eteläperä (VTT)	23/10/2015	ToC edits and first inputs
0.3	Jussi Kiljander (VTT)	23/10/2015	Executive summary and conclusions
0.4	Tuomo Mattila (VTT)	25/10/2015	IoT gateway software management
1.0	Matti Eteläperä (VTT)	29/10/2015	Final version after internal review

Internal review history:

Reviewed by	Date	Summary of comments
Davide Conzon	28/10/2015	Approved with minor comments
Peeter Kool	28/10/2015	Approved with minor comments

Index:

1. Executive summary 4
2. Introduction 5
2.1 Purpose, context and scope of this deliverable 5
2.2 IoT gateway life-cycle 6
3. IoT gateway deployment management 8
3.1 Introduction 8
3.2 Architecture components 8
3.2.1 MQTT Broker 9
3.2.2 Gateway device and software 9
3.2.3 Proximity Manager 10
3.2.4 System Knowledge Base 11
3.2.4.1 SKB REST interface 11
3.2.4.2 SKB UI web service 11
3.3 Select sequence diagrams 18
3.4 IMPReSS test case implementation 20
3.4.1 Bluetooth Low Energy (BLE) sensor-tags 21
4. IoT gateway software management 23
4.1 IoT gateway software management overview 23
4.2 Host-level version controller 24
4.3 Container-level version controller 24
4.4 Security considerations 27
5. Results and discussion 28
6. References 29
Appendix A 30

1. Executive summary

This deliverable describes a novel network management infrastructure for the Internet of Things, which has been developed in the T3.4 - Network Management. The key features of the IoT network management infrastructure are the following:

1. Remote management of IoT gateway software.
2. Low-effort deployment of IoT sensors and actuators.
3. Zero-effort management of IoT device context during runtime.

The goal of the remote software management is to decrease the costs related to management of IoT gateway software both by 1) making it possible to manage the software remotely so that there is no need for maintenance personnel to visit the premises and by 2) making it as easy as possible to package the various libraries and modules into a single software package. The remote software management infrastructure for IoT gateways is implemented on top of Docker virtualization tool.

The goal in the low-effort deployment of IoT devices and runtime management of IoT device context is to decrease the costs related to IoT sensor and actuator deployment by automating the processes related to context creation and maintenance. That is, the aim is to generate the associations between IoT Resources (i.e., sensors and actuators) and IoT Entities (i.e., object of interest that are monitored and controlled by the IoT Resources) automatically. In particular, the goal has been to create automatically these associations the room level. To this end, we assigned IoT gateways to each room in the IoT system that we want to monitor and equip the other IoT Entities with active tags, which advertise their unique ID over the air. The key idea is to use signal strength information of sensor, actuator and active communication in order to associate these devices into certain rooms and IoT Entities.

The approach is suitable for any wireless communication technology. In the proof-of-concept (PoC) implementation, we utilized Bluetooth Low Energy (BLE) radio. With BLE it was possible to achieve automatic association of IoT Resource and IoT Entities on the room level in most cases. However, the variance in the signal strength was also quite high so problems may occur if two IoT Room GWs are located very close to each other and separated only by thin wall.

2. Introduction

This document describes the various network management related technologies developed during IMPReSS project. It is a stand-alone document describing the key components of each technology and also provides links to the source codes.

2.1 Purpose, context and scope of this deliverable

The goal of the network management (Gonçalves et al 2012) in the traditional Internet is to discover, monitor and manage Internet devices such as routers, switches and servers. Simple Network Management Protocol (SNMP) (Case et al 1990) is the de-facto protocol for this purpose in IP based networks.

In IoT the problems of the network management are similar to the traditional Internet. The main difference is the devices that need to be managed. In IoT the challenge is to manage IoT infrastructure devices such as IoT gateways, sensors and actuators that are core parts of the IoT network. These are the most critical and important devices, because they need to be physically located in the IoT system premises. This means that it will be expensive if it is not possible to manage them remotely. MQTT (IBM & Eurotech 2010) and RESTful protocols such as Constrained Application Protocol (CoAP) (Shelby et al 2013) and HTTP (Fielding et al 1999) are the de facto protocols used in IoT and therefore it is natural to use the same protocols also for performing network management activities. This is especially true for resource constrained devices as memory can be saved by eliminating the need to support additional network management protocols such as the SNMP for example.

IoT network management is a crucial bottleneck and problem in highly distributes IoT deployments. The problem arises from the fact that the deployment and maintenance cost of systems rises in a linear fashion with the number of connected devices. We aim to cut this network management effort, by automating the most labour intensive tasks – namely the association between the physical deployment of the IoT gateway and the services hosting system related information.

The IoT network management infrastructure developed in the T3.4 and described in this deliverable consists of two independent solutions: Remote IoT gateway software management and zero-effort IoT network context management. The zero-effort IoT network context management framework provides means to create and manage the context (i.e., IoT Resources, IoT Entities and associations between them) of IoT system. The framework is described in more detail in the chapter 3. To illustrate how the framework works in practise, we apply it in a simulated scenario using the map of the Federal University of Pernambuco (UFPE) campus. The IoT gateway software management framework provides means for simple packaging and remote management of IoT gateway software. It is described in more detail in the chapter 4.

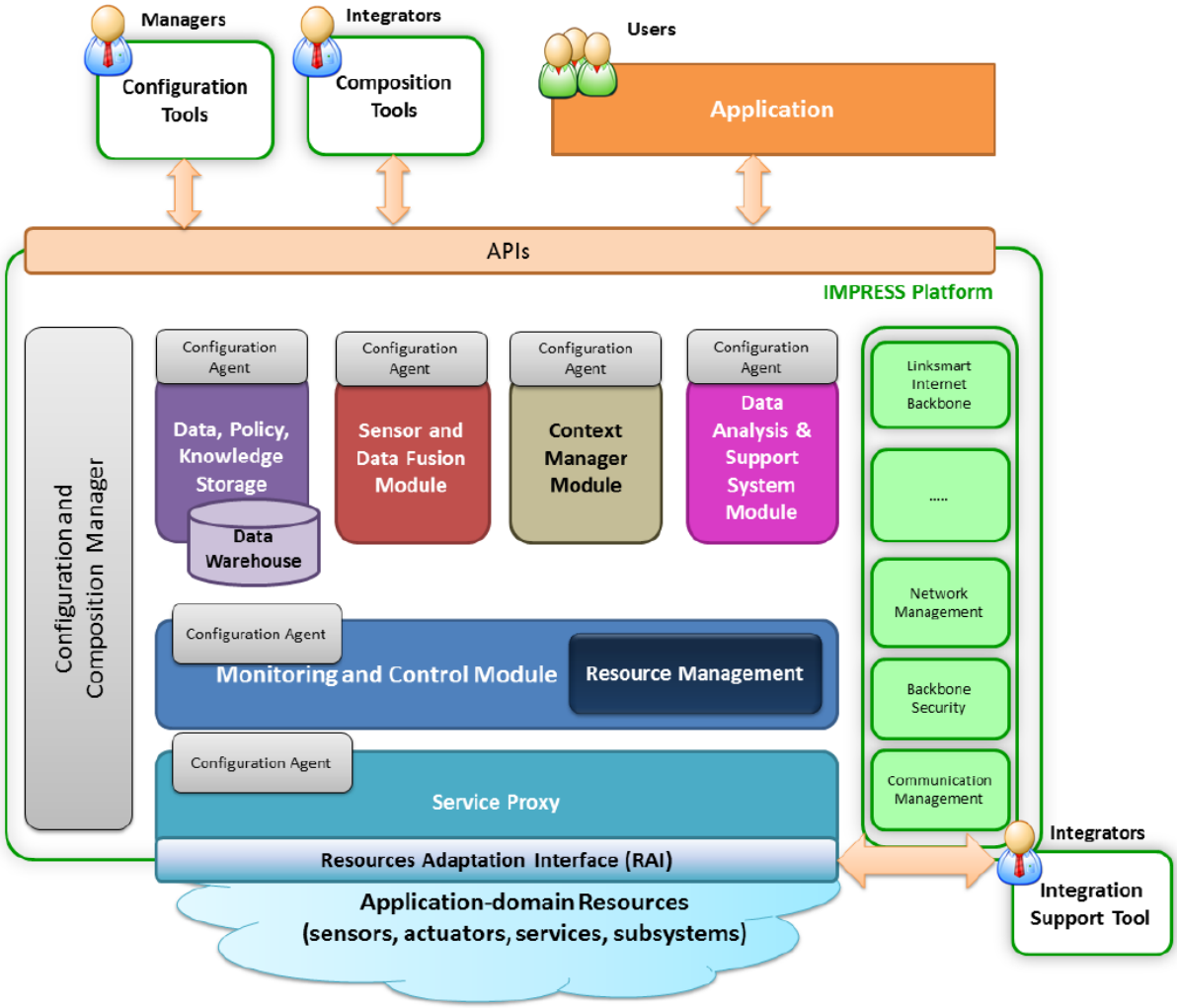


Figure 1. IMPReSS architecture.

Figure 1 presents the original IMPReSS architecture and how network management block (green on the right) is positioned. Network management work in IMPReSS is seen as an integration support action for providing necessary tools for deploying large scale IoT systems rather than a functional block in the architecture.

2.2 IoT gateway life-cycle

In Figure 2 we present the IoT gateway and system lifecycle used in producing tools described in this deliverable. The green dot depicts the start point, leading to starting of the back-end web services required by the gateway component and the process of deploying and running the IoT gateways themselves.

