Target Outcome: b) Sustainable technologies for a Smarter Society

(FP7 614100)

D3.1 Resource Adaptation Interface Framework

June 30, 2014 – Version 1.0

Published by the Impress Consortium

Dissemination Level: Public

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

Document control page

Document file: D3.1 Resource Adaptation Interface Framework_v1.0
Document version: 1.0
Document owner: Enrico Ferrera (ISMB)

Work package: WP3 - Resource Abstraction and IoT Communication Infrastructure
Task: T3.1 Resource Adaptation Interface and Integration Support tool
Deliverable type: P

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

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Davide Conzon	2014/04/18	Initial structure of the document. Contributions in all sections.
0.2	Enrico Ferrera	2014/06/03	Contributions in all sections, added architecture figures and chapter 5.
0.3	Enrico Ferrera, Davide Conzon	2014/06/17	Document ready for internal review.
1.0	Enrico Ferrera	2014/06/30	Document fixed according the internal review.

Internal review history:

Reviewed by	Date	Summary of comments
Peter Rosengren	2014/06/30	Accepted with minor comments

Index:

1 Executive summary

This document describes the role and architecture of the Resource Adaptation Interface (RAI) and its integration within the Service Proxy. Moreover, APIs for leveraging RAI functionalities and for extending it are described.

RAI consists in a set of existing classes and methods that can be used in order to monitor and control application-level resources. In fact, RAI aims to abstract the concept of resource (i.e. physical devices or third-party systems) providing a generic virtual "device" that can be used to seamlessly communicate with resources, despite of technology-specific implementation details. RAI aims to ease and speed up the integration of application-level resources within IMPReSS platform.

The rest of this deliverable is organized as follow: in Section 2 RAI and the concept of abstraction is introduced and defined. Section 3 describes the RAI architecture and the role of the component within the IMPReSS platform and more specifically within Service Proxy, explaining also how to extend RAI adding not-previously-managed resources into a RAI instance. Finally, in Section 4, APIs and procedures for using RAI are specified.

Deliverable D3.1 actually consists in the first prototypical implementation of the RAI, for this reason this document has to be considered a compendium of the software and consequently the number of pages have been kept restricted.

2 Introduction

The Internet-of-Things (IoT) is based on a plethora of heterogeneous objects, each one providing specific functionalities that are accessible through specific communication protocols. For this reason, an *abstraction and adaptation layer* is necessary, in order to blend the access of many different resources in a common language and set of procedures. Standing to [1], an IoT platform architecture is layered as shown in Figure 1. The Object Abstraction component is the very low layer of an IoT platform and it is located just before the IoT objects, taking part of the IoT ecosystem.

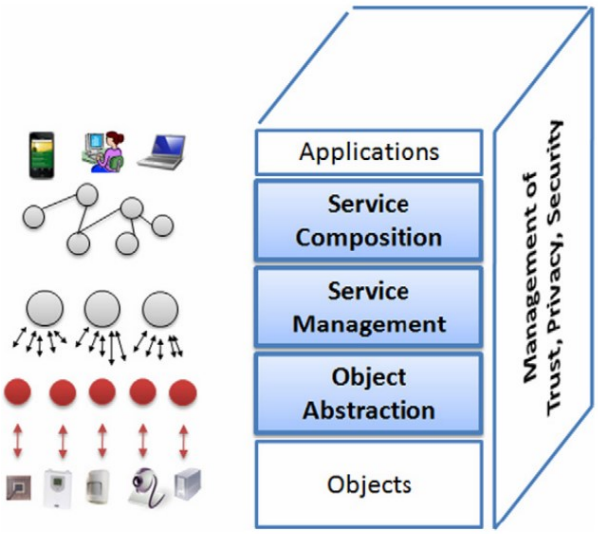

Figure 1: IoT platform architecture

RAI aims at cover the role of abstraction layer. More specifically, RAI is a framework that collects a set of existing classes and methods that can be used in order to monitor and control application-level resources. In fact, RAI aims to abstract the concept of resource (i.e. physical devices or third-party systems), providing a generic "device" that can be used to seamlessly communicate with resources despite of technology-specific implementation details. RAI aims to ease and speed up the integration of application-level resources within IMPReSS platform. The main goal of the Resource Adaptation Interface is to integrate and expose the features provided by heterogeneous physical and non-physical resources (e.g. sensors or actuators, personal devices of engaged citizens, services providing public data), providing to the IMPReSS platform a uniform way to communicate with them, despite the differences in term of hardware and software.

