

Distributed Systems and Middleware Technologies  
The JANET Home Service

Table of Contents

[Introduction 2](#_Toc88032096)

[System Requirements 3](#_Toc88032097)

[Functional Requirements 3](#_Toc88032098)

[Non-Functional Requirements 4](#_Toc88032099)

[Working Hypotheses 4](#_Toc88032100)

[Actors and Use Cases Diagram 5](#_Toc88032101)

[Architectural Design 6](#_Toc88032102)

[Analysis Classes Diagram 6](#_Toc88032103)

[Classes Definitions 7](#_Toc88032104)

[Classes Attributes 8](#_Toc88032105)

[Enumerations Definitions 11](#_Toc88032106)

[Software Architecture 12](#_Toc88032107)

[Cloud Tier 13](#_Toc88032108)

[Local Tier 13](#_Toc88032109)

[Application Dataflow 14](#_Toc88032110)

[JANET Home Simulator 15](#_Toc88032111)

[Overlay Network Architecture 16](#_Toc88032112)

[Nodes Clusters 16](#_Toc88032113)

[Nodes Dataflow 16](#_Toc88032114)

[Nodes Hosts 17](#_Toc88032115)

[Operating Principles 18](#_Toc88032116)

[Start-up Sequence 18](#_Toc88032117)

[Nodes Statuses 18](#_Toc88032118)

[Nodes Management 19](#_Toc88032119)

[System Base Configuration 19](#_Toc88032120)

[JANET Simulator Architecture 20](#_Toc88032121)

[OTP Supervisors 21](#_Toc88032122)

[Mnesia Database 22](#_Toc88032123)

[OTP Processes Description 24](#_Toc88032124)

[JANET Controller Architecture 26](#_Toc88032125)

[OTP Supervisors 27](#_Toc88032126)

[Mnesia Database 27](#_Toc88032127)

[OTP Processes Description 29](#_Toc88032128)

[JANET Device Architecture 31](#_Toc88032129)

[OTP Supervisors 31](#_Toc88032130)

[OTP Processes Description 32](#_Toc88032131)

[Cloud Tier 34](#_Toc88032132)

[Titolo 2 34](#_Toc88032133)

[Titolo 3 34](#_Toc88032134)

[System Deployment 35](#_Toc88032135)

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:

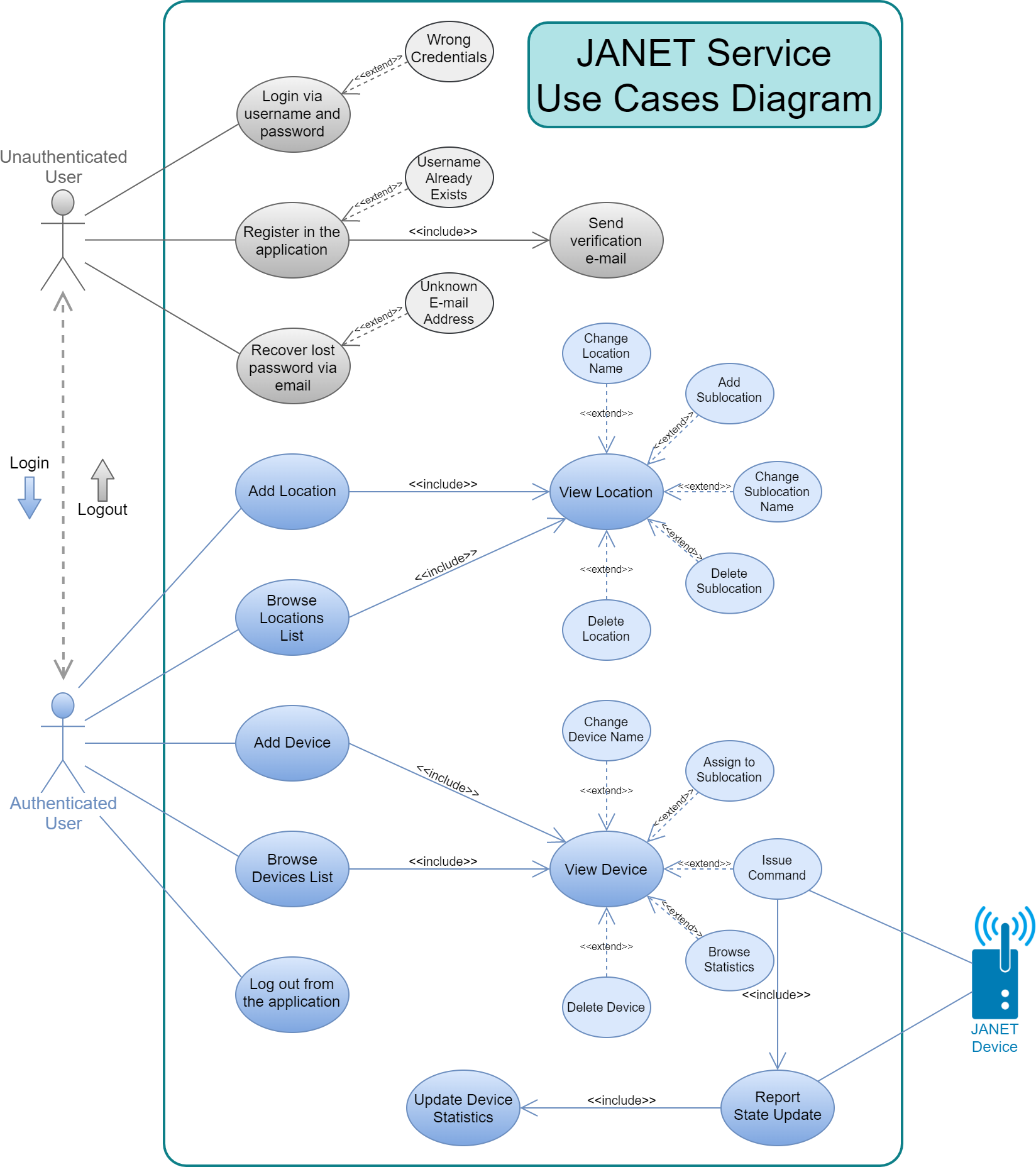
* In order to abstract the devices’ management from their implementation details and to avoid interoperability concerns across products of different vendors, the JANET devices and controllers have been implemented as dedicated *virtual machines* managed through an ad-hoc simulation environment termed *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](https://developers.google.com/assistant/smarthome/traits): *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.