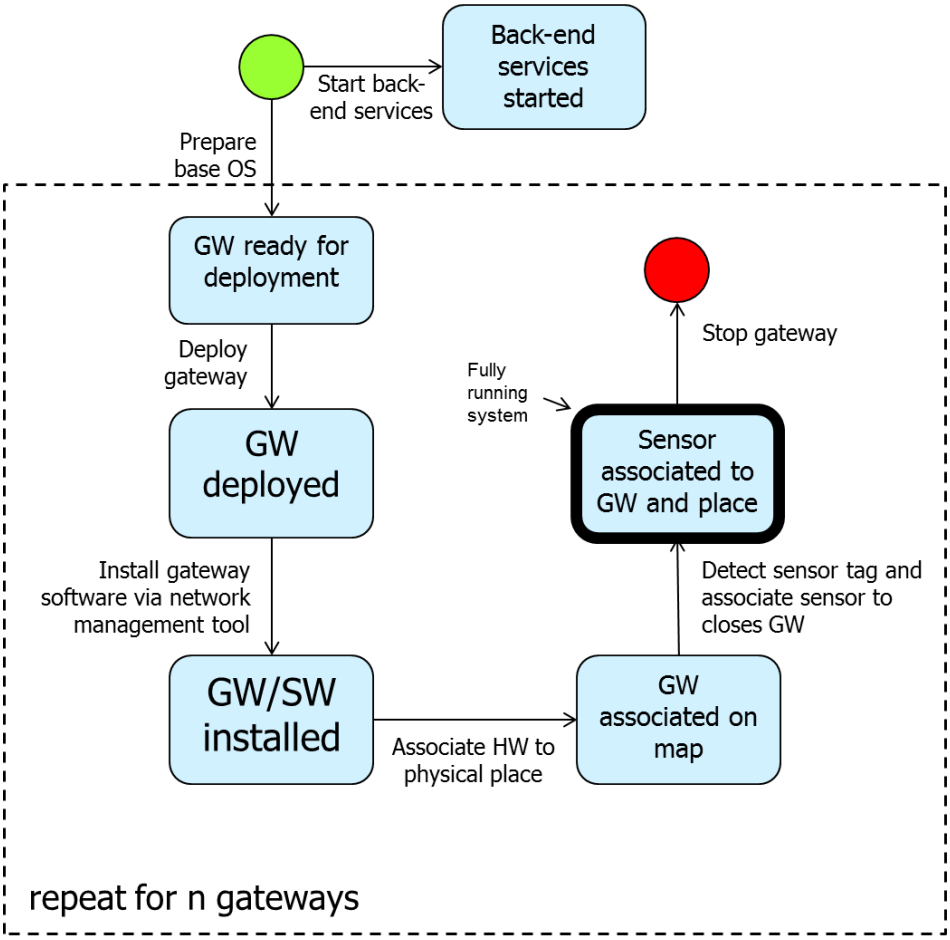


Figure 2. IoT gateway lifecycle.

The back-end services presented in the figure involve both deployment related services as well as run-time management related services, presented in sections 3 and 4 of this document. Also the gateway specific cycle (drawn inside the box in Figure 2) involves steps which can be mapped to either type of management. They are also presented in more detail in the following sections.

3. IoT gateway deployment management

3.1 Introduction

In this section, we present a system for automatically associating IoT devices to a back-end system for both deployment and run-time management.

The system is designed to work particularly well in IoT deployment with multiple gateway devices connected to static or moving wireless sensor or tag devices. Agnostic to the radio interface used, the sensor-tags can be automatically associated to the nearby gateways, or even positioned via sensor strength triangulation. To summarize, the system has two key functionalities to aid IoT gateway and system deployment:

1. Zero effort deployment of IoT gateway devices
2. Automatic association of sensor-tags to gateways

These two functionalities result in an automatic system for associating physical objects of interest, gateway devices, sensor-tags (and measurement values) and static physical places to each other.

3.2 Architecture components

An overview of the IMPReSS deployment management architecture is presented in Figure 3.

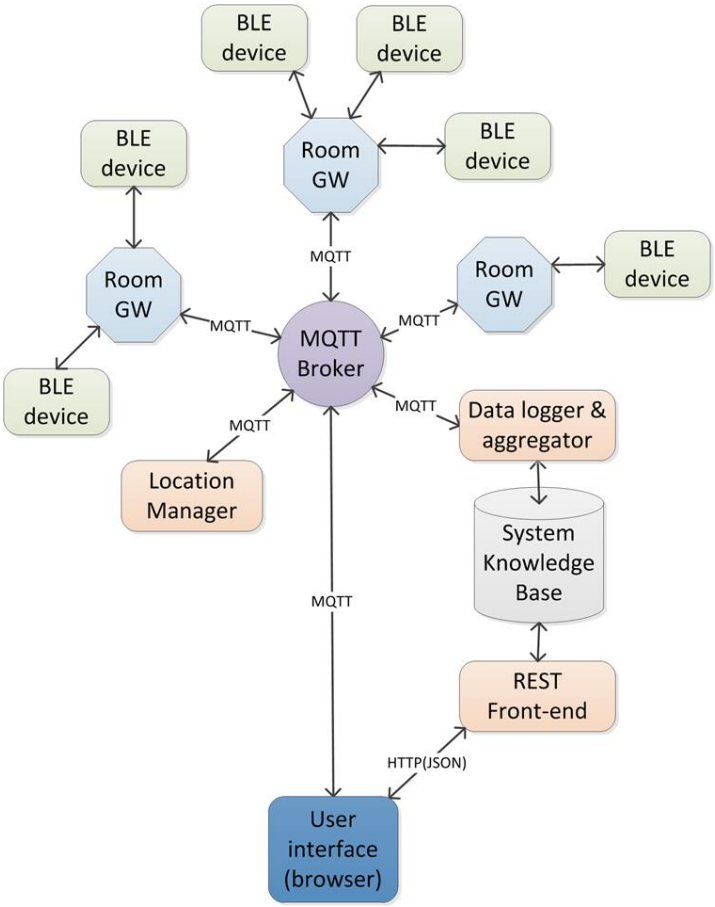


Figure 3. Architecture overview.

In Table 1 the mapping of functional components to the physical architecture is presented.

Table 1. Mapping of functional blocks to physical architecture.

Component	Deployed at	Description
MQTT Broker	Local network or Internet	Has to be accessible from all gateways
Location manager	Local network or Internet	Has to be accessible from all gateways
User interface	Browser based, connected either to local network or Internet.	Has to be accessible from all gateways and MQTT broker

The general architecture created in IMPReSS is presented in more detail in the following subsections.

3.2.1 MQTT Broker

MQTT broker is the main communication medium in the system. The star network formed around the broker allows clients to communicate with each other by publishing and subscribing to topics.

Mosquitto MQTT broker version 1.4 was used in this setup. The list of the opened ports is presented in Table 2. Along the default 1883 port, the broker topics can be accessed using WebSockets. This feature is needed for browser-based interfacing.

Table 2. MQTT broker ports in the IMPReSS scenario.

Protocol	port
MQTT	1883
MQTT over WebSockets	8083

3.2.2 Gateway device and software

As gateway devices we used RaspberryPi (RPI) 2 single board computers (Figure 4) connected to a local network via Ethernet. The RPIs are equipped with radio interfaces to receive data from wireless sensor-tags. Our approach does not limit the type of radio interface used in the tags, it can be for example WIFI, Bluetooth, Ultra-wideband (UWB) or Z-Wave.

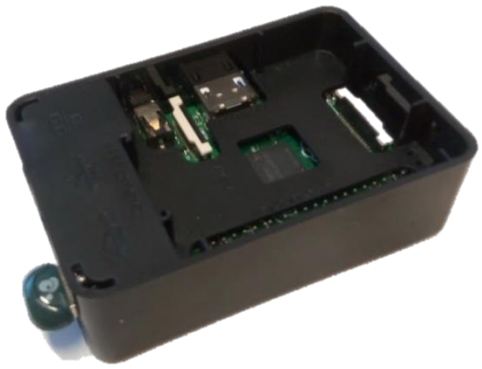


Figure 4. Raspberry Pi 2 gateway and Iomega BLE dongle.

Table 2 presents MQTT operations performed by the RPI, after receiving information from tags. The gateways also send a keep-alive message every 10 seconds. This to notify the system that the gateway is operational even if there is no tag signal received.

PUBLISH to topic	Payload JSON	Note
/gateways/<GW_ID>/tags/<TAG_ID>	{gateway_id: GW_ID, tag_id: TAG_ID, RSSI: rssi, ADV_DATA : advData}	
/gateways/<GW_ID>	{gateway_id: MAC_ID}	Keep-alive sent every 10 seconds

3.2.3 Proximity Manager

Proximity Manager (PM) is responsible for associating a sensor tag to the closest gateway device based on the received signal strength. The gateway devices send to the proximity manager all RSSI signal strength values. PM calculates the signal strength received by each of the sensor devices for each gateway device and averages them in a time window of a predefined length. In the tests a window length of 10 to 30 seconds was used.

SUBSCRIBE to topic	Payload	Note
/gateways/<GW_ID>/tags/<TAG_ID>	{gateway_id: GW_ID, tag_id: TAG_ID, RSSI: rssi, ADV_DATA : advData}	Read RSSI value and TAG_ID

PUBLISH to topic	Payload	Note
/associations/<TAG_ID>	{gateway_id : GW_ID, tag : TAG_ID, RSSI: rssi, ADV_DATA: advData}	Generates

3.2.4 System Knowledge Base

System Knowledge Base (SKB) provides a RESTful interface for accessing information in a request-response manner. SKB also includes a template driven engine for creating a web service for monitoring and managing the IoT deployment.

