AppsGate Developers Documentation

Object

This document gives all information needed to understand AppsGate architecture over OSGi and ApAM middleware. It also describes JSON web client communication protocol.

Last update

May 24, 2014

Version

1.0.0

Authors

Cédric Gérard (UJF engineer)

Overview

This document is divided in five parts. The first one concern the CHMI and the developer guidelines to integrate a new technology support. The second part deals with EHMI and the means to provide the persistency of the context and the end user development environment. All methods available through the client/server protocol API will be details in the third part and explained with sequences diagrams to understand what happened clearly with each calls. The last two parts concern satellites ApAM components used and the configuration GUI.

Newly added

First version released.

Updates needed

Add the DomiCube grammar and updates grammar for exiting devices

The AppsGate global architecture is component oriented on the top of OSGi. AppsGate dedicated components are developed with ApAM (https://github.com/AdeleResearchGroup/ApAM/wiki).

The core human-machine interface (CHMI) and the extended humane-machine interface (EHMI) are loosely coupled and can be use separately or replace with another layer that match developer needs.

In the figure 1 the global architecture overview shows that the CHMI and the EHMI are blocks in which embedded components can only communicate with components in the same block. The only way to trigger command or get state from the CHMI is to use the CHMIProxy component that act as router to call method on any instance in the CHMI. In the same way the EHMIProxy component is a proxy for the EHMI and the only mean to get contextual data or configure your application.

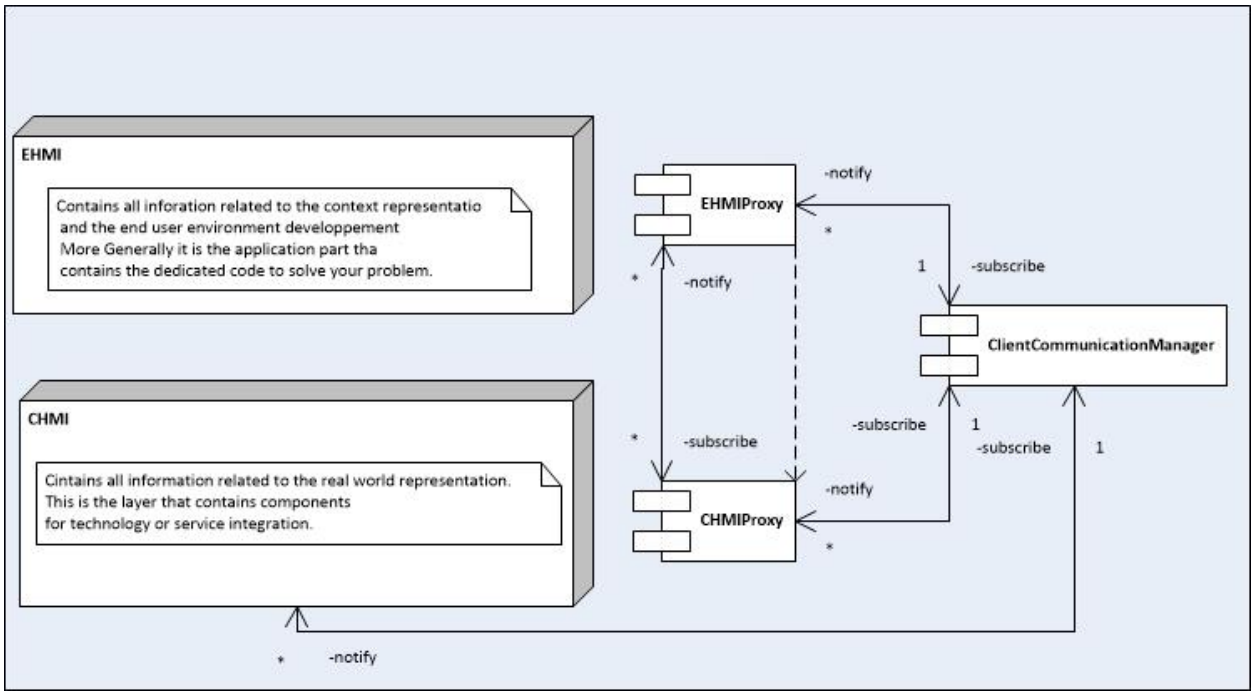

FIGURE 1: OVERVIEW OF THE APPSGATE ARCHITECTURE DECOMPOSITION

1. Core Human Machine Interface Middleware (CHMI)

The CHMI is divided in silos (Figure 2), one for each integrated technology and one for simulation. At This time we have integrated EnOcean and Watteco for Wireless sensor network (WSN), Philips HUE bridge and light bulb for concealed lighting and Google agenda, mail and Yahoo weather as web services. We also have specifically integrated UPnP services such as media browser and media player.

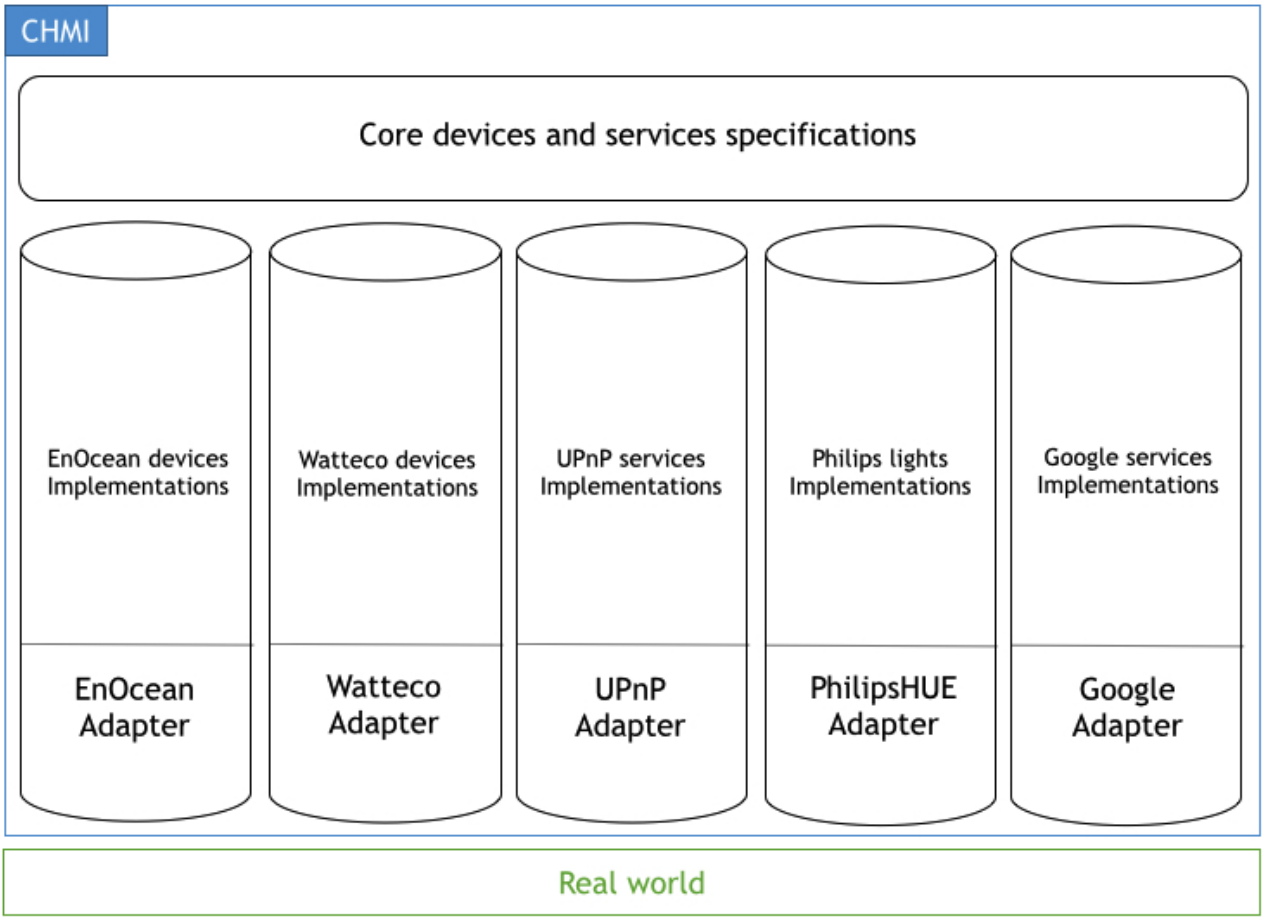

FIGURE 2: GLOBAL VIEW FOR THE CHMI

The figure 2 illustrates layers that map the real world to its virtual representation. For technology or each protocol we have to implement a component to allow AppsGate to "speak" this language. This is the role of all adapters. An adapter have to deal with devices or services in the real world and instantiate each one with the corresponding ApAM implementation, that why when a new technology is added to AppsGate the integrator have to make an adapter and all implementations needed for devices that he want to integrate. The last layer contains AppsGate core specification for devices and services and the core objects specification that normalize instances for external communication.

a. Specifications

The purpose of specifications component is to define and to normalize the behavior of instances from the same kind of devices whatever the technology behind. For example the core temperature sensor specification define that the temperature has to be notify in a specific ApAM message and all implementation have to provide a way to get the temperature on demand. Each implementation could not work in the same way but the result has to be the same. Specifications

are the first level of abstraction to standardize devices handling. The second role of the specification layer is to gather instances under one manageable type.

The first specification is the more generic, it is the CoreObjectSpec. To be collected by the CHMI proxy an instance must implement the java interface CoreObjectSpec. This specification defines the messages communication mechanism by providing the message java interface. It provides the CoreObjectBehavior abstract class. That can by modify to inject behavior needed for the application into instances. All other specification defines a type in AppsGate CHMI. The tab below shows the type values and its corresponding human reading type.

Value	Friendly name	Category	Status
-1	Undefined	Device	Supported
0	Temperature	Device	Supported
1	Illumination	Device	Supported
2	Switch	Device	Supported
3	Contact	Device	Supported
4	Key card switch	Device	Supported
5	Occupancy	Device	Supported
6	Smart plug	Device	Supported
7	Color light	Device	Supported
8	On/off actuator	Device	Supported
9	Co2	Device	Supported
21	System clock	Service	Supported
31	Media player	Service	Supported
36	Media browser	Service	Supported
101	Google calendar	Web service	Supported
102	mail	Web service	Supported
103	Weather	Web service	Supported
200	Mobile device	Device	Supported
210	DomiCube	Device	Supported
794225618	Content directory	Service	Not used
-532540516	?	Service	Unknown
2052964255	Connection manager	Service	Not used
415992004	Av transport	Service	Not used
-164696113	Rendering control	Service	Not used
-1943939940	?	Service	Unknown

TAB 1: APPSGATE CHMI EXISTING TYPE DETAILS

Specifications can be added by third party ton integrate new concept in the CHMI. This doesn't have any impact in the behavior of CHMI. A specification can't be instantiate directly, it is necessary to have an implementation of this specification.

ApAM metadata description holds properties mandatory for all ApAM implementation.

- deviceName (can by empty)

The name of the device when it is instantiate it can be used to provide a default name for a device/service.

- deviceId (mandatory and unique)

The device unique id in the AppsGate CHMI.