RAI is located on the extreme edge of the IMPReSS platform, just before the hardware and software resources, and can cooperate with the LinkSmart middleware, which the platform is based on. The heterogeneous nature of physical devices requires finding a way to interact simply with the resources. For this reason, the architecture of the RAI has been designed to be modular, in order to make it extensible through the addition of new resource drivers abstracting new, previously unhandled, entities. RAI can abstract both physical resources (e.g. sensors, actuators) and third-party services (e.g existing platform such as Xively [2], meteo forecast web service, etc.).

The following of the document provides an overview of the current design and implementation of the RAI. Further additions and refinements to the component are expected in subsequent phases of the project.

3 Resource Adaptation Interface Architecture

3.1 Context

As indicated in [3] the RAI is a sub-component of the Service Proxy. A Service Proxy is an IMPRESS platform component that is responsible for the implementation of the interface between the resources and their users. In other words, a Service Proxy is a software component that is responsible for the abstraction of the resources, hiding the various implementation details (e.g. communication protocols and data format), and instead providing a uniform virtualization for them within the IMPRESS platform. In other words, when applications need to interact with the physical world through ALRs, they can do it through virtual resources (i.e. Service Proxies) that provide a bridge to the actual resources (e.g. sensors and actuators), exposing their offered services. The Service Proxy architecture with the RAI is shown in Figure 2. The role of the RAI is to abstract Application-level Resources and expose their functionalities in a common way through the provided API. The Local Resource Manager can interact with the RAI using the exposed API, handling thus, the requests received by the IMPReSS applications.

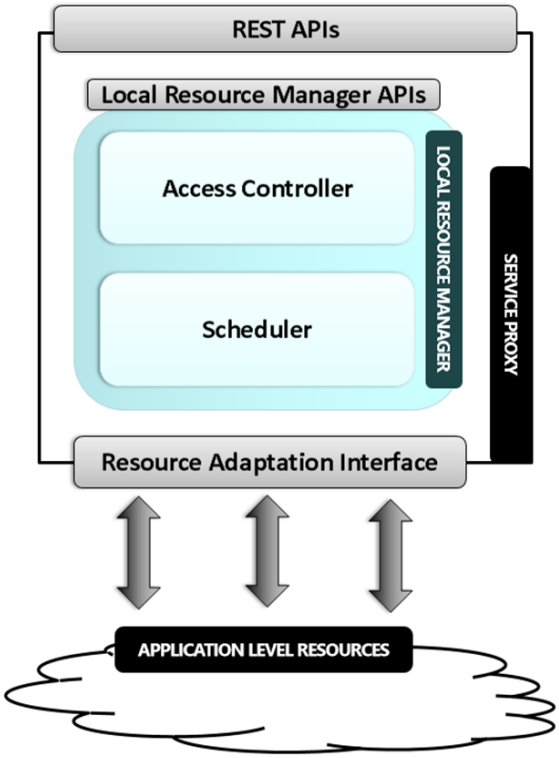

Figure 2 - Service proxy architecture

In the next sections, a deeper RAI architecture description is addressed.

3.2 Architecture description

The RAI architecture is composed by three layers, completely decoupled with each other. In this way, it is possible to change one of them, without requiring many modifications to the others.