3.2.4.1 SKB REST interface

SKB REST interface resources are described in Table 3. A different base URI is given to each system deployment and collections for places, objects, tags and gateways exist.

Table 3. Back-end REST interface.

Resource	GET
http://<base_uri>	Returns links to subcollection in JSON format.
http://<base_uri>/places	Return the places in the system. E.g. [{place_Id : 1342342}, …]
http://<base_uri>/places/<place_id>	Returns representation of the room and links to the resources. E.g. {id : 1342342, name : E355, type : room, links : /objects }
http://<base_uri>/objects/	Returns a list of objects in the whole system. E.g. {links : [{link : /<object_id> }, { link : /<object_id_2> }, …]}
http://<base_uri>/objects/<object_id>	{id : 1342342, name : tempr sensor, type : sensor, links : /attributes }
http://<base_uri>/tags	{attributes : [{ vlink : /temperature, unit : "Celsius"}, {link : /humidity, unit : "%"}, …]}
http://<base_uri>/tags/<tag_id>	Return latest value in format {value : "22.5"}
http://<base_uri>/gateways	Return the places in the system. E.g. [{gateway_Id : 1342342}, …]
http://<base_uri>/gateways/<gateway_id>	Returns representation of the gateways and links resources. E.g. {id : 1342342, name : E355, type : gateway, links : /objects }

In Appendix A an example of a HTTP GET to a gateway collection URI is presented.

3.2.4.2 SKB UI web service

SKB UI includes the following sub-pages, or views:

General view (REST)

- Shows the amount of tags, gateways, places and objects registered in the system.
- See Figure 5.

Figure 5. UI general view.

Map view (REST + MQTT)

- Tool for creating new places on the map
  - User specifies a central point of a room
- Tool for placing new gateways on map by drag and drop
  - Automatically associates gateways to new places if moved
- Tool for linking map/floorplan to geographical coordinates
- Automatically visualizes the association between sensor-tags and closest gateways using a force-driven layout
  - Shows RSSI strengths of the associations
- Automatically shows objects linked to gateways
  - Clicking the object node opens Object view

In Figure 6 the map view UI is presented. The pin circles are gateways positioned inside rooms (E384 etc.) in a floorplan. In the lower section of the figure a tag (white circle) is associated to the gateway names E366. "Jussi" is an object associated to the tag and the label is thus shown in the figure. The user can move gateways to new position to match the actual deployment. New, not associated gateways appear in the map view at a fixed location and are easy to associate to a place in the floorplan.

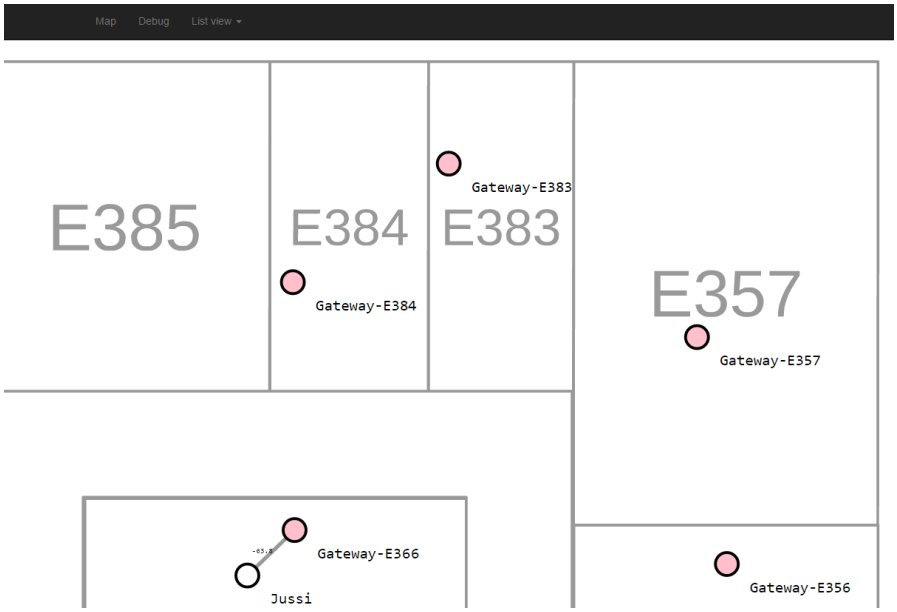
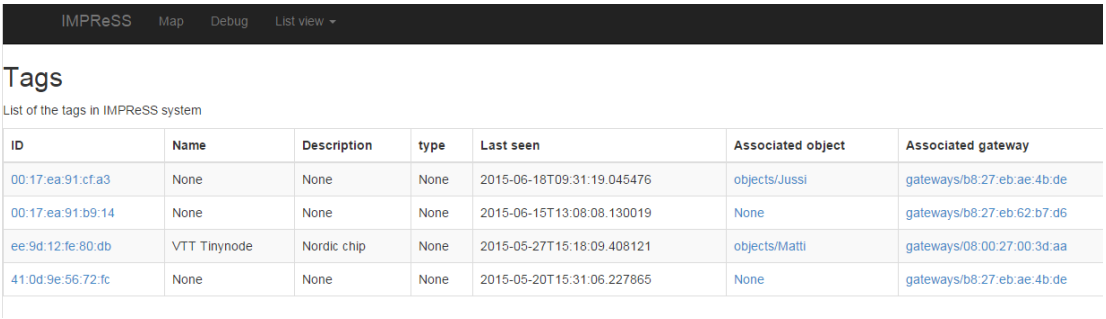


Figure 6. Map view.

Collection views

- Tags collection view (REST)
  - Information shown: ID, description, last seen, associated gateways and objects
  - See Figure 7.



ID	Name	Description	type	Last seen	Associated object	Associated gateway
00:17:ea:91:cf:a3	None	None	None	2015-06-18T09:31:19.045476	objects/Jussi	gateways/b8:27:eb:ae:4b:de
00:17:ea:91:b9:14	None	None	None	2015-06-15T13:08:08.130019	None	gateways/b8:27:eb:62:b7:d6
ee:9d:12:fe:80:db	VTT Tinynode	Nordic chip	None	2015-05-27T15:18:09.408121	objects/Matti	gateways/08:00:27:00:3d:aa
41:0d:9e:56:72:fc	None	None	None	2015-05-20T15:31:06.227865	None	gateways/b8:27:eb:ae:4b:de

Figure 7. Tags collection view.

- Gateways collection view (REST)
  - Information shown: ID, description, last seen, associated place
  - See Figure 8.

ID	Name	Description	type	Last seen	Associated place
b8:27:eb:cd:90:5b	Gateway-E357	add description	None	2015-06-25T15:42:05.443558	places/E357
b8:27:eb:69:1a:61	Gateway-E357	add description	None	2015-06-25T15:42:04.323222	places/E357
b8:27:eb:32:58:29	Gateway-E357	add description	None	2015-06-25T15:42:00.989635	places/E357
b8:27:eb:ae:4b:de	Gateway-E357	add description	None	2015-06-25T15:42:00.330857	places/E357

Figure 8. Gateways collection view.

- Objects collection view (REST)
  - User can create a new object
  - See Figure 9.

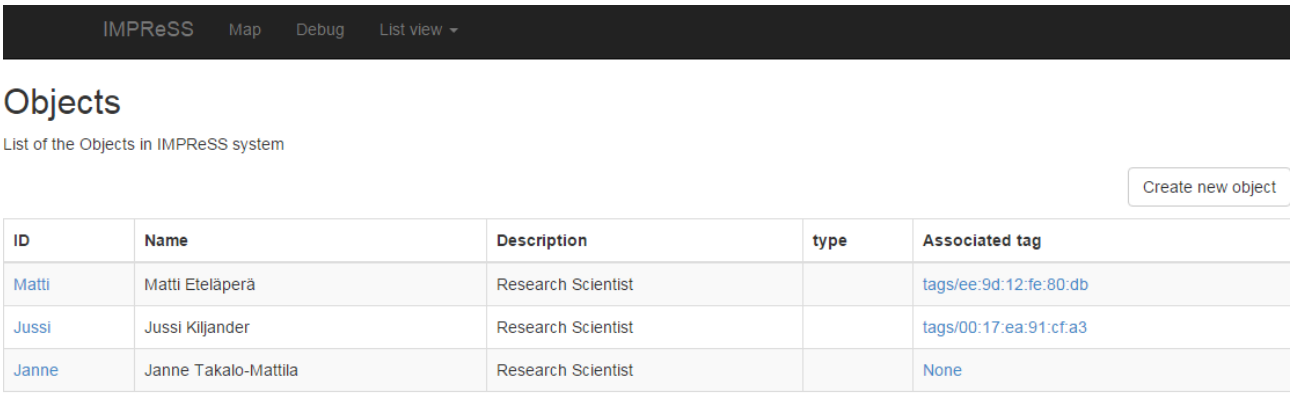


Figure 9. Object collection view.

- Places collection view (REST)
  - Information shown: ID, description, type, location, associated gateway
  - User can create a new place
  - See Figure 10.