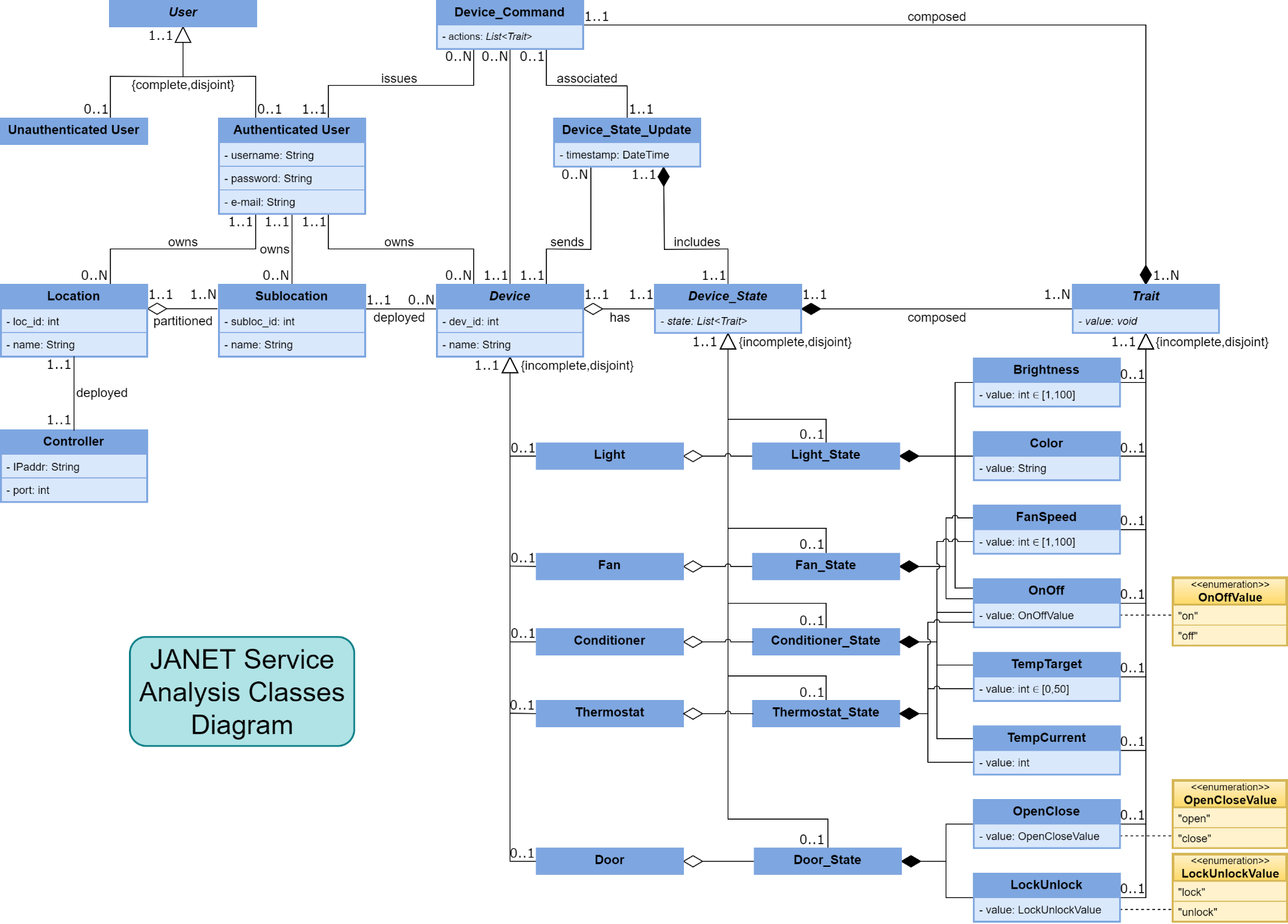
The use cases associated with each actor is outlined in the diagram below:



