UNIVERSITÀ DI PISA

Distributed Systems and Middleware Technologies

The JANET Home Service

Table of Contents

Introduction

The application that was developed, named *JANET (Java erlAng NETwork) Home Service*, consists in a distributed system designed for the *Internet of Things (IoT)* consumer market allowing for the remote management and monitoring of networks of *smart devices*, where each user is associated with the collection of smart devices he or she owns, including smart fans, smart lights, smart thermostats and more, that are deployed in a set of user-defined *locations* (e.g. home, office, apartment) and *sublocations*, representing arbitrary logical partitions of the locations (e.g. living room, bedroom workshop).
The set of devices deployed in each location is managed by a local JANET Controller device acting as a border gateway towards the rest of the JANET service, consisting in a *cloud infrastructure* comprised on the one hand of a back-end dedicated to storing and processing all information relevant to the application, such as the locations and sublocations configurations, the history of device state updates, user credentials and more, and on the other of a front-end offering authenticated users functionalities such as creating and deleting locations and sublocations, adding or deleting devices, moving devices between sublocations, monitoring and changing the devices' states and browsing a set of graphical device usage statistics derived from their states' histories.
As a simplifying hypothesis aimed at abstracting the devices' management from their implementation details and at avoiding interoperability concerns across products of different vendors, in the first release of the system the JANET devices and controllers have been implemented as dedicated *virtual machines* managed through an ad-hoc simulation environment termed *JANET Home Simulator*.

System Requirements

Functional Requirements

The functional requirements of the application are outlined below:

- The system shall present a set of distributed components, collectively termed *cloud infrastructure*, comprised on the one hand of a *back-end* dedicated to storing and processing all information relevant to the application, and on the other of a *front-end* offering users authenticated in the application a *web interface* for accessing its functionalities.
- Users in the applications shall be divided into *unauthenticated users* and *authenticated* (or *registered*) *users*.
- Unauthenticated users in the application shall be allowed the following functionalities:
  - Log into the application via a *username/password* authentication mechanism.
  - Register into the application as a new authenticated user by providing a *username*, a *password* and an *e-mail address*, where the username must be globally unique and with the registration process requiring users to confirm their e-mail address by visiting an activation link provided in a verification e-mail sent submitting their account information.
  - Recover the lost or forgotten password associated with their account by having it forwarded to the account's e-mail address.
- Authenticated users in the application shall be allowed the following operations:
  - Add any number of *locations* of custom symbolic *names* to their private set of locations, each representing a defined operating environment corresponding from a networking perspective to a *Local Area Network (LAN)*.
  - Add any number of *sublocations* of custom symbolic *names* to their private set of sublocations, each representing a logical partitioning, typically associated with a well-defined sub-environment, of a specific location.
  - Add any number of *devices*, or *JANET devices*, of custom symbolic *names* to their private set of devices, each representing a smart appliance of a specific device *type* defining the attributes, or *traits*, comprising its *state*, as well as the set of *commands*, or *actions*, aimed at changing such state the device is capable of processing.
  - Assign, or deploy, each of their devices in one of the sublocations within the locations they have defined.
  - Browse their private sets of locations, sublocations and devices.
  - View, depending on whether they are currently online, the real-time or the last known states of their devices
  - Change the custom symbolic names associated with their locations, sublocations and devices.
  - Delete any of their locations, sublocations and devices.
  - Issue commands to their devices aimed at changing their state.
  - Browse a set of graphical device usage statistics derived from their states' histories.
  - Log out from the application.
- Each location shall comprise a dedicated *JANET controller* device acting as a border gateway between the devices deployed in the location and the cloud infrastructure, routing commands and state updates to their destinations.
- JANET devices shall report every state change, whether caused by direct user intervention (by issuing a command, but also via on-site physical interaction) or by sensor readings, to the cloud infrastructure.

Non-Functional Requirements

The main non-functional requirements of the application are discussed below:

- Timeliness
  The latency between the issuing and the displaying of the results of operations submitted via the web interface should be minimized (ideally <2 seconds and in no case greater than 5 seconds) so as to prevent the degradation of the Quality of Service (QoS) experienced by end-users, with the overall service attuning to the definition of a soft real-time system.
- Security
  The system shall enforce standard security practices in preventing the unauthorized access to its functionalities, both in the cloud infrastructure, where the users' passwords must be stored in encrypted form, and in the users' operating environments, where only devices previously registered via the web interface should be allowed connection with a JANET controller.
- Usability
  The system shall present a commercial-level degree of usability in the functionalities offered to end users, which should be provided in an intuitive format without requiring expert knowledge in the IT sector.

Working Hypotheses

The system's development in its first release has been based upon the following working hypotheses:

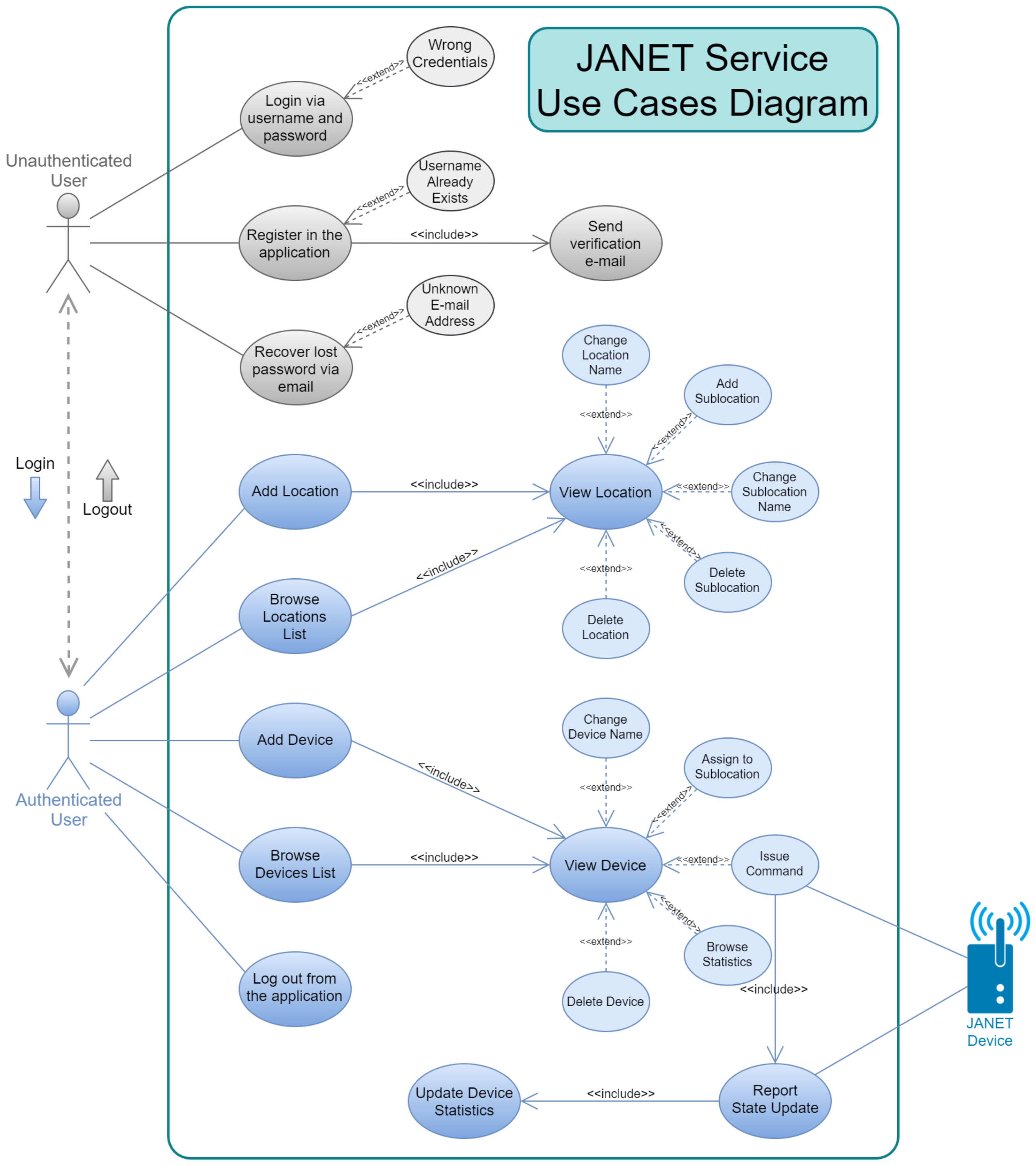
- In order to abstract the devices' management from their implementation details and to avoid interoperability concerns across products of different vendors, the JANET devices and controllers have been implemented as dedicated *virtual machines* managed through an ad-hoc simulation environment termed *JANET Home Simulator*.
- Only the following device types are supported, whose *state* and *commands* were selected as subsets of the *traits* and *actions* associated with such types as defined in the Google Smart Home Interface: *fan*, *light*, *door*, *thermostat*, *conditioner*.
- Being the users' operating environments simulated, the non-functional requirement of security was relaxed by not encrypting the communications between the JANET controllers and the cloud infrastructure components as well as between the JANET controllers and the JANET devices.
- Existing regulations and concerns relative to the users' data privacy were not taken into account.
- Advanced aspects typical of commercial-grade distributed systems such as data redundancy, disaster recovery and load balancing were not taken into consideration.

Actors and Use Cases Diagram

From the analysis of its functional requirements, three actors were identified in the system:

- The Unauthenticated Users, who are allowed only a restrict subset of the application's functionalities.
- The Authenticated Users, who are allowed all of the application's main functionalities.
- The JANET devices, which report state updates to the cloud infrastructure.

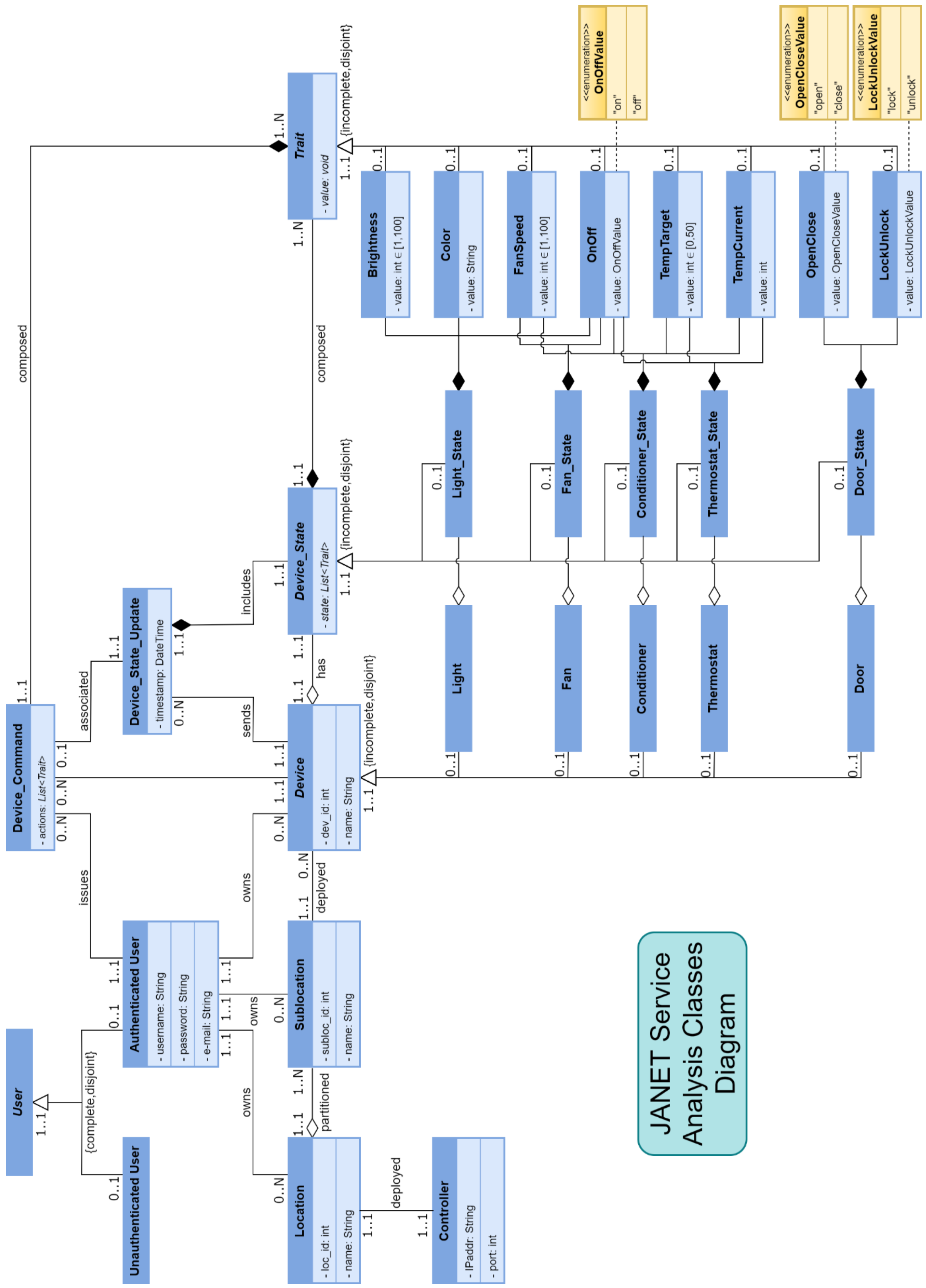
The use cases associated with each actor is outlined in the diagram below:



Architectural Design

Analysis Classes Diagram

From the service's requirements and its working hypotheses the following analysis classes were identified within the application:



Classes Definitions

Class	Description
User	An abstract class representing a user in the application
Unauthenticated User	An unauthenticated user in the application
Authenticated User	An authenticated user in the application
Location	An operating environment corresponding to a LAN owned by a user
Controller	The JANET controller acting as a border gateway between the devices deployed in the location and the cloud infrastructure
Sublocation	A logical partitioning of a specific location
Device	An abstract class representing a smart device owned by a user and deployed in one of its sublocations
Light	A *Device* of type "Light"
Fan	A *Device* of type "Fan"
Conditioner	A *Device* of type "Conditioner"
Thermostat	A *Device* of type "Thermostat"
Door	A *Device* of type "Door"
Device_State	An abstract class representing the state of a device, which is composed of a set of traits depending on its type
Light_State	The state for a device of type "Light"
Fan_State	The state for a device of type "Fan"
Conditioner_State	The state for a device of type "Conditioner"
Thermostat_State	The state for a device of type "Thermostat"
Door_State	The state for a device of type "Door"
Trait	An abstract class representing an attribute of a device's state
Brightness	The device's relative brightness level ∈ [1,100] ("Light")
Color	The device's color in hexadecimal format ("Light")
FanSpeed	The device's relative fan speed ∈ [1,100] ("Fan", "Conditioner")
OnOff	Whether the device is on or off ("Light", "Fan", "Thermostat", "Conditioner")
TempTarget	The desired temperature for the device's operating environment ∈ [0,50] ("thermostat", "conditioner")
TempCurrent	The current temperature of the device's operating environment ("thermostat", "conditioner")
OpenClose	Whether the device is open or closed ("door")

LockUnlock	Whether the device is locked or unlocked ("door")
Device_Command	A command issued by a user to change the state of one of their devices
Device_State_Update	The updated state reported by a device

Classes Attributes

User		
Attribute	**Type**	**Description**
none		

Unauthenticated User		
Attribute	**Type**	**Description**
none		

Authenticated User		
Attribute	**Type**	**Description**
username	String	A unique string identifying the user, which is also used to access the application
password	String	The password required for the user to access the application
e-mail	String	The user's email (optional)

Location		
Attribute	**Type**	**Description**
loc_id	int	The location's unique identifier
name	String	The location symbolic name

Controller		
Attribute	**Type**	**Description**
IPaddr	String	The IP address by which the controller can be reached by the cloud infrastructure
port	int	The port by which the controller can be reached by the cloud infrastructure

Sublocation		
Attribute	**Type**	**Description**
subloc_id	int	The sublocation's unique identifier
name	String	The sublocation's symbolic name

Device		
Attribute	**Type**	**Description**
dev_id	int	The device’s unique identifier
name	String	The device’s symbolic name

Light		
Attribute	**Type**	**Description**
none (same as the superclass)		