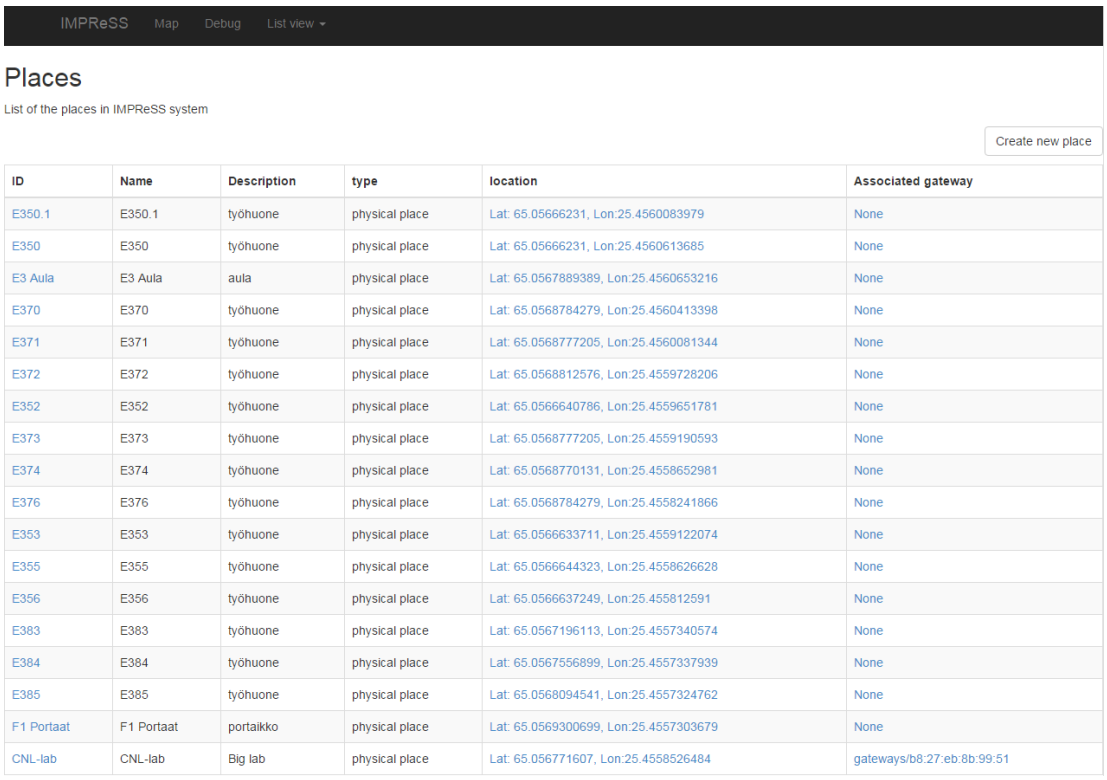
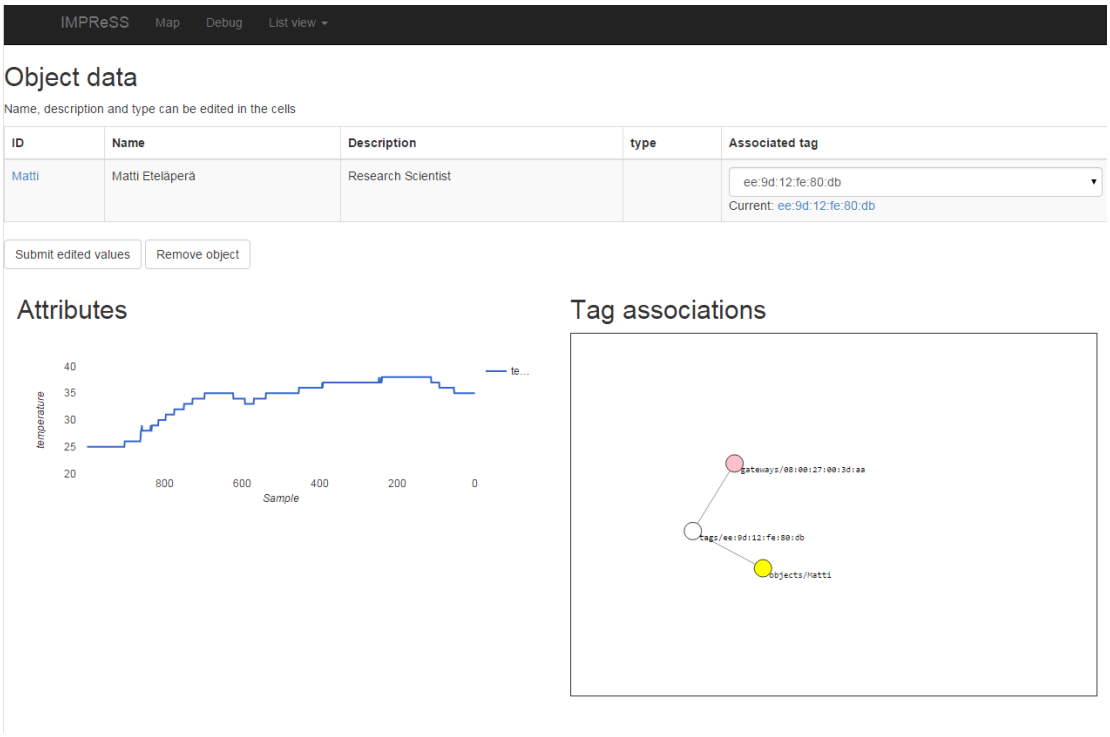


Figure 10. Places collection view.

Individual views

- Object resource view (REST)

- Shows associations between gateways, tags and objects
- Shows the historical graph of the previous sensor attributes
- See Figure 11.



- Figure 11. Object resource view.

- Tag resource view (REST)
  - Shows associations between gateways, tags and objects
  - Shows the historical graph of the previous sensor attributes
  - See Figure 12.

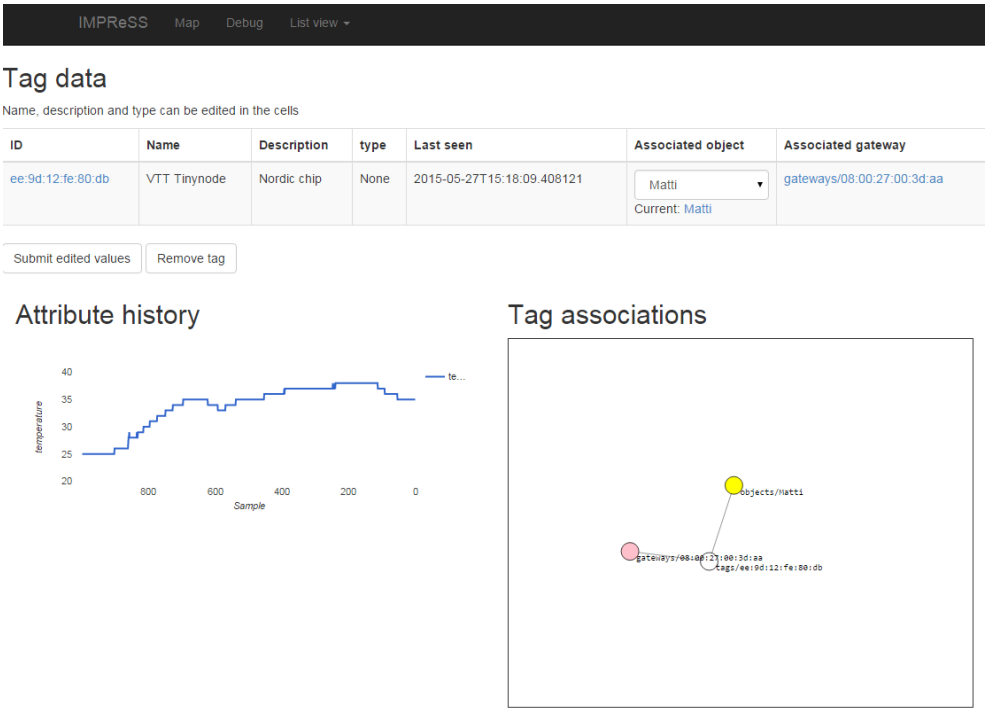


Figure 12. Tag resource view.

- Gateway resource view (REST)
  - Edit individual gateway attributes
  - See Figure 13.

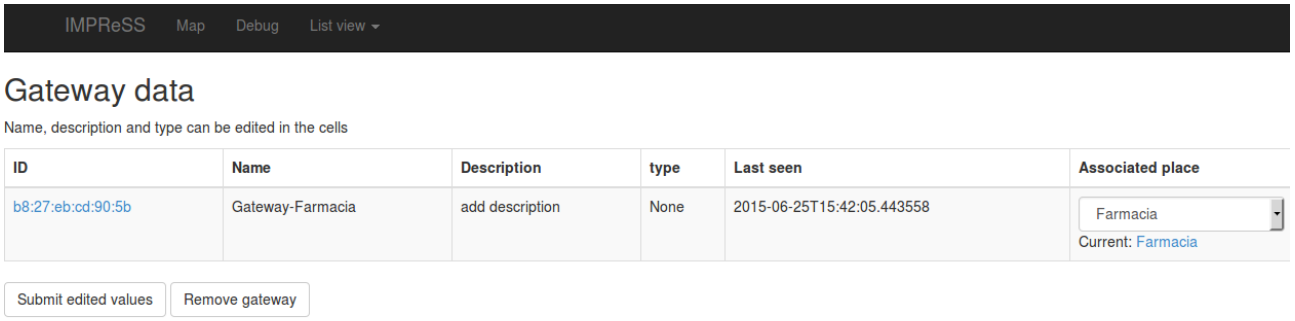


Figure 13. Gateway resource view.

- Place resource view (REST)
  - See Figure 14.