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 |

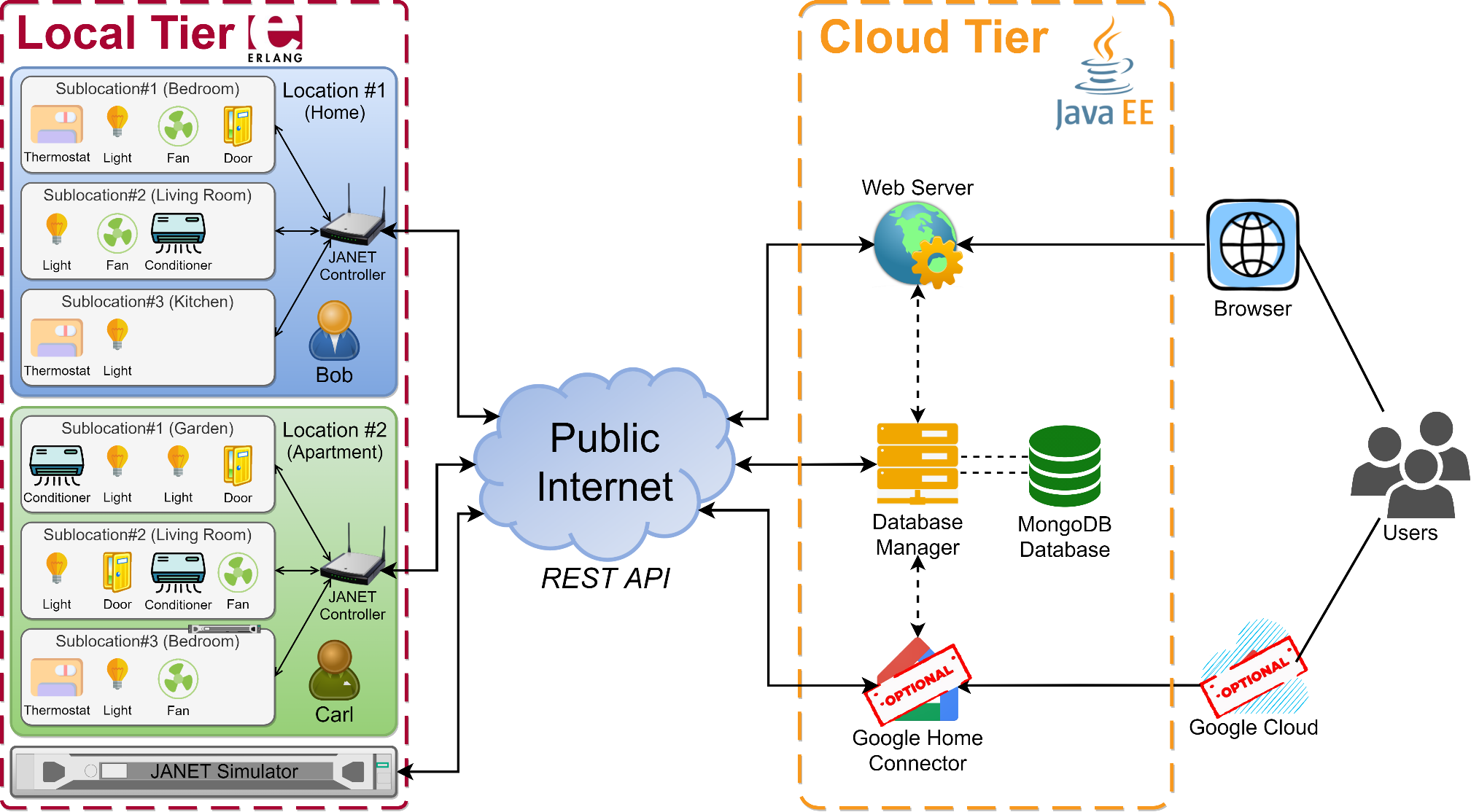
|  |  |  |
| --- | --- | --- |
| Device\_State\_Update | | |
| **Attribute** | **Type** | **Description** |
| timestamp | DateTime | The device’s local time at which the state update occurred |

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:



UPDATE

Cloud Tier

UPDATE

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.
* A *Database Manager* module, routing data between the other application modules and an instance of the MongoDB database and implementing functionalities such as persisting state and sensor reading updates received from smart devices, as well as validating users’ authentication attempts by checking the login credentials passed by the web server module against the account information stored in the database.
* A *MongoDB Database* module, storing most of the data relevant for the service, such as:
  + The users’ account information (username, password, email address).
  + The set of smart devices owned by each user.
  + The set of locations and sub-locations defined by each user.
  + The sub-location each smart device is assigned to.
  + The devices statuses and sensor readings logs.
* An optional *Google Home Connector* module, acting as a bridge allowing users to access the application via the web interface provided by *Google Home Services*, whose integration as previously discussed will be considered during development.

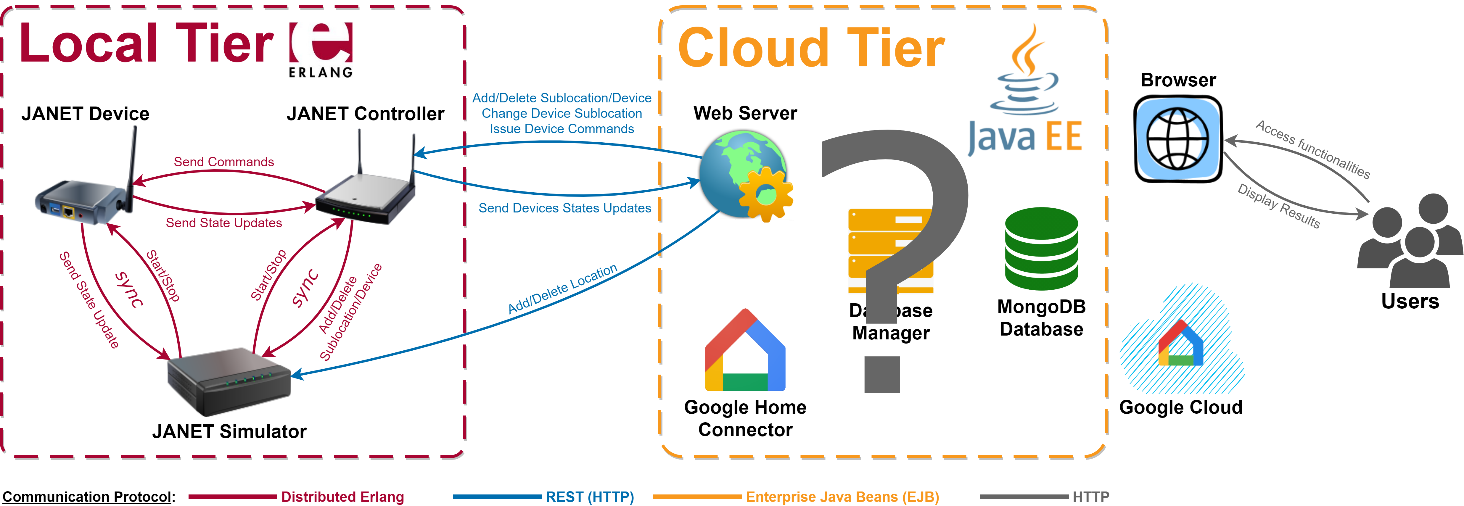
Local Tier

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:



UPDATE

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](https://rebar3.readme.io/docs/getting-started) project management tool, and includes the following middleware OTP applications:

* [Mnesia](https://github.com/erlang/otp/tree/master/lib/mnesia), a built-in Database Management System (DBMS).
* [Cowboy](https://github.com/ninenines/cowboy), an HTTP server.
* [Gun](https://github.com/ninenines/gun), an HTTP client.
* [Ranch](https://github.com/ninenines/ranch), a TCP socket acceptor pool manager.
* [Cowlib](https://github.com/ninenines/cowlib), a library application used by the *Cowboy* and *Gun* applications.
* [JSONE](https://github.com/sile/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:

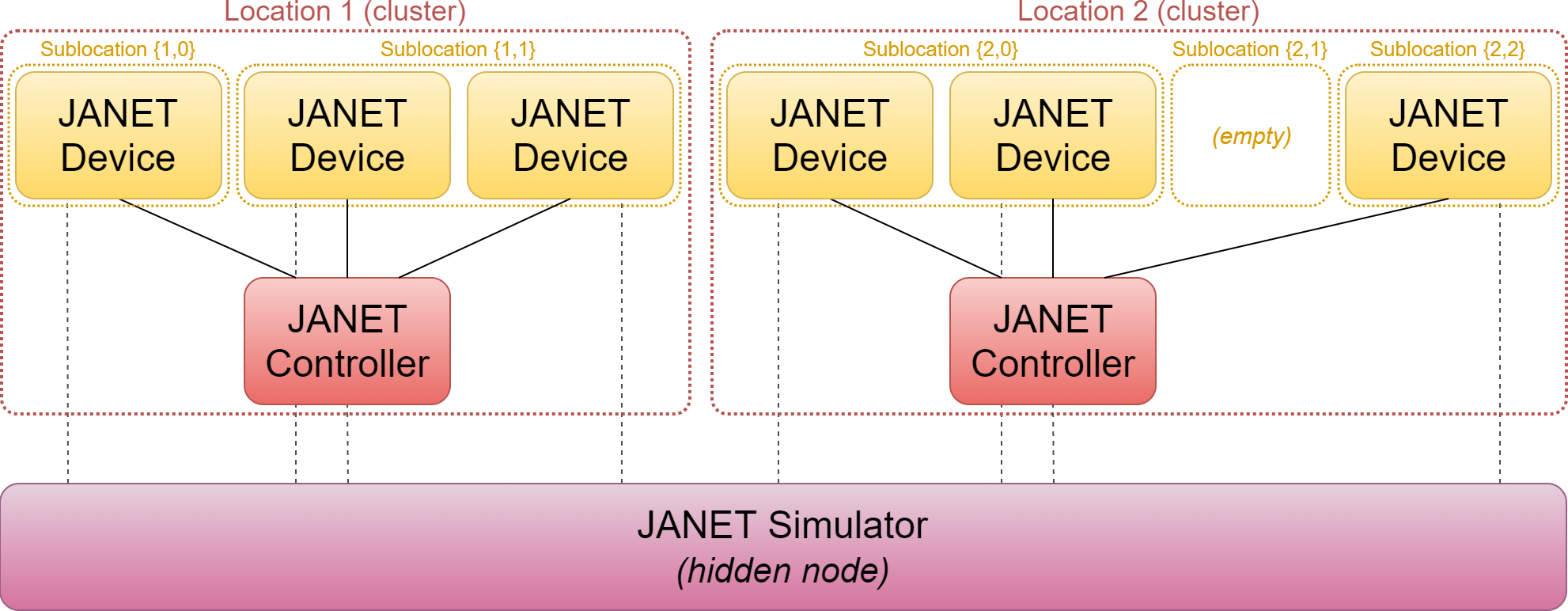
* A *system installation and configuration guide*, outlining the prerequisites and the steps for deploying the simulation environment.
* A *user guide*, highlighting the main commands and functionalities offered by the system.

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.