- deviceType (mandatory)

The device technological type, for example EnOcean_DEVICE.

- pictureId (can be empty)

A default iconographic representation of the device

- userType (mandatory)

The user type of the device, for example Temperature or Switch

- status (mandatory)

If the device is connected or not.

- isPaired (use for sensors/actuators)

If the device actually paired or not.

- signal(use for sensors/actuators)

The signal power for a device.

a. Technological silos

A silo is a group of components that correspond to the integration of a dedicated technology (device or service). A technological silo regroup all implementation of integrated devices or services and most the time an adapter to map a specific protocol and to handle the discovery and instantiation procedures. An Adapter is ApAM designed in the sense that it is composed of two bundles. The first bundle is a specification that determines OSGi services provides by the Adapter and the second is the implementation.

i. EnOcean

EnOcean is a protocol for wireless sensors network that make energy harvesting. This silo is based on Ubikit bundles provided by third party, Immotronic. Source code for Ubikit EnOcean is not

provided and it is not open source. The serial broker through RXTX needs native libraries available under the “natives” directory at the root of any AppsGate distribution. Some script are provides to launch AppsGate distributions with the correct parameter in order to use natives that match your platform (apam for Linux, apamMacOSX for MacOS, apam.bat for Windows).

We use EnOcean PEM (Physical environment model) and Ubikit services through an adapter name UbikitAdapter. The UbikitAdapterSpec provides services to manage pairing listening mode, to manage Ubikit items and to send actuator commands. The UbikitAdapter implementation is connected to the EnOcean PEM via the Ubikit API and gets all telegrams from EnOcean network. They are displayed in the OSGi console and log in the debug.log file in the distribution root. Ubikit provides event mechanism to catch EnOcean telegrams for paired sensors. The UbikitAdapter implementation catches those events and updates corresponding ApAM instances properties.

Dependence	UbikitAdapterImpl	UbikitAdapterSpec
maven	UbikitAdapterSpec	JSON
	ipojo.annotation	
	apam-core	
	Ubikit	
	pem.enocean	
	ClientCommunicationManager	
service	ListenerService	
	SendWebsocketsService	
	HttpService	
	PhysicalEnvironmentModelService	

TAB 2: UBIKITADAPTER DEPENDENCIES

Some EnOcean profiles have been integrated in AppsGate each have its own ApAM implementation.

- EnoceanUndefinedSensorImpl

This implementation is used when a sensor is paired but its profile is not supported yet. No event is tracked.

Dependence	EnoceanUndefinedSensorImpl
maven	CoreUndefinedSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 3: ENOCEAN UNDEFINED SENSOR DEPENDENCIES

- EnoceanTempSensorImpl

The EnOcean temperature implementation is use with temperature sensor and old a temperature property updated each time the UbikitAdapter received a temperature update telegram. It allows also getting the last received temperature value.

Dependence	EnoceanTempSensorImpl
maven	CoreTempSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 4: ENOCEAN TEMPERATURE SENSOR DEPENDENCIES

- EnoceanLumSensorImpl

The EnOcean illumination implementation hold a “currentIllumination” property updated each time the UbikitAdapter received an illumination value. It allows getting the last received illumination value.

Dependence	EnoceanLumSensorImpl
maven	CoreLumSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 5: ENOCEAN ILLUMINATION SENSOR DEPENDENCIES

- EnoceanSwitchSensorImpl

The EnOcean switch implementation hold three properties in order to manage the button number, the position of that button and the release state. It sends notification when button state change and allow getting the last button status.

Dependence	EnoceanSwitchSensorImpl
maven	CoreSwitchSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 6: ENOCEAN SWITCH SENSOR DEPENDENCIES

- EnoceanContactSensorImpl

The EnOcean contact sensor implementation holds a property for the current contact state. It allows also getting the last received value.

Dependence	EnoceanContactSensorImpl
maven	CoreContactSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 7: ENOCEAN CONTACT SENSOR DEPENDENCIES

- EnoceanKeyCardSensorImpl

The EnOcean Key card switch is a switch that works with a card. It detects when a card is inserted or when a card is taken away. It holds a property "currentStatus" that is a boolean value. True means a card is inserted and false means no card. It allows also getting the last received state.

Dependence	EnoceanKeyCardSensorImpl
maven	CoreKeyCardSensorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 8: ENOCEAN KEY CARD SENSOR DEPENDENCIES

- EnoceanOnOffActuatorImpl

The EnOcean ON/OFF actuator implementation is a generic implementation of each device that can be turned on or off. This implementation holds a state property (isOn) that represents the current state of the device. It allows getting the last received state and to change the current state with On() and Off() methods.

Dependence	EnoceanOnOffSensorImpl
maven	CoreOnOffActuatorSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 9: ENOCEAN ON/OFF ACTUATOR DEPENDENCIES

- EnoceanPlugImpl

The EnOcean smart plug implementation stands for EnOcean device that act as a remote controlled plug but they can calculate the current consumption. This implementation uses four properties for the plugState, the consumption, the calculate activeEnergy and the time of the last request received. An instance of this implementation notify every 10 second the current consumption if it is turned on. It can be turn on or off remotely and application can access the last state and the last consumption value received on demand.

Dependence	EnoceanPlugSensorImpl
maven	CoreSmartPlugSpec
	ipojo.annotation
	UbikitAdapterSpec
	CoreObjectSpec
service	UbikitAdapterService

TAB 10: ENOCEAN SMART PLUG SENSOR/ACTUATOR DEPENDENCIES

ii. Watteco

Watteco is another sensor network technology base on 6LoWPAN. This silo is based on a little C script that configures a virtual IPv6 network interface over a serial connection with Watteco USB dongle. The WattecoAdapter is design to launch the C script and manage to maintain the virtual interface up. After it established serial connection with the Watteco USB dongle and get the network table of detected Watteco devices. The Watteco specification provides java interface of service use to manage Watteco sensor network. The WattecoAdapter at the device instantiation set a default configuration for metering reporting to each device in the network.

Dependence	WatetcoAdapterImpl	WattecoAdapterSpec
maven	WattecoAdapterSpec	
	ipojo.annotation	
	apam-core	
	ClientCommunicationManager	
service	ListenerService	
	SendWebsocketsService	
	HttpService	

TAB 11: WATTECOADAPTER DEPENDENCIES

- WattecoTempImpl

The Watteco Temperature implementation is used with temperature sensor and old a temperature property updated each time the sensor send a new value. It allows also requesting the current temperature value of the temperature sensor

Dependence	WattecoTempSensorImpl
maven	CoreTempSensorSpec
	ipojo.annotation
	WattecoAdapterSpec
	CoreObjectSpec
service	WattecoIOService

TAB 12: WATTECO TEMPERATURE SENSOR DEPENDENCIES

- WattecoSmartPlugImpl

The Watteco smart plug implementation stands for Watteco device that act as a remote controlled plug but they can calculate the current consumption. This implementation uses two properties for the plugState and the consumption. With default configuration a sensor send the consumption every two watts consumption variation and Every 10 seconds. It is possible to get the consumption, the active energy and the plug state on demand directly from the sensor.

Dependence	WattecoSmartPlugImpl
maven	CoreSmartPlugSpec
	ipojo.annotation
	WattecoAdapterSpec
	CoreObjectSpec
service	WattecoIOService

TAB 13: WATTECO ENOCEANPLUG SENSOR/ACTUATOR DEPENDENCIES

- WattecoCO2Impl

The Watteco Co2 implementation is used with Co2 sensor and old a integer value that is the ppm (part per million) in a property updated each time the sensor send a new value. It allow also to request the current CO2 value of the temperature sensor

Dependence	WattecoCO2SensorImpl
maven	CoreCO2SensorSpec
	ipojo.annotation
	WattecoAdapterSpec
	CoreObjectSpec
service	WattecoIOService

TAB 14: WATTECO CO2 SENSOR DEPENDENCIES

- WattecoOccupancyImpl

The Watteco occupancy implementation is used with occupancy sensor and old a boolean property updated each time the sensor send a PIR event. It allows also requesting the current status value of the occupancy sensor. The occupied property is true when the sensor detect a motion and false when no motion is detect from a little period of time.

Dependence	WattecoOccupancySensorImpl
maven	CoreOccupancySpec
	ipojo.annotation
	WattecoAdapterSpec
	CoreObjectSpec
service	WattecoIOService

TAB 15: WATTECO OCCUPANCY SENSOR DEPENDENCIES

iii. Philips HUE

Philips HUE is a technology developed to control Philips lights (HUE, living color, etc.) light through a gate using a REST API. The PHilipsHUEAdapter is based on the PhilipsHUE sdk available on the HUE official web site. This adapter manages the REST API call and response and the bridge discovery. The bridge connection (application registration and connection) is also made via the PhilipsHUEAdapter. It is design to manage HUE network failure and update to keep HUE instance up to date.

Dependence	PhilipsHUEAdapterImpl	PhilipsHUEAdapterSpec
maven	PhilipsHUEAdapterSpec	JSON
	ipojo.annotation	
	apam-core	
	ClientCommunicationManager	
	slf4j-api	
service	ListenerService	
	SendWebsocketsService	
	HttpService	

TAB 16: PHILIPSHUEADAPTER DEPENDENCIES

For now only the Philips HUE lights have been included in AppsGate. This integration is only made through the bridge API not directly with Zigbee protocol. The implementation of PhilipsHUE allows to remote control the light bulb. It is possible to turn it On or Off, change its color from

HUE set, change its brightness, change its color saturation, change its color mode and change its transition time.

Dependence	PhilipsHUEImpl
maven	CoreColorLightBulbSpec
	ipojo.annotation
	PhilipsHUEAdapterSpec
	CoreObjectSpec
	slf4j-api
service	PhilipsHUEService

TAB 17: PHILIPS HUE LIGHT IMPLEMENTATION DEPENDENCIES

iv. Google

We integrate two google services, Google Agenda and Gmail. The Google adapter act as a broker for Google services authentication. In the opposite of other adapters the Google adapter does not instantiate any service implementation but only provides, through OSGi, method to link implementation with real web services in the cloud of Google. The GoogleAgenda implementation and Gmail implementation are use with user profile and instantiate when a user synchronizes a service account with its profile.

Dependence	WatetcoAdapterImpl	WattecoAdapterSpec
maven	ipojo.annotation	
	slf4j-api	
	google.gdata-calendar	

TAB 18: GOOGLEADAPTER DEPENDENCIES

- GoogleCalendarImpl

This implementation is use to instantiate a Google agenda service. An instance of this service is related to only one remote agenda for a user account.

Dependence	GoogleAgendaImpl
maven	CoreCalendarSpec
	CoreClockSpec
	GoogleAdapter
	CoreObjectSpec
	slf4j-api
	ical4j

TAB 19: GOOGLE CALENDAR IMPLEMENTATION DEPENDENCIES

- GMailImpl

This implementation is use to synchronize mail events with an AppsGate user account. An instance of this implementation covers only one mail account for one user.