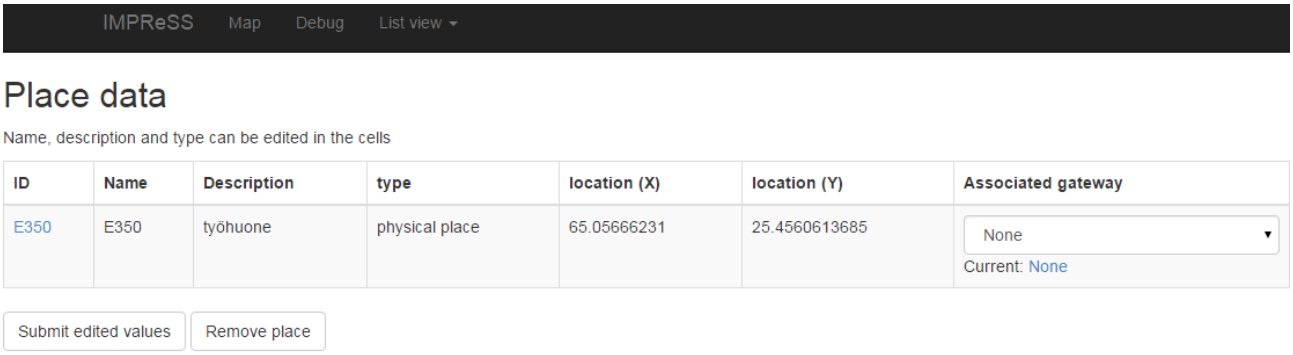


Figure 14. Place resource view.

Map view MQTT topics

The following tables describe the MQTT subscribe and publish interface used by the map view.

Map view includes setup modes for new floorplans, or "maps". The user can calibrate the x-y coordinates used by the UI to real latitudes and longitudes by defining two points on the map and inputting their respective geographical coordinates. This information is published to the MQTT broker with a "retain" flag so that the browser can receive the latitude and longitude values after refreshing the page. The x-y positions of gateway nodes are also published with the retain flag set.

Table 4. Map view MQTT publish topics.

PUBLISH to topics	Retain flag	Payload JSON	Note
/latlons/<MAP_URN>	1	{map_uri: MAP_URI, latlons: {lat_1: latlon_1.lat, lon_1: latlon_1.lon, lat_2 : lat, lon_2 : lon}, xys : { x_1 : xys.x, y_1: xys.y, x_2 : x, y_2 : y }	Maps d3.js xy-coordinates to geographical coordinates to each map used.
/pos_map/<GW_ID>	1	{gateway_id: MAC_ID, coordinates:[LAT, LON]}	Loads saves gateway geographical positions.

Table 5. Map view MQTT subscribe topics.

SUBSCRIBE to topic	Payload JSON	Note
/gateways/<GW_ID>	{gateway_id: MAC_ID}	Keep-alive
/associations/<TAG_ID>	{gateway_id : GW_ID, tag : TAG_ID, RSSI: rssi, ADV_DATA: advData}	Gateway <-> tag association
/pos_map/<GW_ID>	{gateway_id: MAC_ID, coordinates:[LAT, LON]}	Loads gateway positions.
/latlons/<MAP_URN>	{map_uri: MAP_URI, latlons: {lat_1: latlon_1.lat, lon_1: latlon_1.lon, lat_2 : lat, lon_2 : lon}, xys : { x_1 : xys.x, y_1: xys.y, x_2 : x, y_2 : y }	Maps d3.js xy-coordinates to geographical coordinates to each map used.

3.3 Select sequence diagrams

In this section, we present some of the key communication patterns involved in the IMPReSS zero-effort deployment architecture.

Figure 15 shows how the Proximity Manager receives signal strength values from each nearby tag and determines the closest tag based on this information. The calculation is an average of time t seconds (30s by default) and the evaluation is performed for each received signal strength message. The strongest signal value is translated into an association message, linking a tag to the gateway. This message is used by the UI and System Knowledge Base components.

All communication after the Gateway go through the MQTT broker, but it is not drawn in the sequence diagram for simplicity's sake.

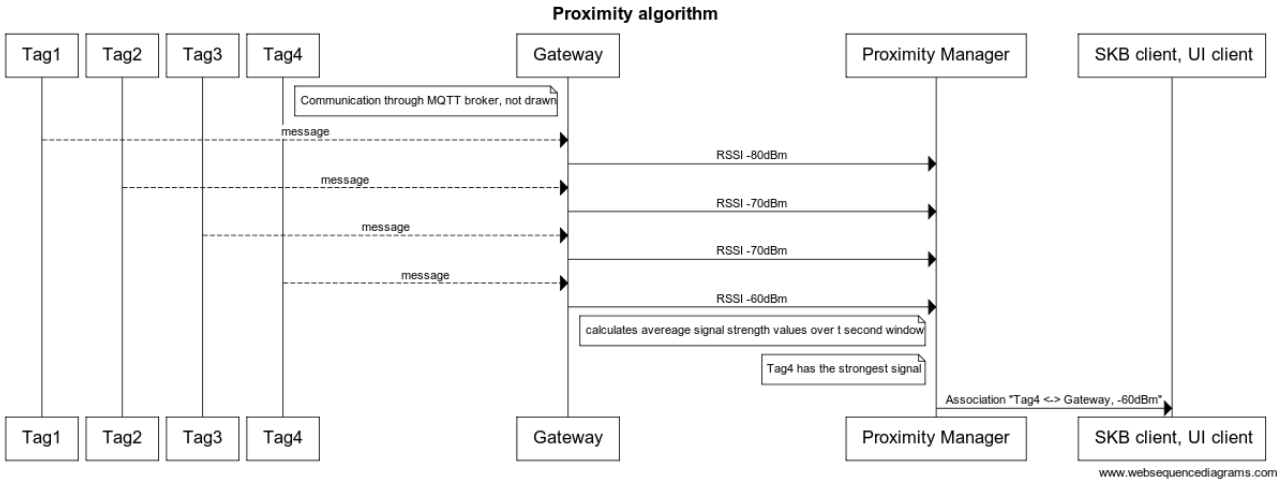


Figure 15. Proximity algorithm sequence diagram (MQTT omitted).

Figure 16 presents the information flow from the sensor-tag to the SKB. The message sent by the tag is presented in a bit more detail. In this example the tag sends TAG_ID and temperature values. The Gateway forwards this message to the MQTT broker to a topic matching the gateway and tag, to which SKB is subscribed via a wildcard. After SKB gets notification of the message it updates a number of RESTful resources and thus makes the tag data available.

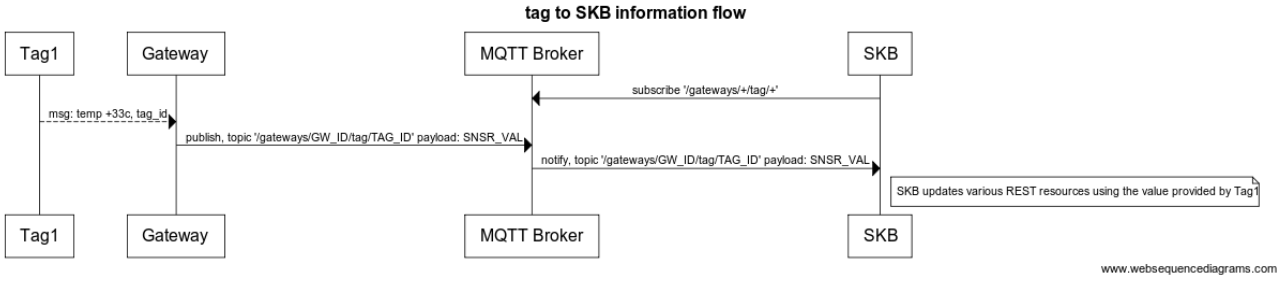


Figure 16. Information flow from sensor-tag to System Knowledge Base.

Figure 17. Object association Tag to System Knowledge Base.

Figure 18 shows the different interactions among the Web UI Map view and the other components. Both MQTT and REST API are used in order to both visualize events in real time and data stored in the SKB. The figure highlights the complexity and the distributed nature of the system.

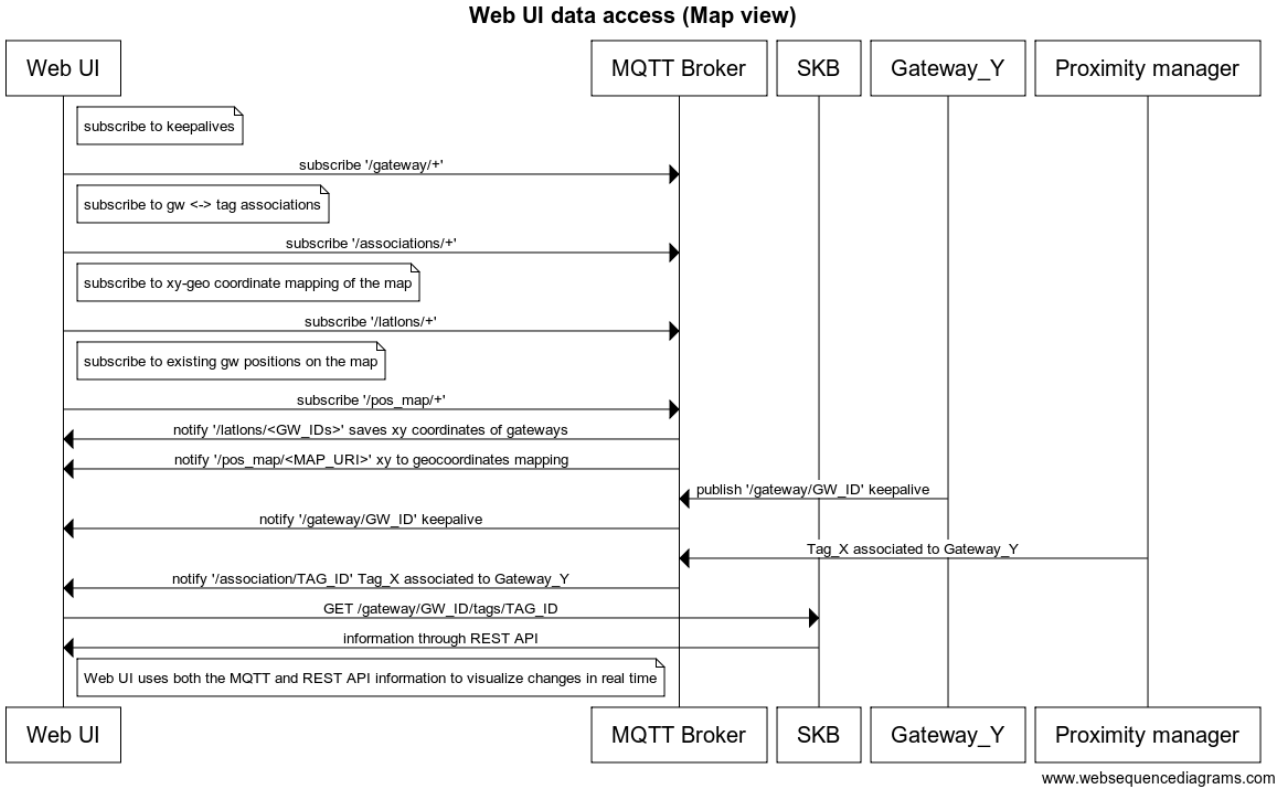


Figure 18. Web UI data access for map view.