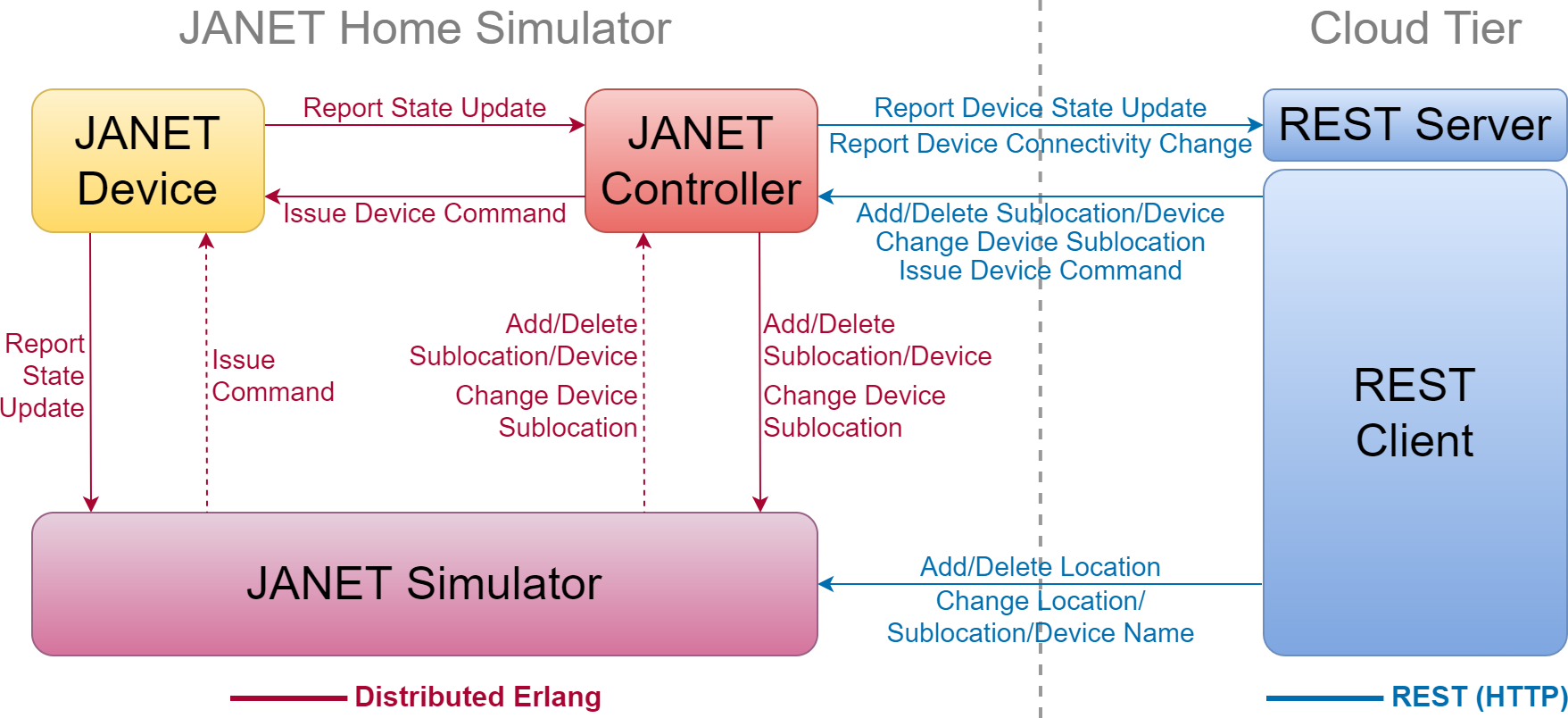
The JANET Simulator node, on the other hand, does not belong to any shared cluster, and represents a *hidden* node that is always connected with every other node in the system.



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:



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.