The RAI virtualizes each resource as a virtual Device that exposes features or functionalities, provided by physical devices or third-party services, through a set of methods and parameters defined by specific Java interfaces. These interfaces are derived from a set of resource models that describe minimal methods/services of a basic set of resource types (e.g. temperature sensor, humidity sensor, switch, etc.). The models must be defined whenever a new resource type have to be integrated within the platform and it is not previously managed by the RAI. For example, if we want to integrate within the platform a temperature sensor for the first time, we have to define a minimal model for the temperature sensor type. Whenever a new temperature sensor, using different technology and

protocol, have to be integrated within the platform, the RAI virtualized device must implement the interface derived from the previously defined temperature sensor model.

The interfaces provided by RAI are used by the LRM to interact with the available resources. The methods defined by the specific resources interfaces are passed as parameters to the method *requestResource(String appID, String operation)* provided by the LRM APIs – see [3].

In Figure 3 is shown the layered architecture of the Resource Adaptation Interface.

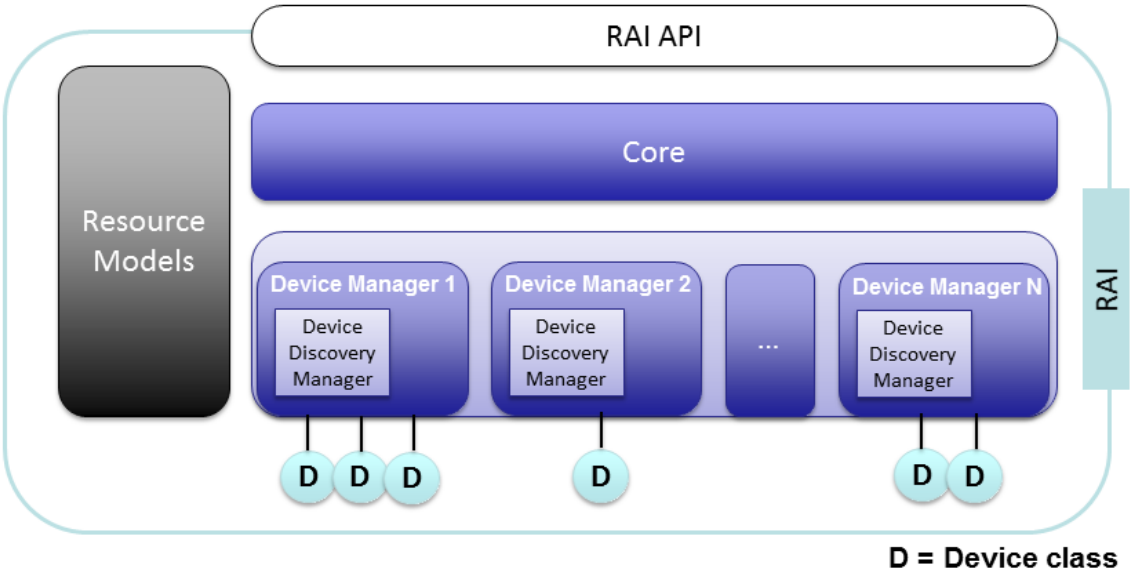

Figure 3 - Resource Adaptation Interface Architecture

The lower layer of the architecture consists in a set of technology-specific *DeviceManager*(s) classes that are responsible for the actual integration of different resources. These components are able to handle specific types of networks and furthermore, they contain the implementation of specific device discovery features. The device-level discovery topic will be deeply addressed within deliverable D3.2 due at M18.

A set of application-level resource models, contained in a dedicated repository, are used for the virtual device interface definition. The modelled interfaces are implemented with specific commands depending on the specific resource to be integrated. For the first implementation of the RAI, the models consist just in a set of Java interfaces that define a basic number of methods/services provided by the most common device types. For next versions of RAI, it is going to be investigated the use of ontology descriptions for resource models.

The middle layer is the RAI core, which is in charge to map the southbound devices and to notify upper layers about each network changing.