3.4 IMPReSS test case implementation

The architecture described in the previous sections is a generic and can be used in various scenarios. In IMPReSS, we tested the approach by mapping the Federal University of Pernambuco (UFPE). Although the software was originally designed to be used in the Opera House scenario, we decided to try it in a larger environment.

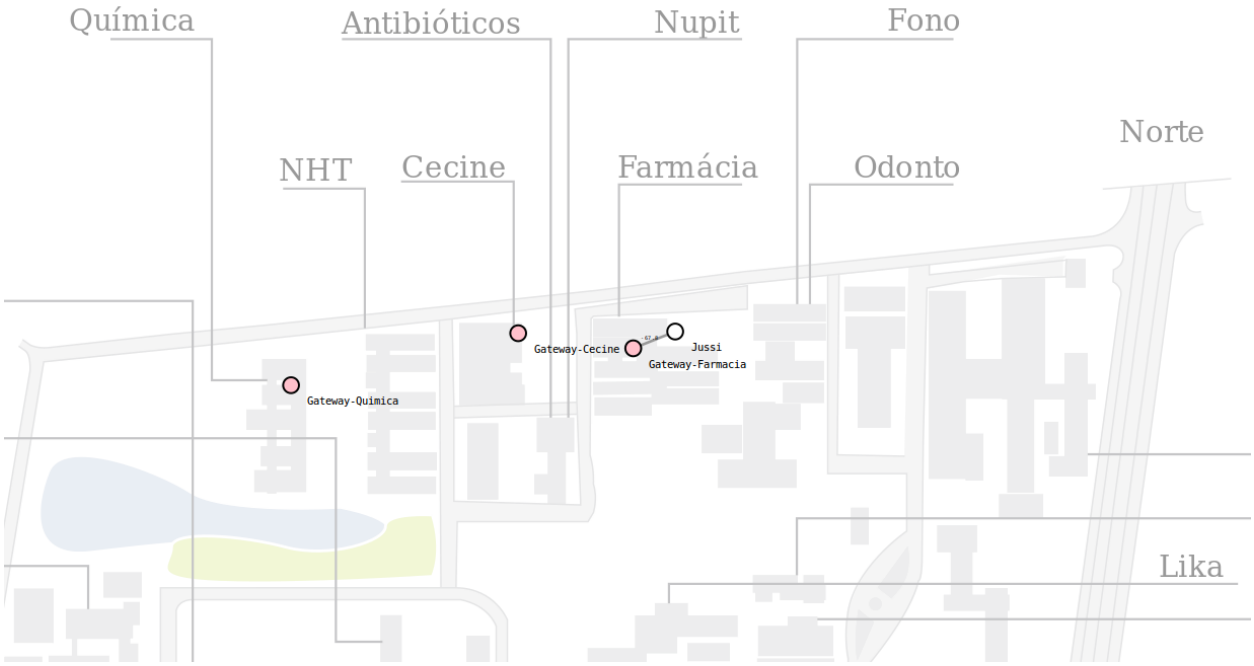


Figure 19. Map view with gateways positioned on the UFPE map.

Figure 19 shows the map of UFPE campus, with three simulated gateways (pink nodes) and one sensor-tag (white node) associated to object "Jussi". In this deployment, the gateway devices were not actually present at the UFPE premises, but this was not needed for demonstrating the deployment system.

The simulation showed that it was easy to adapt the IMPReSS system to use the UFPE campus map. In the early phase of the work the system was also simulated in the IMPReSS Opera House scenario, and the only change required was to use a specific vector graphics map instead of a floorplan. Gateways appear on the map approximately 20 seconds after they are plugged in. The delay is because of Rapsberry Pi microcomputers booting.

The following deployment steps were taken:

1. Open web UI object view
    a. Create some objects of interest to be monitored by the system
        i. Insert name, type, description
2. Open web browser at the Map View
    a. Press "A" key to enable "add place" mode.
        i. Click on map, input place name and description
        ii. Repeat for as many places needed
    b. Press "L" key to enable mapping of map pixels to geographical coordinates

i. Click on a known spot #1 and input latitude and longitude values
ii. Click on a known spot #2 and input latitude and longitude values
iii. Places and gateways now get real geographical coordinates in the system

3. Switch the gateway devices on
    a. Unassociated gateways appear on the map after 20 seconds (booting time)
    b. Select a gateway (it turns red) by clicking it
    c. Click on the map and the gateway is
        i. Moved to the new position
        ii. Automatically associated to the closest place coordinate
4. Switch on a sensor-tag
    a. Appears immediately on the map view as a white node
    b. Linked to the closest gateway
    c. Open tag view and associate an object to the tag
5. Move tag to another position
    a. Sensor values are automatically associated to the new place in System Knowledge Base.

3.4.1 Bluetooth Low Energy (BLE) sensor-tags

Any radio technology from which the signal strength can be obtained could be used in the system, but we chose VTT's TinyNode BLE in this setup. Raspberry Pi gateway devices were equipped with an Iogear USB dongle and a matching driver for receiving the BLE signal and publishing to the MQTT broker with a suitable payload.

The sensor-tags are based on Nordic Semiconductror N51422 chips and are programmable. The BLE sensor-tags used in the setup send an advertisement beacon every 1 second. The transmission rate is configurable and the advertisement message can include various sensor value readings, depending on the hardware used. In our proof-of-concept we transmitted a temperature value encoded in the advertisement message. The sensor-tag used is seen in Figure 20.

Figure 20. Coin sized VTT TinyNode BLE sensor-tag.

In Figure 21 the relation between sensor-tag battery life with a 220mAh CR2032 battery compared to advertisement message is shown. The data points are for 1, 10, 30, 60, 120, 240, 480 and 600 second intervals. One can see that the battery life starts degrading rapidly for transmission intervals under 60 seconds. Our setup used an interval of 1 second for easier debugging, leading to a limited battery life of approximately 3 months.

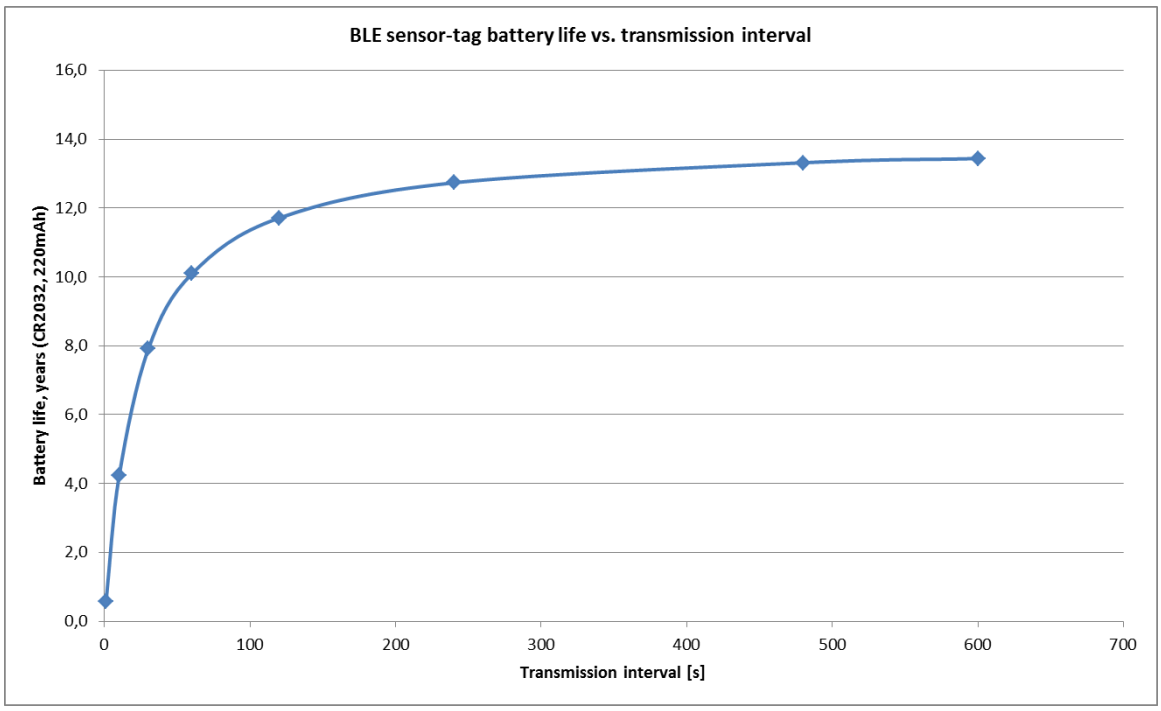


Figure 21. BLE sensor-tag battery life vs. transmission interval.