Dependence	GMailImpl
maven	CoreMailSpec
	java.mail
	felix.ipojo
	ipojo.annotation
	slf4j-api
	felix.gogo
	CoreObjectSpec
	felix
	osgi.core
	apam-core

TAB 20: GMAIL IMPLEMENTATION DEPENDENCIES

- Weahter

This service does not have any adapter. The yahoo weather implementation acts as a standalone. This implementation offers a service to add a location (city name) to add an point of interest from where to get the forecast from yahoo web service.

Dependence	YahooWeahterImpl
maven	CoreWeahterServiceSpec
	felix.ipojo
	ipojo.annotation
	felix.gogo
	felix.gogo.runtime
	GoogleAdapter
	CoreObjectSpec
	slf4j-api
	osgi.core
	apam-core

TAB 21: YAHOO WEATHER IMPLEMENTATION DEPENDENCIES

v. UPnP

Implementation for UPnP services are generated from the UPnPGenerator. The UPnPAdapter is used to be compliant with the AppsGate integration rules, it manages the discovery phase and instantiate supported services.

We currently support UPnP media player and UPnP media browser. From our first integration it is possible to access directly to UPnP service through a direct mapping instance of each services. Supported services are:

- AvTransport
- ContentDirectory
- ConnectionManager
- RenderingControl

Dependence	all UPnP
maven	CoreObjectSpec
	ipojo
	apam-core
	ipojo.annotation
	devicegen
	osgi.core

TAB 21: ALL UPNP DEPENDENCIES

b. CHMI proxy

This component acts as proxy for the complete CHMI. It allows invoking generically all methods on every CoreObject that implements the CoreObjectSpec java interface. It offers a notification mechanism to send the JSON description of any discovered device or service. This CHMI provide an expert interface to control and configure adapters, devices and services.

This proxy is divided in two parts the CHMIProxySpec goals to expose OSGi services for the CHMI proxy. The CHMIProxyImpl implements methods described in the specification and it the only access point to instances of the CHMI components. Notifications that are triggered from device/service instances can be access through ApAM message events mechanism from anywhere.

This component is a singleton and depends on ClientCommunicationManager service. This service allows remote application communicating through the CHMIProxy to any CHMI component and to be notified for each change.

Dependence	CHMIProxyImpl	CHMIProxySpec
maven	CHMIProxySpec	ClientCommunicationManager
	EHMIProxySpec	CoreObjectSpec
	apam-core	slf4j-api
	ipojo.annotation	
service	CoreObjectSpec[]	
	addListenerService	
	sendToClientService	
	EHMIProxySpec	

TAB 22: CHMI DEPENDENCIES

c. CHMI Integration guidelines

Add a new technology in AppsGate is, in the worst case, a four steps process. With a top down point of view, developers have to add an adapter related to the technology their devices or services use to communicate (radio protocol on serial port, Bluetooth, ZigBee, REST, etc.). This component is shared by all device or service implementation and hold only protocol features. It aims to find devices/services by itself and instantiate the corresponding ApAM implementations. It can provide an html interface to configure some parameters or allow experts users to control the adapter and all instances. This adapter keeps a track of all instances and if the protocol needs it, it can delete any existing instance.

For the second step developers have to check if an existing specification match the behavior of their devices/services. In this case they have to develop an ApAM implementation related to this specification to be AppsGate compliant to this profile. If no specification exist, it is necessary to add a new one. A specification is design to provide the most abstract behavior as possible, it is not necessary to describe all device or service features cause some are technology or brand related. For EHMI and end user development environment only feature describe in specification are available. A remote dashboard using all device features can be developed as well.

A specification describes generic feature in the java interface and implement ApAM messages that the device can/must trigger on updates. The implementation must implements the CoreObjectSpec in order to be discovered and exposed by the AppsGate CHMI. It must implements the interface provides by its specification and it may extend the CoreObjectBehavior class that allows an instance to provide a JSON self-description of its states and events.

To summarize:

1- Develop an Adapter for the protocol communication
2- Check existing specifications
3- Add missing specification (if needed)
4- Develop device/service implementations

2. EHMI

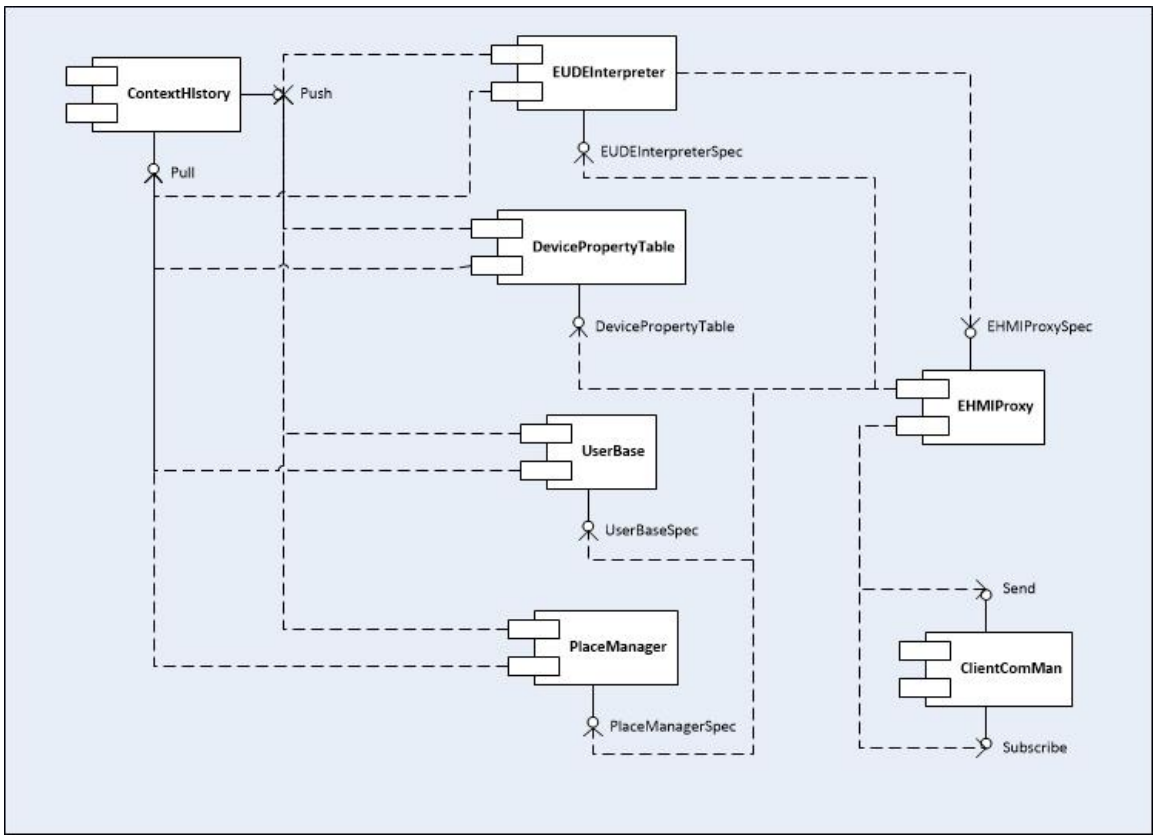

FIGURE 3: CHMI ARCHITECTURE OVERVIEW

a. EHMI proxy

This component aims to manage the link between EHMI and remote clients and between EHMI and CHMI layer. All remotes calls from clients or the EUDE interpreter pass through the EHMIProxyImpl. The EHMI proxy exposes the AppsGate server IP via UPnP and allows any application to discover, over the local network, the server IP address and web socket port use to attempt a connection. All client Web application can be deployed in the Felix OSGi web server via the EHMIProxyImpl.

Dependence	EHMIProxyImpl	EHMIProxySpec
maven	EHMIProxySpec	org.json
	ipojo.annotation	
	upnp.extra	
	osgi.compendium	
	osgi.core	
service	UbikitAdapterService	
	CHMIProxySpec	

	DevicePropertyTableSpec	
	UserBaseSpec	
	PlaceManagerSpec	
	EUDEInterpreterSpec	

TAB 23: EHMI PROXY COMPONENT DEPENDENCIES

The EHMI proxy generates for each client web application call or EUDE method call a GenericCommand object. This object embeds a runnable object to invoke the method on the asynchronously and get the result. This communication mode will be describes in the chapter 3 (Communication).

The EHMIProxySpec is the interface description for remote client application. All method in the interface allows applications to control all EHMI features for Users, places, object properties and programs. This specification is also describes in details in the chapter 3.

b. UserBase

The user base manages all users' profiles and associated services accounts. In the user base all user are authenticate with and id (login) and a password. In the user creation process a last name and a first name has to be provided. A user can also have a role in the household, a list of associated service and all devices he owned.

Dependence	EHMIProxyImpl	EHMIProxySpec
maven	UserBaseSpec	jbcrypt
		apam.core
service	ContextHistorySpec	

TAB 24: USER BASE COMPONENT DEPENDENCIES

Add a user to the AppsGate user base and return true if the user is correctly added, false otherwise.

public boolean adduser(String id, String password, String LastName, String firstName, String role);

Remove a user from the AppsGate user base and return true if the user is removed, false otherwise

public boolean removeUser(String id, String password);

Get the user list as JSON description

public JSONArray getUsers();

Get a user details

public JSONObject getUserDetails(String id);

Check if the id is already use or not.

public boolean checkIfIdIsFree(String id);

Add a service account to this user account and return true is the account is correctly added.

public boolean addAccount(String id, String password, JSONObject accountDetails);

Remove a service account from this user account and return true if the account is correctly removed

public boolean removeAccount(String id, String password, JSONObject accountDetails);

Get all account details

public JSONArray getAccountsDetails(String id);

Associate a device to the user and return true if the device is correctly added.

public boolean addDevice(String id, String password, String deviceId);

Dissociate a device from the user and return true if the device is correctly removed

public boolean removeDevice(String id, String password, String deviceId);

Get all associated device id

public JSONArray getAssociatedDevices(String id);

Get the user base as a tree.

public JSONObject getHierarchy();

Set the hierarchy.

public void setHierarchy(JSONObject hierarchy);

The user base does not use notification messages.

c. DevicePropertiesTable

This component hold properties associated to any devices or services, for instance it hold the name for each device for each user profile.

Dependence	DevicePropertyTableImpl	DevicePropertyTableSpec
maven	DevicePropertyTableSpec	EHMIProxySpec
service	ContextHistorySpec	

TAB 25: DEVICE PROPERTIES TABLE COMPONENT DEPENDENCIES

Add a new name

public void addName(String objectId, String usrId, String newName);

Delete an existing object name

public void deleteName(String objectId, String usrId);

Get the name given to a device by a specified user

public String getName(String objectId, String usrId);

Add a grammar for a device type and return true if the grammar is new, false if a previous grammar has been replaced

public boolean addGrammarForDeviceType(String deviceType, JSONObject grammar);

Remove a grammar associate to a device type

public boolean removeGrammarForDeviceType(String deviceType);

Get the grammar associate to a device type

public JSONObject getGrammarFromType(String deviceType);

d. PlaceManager

The place manager allow AppsGate client to build a hierarchical set of places. Associates devices/services with places.