Fan		
Attribute	**Type**	**Description**
none (same as the superclass)		

Conditioner		
Attribute	**Type**	**Description**
none (same as the superclass)		

Thermostat		
Attribute	**Type**	**Description**
none (same as the superclass)		

Door		
Attribute	**Type**	**Description**
none (same as the superclass)		

Device_State		
Attribute	**Type**	**Description**
state	*List<Trait>*	The list of traits composing the device’s state

Light_State		
Attribute (Traits)	**Type**	**Description**
onOff	OnOffValue	Whether the light is on or off
brightness	int	The light’s relative brightness level ∈ [1,100]
color	String	The light’s color in hexadecimal format

Fan_State		
Attribute (Traits)	**Type**	**Description**
onOff	OnOffValue	Whether the fan is on or off
fanSpeed	int	The fan relative speed ∈ [1,100]

Conditioner_State		
Attribute (Traits)	**Type**	**Description**
onOff	OnOffValue	Whether the conditioner is on or off
tempTarget	int	The desired temperature for the conditioner's operating environment ∈ [0,50]
tempCurrent	int	The current temperature of the conditioner's operating environment
fanSpeed	int	The conditioner's fan relative speed ∈ [1,100]

Thermostat_State		
Attribute (Traits)	**Type**	**Description**
onOff	OnOffValue	Whether the thermostat is on or off
tempTarget	int	The desired temperature for the thermostat's operating environment ∈ [0,50]
tempCurrent	int	The current temperature of the thermostat's operating environment

Door_State		
Attribute (Traits)	**Type**	**Description**
openClose	OpenCloseValue	Whether the door is open or closed
lockUnlock	LockUnlockValue	Whether the door is locked or unlocked

Trait		
Attribute	**Type**	**Description**
value	void	The trait's value

Brightness		
Attribute	**Type**	**Description**
value	int	The device's relative brightness level ∈ [1,100] ("light")

Color		
Attribute	**Type**	**Description**
value	String	The device's color in hexadecimal format ("light")

FanSpeed		
Attribute	**Type**	**Description**
value	int	The device's relative fan speed ∈ [1,100] ("Fan", "Conditioner")

OnOff		
Attribute	**Type**	**Description**
value	OnOffValue	Whether the device is on or off ("Light", "Fan", "Thermostat", "Conditioner")

TempTarget		
Attribute	**Type**	**Description**
value	int	The desired temperature for the device's operating environment ∈ [0,50] ("thermostat", "conditioner")

TempCurrent		
Attribute	**Type**	**Description**
value	int	The current temperature of the device's operating environment ("thermostat", "conditioner")

OpenClose		
Attribute	**Type**	**Description**
value	OpenCloseValue	Whether the device is open or closed ("door")

LockUnlock		
Attribute	**Type**	**Description**
value	LockUnlockValue	Whether the device is locked or unlocked ("door")

Device_Command		
Attribute	**Type**	**Description**
actions	*List<Trait>*	The list of traits with their desired values to be issued to the device

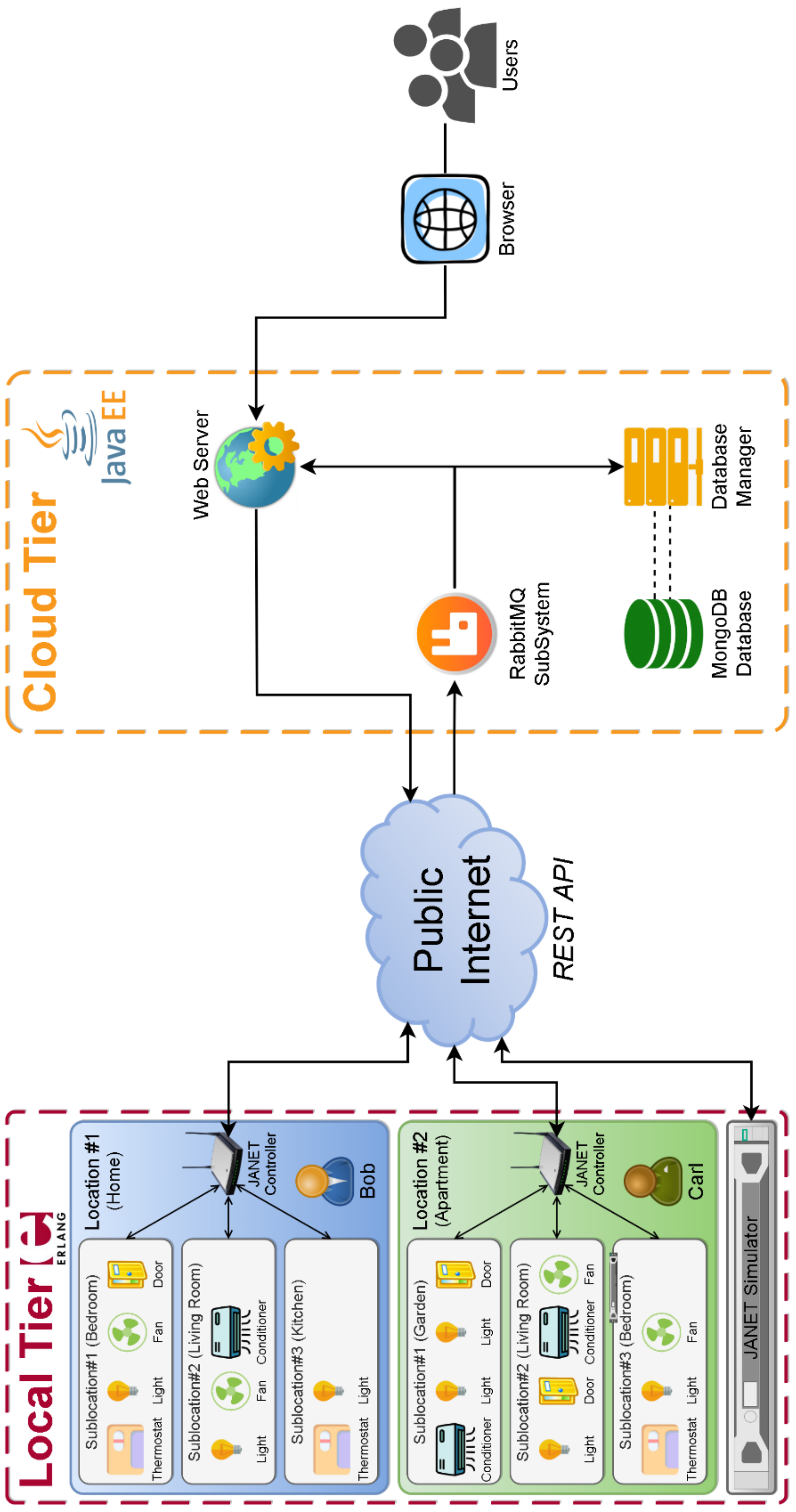
Device_State_Update		
Attribute	**Type**	**Description**
timestamp	DateTime	The device's local time at which the state update occurred

Enumerations Definitions

Enumeration	Possible Values
OnOffValue	"on", "off"
OpenCloseValue	"open", "close"
LockUnlockValue	"lock", "unlock"

Software Architecture

In terms of its software architecture, the service is logically divided into a *Cloud* and a *Local* tier communicating over the public internet via RESTful interface:



Cloud Tier

The cloud tier is composed of four software modules interacting via network communication, allowing their deployment on different systems depending on the available infrastructure, and will be implemented using the *Java Enterprise Edition*:

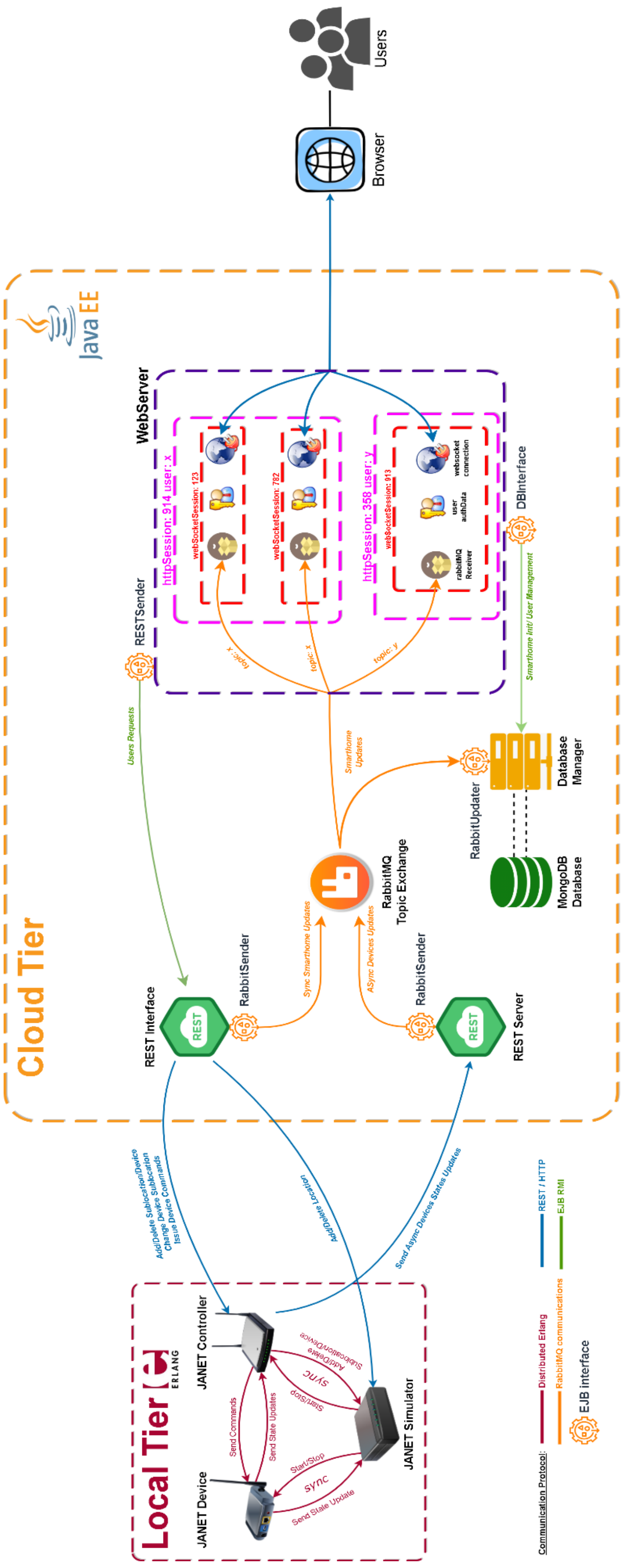
- A *Web Server* module, implementing the web interface offered to users for interacting with the application and a component for issuing commands for the smart devices, which will be developed using the *Glassfish* application server. The component is connected to:
    - The *Local Tier* through a REST interface for sending user request to their devices
    - The *RabbitMQ Subsystem* to be able to send/receive updated to/from the smarthomes
    - The *Database Manager* for managing users(login/registration/information update/smarthome retrieval)
- A *RabbitMQ Subsystem* module, to route device updates to the various network components. The choice to adopt such a complex Public/Subscribe messaging mechanism is due to two factors:
    - Allow an evolution of the system over time making the addition of new components simple and perfectly scalable
    - The delegation of mutual exclusion problems to an external entity, the messages are delivered to the various modules involved that can process them in a totally independent manner
- A *Database Manager* module, to permanently maintain the application data. The data can be distinguished into:
    - Information about users, to allow them to register an account and log into the application
    - Information about the users' smarthomes, to enable initialization of their sessions with an updated copy of the status of their devices
    - Information about device operations, in order to make statistics about their use and make them available to users

Local Tier

The local tier comprises the set of the authenticated users' operating environments, which include the locations, sublocations and devices they have defined, as well as the JANET controller deployed in each location.
As previously introduced, in the first system release this tier has been implemented in an ad-hoc simulation environment termed *JANET Home Simulator*, in which JANET controllers and devices consist of dedicated virtual machines, or Erlang nodes, implementing all behaviours and communication protocols of their physical counterparts, and whose execution is managed through the use of a hidden simulation manager node termed *JANET Simulator*.

Application Dataflow

The main data flows between the service's actors and software components and their application-level protocols are summarized in the diagram below:



JANET Home Simulator

The *JANET Home Simulator* is a self-sufficient simulation environment implementing via virtual machines the authenticated users' operating environments, or *local tier*, of the JANET Home Service.
The system was developed in Erlang/OTP v.24 and consists of three main OTP applications to be executed in distinct *Erlang Run Time Systems* (ERTS), or nodes, each being labelled in the context of the environment as the main application it executes:

- *JANET Device* nodes represent smart appliances owned by authenticated users and deployed in a specific sublocation within a location, and as previously discussed are characterized by a *type* and a *state* whose updates are forwarded towards the cloud tier via their locations' JANET Controllers.
- *JANET Controller* nodes act as border gateways between the JANET devices deployed in their location and the cloud tier, routing device commands and device state updates to their destinations.
- The *JANET Simulator* is a special node managing the entire simulation environment by dynamically starting, stopping and synchronizing with the other nodes.

The system was developed using the Rebar3 project management tool, and includes the following middleware OTP applications:

- Mnesia, a built-in Database Management System (DBMS).
- Cowboy, an HTTP server.
- Gun, an HTTP client.
- Ranch, a TCP socket acceptor pool manager.
- Cowlib, a library application used by the *Cowboy* and *Gun* applications.
- JSONE, a library application for encoding Erlang terms in JSON data and vice versa.

In addition to the simulation environment architectural details, thoroughly described in the following sections, the two following complementary guides are available in their accompanying documents:

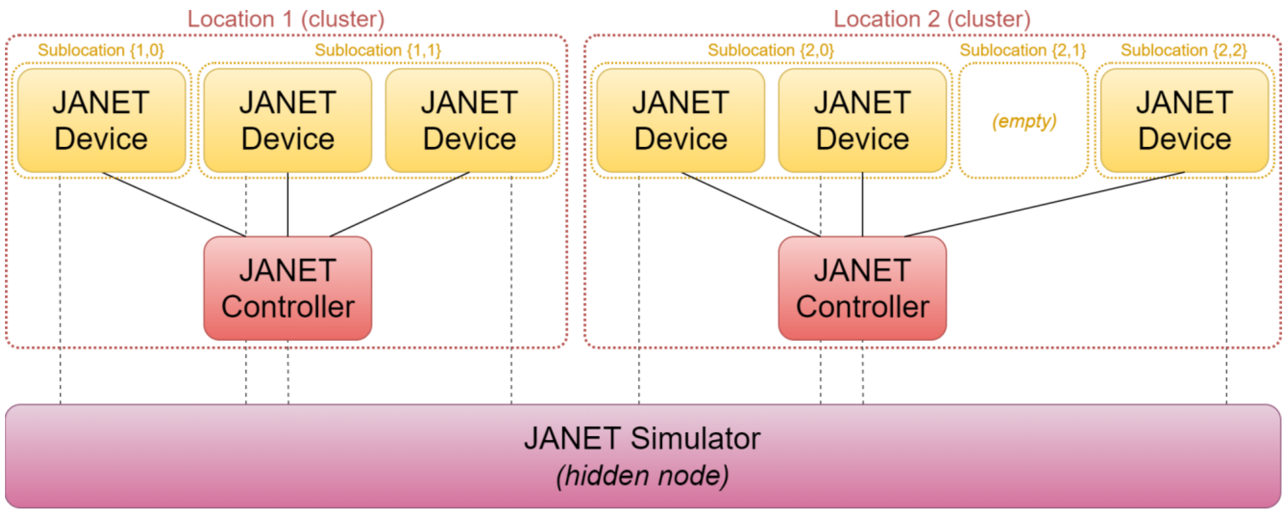
- A *system installation and configuration guide*, outlining the prerequisites and the steps for deploying the simulation environment.
- A *user guide*, highlighting the main commands and functionalities offered by the system.

Overlay Network Architecture

Nodes Clusters

JANET Controller and JANET Device nodes are organized via the Erlang *cookie* mechanism in clusters corresponding to locations, each comprising its controller and an arbitrary number of device nodes logically distributed in an arbitrary number of sublocations, with each device establishing a point-to-point connection with its location's controller obtained by disabling the default transitivity property of inter-node connections.