The upper layer is responsible for the exposition of the methods/services provided by the resources. This layer is made of the APIs offered by the RAI core, in order to retrieve and manage virtual devices and call their resource-specific methods.

4 RAI Development Kit

The RAI Development Kit (RDK) is a set of interfaces, classes and models useful to extend RAI features with new, previously unhandled, resources. In the next sections the whole RDK is described, using also UML class diagrams.

4.1 Interface Device

This is the interface, which all devices are based on. It is extended by all the device models interfaces. By extending it, each modelled device interface shares the same base object. In this way, every device discovered (or removed) can be forwarded to the RAI core as a generic Device object, which can be transformed into a strongly typed object making a simple caste based on the *DeviceType* field. It is also possible to retrieve the NetworkType to which each device belongs to in order to allow the application to query the RAI asking for a specific device type belonging to a specific devices network.

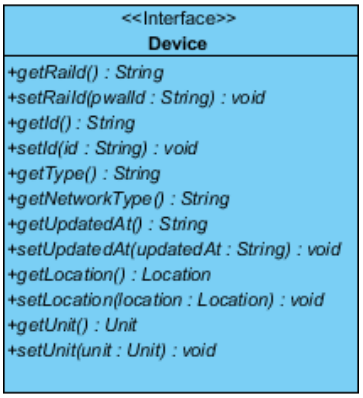

Figure 4 - Interface Device

4.2 Class DeviceType

The *DeviceType* class is a utility class, which contains a set of static strings that define the type of devices established during the devices integration phases. When new devices types are added to this class, these will be shared and make them available at every level of the architecture.

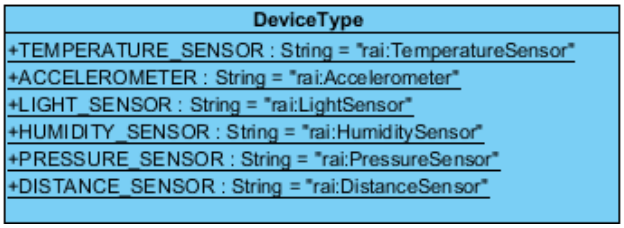

Figure 5 - Interface DeviceType

4.3 Class DeviceNetworkType

This class, as the previous one, is a utility class containing static strings, defining type of networks established during devices integration phases. When new networks types are added to this class, these will be shared and make available at every level of the architecture. Despite the abstraction goal of the RAI, it can be required by the application the original sensor type in order to, for instance, classify them or for debug purposes.

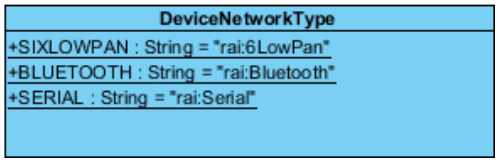

Figure 6 - Class DeviceNetworkType

4.4 Enum DeviceManagerStatus

This is an enum, which represent all the possible devices managers' status (currently only STARTED and STOPPED are supported). This information is useful to avoid, for example, start commands or stop commands sent twice.

4.5 Device models interfaces

The device models interfaces are a set of interfaces that model resources and provide a set of basic functionalities that have to be implemented, in order to be compliant with a specific resource type. The actual implementation of the device models interfaces changes depending on the specific communication protocol used by each resource. The definition of basic device models can be considered as a strong commitment, but these interfaces can be extended to include different functionalities, so all kind of devices can be modelled just adding new interfaces.

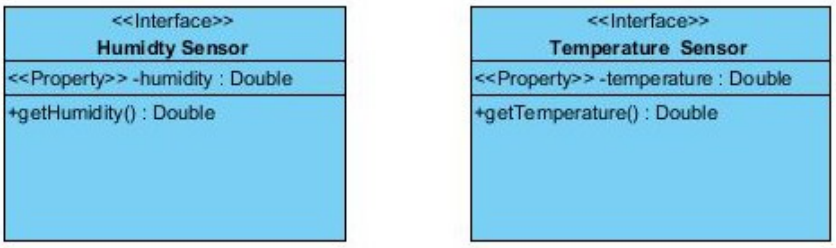

Figure 7 - example of devices' model interface

The set of models are collected in a dedicated repository. For the first implementation of the RAI, the models consist just in a bunch of Java interfaces that define a basic number of methods/services provided by the most common device types. For next versions of RAI, it is going to be investigated the use of ontology descriptions for resource models.