4. IoT gateway software management

4.1 IoT gateway software management overview

To provide capabilities for maintaining and upgrading the software of the IoT gateways, a version control system must be in place. To facilitate this, a software container system (also known as operating-system-level virtualization) is used.

There are a number of benefits to using a software container system. For a reasonably small system resource overhead cost it grants a degree of isolation to the containers. This allows for somewhat effortless switching between different versions for each containerized software component while guaranteeing a desired set of runtime libraries and filesystem state inside each container. Thus, the version management system can safely roll back software updates if necessary or have software components with conflicting runtime library dependencies.

In this setup, we use Docker as the software container implementation, running on the Raspbian distribution of GNU/Linux. It provides virtualization capabilities by utilizing facilities of the Linux kernel, such as cgroups and namespaces. It is controlled via a high-level API which the version controller software uses.

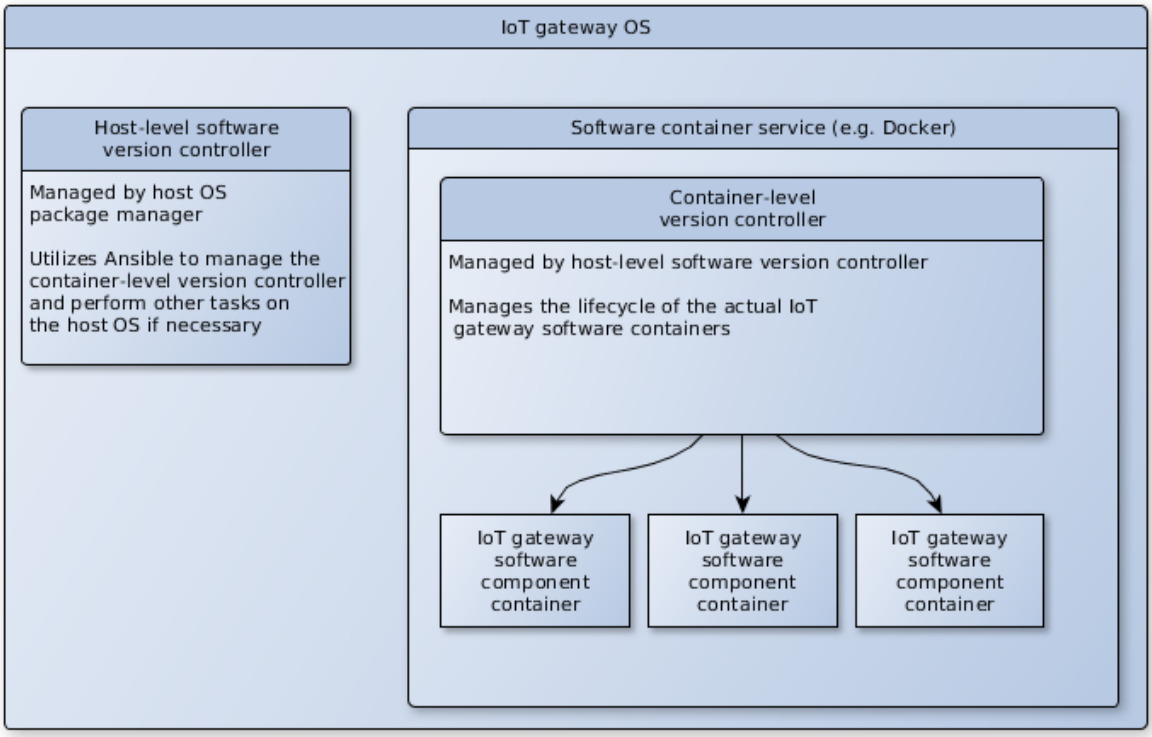


Figure 22. IoT gateway software management architecture.

The version control system consists of, if not counting the OS services and Docker, two separate components (see Figure 22): the host-level controller and the container-level controller. The rationale for this division stems from the fact that the version controller itself may need updating and since any update runs the risk of something going wrong with the update, this risk needs to be mitigated by having a separate component that is as simple as possible and preferably never updated. Thus, if an update of the container-level version controller fails or crashes due to a bug, the host-level controller acts as a rescue system.

4.2 Host-level version controller

Since this setup runs on a variant of the well-established GNU/Linux distribution Debian, we are able to make use of its native packaging system for installing the host-level version controller, which grants it additional robustness by making sure it and all its software dependencies are properly installed. The host-level software version controller itself is a rather simple, periodically run script-like application whose most important function is to perform an externally defined set of tasks, one of which is checking that the container-level version controller is working normally.

In practice, this is done utilizing a software called Ansible in so-called "pull mode". Ansible is a configuration management software platform, which can be used to perform a very large variety of tasks on a remote system. In our case, Ansible is installed as a dependency for the host-level software version controller. Periodically, the version controller retrieves a set of tasks (called a playbook) from the cloud. These tasks can include managing package installations on the IoT gateway host OS, modifying configuration files, starting and stopping services, and so on. Ansible has a number of plugins, which enable running these tasks in a robust manner. The basic tasks are ensuring that the desired version of Docker is installed and running and that the desired version of the container-level version controller is installed and running and not crashed or frozen. The latter check is performed using a subset of the REST API described later in this chapter.

4.3 Container-level version controller

The other component of the version controller is the container-level version controller, which manages the actual IoT gateway software components. It uses the Docker API to query Docker for information regarding running containers, and to control their starting, stopping, restarting and removal according to concurrent update information it receives from the cloud.

The update process is visualized in Figure 23. To receive new update information, the container-level version controller subscribes to an MQTT topic on a broker in the cloud. Thus, it receives new update information, as soon as it is published on the relevant MQTT topic. The update information, published in JSON format, contains Docker container configuration parameters, for each component container that is supposed to be running after the update. The data content of the update information message is described in Table 6.

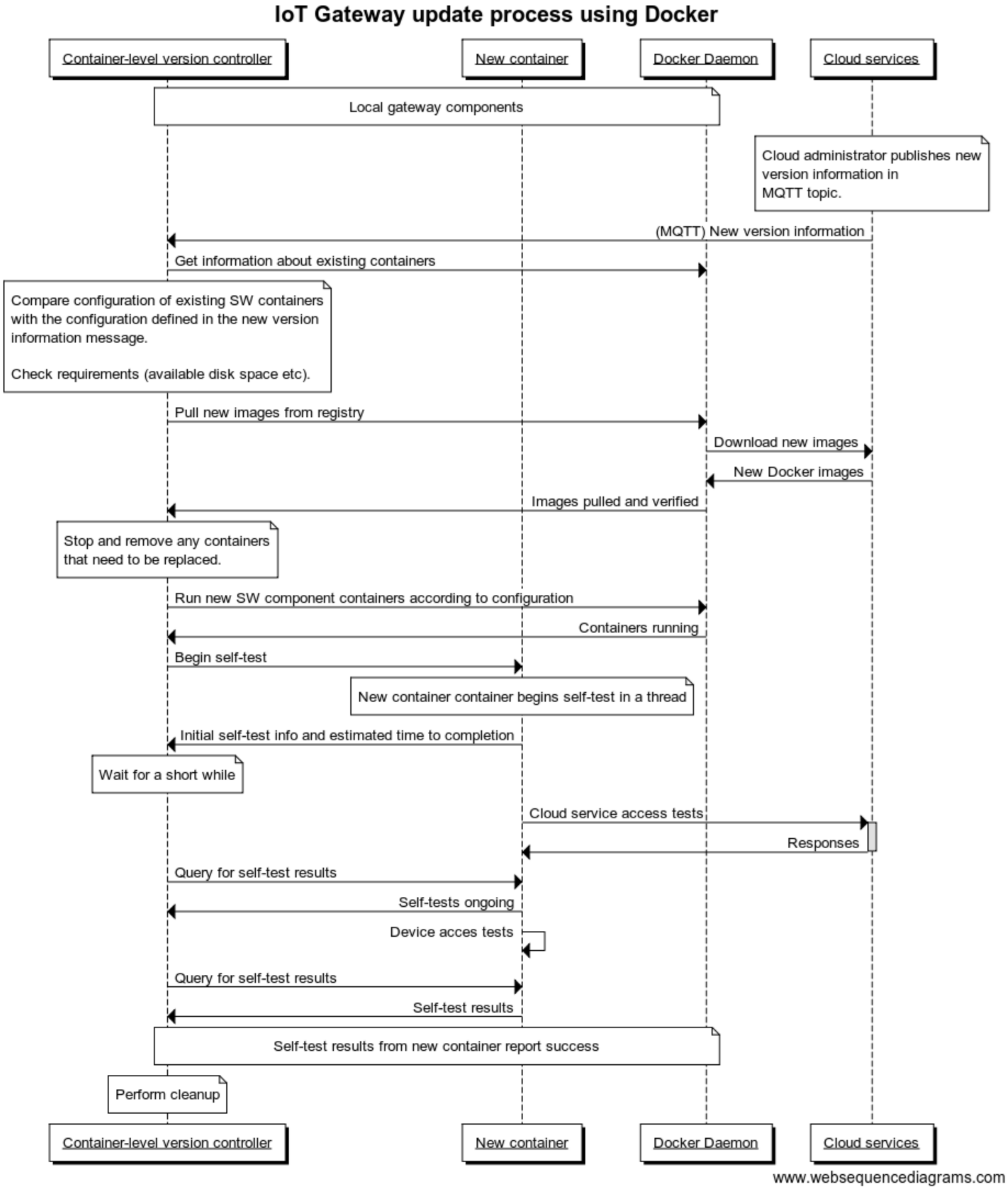


Figure 23. IoT gateway update process sequence diagram.