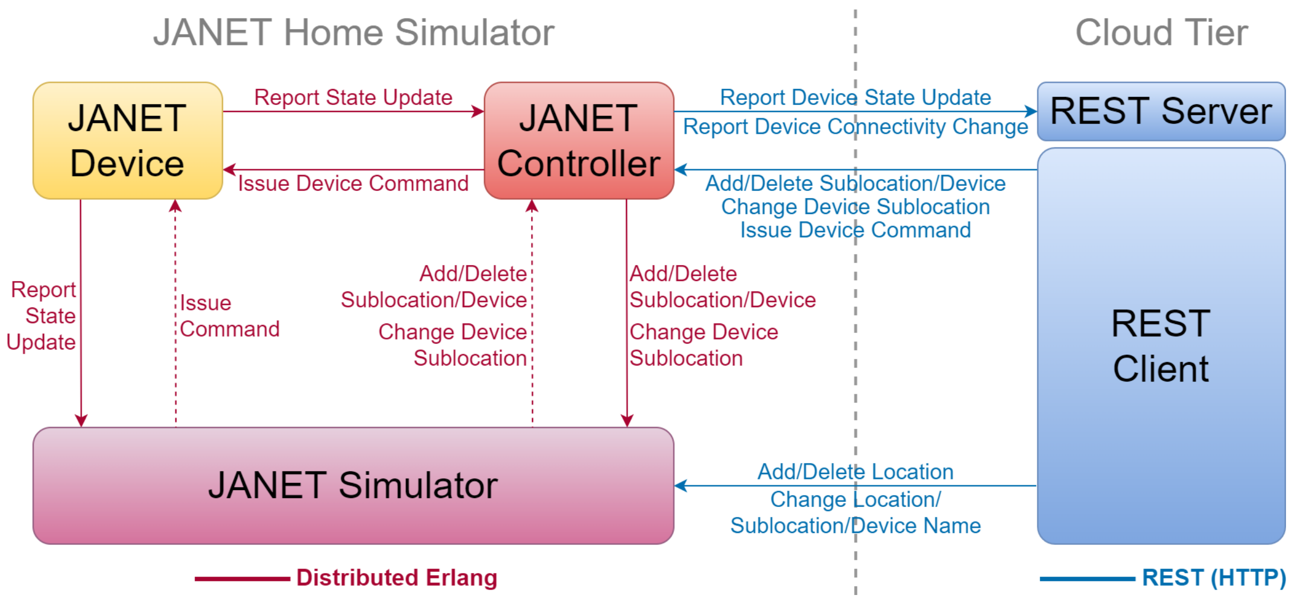
The JANET Simulator node, on the other hand, does not belong to any shared cluster, and represents a *hidden* node that is always connected with every other node in the system.



Nodes Dataflow

Within the overlay network JANET nodes exchange data using the message-passing constructs offered by the *Distributed Erlang* paradigm, typically via the *call* and *cast* callback functions exported by the standard *gen_** OTP behaviours, while communication with the cloud tier is implemented by the JANET Simulator and JANET Controllers exposing two different REST interfaces, collectively allowing for the remote use and management of the entire simulation environment, with controllers also presenting an HTTP client for sending asynchronous device state and connectivity updates to the remote REST server.

A summary of the nodes' functional interactions, i.e. the data exchanges directly implementing one or more of the system's functional requirements when at the steady state, is outlined below:


JANET Home Simulator
Cloud Tier
JANET Device
JANET Controller
Report State Update
Issue Device Command
Report Device State Update
Report Device Connectivity Change
REST Server
Add/Delete Sublocation/Device
Change Device Sublocation
Issue Device Command
Report State Update
Issue Command
Add/Delete Sublocation/Device
Change Device Sublocation
Add/Delete Sublocation/Device
Change Device Sublocation
REST Client
JANET Simulator
Add/Delete Location
Change Location/ Sublocation/Device Name
Distributed Erlang
REST (HTTP)

It should also be noted that all functionalities exposed to the remote REST client via the two REST interfaces are also offered internally via the JANET Simulator API, enabling the simulation environment to be used as a fully autonomous system.

Nodes Hosts

The simulation environment allows its nodes to be distributed among a set of predefined hosts, where:

- The JANET Simulator can be started on an arbitrary host by launching the homonymous OTP application.
- JANET Controller and JANET Device nodes are dynamically started and stopped by the JANET Simulator node, which deploys them on their designated *nodes hosts* defined at the moment their associated location or device was added into the system.

Operating Principles

Start-up Sequence

The simulation environment is started by launching the JANET Simulator application on its designated node, which initiates the following start-up sequence whose details will be more thoroughly explained later in the document:

1) The configurations of the users' locations and devices are retrieved from the Mnesia database installed on the JANET Simulator node.
2) Each location and device is assigned in the JANET Simulator a *controller* or *device manager* process, which attempts to start its ERTS on its designated *nodes host*.
3) After the ERTS has been started, each *controller* and *device manager* starts on its managed node the JANET Controller or the JANET Device application via a remote procedure call (RPC), also passing the information required for its initialization.
4) JANET Controllers and JANET Device nodes perform their own start-up operations, which they complete by their processes designed to exchange information with the JANET Simulator (in particular, the *ctr_simserver* process for controller nodes and the *dev_server* process for device nodes) sending their PIDs to their associated manager processes on the JANET Simulator.
5) After their start-up operations have completed:
   - Device nodes start executing the state machine associated with their device *type* while also attempting to *pair* with their location's controller by contacting its *ctr_pairserver* process.
   - Controller nodes accept via their *ctr_pairserver* processes pairing requests coming from device nodes being recognized as belonging to their location, spawning for each of them a *ctr_devhandler* process used for exchanging information with the device *dev_server* process, while also attempting via the *Gun* HTTP Client to establish a persistent connection with the remote REST server in the cloud tier.
6) Once device nodes are paired with their location's controller and controller nodes have established a persistent connection with the remote REST server the system is fully operational.

Nodes Statuses

During the system's execution controller and device nodes can be in one of the following *statuses*, which are tracked and updated by their associated *controller* and *device manager* processes in the JANET Simulator:

Status	Controller Node	Device Node
NOT_STARTED	The node's manager in the JANET Simulator, and thus the node itself, have not been started yet	
BOOTING	The node's manager in the JANET Simulator has started, but the process on its managed node used for exchanging information has not yet sent its PID to *register* the node.	
CONNECTING	The controller has registered within the JANET Simulator, but has not yet established an HTTP connection with the remote REST server	The device has registered within the JANET Simulator, but has not yet *paired* with its location's controller
ONLINE	The controller has established an HTTP connection with the remote REST server	The device has *paired* with its location's controller
STOPPED	The node and its manager in the JANET Simulator have been temporarily stopped	

Nodes Management

JANET Controller and JANET Device nodes are both *linked* with and *monitored* by their associated *controller* and *device* managers in the JANET Simulator, which allows for:

- Their automatic restart should they stop or crash, where as a measure for preventing data inconsistencies in the simulation environment the OTP supervision trees in controller and device nodes do not attempt to restart processes exiting with a reason other than *normal*, causing instead such exit signal to propagate up in the OTP tree up to the application master, which being the JANET Controller and JANET Device nodes defined as *persistent* applications shuts down the entire node with such error reason, which in turn is propagated to the *linked* controller or device manager in the JANET Simulator that, after being restarted by its own supervisor, restarts in turn its managed node by initializing it with the data in the JANET Simulator database.
- Their automatic shutdown when their associated location or device is deleted from the JANET Simulator or when the simulation environment is shut down, with the *shutdown* exit reason being propagated from the managers to their associated nodes.
- The possibility of temporarily stopping and dynamically restarting each node, which is obtained by stopping and restarting their manager processes via a dedicated API offered by the JANET Simulator, which makes it possible to simulate the condition of their physical devices being powered off or in any case disconnected from their operating environments' local area networks.

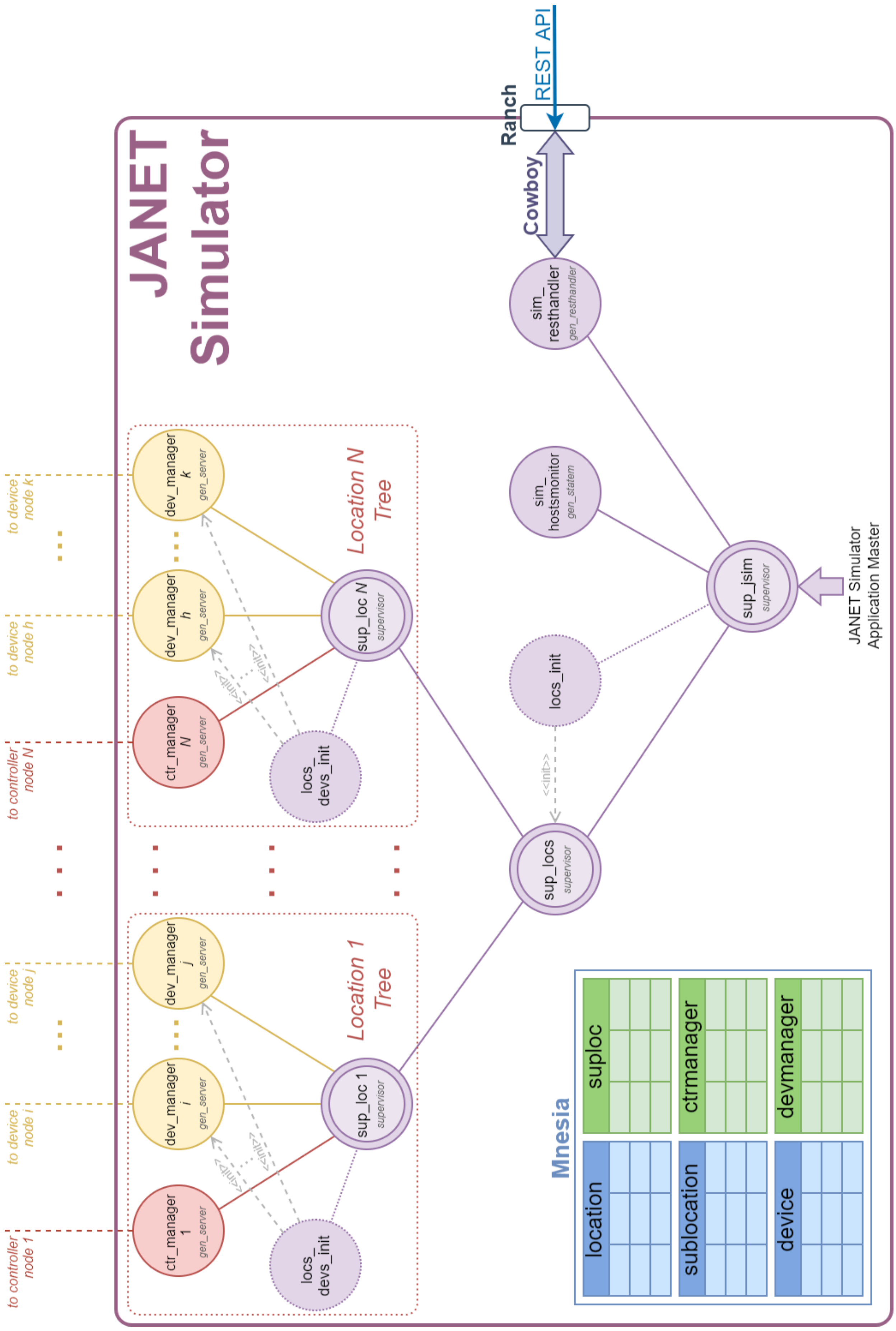
System Base Configuration

The simulation environment base configuration is set by editing the values of the following parameters defined in the *sys.config* configuration file used to start the JANET Simulator node:

Parameter	Description	Allowed Values	Note
sim_rest_port	The OS port to be used by the JANET Simulator REST server	integer() > 0	-
remote_rest_server_addr	The address of the remote server accepting REST requests from JANET Controller nodes	list() / string()	If deployed on the same host, the machine name (e.g. "yourHost") must be used over "localhost"
remote_rest_server_port	The port of the remote server accepting REST requests from the JANET Controller nodes	integer() > 0	-
remote_rest_server_path	The remote REST server path where to send device state and connectivity updates	list() / string()	-
nodes_hosts	The list of hostnames JANET nodes can be deployed in	[list() / string()]	If the JANET Simulator host is to be included, use the full machine name (e.g. "yourHost") over "localhost"

The rest of the environment persistent configuration, including the definitions of the users' locations, sublocations and devices, are instead stored in the Mnesia database installed in the JANET Simulator node, which will be discussed later.

JANET Simulator Architecture



OTP Supervisors

The configurations and children specifications of the supervisors used in the JANET Simulator OTP tree are outlined below:

Name	Role	Strategy	Intensity	Period
sup_jsim	JANET Simulator root supervisor	*rest_for_one*	2	30
sup_locs	Locations' trees supervisor	*simple_one_for_one*	2	30
sup_loc	Location supervisor	*one_for_one*	2	30

sup_jsim Children Specification					
ChildID	**Description**	**Multiplicity**	**Type**	**Restart**	**Shutdown**
sup_locs	Locations trees supervisor	1..1	*supervisor*	*permanent*	15000
locs_init	Locations trees initializer	1..1	*worker*	*transient*	5000
sim_hostsmonitor	Remote hosts monitor	1..1	*worker*	*permanent*	5000
sim_resthandler	REST server handler	1..1	*worker*	*transient*	8000

sup_locs Children Specification					
ChildID	**Description**	**Multiplicity**	**Type**	**Restart**	**Shutdown**
sup_loc	Location tree supervisor	0..N	*supervisor*	*permanent*	14500

sup_loc Children Specification					
ChildID	**Description**	**Multiplicity**	**Type**	**Restart**	**Shutdown**
ctr_manger_*i*	Controller manager	1..1	*worker*	*transient*	14000
locs_devs_init	Location device managers initializer	1..1	*worker*	*transient*	200
dev_manager_j	Device manager	0..M	*worker*	*transient*	14000

Mnesia Database

As previously introduced, all persistent information in the simulation environment is stored in the Mnesia database installed on the JANET Simulator node, whose structure in terms of tables, their records definitions and relationships is presented below:

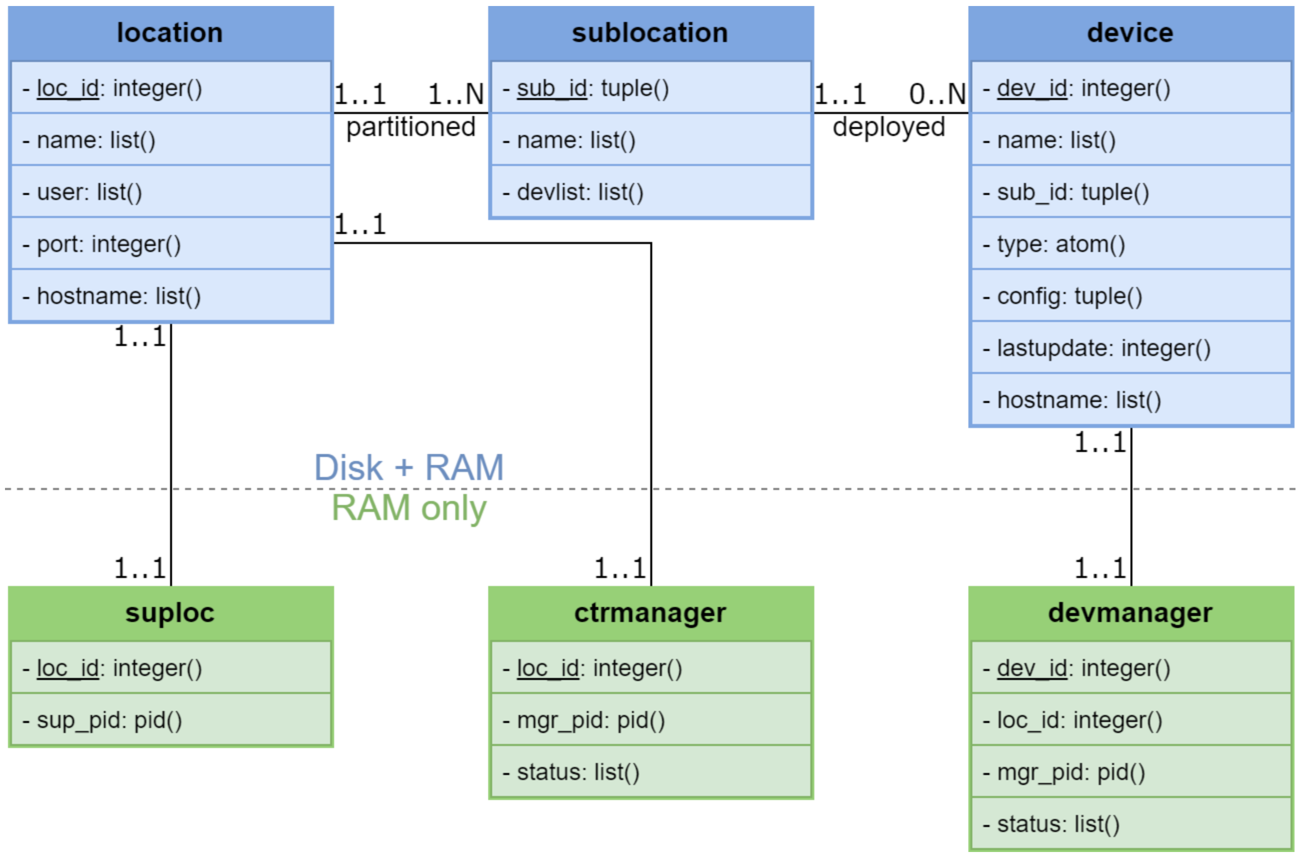


Table	Type	Description	Stored in	Notes
location	*set*	Location information	Disk + RAM *(disc_copies)*	-
sublocation	*ordered _set*	Sublocation information	Disk + RAM *(disc_copies)*	For each location "i" a *default sublocation* {i,0} is defined that cannot be deleted or renamed
device	*set*	Device information	Disk + RAM *(disc_copies)*	-
suploc	*set*	Location supervisors dynamic naming registry	RAM Only *(ram_copies)*	Used over the default registry for circumventing the necessity of dynamically generating atoms
ctrmanager	*set*	Controller nodes information	RAM Only *(ram_copies)*	Entries are managed by their associated *ctr_manager* processes
devmanager	*set*	Device nodes information	RAM Only *(ram_copies)*	Entries are managed by their associated *dev_manager* processes

location Record Definition			
Element	**Type**	**Description**	**Note**
loc_id	unique integer() > 0	The location unique identifier	Record key
name	list()	The location name	-
user	list()	The location user	Indexed element
port	unique integer() > 30000	The OS port to be used by the controller's REST server	- >30000 to reduce the risk of port allocation conflicts on the host OS - Indexed element
hostname	list()	The location controller's *nodes host*	Must belong to a predefined set of *allowed nodes hosts*

sublocation Record Definition			
Element	**Type**	**Description**	**Note**
sub_id	unique tuple()	The sublocation unique identifier	- Expressed as {*loc_id,subloc_id*} - Record key
name	list()	The sublocation name	-
devlist	list()	The list of devices deployed in the sublocation	A redundancy introduced to enhance read performance

device Record Definition			
Element	**Type**	**Description**	**Note**
dev_id	unique integer() > 0	The device unique identifier	Record key
name	list()	The device name	-
sub_id	tuple()	The sublocation the device is deployed in	- Expressed as {*loc_id,subloc_id*} - Indexed element
type	atom()	The device's type	Must belong to a predefined set of *allowed device types*
config	tuple()	The device last configuration, or state	Expressed as a tuple containing its traits' values
lastupdate	integer() > 0	The timestamp in Unix time of the device's latest configuration	-
hostname	list()	The device's *nodes host*	Must belong to a predefined set of *allowed nodes hosts*

suploc Record Definition			
Element	**Type**	**Description**	**Note**
loc_id	unique integer() > 0	The location unique identifier	Record key
sup_pid	pid()	The PID of the location's 'sup_loc' supervisor	Dynamic naming registry purposes

ctrmanager Record Definition			
Element	**Type**	**Description**	**Note**
loc_id	unique integer() > 0	The location unique identifier	Record key
mgr_pid	pid()	The PID of the controller node's manager	Dynamic naming registry purposes
status	list()	The controller node's status	Possible values: "BOOTING", "CONNECTING", "ONLINE" (no entry = "NOT_STARTED")

devmanager Record Definition			
Element	**Type**	**Description**	**Note**
dev_id	unique integer() > 0	The device unique identifier	Record key
mgr_pid	pid()	The PID of the device node's manager	Dynamic naming registry purposes
status	list()	The device node's status	Possible values: "BOOTING", "CONNECTING", "ONLINE" (no entry = "NOT_STARTED")

Most of the CRUD operations on the Mnesia database have been implemented using *transactions*, and the complete list of database API offered to the JANET Simulator application can be found in the accompanying *user guide* document.

OTP Processes Description

An overview of the processes used in the OTP tree of the JANET Simulator application, along with their associated OTP behaviours, is presented below:

locs_init

This transient process is used for spawning the locations' 'sup_loc' supervisors during the JANET Simulator start-up sequence.

loc_devs_init

This transient process is used for spawning the *dev_manager* processes of all devices belonging to the location when its 'sup_loc' supervisor is started.

ctr_manager/dev_manager *[gen_server]*

As previously discussed in the system start-up sequence and nodes' management chapters, these are the JANET Controller and JANET Device nodes manager processes, which create and maintain the environment overlay network by starting their associated nodes on their designated *nodes hosts*, restarting them should they stop and crash and stopping them when simulating their physical disconnection from their LANs, when their associated location or device is deleted, or when the simulation environment is shut down.

Each manager is *linked* with its node and, following its *registration*, *monitors* its process dedicated to exchanging data with the JANET Simulator (in particular, the *ctr_simserver* process for controller and the *dev_server* process for device nodes), and managed nodes are in fact started by using transient *node_starter* processes for the purpose of enabling their parent managers to service events coming from within the JANET Simulator (notably, *shutdown* exit signals received from their 'sup_loc' location supervisors).

After their registration, the information exchanged between the managers and their associated nodes includes:

- Receiving node statuses updates, which are mirrored in their associated entries in the *ctrmanager* and *devmanager* tables.

- Controller managers perform on the JANET Simulator database the operations requested by the controllers' *ctr_resthandler* processes routed via the *ctr_simserver* processes, returning their results.
- Device managers receive *device state updates* with their relative timestamps from their devices' *dev_server* processes, mirroring them in their associated entries in the *device* table.

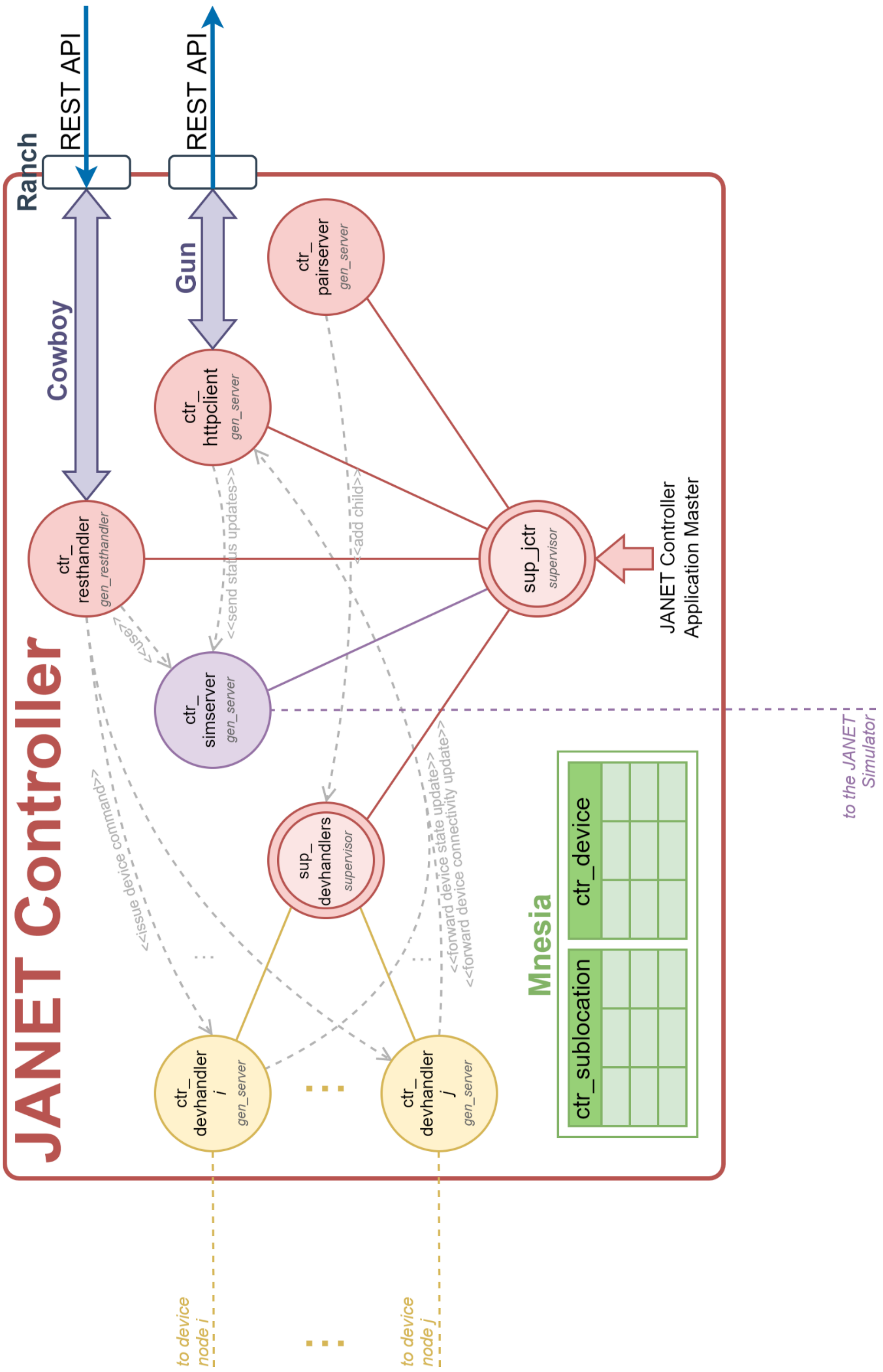
sim_hostsmonitor *[gen_statem]*

This process was introduced to experiment with advanced features that were later deferred to future application developments, and in its current incarnation, after determining the set of *remote hosts* used by the simulation environment by removing from the *nodes hosts* and the remote REST server all names and addresses mapping to the *localhost*, monitors their connectivity statuses by periodically *pinging* them via the underlying OS, reporting any connectivity status change to the user via logging.

sim_resthandler *[gen_resthandler]*

This process implements the JANET Simulator REST server handling logic by performing on the Mnesia database the operations requested by the remote REST client and returning their results, where the complete interface offered by the REST server can be found in the accompanying *JANET Simulator REST Interface* document.

It should also be noted that, in order to reuse the same handling logic for different REST paths, the process was implemented as a callback module of a custom *gen_resthandler* behaviour representing a bridge between the handling logic and the *Cowboy* HTTP server.


JANET Controller
Ranch
REST API
REST API
Cowboy
Gun
ctr_resthandler
gen_resthandler
ctr_httpclient
gen_server
ctr_pairserver
gen_server
sup_jctr
supervisor
JANET Controller Application Master
ctr_simserver
gen_server
sup_devhandlers
supervisor
ctr_devhandler i
gen_server
ctr_devhandler j
gen_server
<<use>>
<<send status updates>>
<<add child>>
<<issue device command>>
<<forward device state update>>
<<forward device connectivity update>>
Mnesia
ctr_sublocation
ctr_device
to the JANET Simulator
to device node i
to device node j

OTP Supervisors

The configuration and children specifications of the supervisors used in the JANET Controller OTP tree are outlined below:

Name	Role	Strategy	Intensity	Period
sup_jctr	JANET Controller root supervisor	*one_for_one*	0	1
sup_devhandlers	Device handlers supervisor	*simple_one_for_one*	0	1

sup_jctr Children Specification					
ChildID	Description	Multiplicity	Type	Restart	Shutdown
sup_devhandlers	Device handlers supervisor	1..1	*supervisor*	*permanent*	5000
ctr_simserver	Simulation server	1..1	*worker*	*permanent*	4000
ctr_resthandler	REST server handler	1..1	*worker*	*transient*	5000
ctr_httpclient	REST Client	1..1	*worker*	*permanent*	5000
ctr_pairserver	Devices Pairing Server	1..1	*worker*	*permanent*	1000

sup_devhandlers Children Specification					
ChildID	Description	Multiplicity	Type	Restart	Shutdown
ctr_devhandler_*i*	Device Handler	0..M	*worker*	*temporary*	4800

As previously discussed in the nodes' management chapter, as a measure for preventing data inconsistencies in the simulation environment, the *intensity* (or *restart intensity*) of every supervisor in the JANET Controller OTP tree has been set to '0' so as to allow, in case of errors, the entire node to be shut down and re-initialized from the data in the JANET Simulator master database.

Mnesia Database

As a measure for reducing the bottleneck associated with redirecting every *read* and *write* operation to the JANET Simulation Mnesia database, also allowing for a more accurate modelling of the simulated equipment, JANET Controllers were provided with an in-memory cut-down replica of its database limited to the functional information relative to their location, which is initialized from the data passed by their controller managers when the nodes are started to then evolve autonomously, where synchronization with the master database is implemented via explicit message passing (introducing an *eventual consistency* aspect between the JANET Simulator and JANET Controller database) and where inconsistencies are handled, as previously discussed, by shutting down and re-initializing the controller node and its Mnesia database from the data in the JANET Simulator master database.

The structure of the JANET Controller Mnesia database in terms of tables, their records definitions and relationships is presented below:

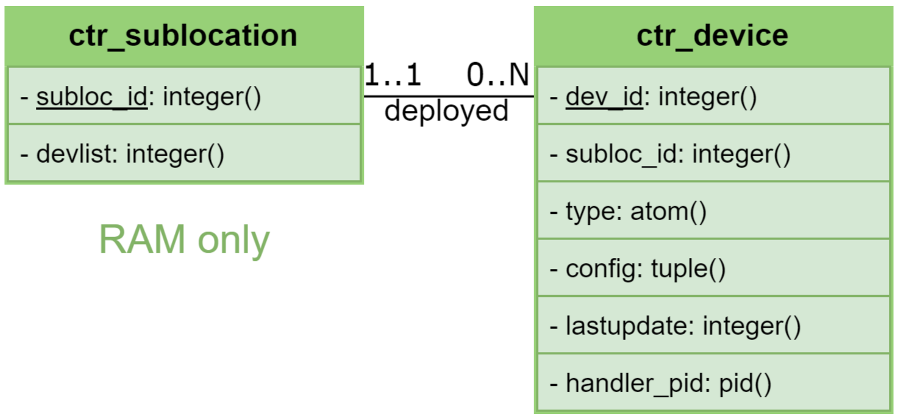


Table	Type	Description	Stored in	Notes
ctr_sublocation	*set*	Sublocation information	RAM Only *(ram_copies)*	-
ctr_device	*set*	Device information	RAM Only *(ram_copies)*	-

ctr_sublocation Record Definition			
Element	Type	Description	Note
<u>subloc_id</u>	unique integer() ≥ 0	The sublocation identifier within the location	- The *default sublocation* "0" cannot be deleted or renamed - Record key
devlist	list()	The list of devices deployed in the sublocation	A redundancy introduced to enhance read performance

ctr_device Record Definition			
Element	Type	Description	Note
<u>dev_id</u>	unique integer() > 0	The device unique identifier	Record key
subloc_id	integer() ≥ 0	The sublocation within the location the device is deployed in	-
type	atom()	The device's type	Must belong to a predefined set of *allowed device types*
config	tuple()	The device latest configuration, or state	Expressed as a tuple containing its traits' values
lastupdate	integer() > 0	The timestamp in Unix time of the device's last configuration	-
handler_pid	pid()	The PID of the device's handler process	Dynamic naming registry purposes

Most of the CRUD operations on the Mnesia database have again been implemented using *transactions*, and in general represent cut-down versions of their equivalents in the JANET Simulator Mnesia database.