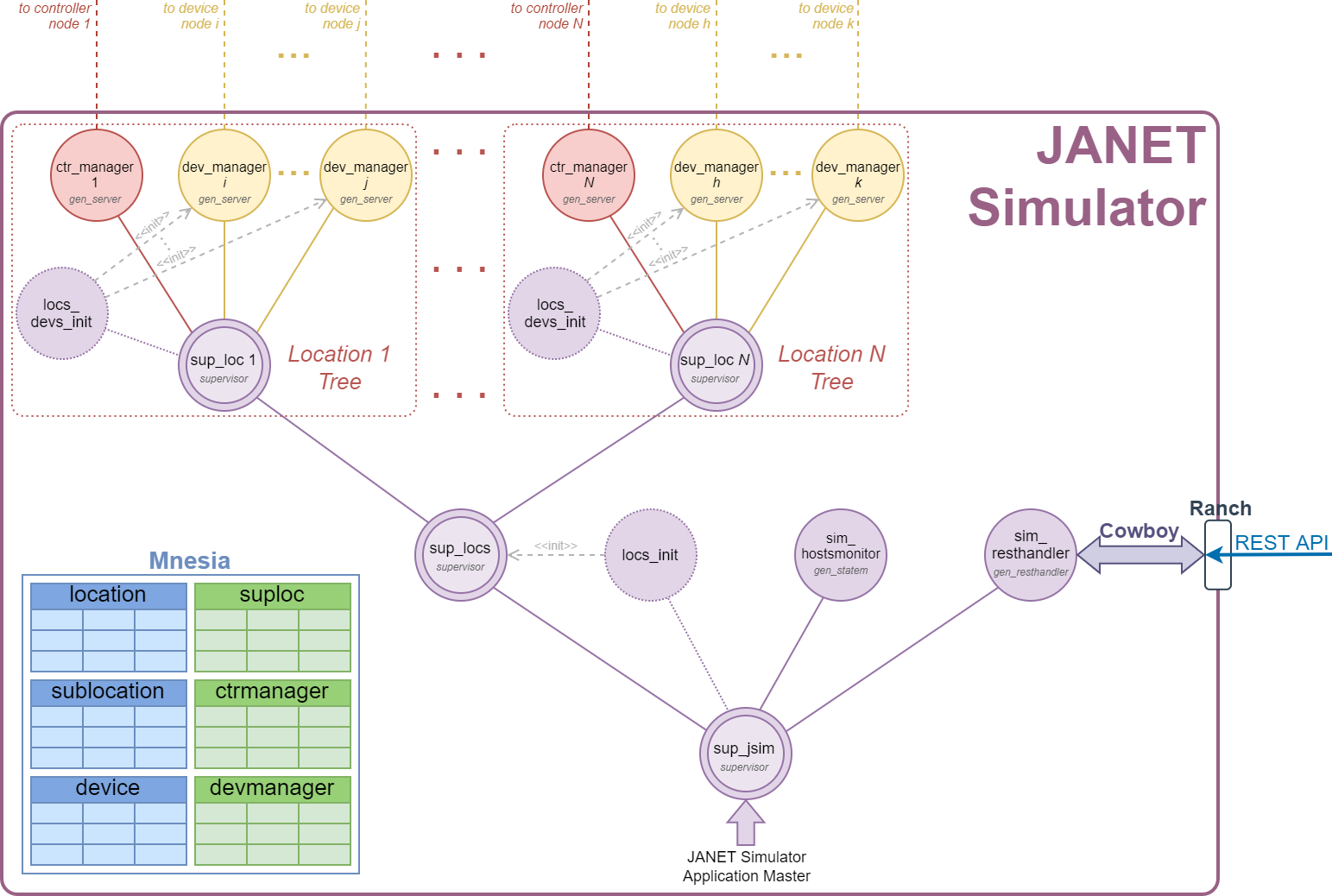
System Base Configuration

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 |

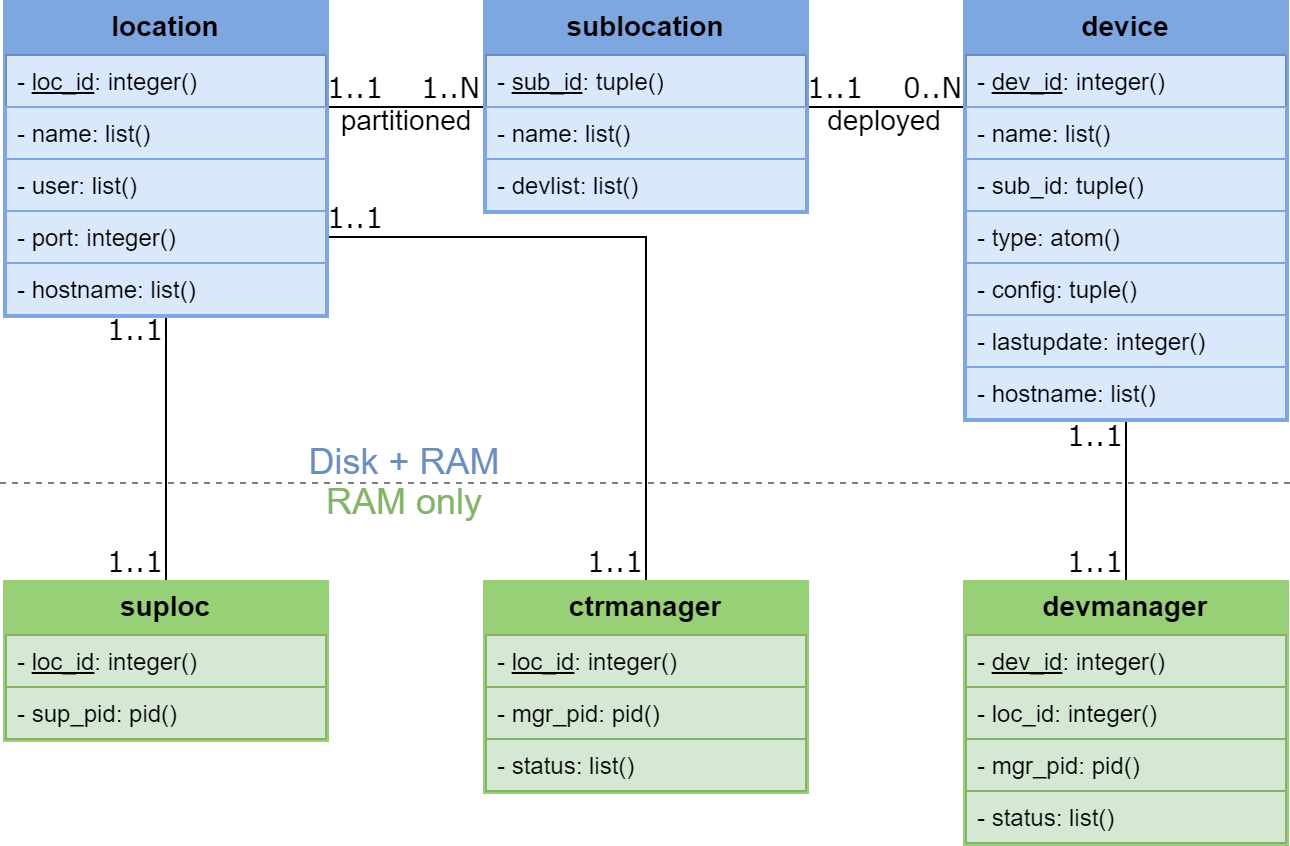
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sup\_jsim Children Specification | | | | | |
| **ChildID** | **Description** | **Multiplicity** | **Type** | **Restart** | **Shutdown** |
| sup\_locs | Locations trees supervisor | 1..1 | *supervisor* | *permanent* | 15000 |
| locs\_init | Locations 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](#JANET_Simulator_Startup_Sequence_1).

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](#JANET_Simulator_Startup_Sequence_2) and [nodes’ management](#JANET_Simulator_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](#Nodes_Statuses), which are mirrored in their associated entries in the [*ctrmanager*](#ctrmanager_status) and [*devmanager*](#devmanager_status) 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](#device_config).