Table 6. Payload parameters.

Datum name	Example data	Description
version	0.12	Software component version
status	production	e.g. "production", "testing"
repository	registry.impress.eu/iotgateway/gateway_db	Docker image repository name
tag	0.12	Docker image repository tag
virtual_size_KiB	209408	Image disk space requirement
image_id	3e742de7ced0	Numerical id of the Docker image
cmd	/impress/startdb.sh	Command (and arguments) to run inside the container
publish_ports	["3306/tcp"]	Array of port mappings from the container to the host
lvc_api_version	0.10	Version of API used by the version controller to query and control the software component
lvc_api_port	8086	Port used for version control API communication
containername	gateway_db	Locally unique name for the container used by Docker
container_labels	["impress", "iot_gateway"]	Additional identifying labels used by Docker
volumes	["/var/lib/impress/db:/var/lib/mysql"]	Array of host paths to be mounted into the container
volumes_from	[]	Array of paths from other containers to be mounted into the container
links	[]	Array of local containers by name, to which this container should have private network access

Upon receiving new version information, the version controller determines which containers, if any, should be updated, created or removed. Any containers, identified by name, whose configuration has changed from that which is running, need to be updated, as well as any containers linking to or mounting volumes from them. Any container configurations not specified in the version information should be stopped.

The container-level version controller also utilizes a REST API to communicate with the software components in each container. This API (described in Table 7) is used to query and control the running of the software component residing in a container. Some containers, database containers for example, may require a controlled shutdown to ensure data integrity for any persistent data. When updating a container, the version controller will tell the software component to run its self-test-suite to check, for example, that it is able to access any network or device resources it requires. If the software component reports a problem or fails to perform the tests, the version controller will know to report the error to the cloud and roll back the update if necessary.

Table 7. REST interface description.

Verb	Resource (/impress/lvc/v010/..)	Description
GET	../info	General info
POST	../tests/selftest	Initiate component self-test
GET	../tests/selftest	Get self-test results from component
POST	../status/quit	Tell component to exit gracefully

4.4 Security considerations

The "wild card" nature of the setup described above raises a number of questions regarding security that need to be considered with due seriousness.

First of all, all communication must be secured cryptographically with both ends' identities verified. The standard way of accomplishing this is to use Transport Layer Security (TLS) protocol, which provides endpoint-authenticated, encypted, integrity-checked communications between the IoT gateway and any cloud service. In our setup we use pre-shared server and client certificates for authentication. An important detail is to make sure that sufficiently large key lengths are used, both in certificate signatures and communication encryption.

Secondly, one trade-off for having the host-level component of the management system as simple as possible, is that it ultimately runs arbitrary commands with super-user privileges on the host system. To counter malicious exploitation of this functionality, any Ansible playbooks or other scripts must be verified by the version controller to originate from a trusted source. To accomplish this, the host-level version controller uses a pre-installed GnuPG public key to verify the playbooks, in addition to verifying the communication with the cloud services. If keys of finite validity duration are used, however, the host-level version controller package may need to be updated from time to time to install renewed keys.

Thirdly, the authenticity of the Docker container configuration information, which the container-level version controller receives from an MQTT topic it subscribes to, must be ensured. Here again, the communication security provided by TLS is not enough. While MQTT traffic is easily secured by TLS, steps must be taken to make sure no unauthorized party can publish data to the topic from which the version controller receives the new version information. This can be accomplished by configuring the broker with access control lists that restrict publishing to that topic to clients certified with either a password or a TLS client certificate. In addition, the data payload itself may be signed with GnuPG, similarly to what the host-level version controller requires of the Ansible playbooks it runs.

Finally, there are a number of smaller concerns. One such is the risk that the Docker daemon may download a fake image. Measures to prevent this include using a private image registry, securing communications with TLS, with the identity of the image registry verified, and providing hard-to-fake identifying information about the images within the update information message. Another concern worth considering is the communication security between Docker containers running inside the IoT gateway. This should be a rather hypothetical concern since the containers are expected to reside within a single computer and only communicate via virtual network interfaces, but special cases or misconfiguration may compromise security.

5. Results and discussion

In this deliverable a novel IoT network management infrastructure was presented. This work has been executed in the task 3.4 - Network Management. The IoT network management infrastructure provides the system administrator with following features: 1) remote management of IoT gateway software, 2) low-effort deployment of IoT sensors and actuators, and 3) zero-effort management of IoT device context during runtime.

The IoT network management infrastructure consists of two individual frameworks: remote software management and IoT network context management. The remote software management framework is a Docker based solution that makes it possible to decrease the costs related to software management of IoT gateways by 1) making it possible to manage the software remotely and by 2) simplifying the process related to software packaging.

The IoT network context management framework provides means for low-effort deployment of IoT devices and zero-effort runtime management of IoT device context. The framework makes it possible to create and visualise IoT Entities and manage the associations between IoT Entities and IoT Resource at a room level. There is a IoT gateway, called Room GW, located in each room (i.e., IoT Entity) that needs to be monitored. Other type of IoT Entities are equipped with radio interfaces (BLE based active tags in this scenario, other radio technologies are also possible). When new IoT Resources are deployed or when the context of the system changes (e.g. IoT Entities move from one room to another) they system automatically associates the IoT Resources and IoT Entities with the room and with each other (i.e., ambient temperature sensor located in a room is associated to all the other objects located in the room). In practise, the associations are deduced from proximity information calculated from signal strength data sent by the tags and the sensor/actuator devices. By automating the processes related to context creation and maintenance, this framework makes it possible to decrease the costs related to IoT deployment and maintenance.

In our experiments, the signal strength data obtained from the BLE communication was sufficient for deducing the associations at the room level. However, there were some problems if two IoT Room GWs were located very close each other and the wall between them as thin. Therefore, if more accurate measurements or finer granularity level of association (i.e., associations within a room) are required different radio technology should be investigated. For example, active tags utilizing Ultra-wideband (UWB) radio (Reed 2005) would be good candidates for this purpose.

6. References

(Case et al1990) Case JD, Fedor M, Schoffstall ML & Davin J. (1990) Simple Network Management Protocol (SNMP).

(Gonçalves et al 2012) Gonçalves P, Oliveira JL & Aguiar RL. (2012) A study of encoding overhead in network management protocols. Intrnl.Journal of Network Management - IJNM 22(6): 435.

(IBM & Eurotech 2010) IBM & Eurotech. (2010) MQTT V3.1 Protocol Specification. URI: http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf. 2014(9/3).

(Shelby et al 2013) Shelby Z, Hartke K & Bormann C. (2013) Constrained Application Protocol (CoAP) draft-ietf-core-coap-18, RFC 7252. URI: http://datatracker.ietf.org/doc/draft-ietf-core-coap/. 2014(4/5).

(Fielding et al 1999) Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P & Berners-Lee T. (1999) Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616, URI: https://www.ietf.org/rfc/rfc2616.txt. 2014(10/23).

(Reed 2005) Reed J. (2005) Introduction to Ultra Wideband Communication Systems, an. Upper Saddle River, NJ, USA: Prentice Hall Press.

Appendix A

REST API output example for HTTP GET to: <BASE_URI>/gateways:


```
[
  {
    "name": "Gateway-Farmacia",
    "links": [
      {
        "href": "gateways/b8:27:eb:cd:90:5b",
        "rel": "self",
        "title": "Gateways"
      },
      {
        "href": "places/Farmacia",
        "rel": "place",
        "title": "Places"
      }
    ],
    "lastSeen": "2015-06-25T15:42:05.443558",
    "_id": {
      "$oid": "554b0e1889756465f66c44db"
    },
    "type": null,
    "id": "b8:27:eb:cd:90:5b",
    "description": "add description"
  },
  {
    "name": "Gateway-Cecine",
    "links": [
      {
        "href": "gateways/b8:27:eb:69:1a:61",
        "rel": "self",
        "title": "Gateways"
      },
      {
        "href": "places/Cecine",
        "rel": "place",
        "title": "Places"
      }
    ],
    "lastSeen": "2015-06-25T15:42:04.323222",
    "_id": {
      "$oid": "554b0e1f89756465f66c44ec"
    },
    "type": null,
    "id": "b8:27:eb:69:1a:61",
    "description": "add description"
  },
  {
```

```
    "name": "Gateway-Quimica",
    "links": [
      {
        "href": "gateways/b8:27:eb:32:58:29",
        "rel": "self",
        "title": "Gateways"
      },
      {
        "href": "places/Quimica",
        "rel": "place",
        "title": "Places"
      }
    ],
    "lastSeen": "2015-06-25T15:42:00.989635",
    "_id": {
      "$oid": "554b0e1d89756465f66c44e9"
    },
    "type": null,
    "id": "b8:27:eb:32:58:29",
    "description": "add description"
  },
  {
    "name": "Gateway-Quimica",
    "links": [
      {
        "href": "gateways/b8:27:eb:ae:4b:de",
        "rel": "self",
        "title": "Gateways"
      },
      {
        "href": "places/Quimica",
        "rel": "place",
        "title": "Places"
      }
    ],
    "lastSeen": "2015-06-25T15:42:00.330857",
    "_id": {
      "$oid": "554b0e1889756465f66c44da"
    },
    "type": null,
    "id": "b8:27:eb:ae:4b:de",
    "description": "add description"
  }
]
```