OTP Processes Description

An overview of the processes used in the OTP tree of the JANET Controller application, along with their associated OTP behaviours, is presented below:

ctr_devhandler *[gen_server]*

As previously introduced in the system start-up sequence these device handler processes are dynamically started under their *sup_devhandlers* supervisor by the *ctr_pairserver* process whenever a device *pairs* with its location's controller, and are used for exchanging information between the controller and the device's *dev_server* process, including:

- Forwarding to the *dev_server* the device commands requested by the *ctr_resthandler* process.
- Receiving from the *dev_server device state updates* with their relative timestamps, which may be:
    - *Synchronous*, if received as a response to a device command, which are returned to the *ctr_resthandler* process.
    - *Asynchronous*, due to the device's state machine autonomously changing its state, which are delivered to the *ctr_httpclient* process for them to be forwarded to the remote REST server.

  It should also be noted that, as a measure for saving on transmission bandwidth and processing resources on the cloud tier, the device states delivered to the *ctr_resthandler* or *ctr_httpclient* processes include only the traits that have changed from the device's latest known state in the Mnesia database, possibly none (in which case asynchronous state updates are silently dropped), changes that are then reflected in the database.

When created, a device handler also establishes a *monitor* towards its associated *dev_server* process, which is used for terminating it should the *dev_server* (and consequently its device node) stop, with the creation and termination of device handlers being also used for sending *device connectivity updates* to the remote REST server via the *ctr_httpclient*.

ctr_simserver *[gen_server]*

This process is used for exchanging information between the controller node and its *ctr_manager* process in the JANET Simulator, including:

- *Registering* the controller during its start-up sequence by sending its PID to the *ctr_manager* process.
- Performing on the Mnesia database the operation requested by the *ctr_manager* to keep it synchronized with the JANET Simulator master database.
- Forwarding to the *ctr_manager* the operations to be performed on the JANET Simulator database requested by the *ctr_resthandler* process.
- Forwarding to the *ctr_manager* the controller status updates received from the *ctr_httpclient* process when it establishes a connection or disconnects from the remote REST server (corresponding to the "ONLINE" and "CONNECTING" statuses respectively).

ctr_resthandler *[gen_resthandler]*

This process implements the JANET Controller REST server handling logic, where:

- To ensure data consistency, requests aimed at changing the controller's configuration (e.g. *add_sublocation*, *delete_device*, etc.) are first executed on the JANET Simulator database by routing them via the *ctr_simserver* process to the controller's manager, and depending on their results:
    - If they are unsuccessful, their associated error is returned to the REST client without attempting them on the JANET Controller database.

    - If they are successful, they are attempted on the JANET Controller database, where errors and their associated data inconsistencies in this case are *concealed* from the REST client by returning the success of the operation, after which the controller node is silently shut down and reinitialized with the data from the JANET Simulator master database.

- Device commands are routed towards their target devices via their associated *ctr_devhandler* processes, with the handling logic supporting multiple commands being issued within the same REST requests, which are forwarded in parallel to their destinations through the use of a transient *cmdclient* process performing non-blocking *gen* calls (being a synchronous *multicall* towards processes with different registered names not directly offered by the *gen_server* behaviour), and once all replies have been collected (or a predefined timeout expires), commands are classified as:
    - *Successful*, if they were accepted and replied by the device with a *device state update* with its associated timestamp.
    - *Unsuccessful*, if they were rejected by the device, which occurs if the command would have put it in an invalid state.
    - *Invalid*, if they could not be forwarded in the first place, for reasons such as invalid syntax, targeting a device not belonging to the location or currently not paired with the controller, and so on.

  Each individual response with its result is then aggregated and returned to REST client within a single HTTP response.

As for the JANET Simulator REST server, in order to reuse the same handling logic for different REST paths, the *ctr_resthandler* process was implemented as a callback module of a custom *gen_resthandler* behaviour representing a bridge between the handling logic and the *Cowboy* HTTP server, and the complete interface offered by the JANET Controller REST server can be found in the accompanying *JANET Controller REST Interface* document.

ctr_httpclient *[gen_server]*

This process uses the *Gun* HTTP client for establishing and maintaining a persistent connection with the remote REST server and:

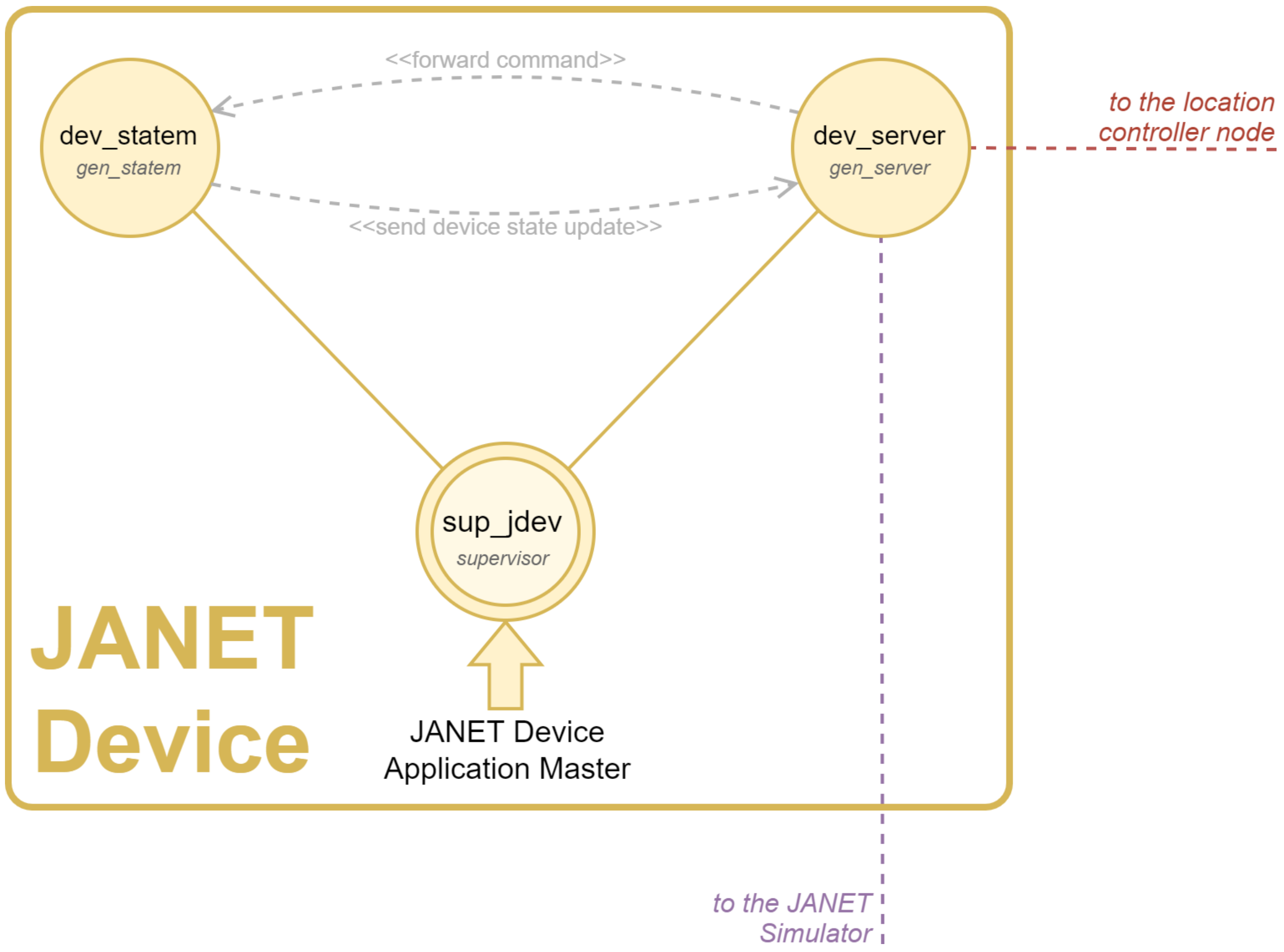
- Forwards any connection state change with the remote REST server (associated with the "CONNECTING" and "ONLINE" controller node statuses) to the controller manager in the JANET Simulator via the *ctr_simserver* process.
- Forwards to the remote REST server the device state and connectivity updates with their relative timestamps received from *ctr_devhandler* processes, postponing them through the use of backlogs while not connected.

ctr_pairserver *[gen_server]*

As previously mentioned in the system start-up sequence this process, whose locally registered name is known in advance to device nodes, is contacted by their *dev_server* processes to *pair* the devices with their location's controller by passing the device ID and the *dev_server* PID, where:

- If the device is recognized as belonging to the location and its associated *ctr_devhandler* is not running, such handler is spawned under the *sup_devhandlers* supervisor and its PID returned to the caller.
- If the device is recognized as belonging to the location but its associated *ctr_devhandler* is already running (which is checked via the *handler_pid* element in the *ctr_device* entry associated with the device), its PID is returned to the caller.
- If the device is not recognized as belonging to the location (which is relative to the fact that no entry with such device ID is present in the *ctr_device* table), the pairing request is rejected by returning an error to the caller.

JANET Device Architecture



OTP Supervisors

The configuration and children specifications of the only supervisor used in the JANET Device OTP tree are outlined below:

Name	Role	Strategy	Intensity	Period
sup_jdev	JANET Device root supervisor	*one_for_all*	0	1

***sup_jdev* Children Specification**					
ChildID	**Description**	**Multiplicity**	**Type**	**Restart**	**Shutdown**
dev_statem	Device state machine	1..1	*worker*	*permanent*	2000
dev_server	Device communication server	1..1	*worker*	*permanent*	4500

As previously discussed in the nodes' management chapter, as a measure for preventing data inconsistencies in the simulation environment, the *intensity* (or *restart intensity*) of the JANET Device root supervisor has been set to '0' so that, in case of errors, the entire node can be shut down and re-initialized from the data in the JANET Simulator master database.

OTP Processes Description

An overview of the processes used in the OTP tree of the JANET Device application, along with their associated OTP behaviours, is presented below:

dev_statem *[gen_statem]*

This process implements the state machine associated with the device's *type* which, as previously mentioned, determines the set of *traits* comprising its *state*, or *configuration*, with the full list of supported device types with their traits can be found in the JANET Service specification.

As previously introduced, during its execution the state machine's configuration can evolve:

- *Synchronously*, upon receiving from the *dev_server* process a command requested by the location controller's REST server (or, alternatively, from the JANET Simulator), which is applied provided it does not bring the device in an invalid state (e.g. the {*open*,*lock*} state for *door* devices).
- *Asynchronously*, when the machine autonomously changes its state, which occurs:
    - When simulating the user physical interaction with the device, a feature that is implemented via a recurrent *simulated_activity_timer* triggering at normally distributed random intervals (and, for an easier system verification, after a minimum time interval has passed from the last received command), where at its activations each of the device traits is assigned a given probability of changing value, which is affected by factors such as its current value (e.g. a *thermostat temp_target* cannot increase if already at its maximum), the values of other traits (e.g. a *fan* may change its *fanspeed* only if it is *on*), and the hour of day (e.g. *a light* has a higher probability of turning *on* in the evening than during the afternoon hours).
    - When simulating a change in the device's temperature sensor, corresponding to its *temp_current* trait (*thermostat* and *conditioner* devices only), which again is implemented via a recurrent *ambient_temperature_update_timer* triggering at normally distributed random intervals, whose mean and variance are reduced by a factor proportional to the distance between the device's current ambient temperature and its *equilibrium temperature*, which corresponds to:
        - If the device is *on*, the value of its *temp_target* trait.
        - If the device is *off*, a predefined value depending on the hour of day.

      Where at each timer activation the *temp_current* trait is assigned a probability of drifting of 1 degree towards the equilibrium temperature (or oscillating around it once reached)..

At every configuration change the state machine replies (*synchronous*) or sends (*asynchronous*) to the *dev_server* process a *device state update* message specifying its complete state with its associated timestamp expressed in UNIX time.

Finally, as an additional measure for enhancing data consistency and fault-tolerance in the overall JANET service, after a predefined *inactivity timeout* from the last message, state machines automatically send a *device state update* with their current (and unaltered) configuration to the *dev_server.*

dev_server *[gen_server]*

This process is used for exchanging data between the device node and its *dev_manager* process in the JANET Simulator and its assigned *ctr_devhandler* process in the location's controller, connections that as previously discussed in the system start-up sequence are respectively established by the process:

- *Registering* the device by sending its PID to the *dev_manager* process in the JANET Simulator.
- *Pairing* the device with its location's controller, an operation that, to allow the *dev_server* to service other events and messages, is carried out in fact by a transient *ctr_pairer* process which continuously attempts to contact the controller's *ctr_pairserver* process passing the device ID and the PID of its *dev_server* parent, returning to the latter, once successfully paired, the PID of

the *ctr_devhandler* process assigned to the device on the controller, which is in turn *monitored* by the *dev_server* for the purpose of respawning the *ctr_pairer* process should the device handler (and consequently the controller node) stop.

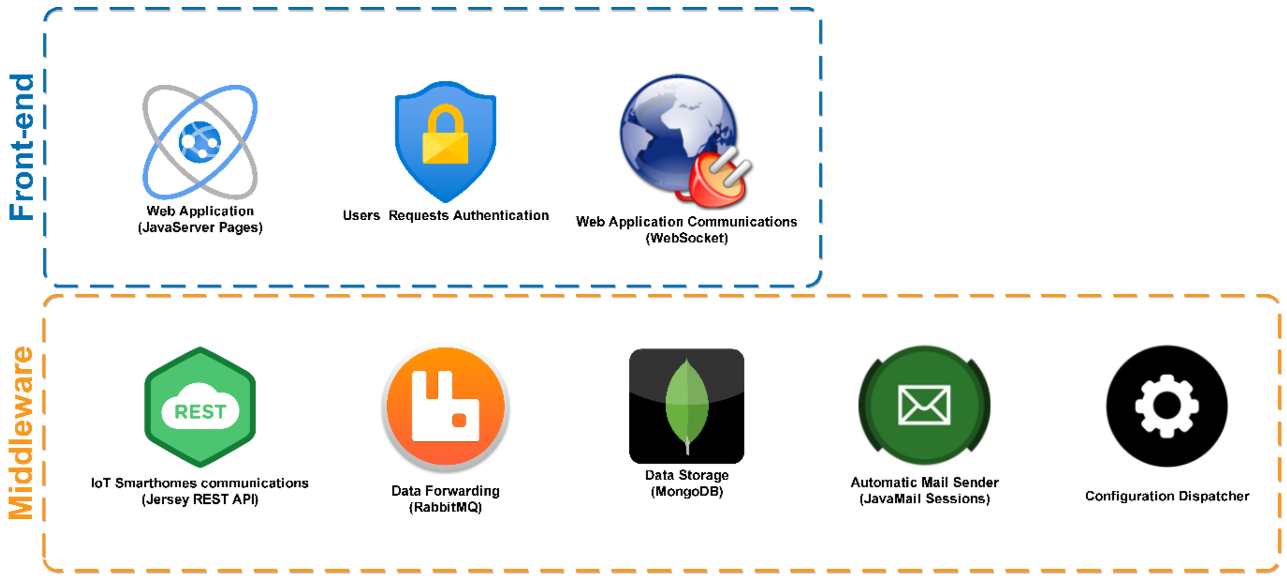
The information routed between nodes by the *dev_server* process includes:

- Receiving commands issued to the device, which are forwarded to the *dev_statem* process.
- Forwarding *device state updates* sent by the *dev_statem* process, whether *synchronously* as a response to a command or *asynchronously* due to the state machine autonomously changing its state, to both the *dev_manager* and the *ctr_devhandler* processes, postponing through the use of a backlog the updates destined to the latter while the device is not paired with its location's controller.
- Informing the *dev_manager* whenever the device pairs or unpairs from its location's controller (corresponding respectively to the "ONLINE" and "CONNECTING" device node statuses).