Dependence	PlaceManagerImpl	PlaceManagerSpec
maven	**PlaceManagerSpec**	
	EHMIProxySpec	
service	**ContextHistorySpec**	

TAB 26: PLACE MANAGER COMPONENT DEPENDENCIES

Add a new place to the hash map and return the id of the new place or null otherwise.

public String addPlace(String name, String parent);

Remove a place from the numeric representation of the smart space. It return true if the place has been removed, false otherwise

public boolean removePlace(String placeId);

Move this place under a new parent place and return true if the place has been moved, false otherwise

public boolean movePlace(String placeId, String newParentId);

Set the tags list of this place

public void setTagsList(String placeId, ArrayList<String> tags);

Empty the list of tag of this place

public void clearTagsList(String placeId);

Add a new tag to this place and return true if the tag has been add, false otherwise

public boolean addTag(String placeId, String tag);

Remove a tag from this place and return true if the tag has been removed, false otherwise

public boolean removeTag(String placeId, String tag);

Set the properties of this place

public void setProperties(String placeId, HashMap<String, String> properties);

Empty the properties list of this place

public void clearPropertiesList(String placeId);

Add a new property to the list or update an existing one and return true if key is a newly added false if the key exist and the value has been changed

public boolean addProperty(String placeId, String key, String value);

Remove an existing property and return true if the property has been removed, false otherwise

public boolean removeProperty(String placeId, String key);

Move a core object to a specify place. If newPalceID attribute equal -1 the object is just removed from this place. It returns true if the device has moved, false otherwise

public boolean moveObject(String objId, String oldPlaceID, String newPlaceID);

Move a core service to a specify place. If newPalceID attribute equal -1 the service is just removed from this place. It returns true if the service has moved, false otherwise

public boolean moveService(String serviceId, String oldPlaceID, String newPlaceID);

Move all objects to the -1 id place.

public void removeAllCoreObject(String placeId);

Rename a place on the smart space and returns true if the place name has been updated, false otherwise

public boolean renamePlace(String placeId, String newName);

Get the root places of the (multiple) hierarchy and returns the root place reference

public ArrayList<SymbolicPlace> getRootPlaces();

Get the symbolic place object from its identifier and return the SymbolicPlace instance

public SymbolicPlace getSymbolicPlace(String placeId);

Get all the places and returns places as an ArrayList<SymbolicPlace>

public ArrayList<SymbolicPlace> getPlaces();

Get a JSON formatted representation of the smart space. It returns a JSON array that describe each place.

public JSONArray getJSONPlaces();

Get all the places it has the name in parameter and returns places as an ArrayList<SymbolicPlace>

public ArrayList<SymbolicPlace> getPlacesWithName(String name);

Get all the places that are tagged with the tags list in parameter and returns places as an ArrayList<SymbolicPlace>

public ArrayList<SymbolicPlace> getPlacesWithTags(ArrayList<String> tags);

Get all the places that have the properties list set and returns places as an ArrayList<SymbolicPlace>

public ArrayList<SymbolicPlace> getPlacesWithProperties(ArrayList<String> propertiesKey);

Get all the places that have the properties list set the specific value and returns places as an ArrayList<SymbolicPlace>

public ArrayList<SymbolicPlace> getPlacesWithPropertiesValue(HashMap<String, String> properties);

Get the place where a device is placed and returns place where this device is located.

public SymbolicPlace getPlaceWithDevice(String deviceId);

Get the place where a service is placed and returns the place where this service is located.

public SymbolicPlace getPlaceWithService(String serviceId);

Get the place identifier of a core object and return the identifier of the place where the core object is placed.

public String getCoreObjectPlaceId(String objId);

e. Context History

This component provides two OSGi services in order to push or pull components states in mongo data base.

Dependence	ContextHistoryMongoImpl	ContextHistorySpec
maven	ContextHistorySpec	
	apam-core	
	mongo-java-driver	
	org.apache.felix.ipojo	
service	MongoDBConfigurationImpl	

TAB 27: CONTEXT HISTORY COMPONENT DEPENDENCIES

Pull service

Pull the last state of the object corresponding to the name give in parameter. It returns a JSON object that contains the last state of the specify object

public JSONObject pullLastObjectVersion(String ObjectName);

Push service

Push the last state of an object name "name". The state is representing by a set of properties and a header represents the last addition of this object. This method specifies the user that triggers the update. It returns true if the save is done correctly false otherwise

public boolean pushData_add(String name, String userID, String objectID, String addedValue, ArrayList<Map.Entry<String, Object>> properties);

Push the last state of an object name "name". The state is representing by a set of properties and a header represent the last removal of this object. This method specifies the user that triggers the update. It returns true if the save is done correctly false otherwise

public boolean pushData_remove(String name, String userID, String objectID, String removedValue, ArrayList<Map.Entry<String, Object>> properties);

Push the last state of an object name "name". The state is representing by a set of properties and a header represents the last change of this object. This method specifies the user that triggers the update. It returns true if the save is done correctly false otherwise.

public boolean pushData_change(String name, String userID, String objectID, String oldValue, String newValue, ArrayList<Map.Entry<String, Object>> properties);

Push the last state of an object name "name". The state is representing by a set of properties and a header represents the last addition of this object. Its returns true if the save is done correctly false otherwise

public boolean pushData_add(String name, String objectID, String addedValue, ArrayList<Map.Entry<String, Object>> properties);

Push the last state of an object name "name". The state is represented by a set of properties and an header represent the last removal of this object. Its returns true if the save is done correctly false otherwise.

public boolean pushData_remove(String name, String objectID, String removedValue, ArrayList<Map.Entry<String, Object>> properties);

Push the last state of an object name "name". The state is representing by a set of properties and a header represents the last change of this object. It returns true if the save is done correctly false otherwise

public boolean pushData_change(String name, String objectID, String oldValue, String newValue, ArrayList<Map.Entry<String, Object>> properties);

Enumeration class for operation to save.

```
public enum OP {
        ADD,
        REMOVE,
        CHANGE;
}
```

f. End user development environment (EUDE)

This component hold program and execute them. It allows adding, removing and managing programs.

Dependence	EUDE_InterpreterImpl	EUDE_InterpreterSpec
maven	**ContextHistorySpec**	**EHMIProxySpec**
	apam-core	
	mongo-java-driver	
	org.apache.felix.ipojo	
service	**MongoDBConfigurationImpl**	

TAB 28: CONTEXT HISTORY COMPONENT DEPENDENCIES

Initialize a program from its JSON representation and return true when succeeded, false when failed.

public boolean addProgram(JSONObject programJSON);

Remove an existing program. Its returns true if the program has been removed, false otherwise.

public boolean removeProgram(String programId);

Update the program name or sources and return true if the program has been updated, false otherwise.

public boolean update(JSONObject jsonProgram);

Launch the interpretation of a program and return true if the program has been successfully launched, false otherwise.

public boolean callProgram(String programId);

Launch the interpretation of a program and return true if the program has been successfully launched, false otherwise.

public boolean callProgram(String programId, JSONArray args);

Stop the program interpretation and return true if the program has been stopped, false otherwise.

public boolean stopProgram(String progamId);

Return a hash map containing all the programs known by the interpreter. Keys are programs' names and values are programs under their JSON format.

public HashMap<String, JSONObject> getListPrograms();

Check if a program is active or not and return true if the program is active, false otherwise

public boolean isProgramActive(String programId);

3. Communication

The communication is dedicated to one service that manages client connection state. We provide one implementation of this services base on web socket. The client communication manager provides two OSGi services. The first one is used to send messages to connected clients (SendWebSocketService) and the other to be registered for incoming messages (ListenerService).

- **SendWebsocketsService**

This service offers multiple methods to send messages.

The send method with a specific key and parameters.

public void send(String key, JSONObject msg);

The send method with a specific key and parameters.

public void send(String key, JSONArray msg);

The send method with a specific key and parameters.

public void send(String msg);

The send method with a specific key and parameters to the specify client.

public void send(int clientId, String key, JSONObject msg);

The send method with a specific key and parameters to the specify client.

public void send(int clientId, String key, JSONArray msg);

The send method to a specific client.

public void send(int clientId, String msg);

- **ListenerService**

This interface is the specification of service for subscribe notification, message and command response to client application. A listener and its referring clients are link through the target given at subscription. For instance the EHMIProxyImpl is notified that a new message has been received when this message contains the

This method allows all callers to subscribe for all commands and events except configuration. It returns true if the command listener is new, false if it has been replaced.

public boolean addCommandListener(CommandListener cmdListener, String target);

Remove the command listener associated to a target and return true if the command listener has been removed, false otherwise

public boolean removeCommandListener(String target);

- **Client application to EHMI**

The EHMIProxySpec defines all method available for remote client applications. In order to reach the EHMIProxyImpl the remote call needs to use the target "EHMI".