sim\_hostsmonitor *[gen\_statem]*

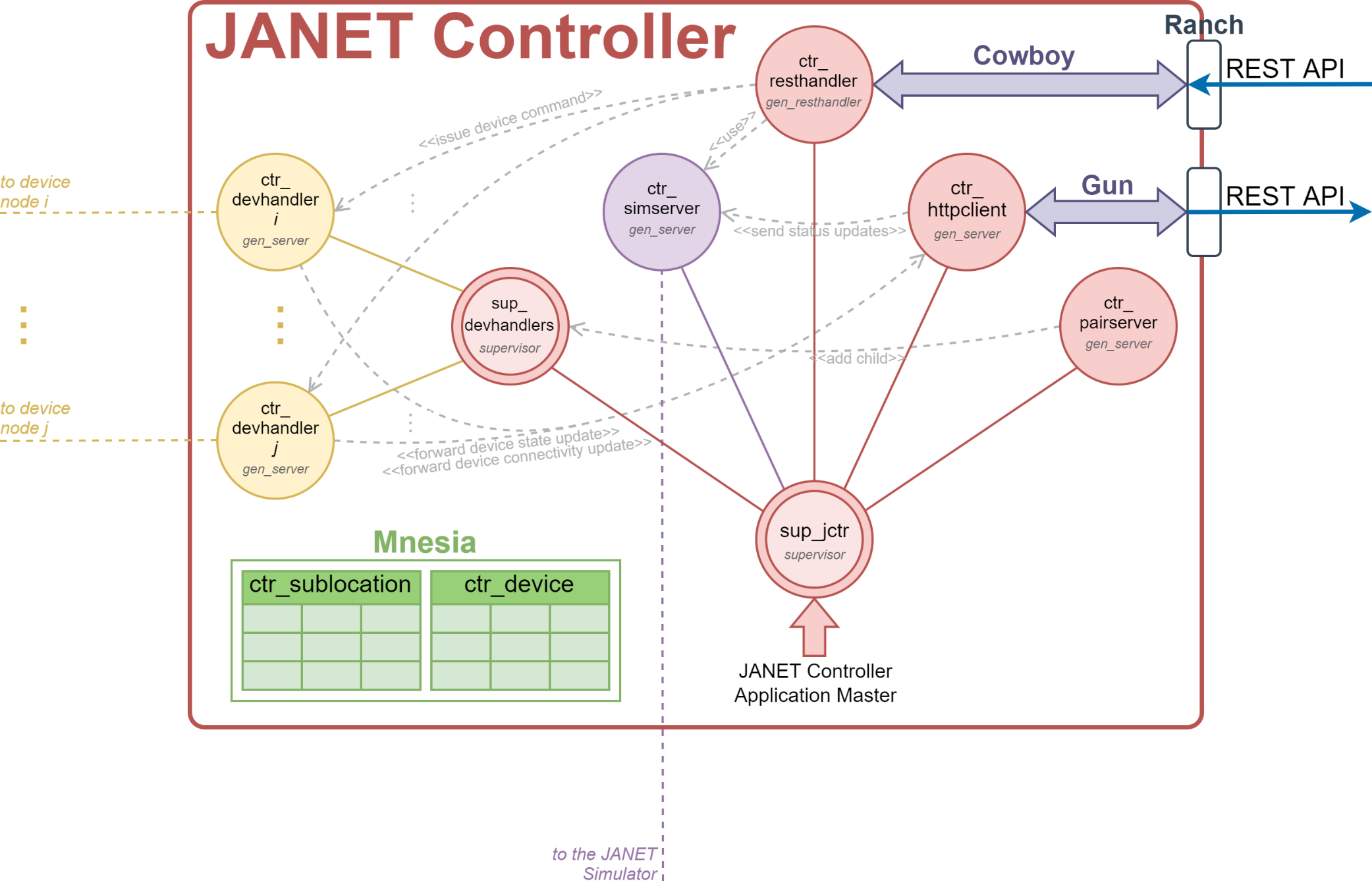
This process was introduced to experiment with advanced features that were later deferred to future application developments, and in its current incarnation, after determining the set of *remote hosts* used by the simulation environment by removing from the *nodes hosts* and the remote REST server all 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 Architecture



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 |

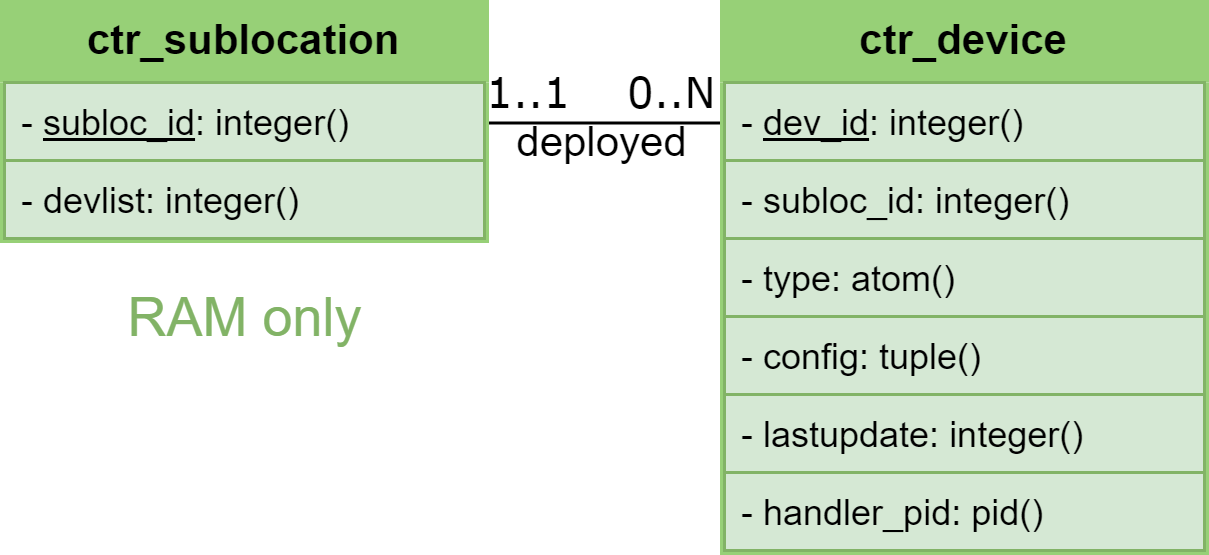
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sup\_devhandlers Children Specification | | | | | |
| **ChildID** | **Description** | **Multiplicity** | **Type** | **Restart** | **Shutdown** |
| ctr\_devhandler\_*i* | Device Handler | 0..M | *worker* | *temporary* | 4800 |