JanetHome Service

The service Janethome is developed as a distributed java enterprise architecture supported by the Glassfish application server. The service consists of two independent layers:

- A **Front-end layer** to implement the Presentation of our service. The layer is based on a JSP-based web server which permits to the users to see the information maintained and allow control and maintenance of their smarthomes
- A **Middleware layer** with the task of providing a standard API interface to the front-end to be able to communicate with the various components of the service in a transparent manner. The layer also gives methods to interact with the **Back-end** layer(IoT smarthome networks) through a REST-based communication


Front-end
Web Application (JavaServer Pages)
Users Requests Authentication
Web Application Communications (WebSocket)
Middleware
REST
IoT Smarthomes communications (Jersey REST API)
Data Forwarding (RabbitMQ)
Data Storage (MongoDB)
Automatic Mail Sender (JavaMail Sessions)
Configuration Dispatcher

Front-end Functionalities

The functionalities of the front-end can be summarized as:

- Web Application: A set of Java Server Pages and Servlets to implement the presentation of the service and offer to the users the ability to view their smarthome, modify it and give commands to their devices
- Users Requests Authentication: To ensure the confidentiality of user information, a login is required to access the features offered by the service. Each request must also be attached to a valid authentication token to be processed by the service
- Web Application Communications: The user pages require an asynchronous messaging for updating their displayed smarthome with the real-time updates coming from devices. For this reason, a mechanism based on WebSocket has been developed. This allows users to send requests to the service and update in real time the status of their smarthomes without the need to refresh their displayed page

Middleware Functionalities

The functionalities of the middleware can be summarized as:

- IoT Smarthome Communications: To communicate with remote smarthomes we have adopted a REST communication. It is therefore present a special form charged to send REST requests and receive asynchronous updates of devices to be distributed among the components of the service
- Data Forwarding: Much information needs to be distributed between various components of the system, to reduce delays due to Lock Contention we have adopted a message passing mechanism of publisher-subscribe type based on RabbitMQ. This allows us to enjoy the benefits of spatial de-coupling of endpoints being able to send updates to an unpredictable number of destinations while maintaining a high degree of control over them
- Data Storage: As storage service we have adopted MongoDB, a document-based database. This above all thanks to its high tolerance to the mutability of the data and the possibility to carry out operations of aggregation and map reduce in order to elaborate statistics on the data of the devices' sensors to be presented to the users
- Automatic Mail Send: Some requests that users can make on the service require the sending of automatic emails for verification purposes (user registration/ password change). A module has been implemented that allows to send mails in a simple and reusable way, choosing between a set of preformatted mails based on html
- Configuration Dispatcher: Almost all the components of the service are configurable, it is therefore entered a single internal service that they can use to obtain their configuration files

Front-end Layer

The front-end level is based on a JSP Web Server, the server consists of the following pages through which our service can delivery its functionalities:

Name	Role	GET Parameters
login.jsp	It's the home page of the service and allows you to access the secondary public pages (registration.jsp/password.jsp) and the private page (webapp.jsp) through a login form. Also through the JSP logic the server is able to verify the presence of any valid authentication credentials and redirect the user directly to the private page	NO PARAM
registration.jsp	Page containing a form for registering a new user	Type = 0
registration.jsp	Page informing the user of the sending of an email to his/her mailbox	Type = 1
registration.jsp	Page confirming the success of the account registration	Type = 2
registration.jsp	Page plotting an error condition to the user	Type > 2
password.jsp	Page for start the change password process by inserting a valid email registered into the service	Type=0
password.jsp	Page for confirm correct request executed	Type=1
password.jsp	Page for concluding the password change by inserting the new password for the account	Type=2
password.jsp	Page for notify an error occurred during any password change phase	Type > 2
webapp.jsp	Main page of the service, permits to manage and see the user smarthome and to interact with it. Through JSP logic it verifies if the user has the authorities for using the page and in case redirect it to the login page	NO PARAM

We can see that for password.jsp and registration.jsp there are several entries, this is because they consists in different pages extracted by the same page using JSP compile pre-processing based on the given type.

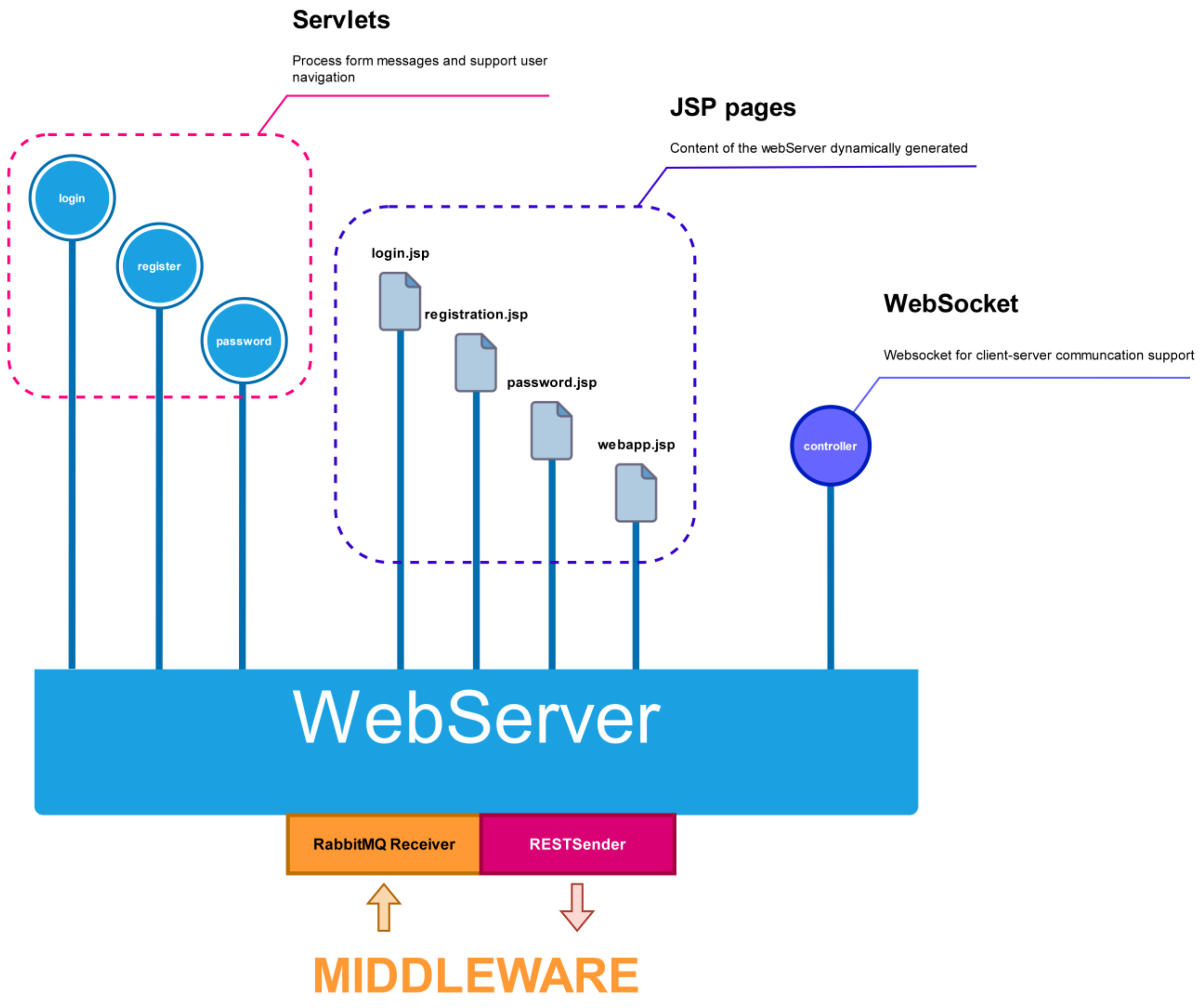
In support of JSP pages there are a set of servlets with the task of receiving the information sent by the forms and providing automatic redirection features:

Name	Role	Position
LoginServlet	The servlet has two main functionalities: • If visited verifies the user credentials and basing of them redirect it to the login.jsp or webapp.jsp • Verification of the login form data	/login
PasswordServlet	The password servlet has two main functionalities: • Verifies the email inserted into the password change form and eventually send an email • Complete the password change method verifying a given token and redirect the user	/password
RegistrationServlet	The servlet verifies has two main functionalities: • Verifies the information inserted are valid and eventually send an email	/registration

	• Complete the registration verifying a given token and redirect the user	
WebappServlet	Support servlet, verifies the user authentication and redirect it to the webapp.jsp or login.jsp pages	/webapp

Finally to permit to communications between the web clients and the service we have three communications channels:

- webSocket: to permit to the webclients to make requests and receive updates without having to refresh their page
- restInterface: given by the RESTsender bean into the Middleware permits to the WebServer to propagate the requests to the back-end layer
- RabbitUpdater: a RabbitMQ input interface for receiving all the updates which involve the user and propagate them to the webclients via the webSocket

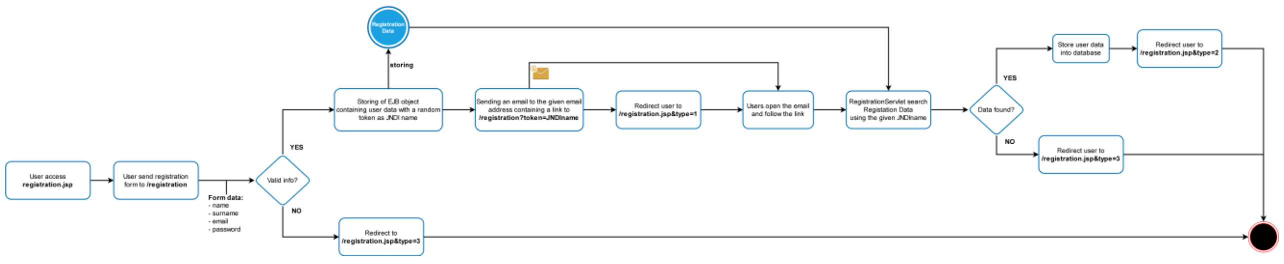


Web Server Behaviour

The following paragraph will show the main interactions with the system and how the web server manages it to generate the final behaviour.

User Registration

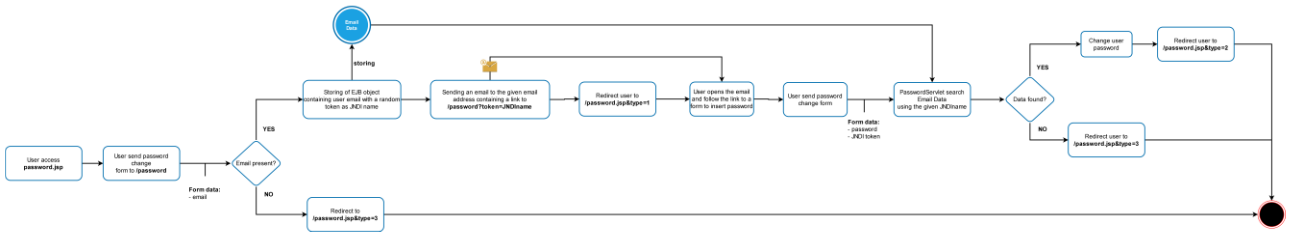
The registration of a new user follows the algorithm shown in the figure. After a user has entered all the information for their account the service verifies the email is not already registered and all fields are valid. If successful, send an email to the address provided by the user containing a link authenticated by the server. This way only the email owner can confirm the registration and we have the verification that it exists and that it is the owner.



User access registration.jsp
User send registration form to /registration
Form data:
- name
- surname
- email
- password
Valid info?
YES
NO
Storing of EJB object containing user data with a random token as JNDI name
storing
Registration Data
Sending an email to the given email address containing a link to /registration?token=JNDIname
Redirect user to /registration.jsp&type=1
Users open the email and follow the link
RegistrationServlet search Registation Data using the given JNDIname
Data found?
YES
Store user data into database
Redirect user to /registration.jsp&type=2
NO
Redirect user to /registration.jsp&type=3
Redirect to /registration.jsp&type=3

Password Change

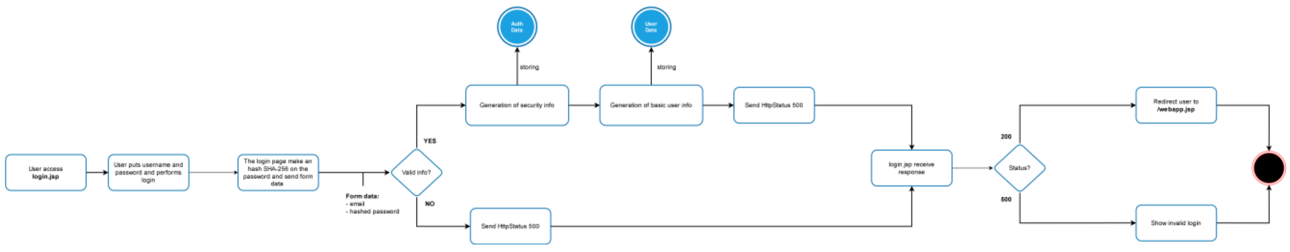
The password change follows the same rules as registering a new account, we need a verification that the person requesting the password change is actually the owner of the account. We then proceed to generate again private information associated with a random JNDI name that is provided to the user through an email.



User access password.jsp
User send password change form to /password
Form data:
- email
Email present?
YES
NO
Storing of EJB object containing user email with a random token as JNDI name
storing
Email Data
Sending an email to the given email address containing a link to /password?token=JNDIname
Redirect user to /password.jsp&type=1
User opens the email and follow the link to a form to insert password
User send password change form
Form data:
- password
- JNDI token
PasswordServlet search Email Data using the given JNDIname
Data found?
YES
Change user password
Redirect user to /password.jsp&type=2
NO
Redirect user to /password.jsp&type=3
Redirect to /password.jsp&type=3

Login

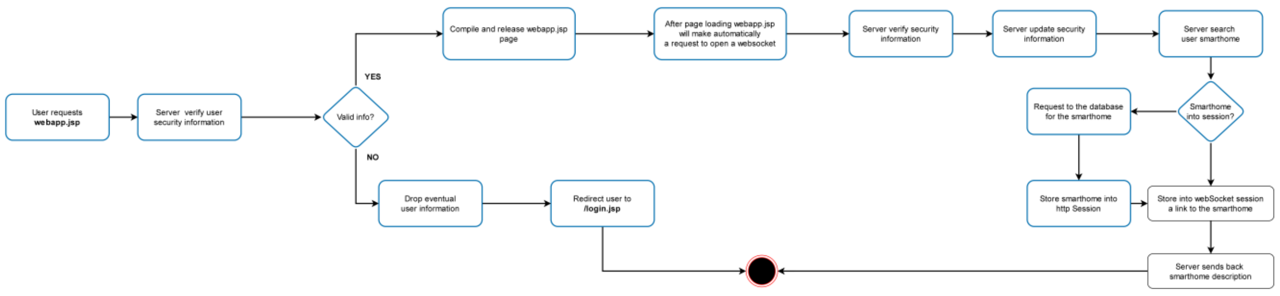
A user providing its username and password within the login form available in login.jsp can access all the service functionalities. The service after a successful login allocates the resources needed to auto-login, this allows users to access to the service without having to provide theirs credentials again. The safety mechanism will be explained in more detail in the next sections of the chapter.



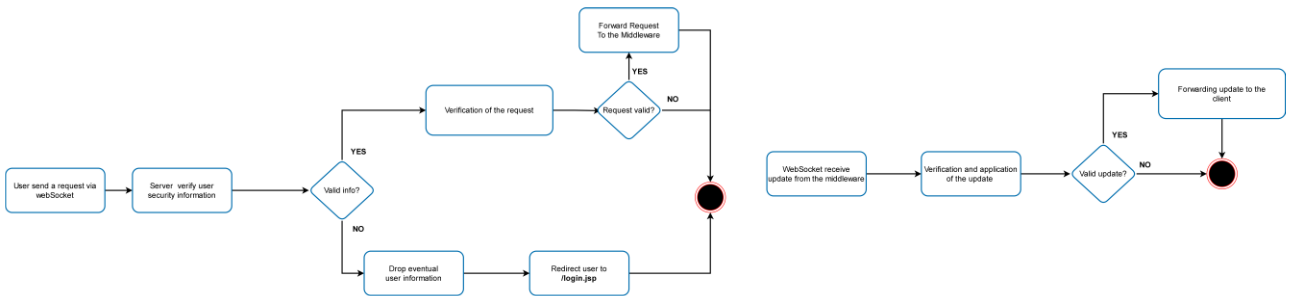
User access login.jsp
User puts username and password and performs login
The login page make an hash SHA-256 on the password and send form data
Form data:
- email
- hashed password
Valid info?
YES
NO
Generation of security info
storing
Auth Data
Generation of basic user info
storing
User Data
Send HttpStatus 500
Send HttpStatus 500
login.jsp receive response
Status?
200
Redirect user to /webapp.jsp
500
Show invalid login

WebApp Usage

After a login the client has all the information to make requests to the core of the service accessible via webapp.jsp, in the absence of this information the page is inaccessible and leads the user to be redirected to login.jsp. The use of webapp goes through several stages, In the first phase the user login and opening a websocket to the server gets a communication channel with which he can make his own requests to the service and get answers and periodic updates from the devices.