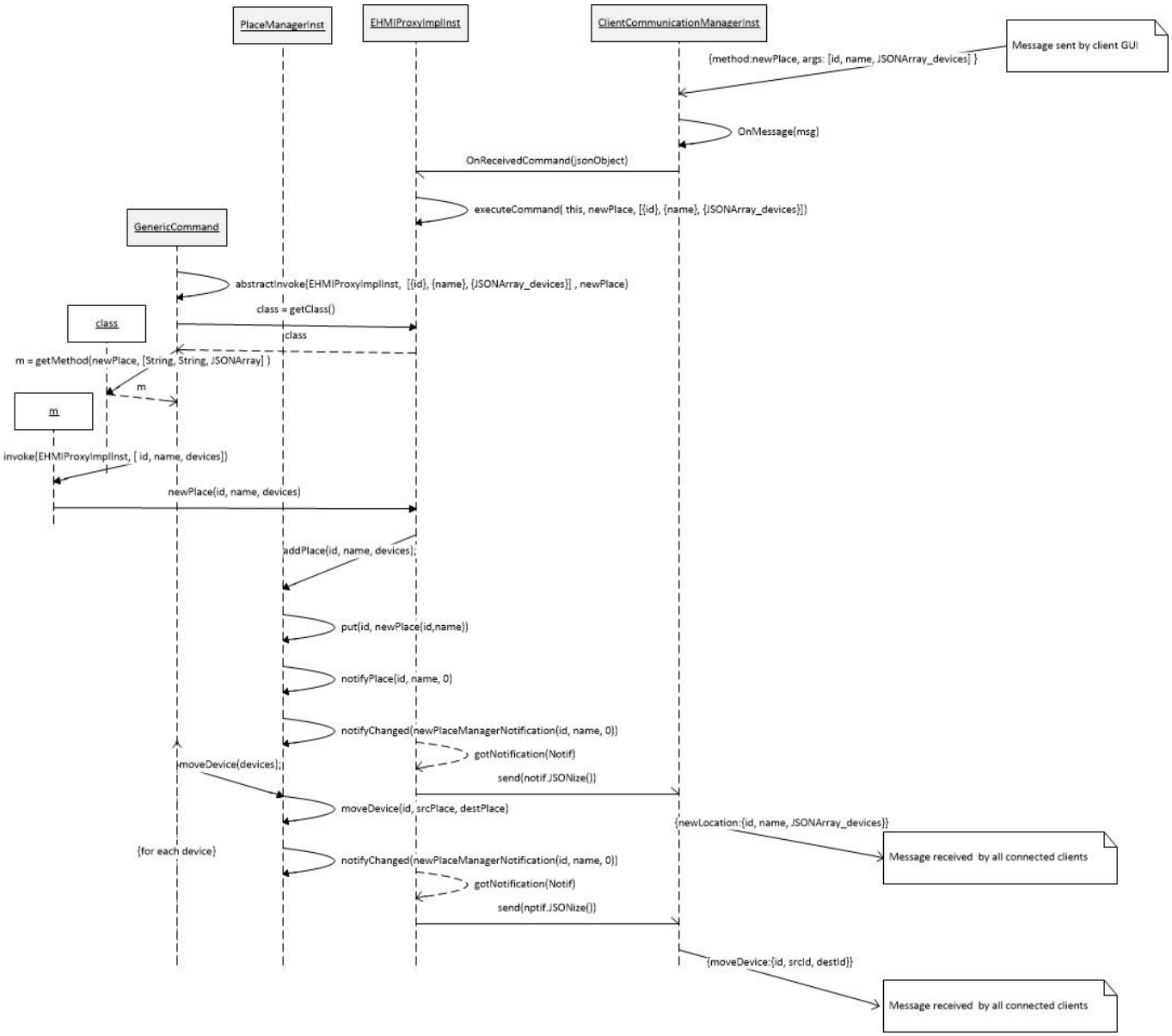

FIG 4: ADD A NEW PLACE IN THE CONTEXT

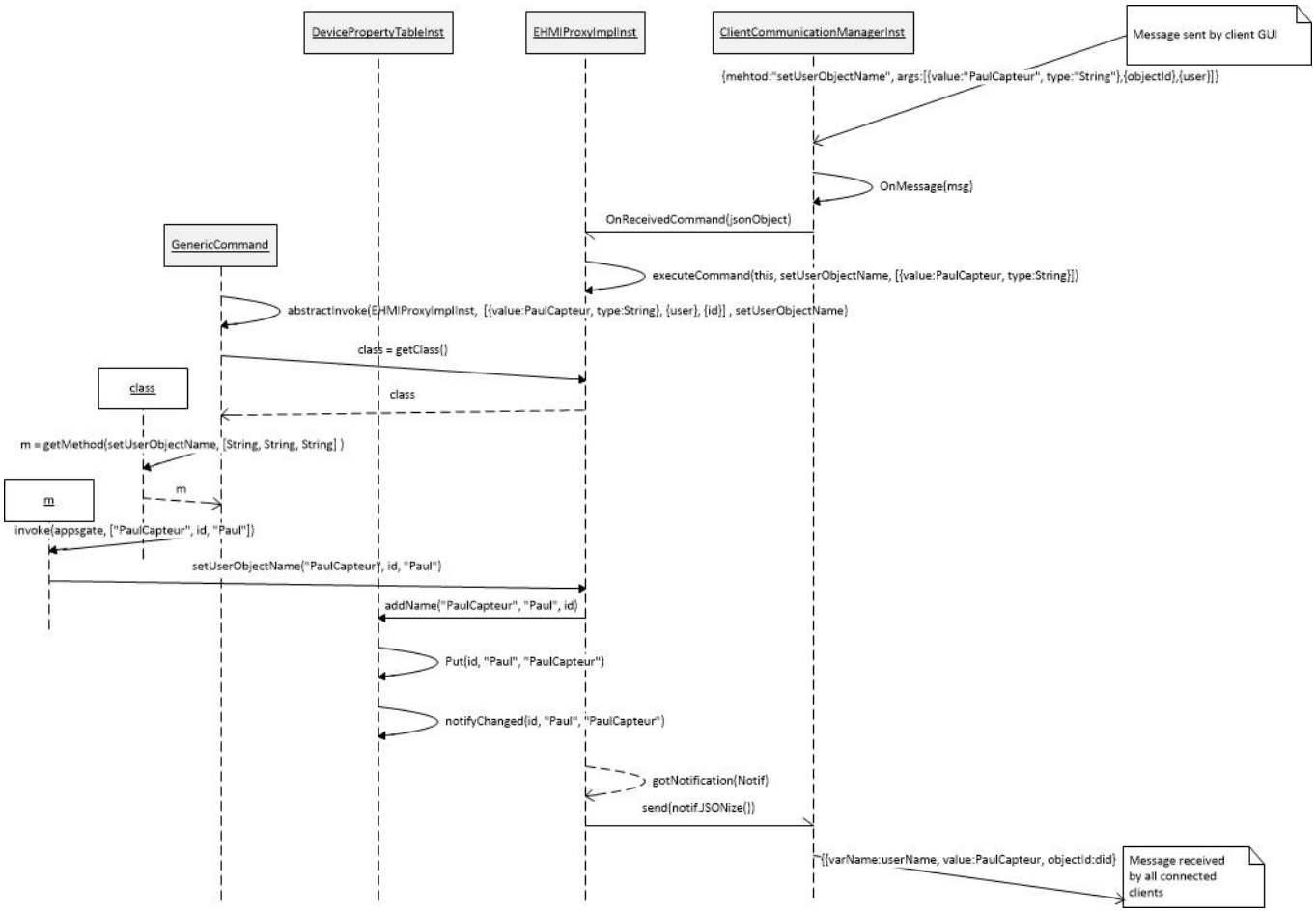

FIG 5: SET A NAME TO A DEVICE IN THE CONTEXT

See below all available methods:

Generic remote call JSON format

{"TARGET":"EHMI", "method":"MEHTOD_NAME", "args[{"type":"JAVA_TYPE", "value":"..."},...,{...}], "clientId":"....", "callId":"..."}

Get all the devices description.

public JSONArray getDevices();

Get device details and return the device description as a JSONObject

public JSONObject getDevice(String deviceId);

Get all the devices of a specify user type and return the device list as a JSONArray

public JSONArray getDevices(String type);

Get the state representation for a device type in the grammar and return the state description.

public StateDescription getEventsFromState(String objectId, String stateName);

Call AppsGate to add a user object name.

public void setUserObjectName(String objectId, String user, String name);

Get the name of an object for a specific user and return the name of the object named by user.

public String getUserObjectName(String objectId, String user);

Delete a name for an object set by a user.

public void deleteUserObjectName(String objectId, String user);

Add grammar in the context properties manager for a new device type and return true if the grammar is really new, false if the grammar has been replaced.

public boolean addGrammar(String deviceType, JSONObject grammarDescription);

Remove grammar associated to a device type return true if the grammar has been removed, false otherwise.

public boolean removeGrammar(String deviceType);

Get the grammar associated to a device type return the grammar as a JSONObject.

public JSONObject getGrammarFromType(String deviceType);

Call AppsGate to get all existing place definition and return a JSON array that describes each place.

public JSONArray getPlaces();

Add a new place and move object in it.

public void newPlace(JSONObject place);

Update a place on the smart place.

public void updatePlace(JSONObject place);

Remove a place from the smart place.

public void removePlace(String id);

Move a device in a specified place.

public void moveDevice(String objId, String srcPlaceId, String destPlaceId);

Move a service in a specified place.

public void moveService(String serviceId, String srcPlaceId, String destPlaceId);

Get the place identifier of a core object and return the identifier of the place where the core object is placed.

public String getCoreObjectPlaceId(String objId);

Return the devices of a list of type presents in the places and return a list of objects contained in these spaces.

public ArrayList<String> getDevicesInSpaces(ArrayList<String> typeList, ArrayList<String> spaces);

Call AppsGate to get all the places that match a specific name and return the places with the name <name> as a JSONArray

public JSONArray getPlacesByName(String name);

Get places that have been tagged with all tags and return places as a JSONArray.

public JSONArray gePlacesWithTags(JSONArray tags);

Get places that contain the properties keys in parameters and returns places list as a JSONArray.

public JSONArray getPlacesWithProperties(JSONArray keys);

Get places that contains the properties keys in parameters./

public JSONArray getPlacesWithPropertiesValue(JSONArray properties);

Get the root places description.

public JSONArray getRootPlaces();

Add a tag to the tag of list of the specified place.

public boolean addTag(String placeId, String tag);

Remove a tag from a place.

public boolean removeTag(String placeId, String tag);

Add a property to a specified place.

public boolean addProperty(String placeId, String key, String value);

Remove a property from a specified place.

public boolean removeProperty(String placeId, String key);

Get the end user list.

public JSONArray getUsers();

Create a new end user.

public boolean createUser(String id, String password, String lastName, String firstName, String role);

Delete an existing end user.

public boolean deleteUser(String id, String password);

Get details on a specify user.

public JSONObject getUserDetails(String id);

Get all information on a specify user.

public JSONObject getUserFullDetails(String id);

Check if the wanted identifier already existing.

public boolean checkIfIdIsFree(String id);

Synchronize a web service account with an end user profile.

public boolean synchronizeAccount(String id, String password, JSONObject accountDetails);

Delete service account synchronization.

public boolean desynchronizedAccount(String id, String password, JSONObject accountDetails);

Associate a device to an end user.

public boolean associateDevice(String id, String password, String deviceId);

Remove end user and device association.

public boolean separateDevice(String id, String password, String deviceId);

Deploy a new end user program in AppsGate system.

public boolean addProgram(JSONObject jsonProgram);

Remove a currently deployed program. Stop it, if it is running.

public boolean removeProgram(String programId);

Update an existing program.

public boolean updateProgram(JSONObject jsonProgram);

Run a deployed end user program .

public boolean callProgram(String programId);

Stop a deployed program execution.

public boolean stopProgram(String programId);

Get the list of current deployed programs.

public JSONArray getPrograms();

Check if a program is active or not.

public boolean isProgramActive(String programId);

This method allows the caller to add a specific coreListener to follow core components state change.

public void addCoreListener(CoreListener coreListener);

This method allows the caller to unsubscribe itself from core component state change.

public void deleteCoreListener(CoreListener coreListener);

Shutdown the AppsGate system (Shutdown the OSGi distribution)

public void shutdown();

Restart the AppsGate system

public void restart();

Get a runnable object that can execute command from a remote device manager asynchronously.

public appsgate.lig.chmi.spec.GenericCommand executeRemoteCommand(String objIdentifier, String method, JSONArray args);

Get the current system time from the local EHMI clock.

public long getCurrentTimeInMillis() ;

- **Client application to CHMI**

In previous AppsGate version client can call directly the CHMIProxyImpl using the target "CHMI". In the StandaloneEHMI (branch) version of AppsGate all command pass through the EHMIProxy. If the command contains the objectId JSON entry the EHMIProxy transmit the call to an available CHMIProxy instance.