As previously discussed in the [nodes’ management](#Nodes_Management_Automatic_Restart) 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** |
| subloc\_id | 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** |
| dev\_id | 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](#JANET_Simulator_Startup_Sequence_devhand) 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](#dev_statem_asynchronous_state_changes), 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](#JANET_Controller_Mnesia_config) 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](#JANET_Controller_Startup_Sequence_simsrv) 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](#Nodes_Statuses_Controller_Connecting) 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:

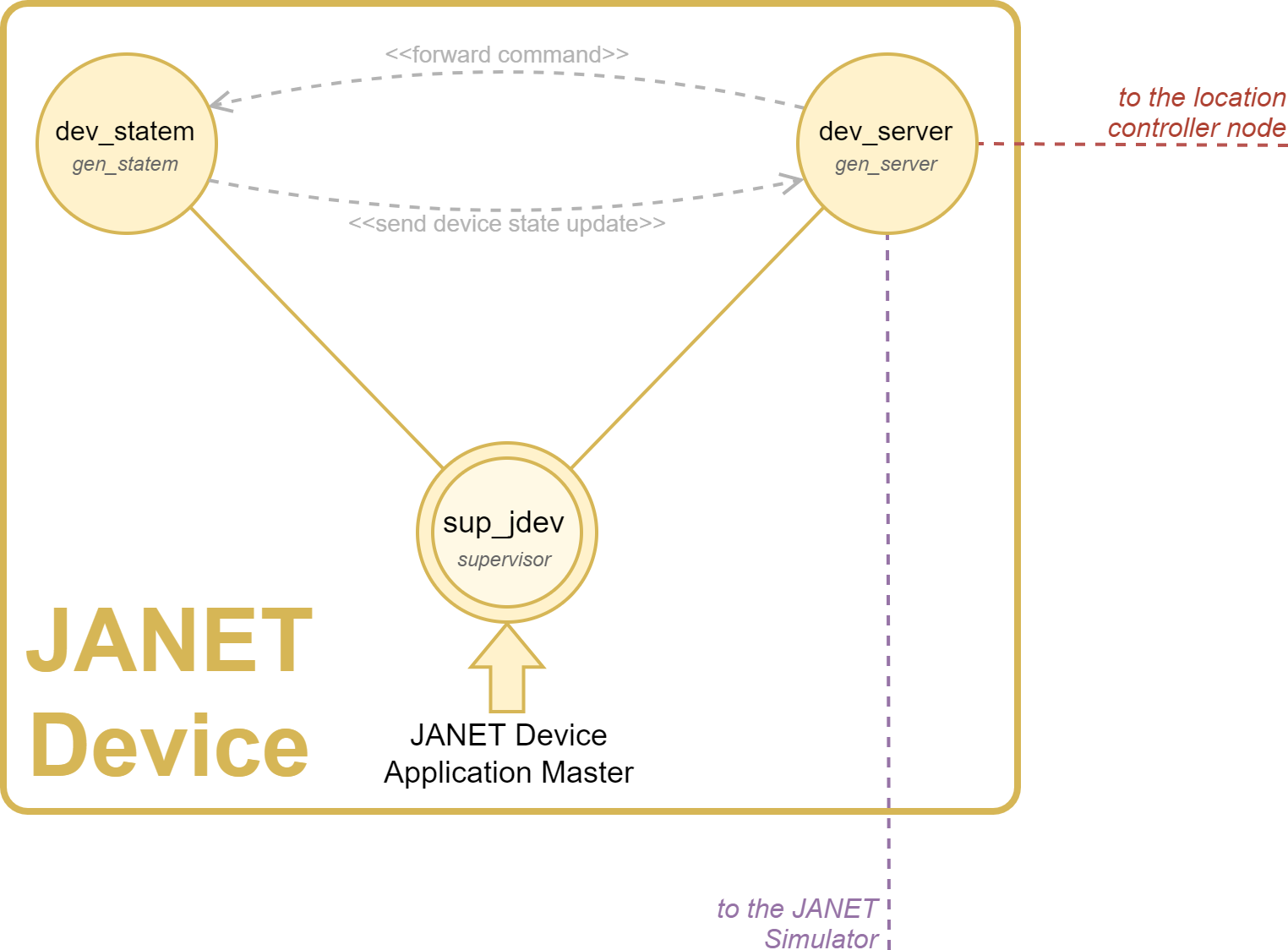
* Forwards any connection state change with the remote REST server (associated with the “CONNECTING” and “ONLINE” [controller node statuses](#Nodes_Statuses_Controller_Connecting)) to the controller manager in the JANET Simulator via the *ctr\_simserver* process.
* Forwards to the remote REST server 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](#JANET_Simulator_Startup_Sequence_pairsrv) 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*](#ctr_device_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](#ctr_device_dev_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](#Nodes_Management_Automatic_Restart) 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](#System_Specification_Device).

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](#JANET_Simulator_Startup_Sequence_devsrv) 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](#dev_statem_asynchronous_state_changes), 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](#Device_Node_Connecting)).

Cloud Tier

Qui cosa ti pare del design in dettaglio della tua parte

Titolo 2

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur.

Titolo 3

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, qui nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur.

Titolo 4

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur.

System Deployment

Verify/add components

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 |
| studente4 | 172.17.0.5 | Web Server, JANET Nodes Host |
| studente5 | 172.17.0.6 | MongoDB Database, JANET Nodes Host |