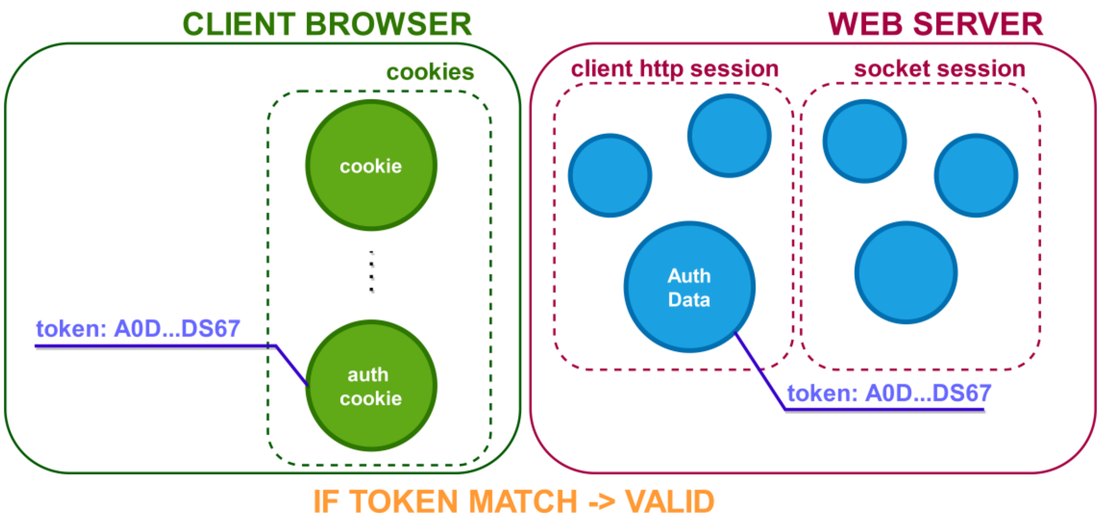

User requests webapp.jsp
Server verify user security information
Valid info?
YES
NO
Compile and release webapp.jsp page
After page loading webapp.jsp will make automatically a request to open a websocket
Server verify security information
Server update security information
Server search user smarthome
Smarthome into session?
Request to the database for the smarthome
Store smarthome into http Session
Store into webSocket session a link to the smarthome
Server sends back smarthome description
Drop eventual user information
Redirect user to /login.jsp

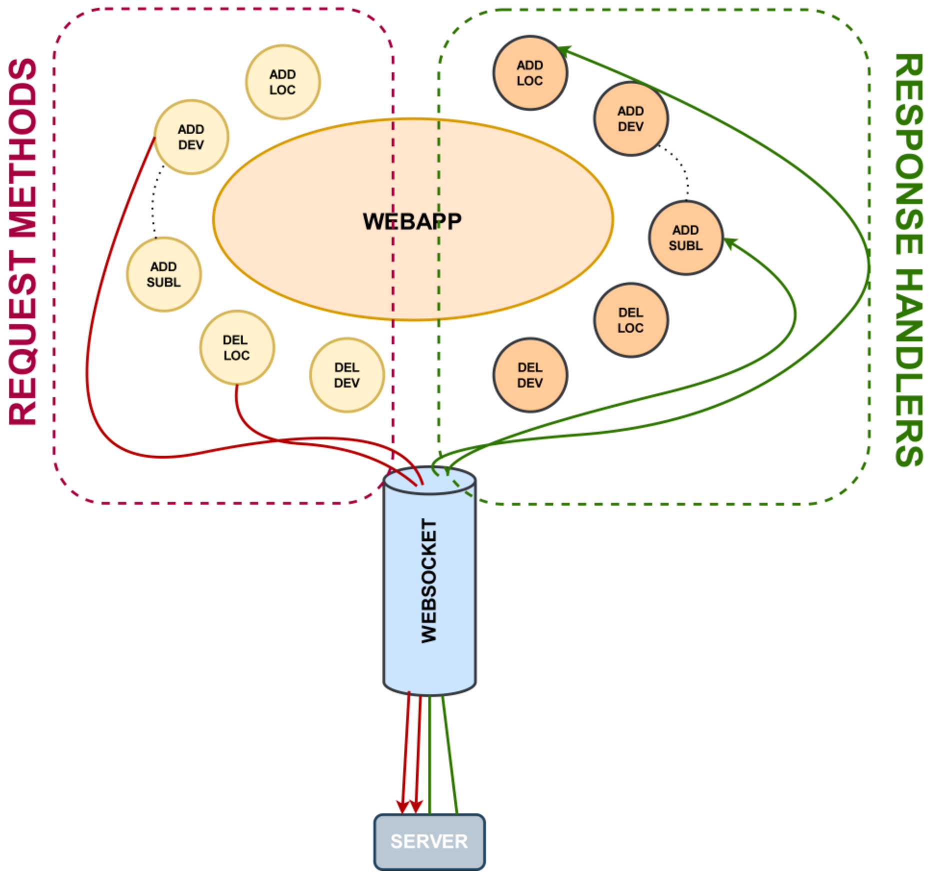
The confirmation message that the WebSocket was opened consists of the description of the user's smarthome, this description will be processed by the web page to go to generate and populate the structure on which the user can go to perform its requests. After opening the websocket the client and the server can talk. Every request of the user is verified using the authentication information and if valid is processed by the service asynchronously. Asynchronous processing means that a client following a request will never receive an answer immediately, the request will be processed by the system and once carried out will produce an update that will be propagated. This way we can use a single mechanism for asynchronous device updates and user requests.


User send a request via webSocket
Server verify user security information
Valid info?
YES
NO
Verification of the request
Request valid?
YES
NO
Forward Request To the Middleware
Drop eventual user information
Redirect user to /login.jsp
WebSocket receive update from the middleware
Verification and application of the update
Valid update?
YES
NO
Forwarding update to the client

Security

System security has been implemented by deriving a simple authenticity verification mechanism from the oAuth2.0 protocol used by all major systems today. The protocol requires https communication to be exploited and is based on generating information shared between client and server that is updated following each request. After a valid login, the system produces a random 120bit token that saves within the user's session and provides it to the client via cookies. Each time a client makes a request the server analyses the cookie and the token kept in the session, if they are both present and match then the user is allowed to perform the operation otherwise it is rejected and redirected to the login.jsp page.





WebSocket Communications

The communication between the client and the server takes place through a WebSocket that is used to generate asynchronous communication. In this, we can define the client as dump because its operations are limited to making demands to the server and to applying of the updates are they happened as a result of own demands or in independent way. The client has not granted any independent action on its data, it is an interface to make requests that will become part of the flow of the service. If your requests are successful sooner or later you will receive an update from the service derived from your request. To achieve this type of architecture its functions have been distinguished into two distinct branches, we have a set of functions dedicated to make requests and a second set of disjoint functions designed to manage the various types of updates that can be sent by the service. All the communications routines are defines into the connection.js file.

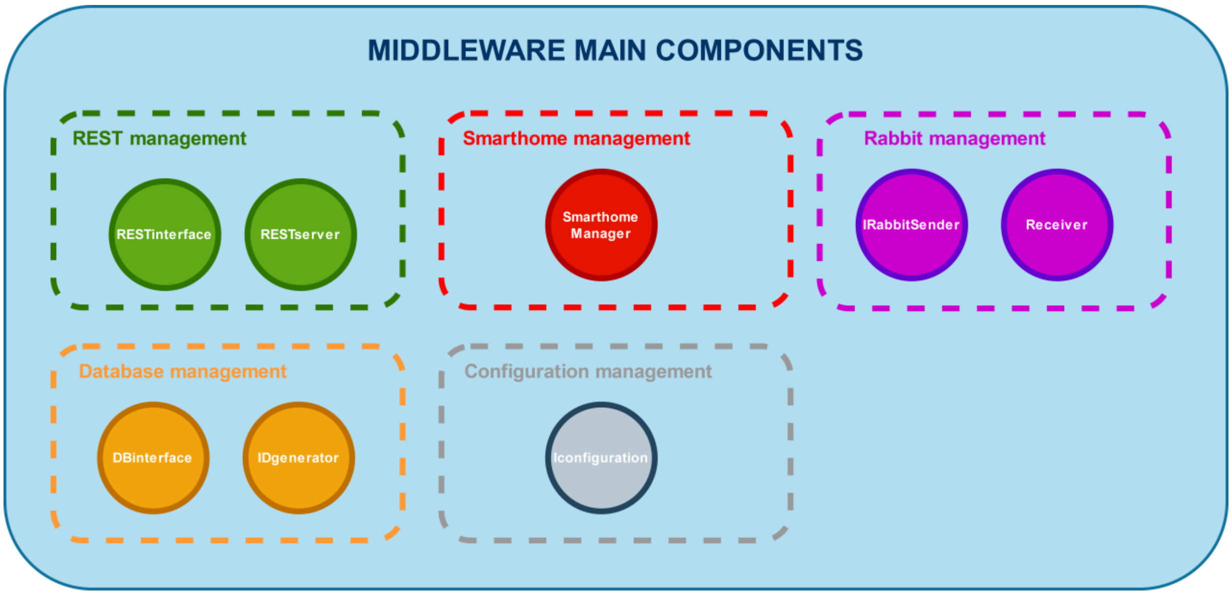
Communications

These are the messages supported by communication through the client and the server. Most messages can be used both to send requests and to receive an update.

Type	Fields	Direction
ADD_LOCATION	Location: name of the location to add Address: hostname or address of the controller Port: port of the controller	IN-OUT
RENAME_LOCATION	old_name: current name of the location new_name: new name to set on the location	IN-OUT
DELETE_LOCATION	location: name of the location to remove	IN-OUT
ADD_SUBLOCATION	location: name of the location in which deploy sublocation sublocation: name of the sublocation	IN-OUT
RENAME_SUBLOCATION	location: location in which search the sublocation old_name: current name of the sublocation new_name: new name to set on the sublocation	IN-OUT
DELETE_SUBLOCATION	location: location in which search the sublocation sublocation: name of the sublocation to remove	IN-OUT
ADD_DEVICE	location: location in which deploy the device sublocation: sublocation in which deploy the device name: name of the device to add type: device type [LIGHT,FAN,DOOR,AC_UNIT,THERMOSTAT]	IN-OUT
RENAME_DEVICE	old_name: current name of the device new_name : new name to set on the device	IN-OUT
DELETE_DEVICE	name: name of the device to remove	IN-OUT
CHANGE_SUBLOC	location: location in which move the device sublocation: sublocation in which move the device name: name of the device to move	IN-OUT
UPDATE	device_name: name of the device to update action: type of action to apply value: value connected with the action	IN-OUT
STATISTIC	device_name: name of the device requiring the statistic statistic: statistic type required start: time from which start the statistic end: time to end the statistic	IN-OUT
STATISTIC	device_name: device name requiring the statistic statistic: statistic type values: data to display	IN
LOGOUT	-	OUT
ERROR_LOCATION	-	IN
EXPIRED_AUTH	-	IN

Middleware Layer

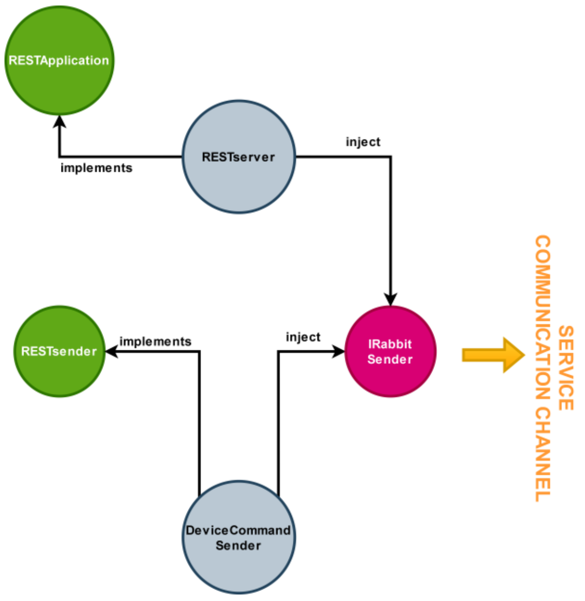
The middleware layer is responsible for providing the functionality exploited by the components of the service and to communicate the service with the back-end layer consisting of the various smarthomes made in Erlang technology. The main middleware components are the following:



- REST management:
  - RESTinterface: Remote EJB implementing a Jersey interface for sending messages to the backend layer giving a hostname and a port
  - RESTserver: Jersey REST server for receiving asynchronous communications from the back-end layer
- Smarthome management: A set of classes for implement the logic behind the IoT management. All the classes are managed by the SmarthomeManager which represents a local in-memory cached and portable database for a smarthome which permits to operate on it
- Rabbit management:
  - IRabbitSender: Remote EJB implementing a RabbitMQ sender for forward updates to all the components of the service
  - Receiver: Skeleton class for RabbitMQ message receival and management. Constitute an abstract class to generate ad-hoc receivers for all the components requiring it
- Database management:
  - DBinterface: Remote EJB implementing an interface for information storing and retrieval from a remote database
  - IDgenerator: Remote EJB implementing an interface for unique token generation
- Configuration management: The most of the components of the service requires a configuration. We have develop a special Remote EJB interface for retrieving a service of configuration dispatcher

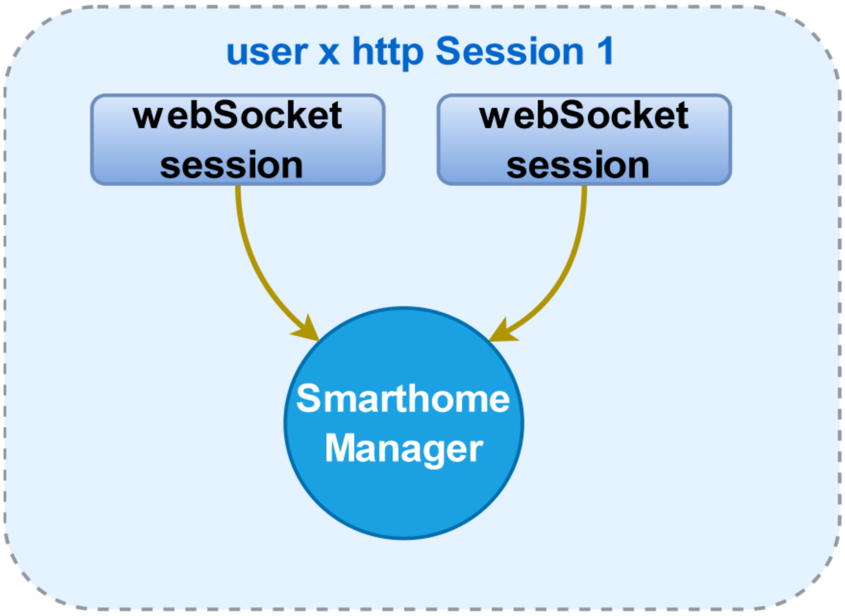
REST Management

The REST communication module consists of two main units, an interface for sending messages to the back-end and a REST server for receiving messages and routing them to service components. The RESTInterface interface is implemented by the DeviceCommandSender class that integrates an instance of the IRabbitSender interface to be able to send messages through the communication channel of the distributed service. The component is then responsible for sending the request to the back-end layer, analyse the response received and eventually acts to bring the operation to success. In case of success in automatic takes charge of the request and puts it into the distributed communication flow. Instead the RESTserver class is based on RESTApplication, the Jersey developing module for REST servers