Generic CHMI core object JSON method call:

{TARGET : "EHMI", "objectId" : "TARGET_OJBJECT_ID", "method" : "MEHTOD_NAME", "args : [{"type" : "JAVA_TYPE", "value" : "..."},...,{...}], "clientId" : "....", "callId" : "..."}

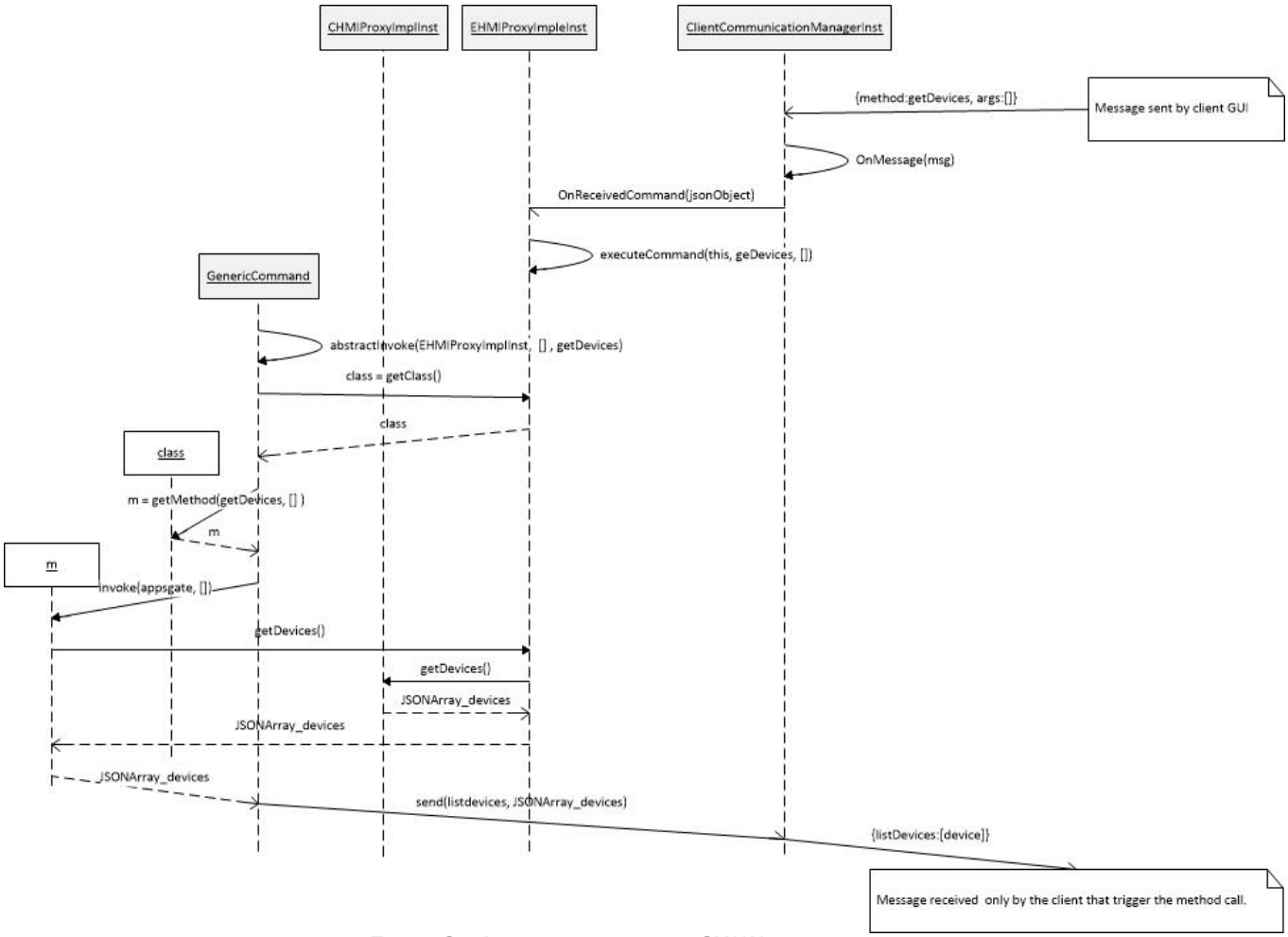

FIG 6: GETDEVICES CALL ON THE CHMIPROXY INSTANCE

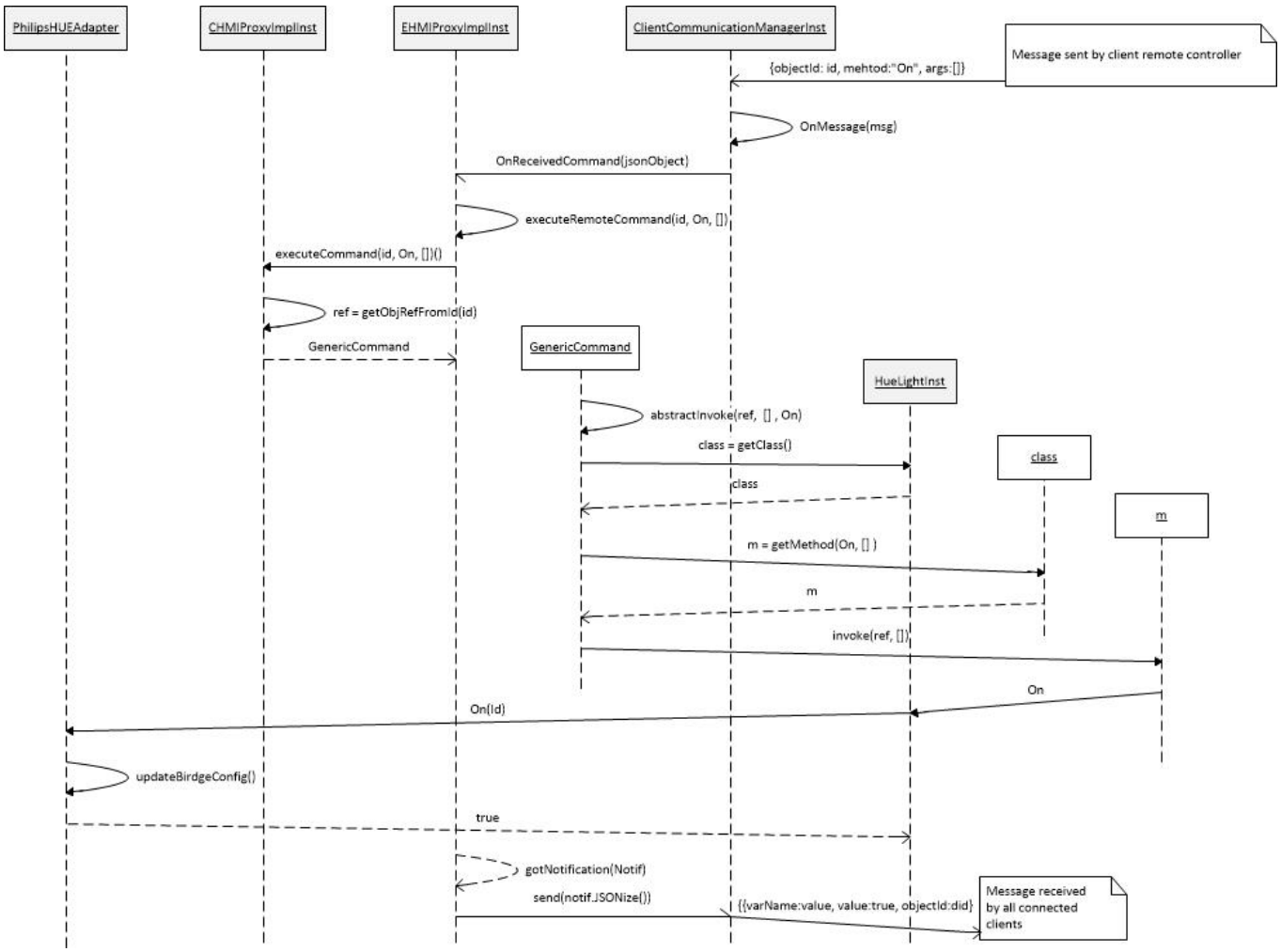

FIG 7: HUE LIGHTS ON CALL ON THE CORE OBJECT HUE LIGHT INSTANCE

Each class of devices or services provides its own method to call. The exhaustive list is provides in the Chapter 4 (Appendix).

- **Bidirectional communication between EHMI and CHMI**

The EHMI use the CHMI as an optional dependency. Without the CHMI the EHMI can't access any device or services but it can work find as a standalone to manage the context (UserBase, PlaceManager, and EUDE.

When the EHMI is connected to an available CHMIProxy instance it subscribes to events notifications (device or service state change) and it subscribes to core update notification (A device or a service appear or disappear).

When a client sends a command that is related to the CHMI (getDevices for instance) the EHMI transmit the call directly using the CHMIProxySpec service.

The CHMI use EHMIProxy listeners to notify each CHMI layer updates. It provides two listeners mechanism to notify when a device or service state change or when an instance is newly instantiated or destroy.

- **Generic methods calls (GenericCommand)**

The GenericCommand is a Runnable object that contains all necessary information (java instance, method name, parameters) to make an asynchronous call of any available method from any specification.

Two GenericCommand class co-exist one for the EHMIProxy that can only trigger method call from the EHMIProxy instance and one for the CHMIProxy that can trigger method from any core instance in the CHMI layer.

When a call request is handling by the EHMI, it returns a GenericCommand from itself or the CHMI that can be executed whenever. The EUDE execute all command itself through the returned GenericCommand object and get the result of the call instead.

- **CHMI notification to client application**

When a change occurred in the CHMI layer this one notifies all subscribers that something happened with subscribed listeners. But it also transmits the notification through the ClientCommunicationManager send service to all connected clients application.

The JSON format of this notification is :

```
{
        objectId :  "CHMI_SOURCE_OBJECT_ID",
        varName : "VARIABLE_NAME_THAT_CHANGED",
        value :  "THE_NEW_VALUE"
}
```

varName attribute value is the same status variable that can be found in object description (See Appendix for device/service description)

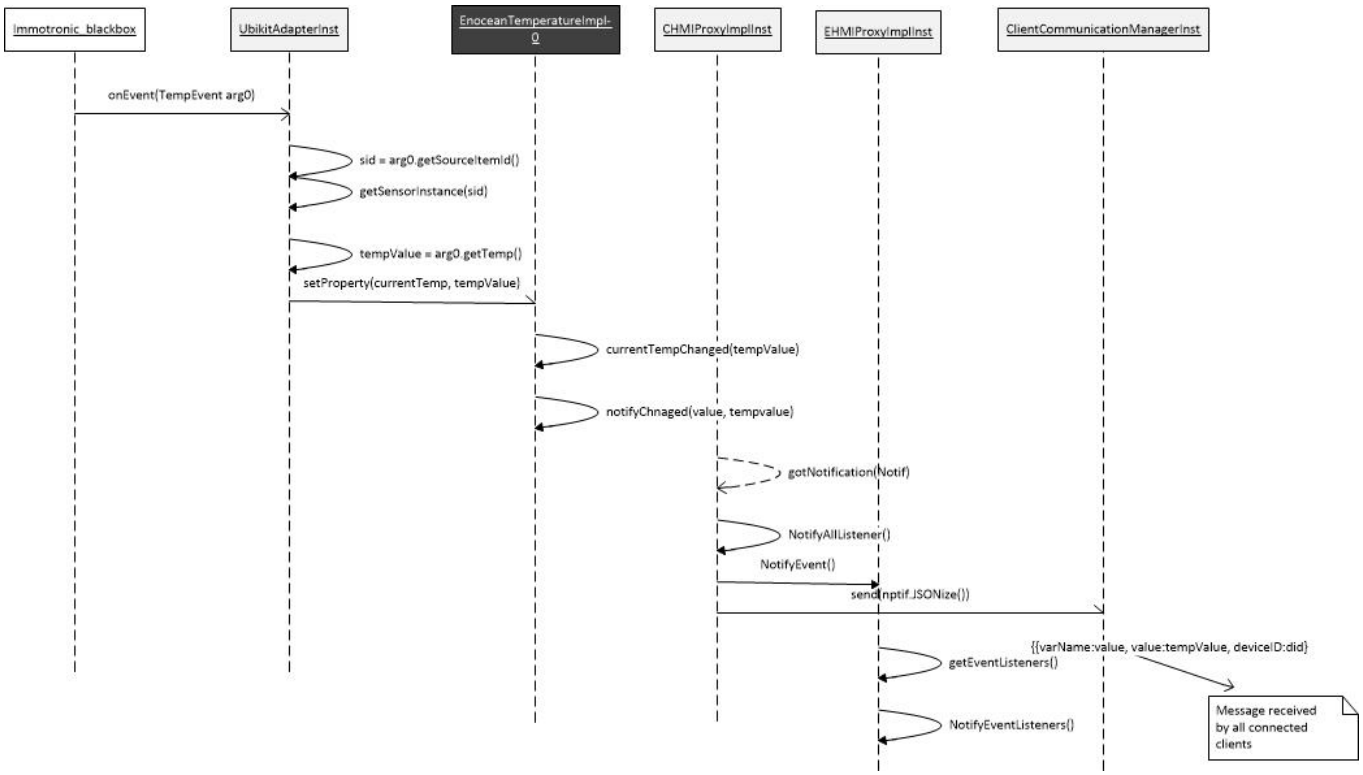

FIG 8: TEMPERATURE SENSOR NOTIFICATION

- **EHMI notification to client application**

As the CHMI the EHMI proxy instance send notification when something change in the context layer. To maintain all connected applications in the same state, the EHMI notify all clients by means of send service provide by the ClientCommunicationManager.

Notification format are describe below:

- **DevicePropertyTable**

```
{
        objectId :  "CHMI_SOURCE_OBJECT_ID",
        varName : "VARIABLE_NAME_THAT_CHANGED",
        value :  "THE_NEW_VALUE",
        userId : "THE-USER_THAT_ADD_ THE PROPERTY"
}
```

- **PlaceManager**

Move object notification

```
{
	moveDevice : {
		deviceId :  "CHMI_SOURCE_OBJECT_ID",
		srcLocationId : "THE SOURCE_LOCATION ID",
		destLocationId : "THE DESTINATION_LOCATION_ID"
	}
	or
	moveService : {
		serviceId :  "CHMI_SOURCE_OBJECT_ID",
		srcLocationId : "THE SOURCE_LOCATION ID",
		destLocationId : "THE DESTINATION_LOCATION_ID"
	}
}
```

Place notification

```
{
	type*: {
		id :  "LOCATION_ID",
		name : "LOCATION_NAME",
		devices : [ALL_DEVICES]
	}
}
```

**type can be addPlace removePlace or updatePlace.*

- **EUDE**

Program Notification

```
{
	changes* : {
		id :  "PROGRAM_ID",
		runningState : "THE_CURRENT_PROGRAM_RUNNING_STATE",
		source :  "THE_PROGRAM_SOURCE",
		userSource: "THE_PROGRAM_USER_METADATA"
	}
}
```

**changes can be empty or describes the change that triggers the update*

Program State notification

```
{
        objectId :  "PROGRAM_ID",
        varName : "VARIABLE_NAME_THAT_CHANGED",
        value :  "THE_NEW_VALUE",
}
```

4. Appendix

This appendix contains all device/service type JSON description with associated grammar and all method for each specification.

Each device has to provide a JSON formatted self-description. This description is commonly use when a remote client ask for the available device or service instance in the CHMI. It can provide a JSON formatted behavior description in a dedicated file (grammar.json). This file is used with the EUDE interpreter to manage state and method calls for each type.

a. DomiCube

The DomiCube is a device from Amiqua4Home team that gives us selected cube face, angular position and battery level notifications. It is integration is made upon MQTT.

- **description**

```
{
        id : "CHMI_DOMICUBE_ID",
        type : "210",
        status : "STATUS_VALUE", #0, 1 or 2
        activeFace : "FACE_VALUE ", #1..6
        batteryLevel : "BATTERY_LEVEL", #0..100
        dimValue : "ANGULAR_VALUE",
        deviceType : "domiCube",
        systemName : "DomiCube"
}
```

- **behavior**

Not found in repository ???

- **methods**

Check if the DomiCube instance has received a message and return true if the DomiCube instance has been synchronized once with real DomiCube.

public boolean hasReceived();

Get the current face number of the DomiCube and return the number of the face as an integer

public int getCurrentFaceNumber();

b. Undefined

This type is used to instantiate device or service not supported yet. That allows end user to receive the object and try to investigate what it is.

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "-1",
        status : "STATUS_VALUE", #0, 1 or 2
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

No behavior for this type.

- **methods**

This method gets the available profiles that corresponding to this ambiguous sensor. If no profiles corresponding to this sensor is not yet implemented. It returns a JSON description that contains the network profile of this sensor and a user friendly interpretation. Ex: For EnOcean switch this method return the following JSONArray

[{"profile":"EEP-05-02-01", "type":"switch sensor"},{"profile":"EEP-05-04-01", "type":"Key card sensor"}]

public JSONArray getCapabilities();

This method send to the EnOcean dump bundle the corresponding profile to that sensor in order to validate the configuration and receive telegrams from it.

public void validate(String profile);

c. Contact

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "3",
        status : "STATUS_VALUE", #0, 1 or 2
        contact : "CONTACT_STATUS" #true or false
```

```
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

```
{
   "friendlyName": "contact sensor",
   "typename":"contactSensor",
   "properties": [
   ],
   "states": [
      {
         "name": "isOpen",
         "stateName" : "getContactStatus",
         "stateValue" : "false",
         "setter": {
            "type": "empty"
         },
         "endEvent": {
            "type": "event",
            "eventName": "contact",
            "eventValue": "true"
         },
         "startEvent": {
            "type": "event",
            "eventName": "contact",
            "eventValue": "false"
         }
      },
      {
         "name": "isClose",
         "stateName" : "getContactStatus",
         "stateValue" : "true",
         "setter": {
            "type": "empty"
         },
         "endEvent": {
            "type": "event",
            "eventName": "contact",
            "eventValue": "false"
         },
         "startEvent": {
            "type": "event",
            "eventName": "contact",
            "eventValue": "true"
         }
      }
   ]
}
```

- **methods**

Get the current contact sensor state and return false if no contact has been detected (Opened) and true otherwise (Closed)

public boolean getContactStatus();

d. Temperature

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "0",
        status : "STATUS_VALUE", #0, 1 or 2
        contact : "TEMPERATURE_VALUE" #true or false
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

```
{
   "friendlyName": "temperature",
   "typename":"temperature",
   "properties": [
   ],
   "states": [
   ]
}
```

- **methods**

public enum TemperatureUnit { Kelvin, Celsius, Rankine, Fahrenheit;}

Get the current temperature unit and return a TemperatureUnit object that represent the temperature unit.

public TemperatureUnit getTemperatureUnit();

Get the current temperature = the last value sent by the temperature sensor and returns the temperature as a float.

public float getTemperature();

e. KeyCard

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "4",
        status : "STATUS_VALUE", #0, 1 or 2
        inserted: "CURRENT_VALUE" #true or false
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

No grammar found in repository

- **methods**

Get the current state of the key card sensor and return true if a card is inserted and false otherwise.

public boolean getCardState();

Get the last card number that has been checked and return the card number

public int getLastCardNumber();

f. On/Off

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "8",
        status : "STATUS_VALUE", #0, 1 or 2
        isOn: "IS_ON_VALUE" #true or false
        deviceType : "ENOCEAN_ACTUATOR"
}
```

- **behavior**

No grammar found in repository

- **methods**

Get the virtual state of this actuator. Nothing is sure that the real device is in this corresponding state. Return true if the device is on and false otherwise.

public boolean getTargetState();

Set the device ON state

public void on();

Set the device OFF state

public void off();

g. Illumination

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "1",
        status : "STATUS_VALUE", #0, 1 or 2
        value: "ILLUMINATION_VALUE",
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

No grammar found in repository

- **methods**

Enum type that defines available luminosity units

public enum LuminosityUnit { Lux,Lumen;}

Get the current luminosity unit and return a LuminosityUnit object that represent the luminosity unit

public LuminosityUnit getLuminosityUnit();

Get the current illumination = the last value sent by the illumination sensor and returns the illumination as an integer

```
public int getIllumination();
```

h. Switch

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "2",
        status : "STATUS_VALUE", #0, 1 or 2
        switchNumber : "SWITCH_NUMBER",
        buttonStatus : "BUTTON_STATUS", #true or false
        deviceType : "ENOCEAN_SENSOR"
}
```

- **behavior**

```
{
   "friendlyName": "sensor",
   "typename":"switchSensor",
   "properties": [
   ],
   "states": [
   ]
}
```

- **methods**

Inner class that define an action when an end user press the switch button

```
public class Action {
        //Swith Number
        byte switchNumber;
        // The state of the button On/Off = Up/Down - none = neutral position
        String state;
}
public Action getLastAction();
```

i. Smart Plug

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "6",
        status : "STATUS_VALUE", #0, 1 or 2
        plugState : "PLUG_STATUS", #true or false
        consumption : "CONSUMPTION_VALUE",
        activeEnergy : "ACTIVE_ENERGY_VALUE",
        deviceType : "ACTUATOR"
}
```

- **behavior**

```
{
   "friendlyName": "Enocean plug",
   "typename":"enoceanPlug",
   "properties": [
   ],
   "states": [
   ]
}
```

- **methods**

Toggles the smartplug, i.e. switches it's on/off state.

public void toggle();

Toggles the smartplug, i.e. switches it's on/off state.

public void on();

Toggles the smartplug, i.e. switches it's on/off state.

public void off();

Return the consumption in Watt

public int activePower();

Return the summation of the active energy in W.h

public int activeEnergy();

Get the relay state

public boolean getRelayState();

j. Philips HUE

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "7",
        status : "STATUS_VALUE", #0, 1 or 2
        value : "CURRENT_STATE", #true or false
        color : "COLOR_VALUE",
        saturation : "SATURATION_VALUE",
        brightness : "BRIGHTNESS_VALUE",
        deviceType : "HUE_ACTUATOR"
}
```

- **behavior**

```
{
   "friendlyName": "lamp",
   "typename": "lamp",
   "properties": [
   ],
   "states": [
      {
         "name": "isOn",
         "stateName" : "getState",
         "stateValue" : "true",
         "setter": {
            "type": "action",
            "methodName": "On"
         },
         "endEvent": {
            "type": "event",
            "eventName": "state",
            "eventValue": "false"
         },
         "startEvent": {
            "type": "event",
            "eventName": "state",
            "eventValue": "true"
         }
      },
      {
         "name": "isOff",
```

```
            "stateName" : "getState",
            "stateValue" : "false",
            "setter": {
                "type": "action",
                "methodName": "Off"
            },
            "endEvent": {
                "type": "event",
                "eventName": "state",
                "eventValue": "true"
            },
            "startEvent": {
                "type": "event",
                "eventName": "state",
                "eventValue": "false"
            }

        }
    ]
}
```