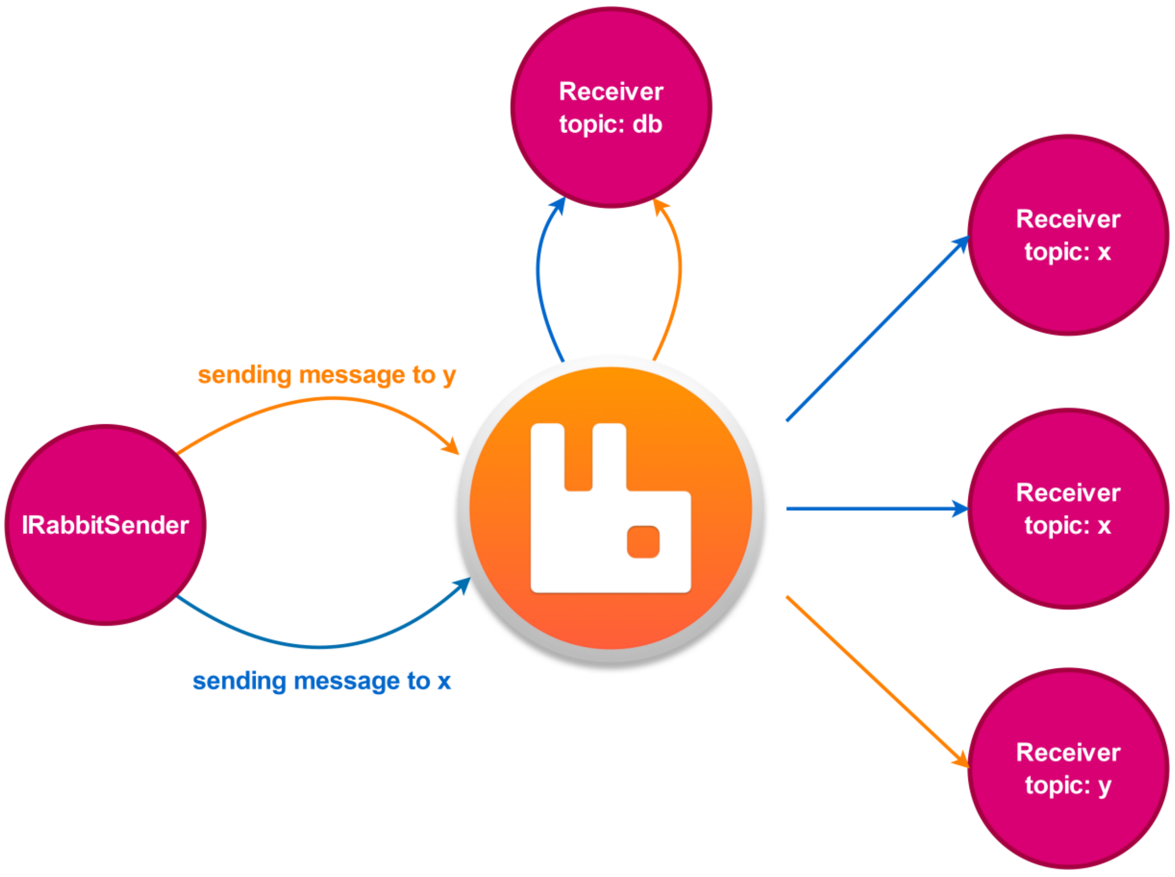
Smarthome Management

The IoT package of the service contains a set of classes that compose the SmarthomeManager, a structure composed of location consisting in turn of sublocation in which to deposit their devices. The smarthome is the central element of the service being the component in charge of verifying and applying updates to the data structure of the system. We made this decision because of the amount of requests that the service can have to handle is too heavy to rely only on the database but it is required some structure in memory, directly accessible from the components however, cause of this, every request transits from this component that, because of the mutual exclusion on it, risks to become a bottleneck of the system for Lock Contention. We therefore decided to adopt a strategy of spreading smarthome structures by relying on the service offered by RabbitMQ. The smarthomes are stored within the http sessions of users, so we do not completely avoid mutual exclusion (multiple webSockets can be opened in the same session and all will share the same smarthome structure) but it strongly limits the number of concurrent accesses to the price to increase the number of concurrent executions of the same demand in separate SmarthomeManager. Another element that we had to take into account in the development of the smarthome is its having to be idempotent to the requests received, the class being shared between multiple webSockets that could be opened at the same time, so it will receive copies of the same request multiple times. For this reason it has been implemented a mechanism of finding duplicate requests that automatically discards after its partial re-execution



RabbitMQ Management

For the generation of a communication channel between the various components of the service we decided to adopt RabbitMQ. Thanks to its high versatility and the spatial spacing that it introduces allows us to distribute requests in an easy way to the various components of the service involved. It also allows the various management elements to be made independent, this on the one hand allows us to completely eliminate the problem of mutual exclusion that is solved by generating various parallel and independent flows and on the other allows us to make very simple a future integration of new parts that would find an endpoint towards the service that allows it to easily get updates and process them to introduce new features. In the implementation of RabbitMQ we decided to use a topic-based exchanger, specifically using the user username as the topic. Each user opens a connection to the web server using a Receiver class allocates a receiver for RabbitMQ communications to which he assigns his username as a keyword. In this way any update related to a user will be reached by all open instances. A special topic consists of *"db"*, this is used by the database and guarantees the reception of any message sent regardless of its destination.



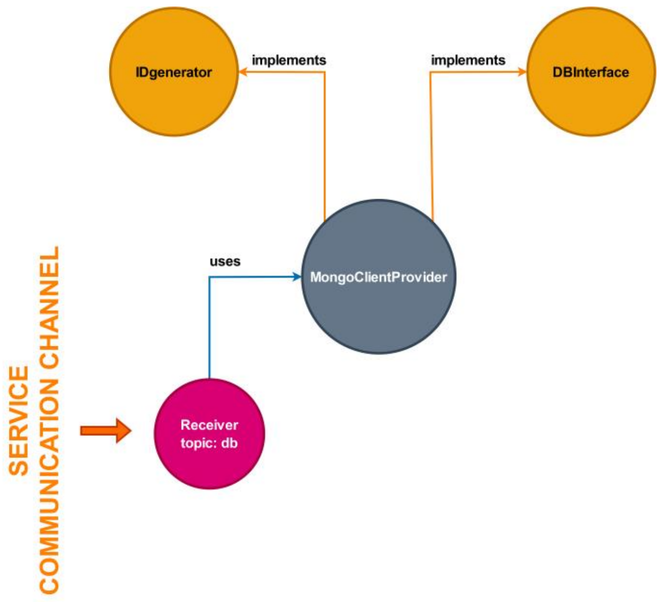
Messages

Type	Fields
ADD_LOCATION	• Location: name of the location to add • LocID: Unique identifier of the location • Address: hostname or address of the controller • Port: port of the controller
RENAME_LOCATION	• old_name: current name of the location • new_name: new name to set on the location
DELETE_LOCATION	• location: name of the location to remove
ADD_SUBLOCATION	• location: name of the location in which deploy sublocation • sublocation: name of the sublocation • sublocationID: Unique identifier of the subLocation
RENAME_SUBLOCATION	• location: location in which search the sublocation • old_name: current name of the sublocation • new_name: new name to set on the sublocation
DELETE_SUBLOCATION	• location: location in which search the sublocation • sublocation: name of the sublocation to remove
ADD_DEVICE	• location: location in which deploy the device • sublocation: sublocation in which deploy the device • Did: unique identifier of the device • name: name of the device to add • type: type of the device
RENAME_DEVICE	• Did: unique identifier of the device • old_name: current name of the device • new_name : new name to set on the device
DELETE_DEVICE	• Did: unique identifier of the device • name: name of the device to remove
CHANGE_SUBLOC	• Did: unique identifier of the device • location: location in which move the device • sublocation: sublocation in which move the device • name: name of the device to move
UPDATE	• dID: unique identifier of the device • action: type of action to apply • value: value connected with the action

Database Management

Although the requests are processed using local structures stored in memory it is still necessary to have a system of persistence of information. This is necessary to ensure the management of user accounts, statistics and the recovery of smarthomes in the event of a system reboot.
The database module is based on the MongoClientProvider class and consists of three elements:

- a DbInterface that allows the Webserver module to manage users (login/registration/change of password) and to initialize its http session (extraction of the user's smarthome)
- a IDgenerator that allows the service to generate unique dID, locationID using the available information into the persistent storage
- an instance of Receiver to receive all the updates that pass on the network and apply them also in the structures maintained in memory



Functionalities

The database takes care of numerous functions that are subtended by various sub-modules:

- Users: managed by the UserProvider class permits to make the following operation on the users:
  - Add a new user
  - Change the user password
  - Verify user presence by its email
  - Verification username/password match
- Smarthomes: managed by the SmarthomeProvider class, permits to make the following operations on the smarthomes:
  - Add a smarthome
  - Extract a smarthome
  - Update a smarthome
- Statistics: managed by the StatisticsProvider class, permits to make the following operations:
  - Add devices' sensor data
  - Make statistics of various type by aggregating the stored data
- IDs: managed by the IDgenerator class, permits to make the following operations:
  - Generate a unique dID
  - Generate a unique locID

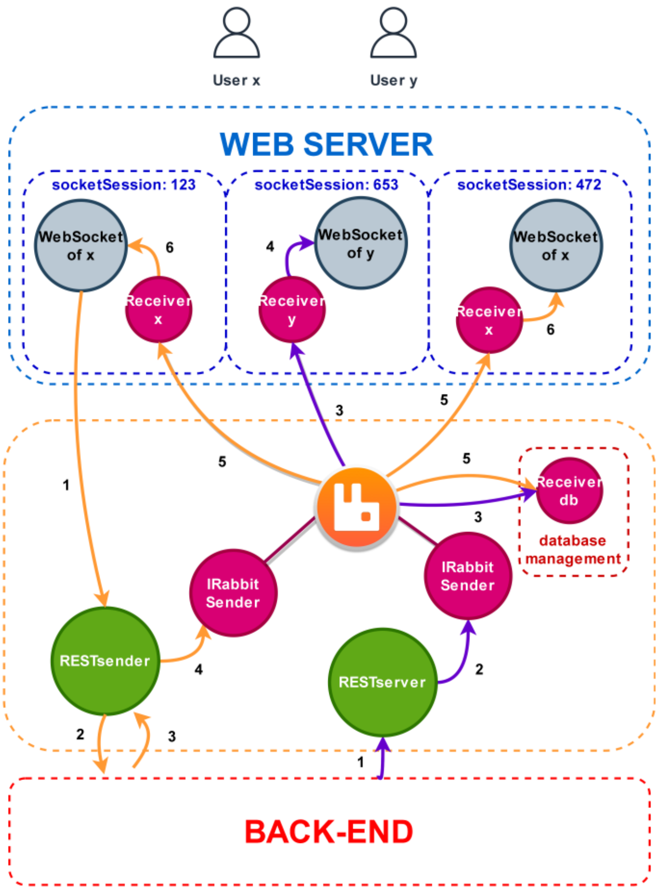
Configuration Management

Almost every component of the service requires a configuration to be able to function. To centralize the management of configurations we decided to implement a special EJB that provides the various components with their own configurations. The configurations are kept in the META-INF folder of the middleware module and consist of the following files:

Configuration	Parameters
rabbit.properties	• hostname: address of RabbitMQ exchange • port: port used by the RabbitMQ exchange • username: username used to access RabbitMQ • password: password used to access RabbitMQ
db.properties	• db_name: name of the database used • hostname: address of MongoDB database • port: port used by MongoDB database
mail.properties	• secure: if 0 use http otherwise if 1 http • port: port used by the webServer • hostname: hostname applied to the webserver • service: name of the service
rest.properties	• control_address: address associated to the Erlang simulator • control_port: port associated to the Erlang simulator

WebServer Request Management

Let's see an example to explain how requests are processed by the service.
In the example shown in the figure we have considered the processing of a user request and an asynchronous update of a device. As we can see for the same user can be present more webSocket sessions, this is due to the ability of the user to access the service from multiple pages simultaneously. Sessions can then be entered in the same http session or in different http sessions if the user uses the same device/browser or not.



User Request Management

1) The webServer from a WebSocket receives a new request from user x, after having verify it, in case of success forward it to a RESTsender
2) The RESTsender forward the request to the back-end
3) The back-end will respond to the request with a success/failure
4) After a success the RESTsender will forward the update to a IRabbitSender specifying the destination user
5) RabbitMQ will forward the request to all the involver components
6) The receivers placed into the WebServer receiving the update will apply it locally on their smarthome and in case of success forward it to the clients via the associated WebSocket

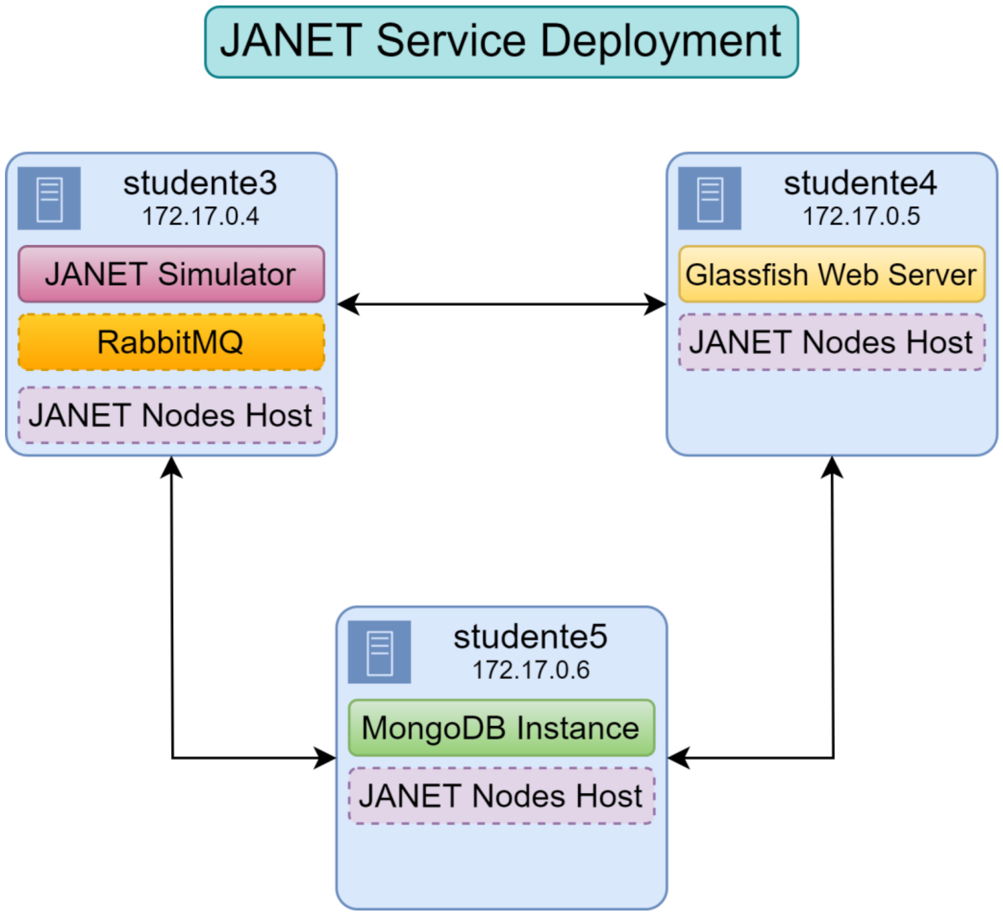
Update Request Management

1) The back-end forward to the service a new update from a device
2) The RESTserver received the request, verifies it and then forward it to a IRabbitSender specifying the destination user
3) RabbitMQ will forward the request to all the involved components
4) The receivers places into the WebServer receiving the update will apply it locally on their smarthome and in case of success forward it to the clients via the associated WebSocket

System Deployment

For demonstration purposes, the software components of the JANET Service were deployed in the following containers offered by the University of Pisa:

Host Name	IP Address	Software Components
studente3	172.17.0.4	JANET Simulator, JANET Nodes Host, RabbitMQ
studente4	172.17.0.5	Glassfish Web Server, JANET Nodes Host
studente5	172.17.0.6	MongoDB Database, JANET Nodes Host



The JANEThome service is available at the address http://janethome.zapto.org/WebServer/login (requires the Unipi VPN).