- **methods**

Get the current light status and return a JSON description of the light (JSON attribute depends on color light implementation)

public JSONObject getLightStatus();

Get the current light color and return the current color as a long integer

public long getLightColor();

Get the current light brightness and return the current brightness as an integer

public int getLightBrightness();

Get the current light color saturation and return the current light color saturation as an integer

public int getLightColorSaturation();

Get the current light state (On/Off) andreturn true if the light is On and false otherwise

public boolean getCurrentState();

Get light manufacturer details and return a JSON object that contain details from manufacturer.

public JSONObject getManufacturerDetails();

Set the complete status of the color light. This method allow to set several attribute at the same time.

public boolean setStatus(JSONObject newStatus);

Switch the light on and return true if the light turn on, false otherwise

public boolean On();

Switch the light off and return true if the light turn off, false otherwise

public boolean Off();

Set the light color

public boolean setColor(long color);

Set the color light brightness

public boolean setBrightness(long brightness);

Set the color saturation

public boolean setSaturation(int saturation);

Set the light color to red

public boolean setRed();

Set the light color to blue

public boolean setBlue();

Set the light color to green

public boolean setGreen();

Set the light color to yellow

public boolean setYellow();

Set the light color to orange

public boolean setOrange();

Set the light color to purple

public boolean setPurple();

Set the light color to pink

public boolean setPink();

Set the light color to bright white

public boolean setWhite();

Set the default light color

public boolean setDefault();

Increase the light brightness by step

public boolean increaseBrightness(int step);

Decrease the light brightness by step

public boolean decreaseBrightness(int step);

Invert the light mode

public boolean toggle();

Make the light blink one time

public boolean blink();

Make the light blink for 30 seconds

public boolean blink30();

Turn the light into a loop between all the color spectrums

public boolean colorLoop();

k. CO2

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "9",
        status : "STATUS_VALUE", #0, 1 or 2
        value: "CO2_VALUE",
        deviceType : "SENSOR"
}
```

- **behavior**

```
{
   "friendlyName": "CO2 sensor",
   "typename":"wattecoCo2Sensor",
   "properties": [
   ],
   "states": [
   ]
}
```

- **methods**

Get the current CO2 concentration and return the Co2 concentration as an integer (ppm unit)

```
public int getCO2Concentration();
```

l. Occupancy

- **description**

```
{
        id : "CHMI_OBJECT_ID",
        type : "5",
        status : "STATUS_VALUE", #0, 1 or 2
        occupied: "OCCUPIED_VALUE", #True or False
        deviceType : "SENSOR"
}
```

- **behavior**

```
{
   "friendlyName": "occupancy sensor",
   "typename":"wattecoOccupancySensor",
   "properties": [
   ],
   "states": [
   ]
}
```

- **methods**

Get the occupied status and return true if the occupied status is true, false otherwise

public boolean getOccupied();

m. Adapters

Adapters can also communicate they use a standalone client communication manager subscription and they use their won target to communication with dedicated web GUI.

UbikitAdaper --> "ENOCEAN"

Watteco --> "WATTECO"

PhilipsHUE --> "PHILIPSHUE"

UPnP --> "UPnP"

All communication is directly redirected to the adapter and does not use the EHMIProxy or CHMIProxy mechanism. See below sequence diagram for EnOcean pairing sensor process.

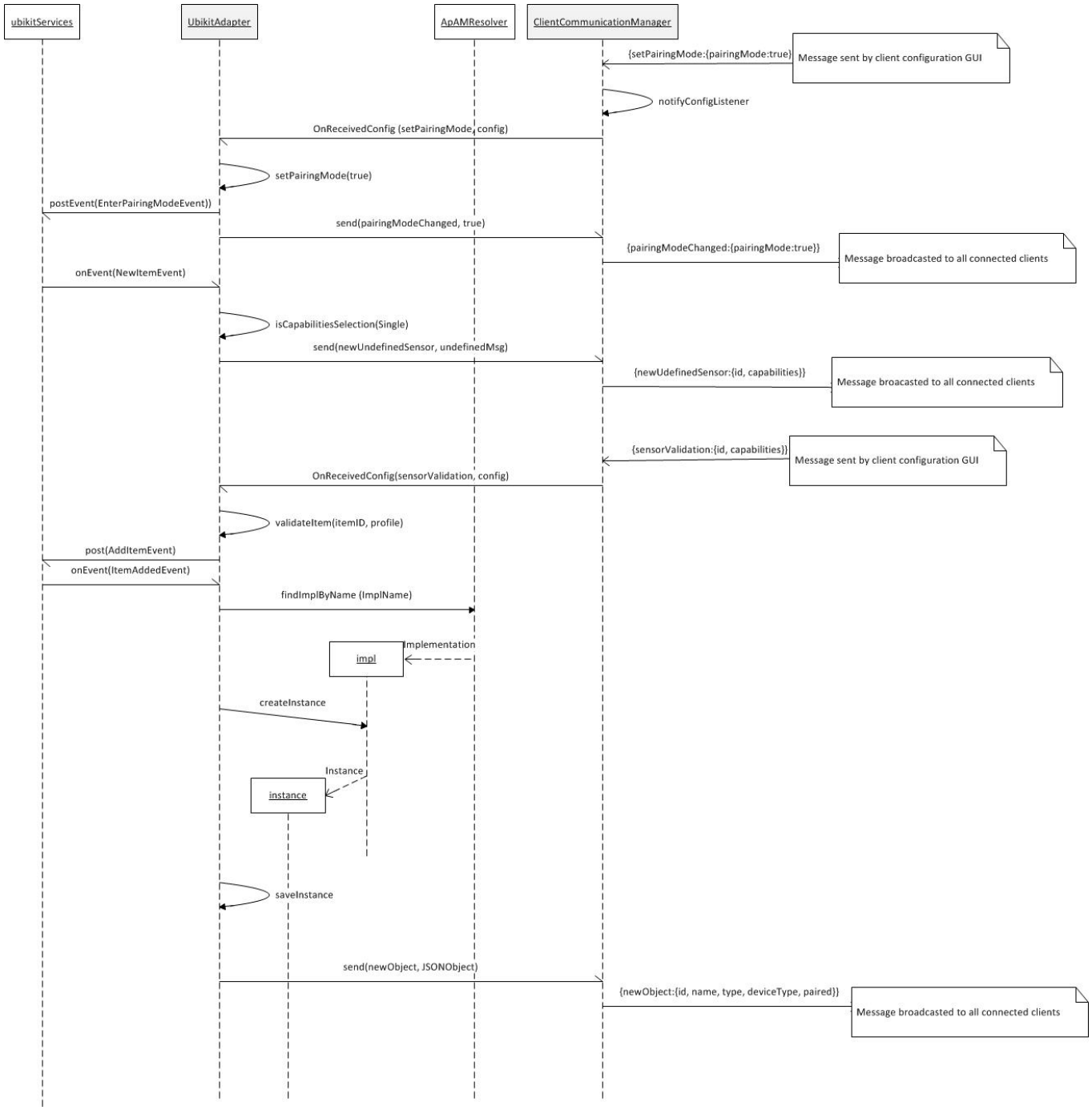

FIG 9: ENOCEAN MANUAL PAIRING

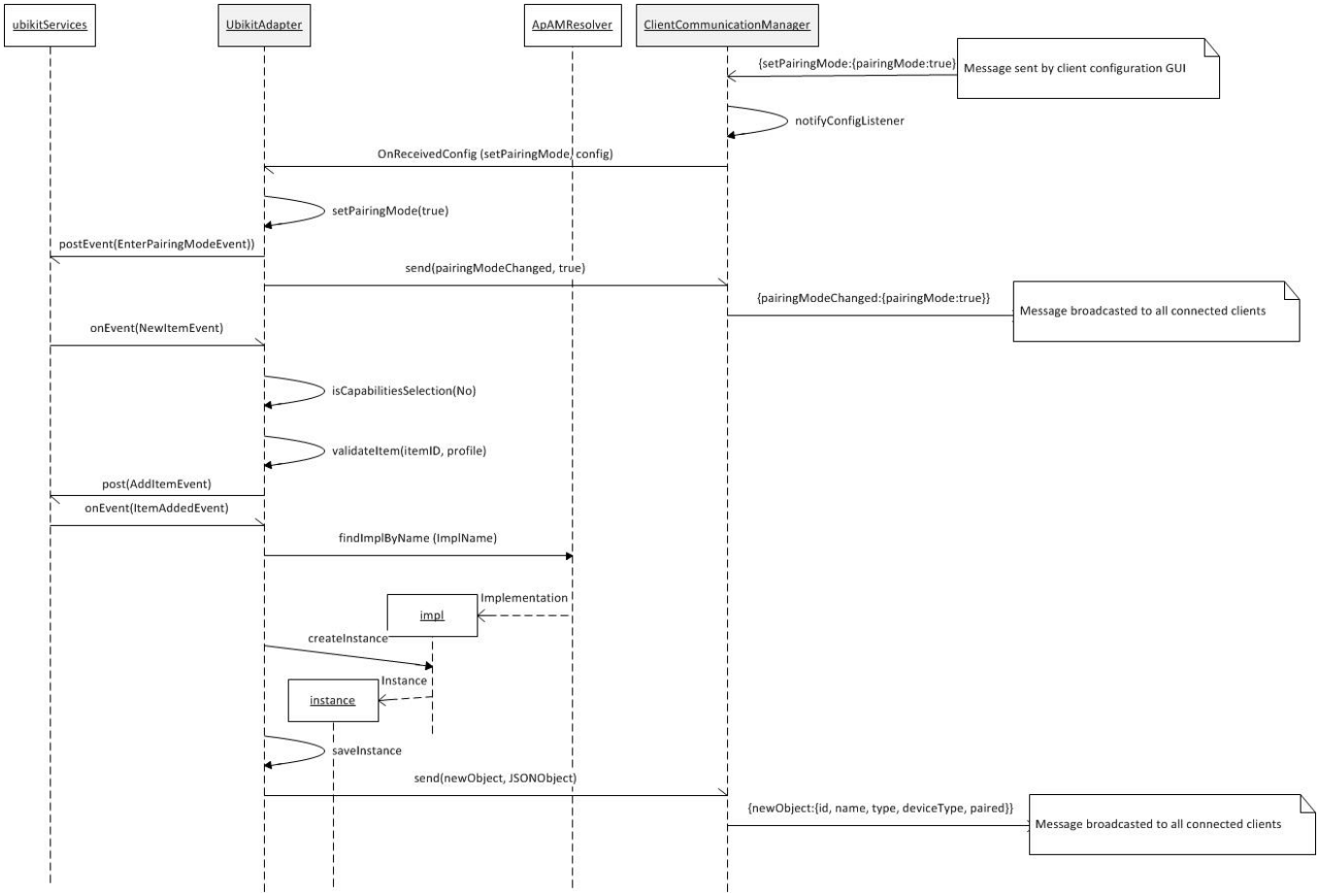

FIG 10: ENOCEAN AUTO-PAIRING PROCESS