4.6 Abstract class DevicesManager

DevicesManager is an abstract class that has to be extended by each device manager. This class implements the Runnable interface, so each device manager runs in a different thread. In this way, it can be started and stopped directly, using the RAI.

A class diagram of the *DevicesManager* class is shown in Figure 8

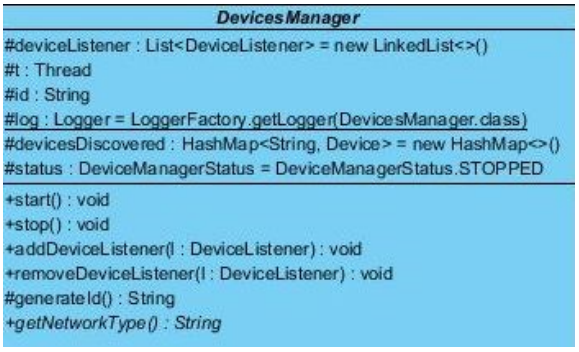

Figure 8 - class DevicesManager

The most significant components of this class are:

- The methods start() and stop() (already implemented), to allow dynamic start and stop of each device manager.
- A hash map (named *devicesDiscovered*), which is used to store the discovered device. It is used to be always aware about which devices are currently available.

- A list of *DeviceListener,* with two methods to add and remove device listeners. The RAI core, which implements the *DeviceListener* interface, is notified every time a device, belonging to a *DevicesManager*, is added or removed.

The next subsection provides an example of implementation of the *DevicesManager* interface.

4.6.1 DevicesManager implementation

```
public class SpecificProtocolManager extends DevicesManager{
private String id;
@Override
public void run() {
log.info("SpecificProtocol manager stared.");
      while(!t.isInterrupted())  {
            // the discovery phase has to
            // be implemented depending on the specific protocol
            List<Device> devices = specificProtocolDiscovery();
            for(Device d : devices) {
                  switch(d.getType()) {
                  case(DeviceType.Thermometer):
                        ThermometerDevice term = new ThermometerDevice();
                        devicesDiscovered.put(this.generateId(), term);
                        for (DeviceListener l : this.deviceListener) {
                              l.notifyDeviceAdded(term);
                        }
                  break;
                  case (…):
                        …… //other devices types
                        break;
                  }
            }
            … … … …
            //discovered a device that has been removed
            this.devicesDiscovered.remove(term);
            for (DeviceListener l : this.deviceListener) {
                  l.notifyDeviceRemoved(term);
            }
      }
}

public void setId(String id) {
      this.id=id;
}

@Override
public String getId() {
      return id;
}

@Override
public String getNetworkType() {
      return DeviceNetworkType.SERIAL;
}

}
```

In a device manager, the entire logic is located into the method named *run*. Since a real discovery phase is not present in all the devices (for example the serial protocol), the *DeviceManager* abstract class does not provide any discovery method, the developers have to handle the discovery phase, as they want.

When a manager discovers a new device, it has to use a Java Object (extending the *Device* interface), which contains the actual protocol-dependent implementations for the device. Then, this object has to be saved in the *discoveredDevices* hash map and a notification has to be sent to the RAI core.

5 Resource Adaptation Interface APIs

This chapter describes the upper layer of the RAI architecture: the application programming interfaces. Every software entity that want to access RAI has to leverage on its APIs, which allow to retrieve and manage the available resources and to monitor and control them through the methods offered by the virtual Device(s).

In Figure 9 is shown the typical architecture design using the RAI APIs.

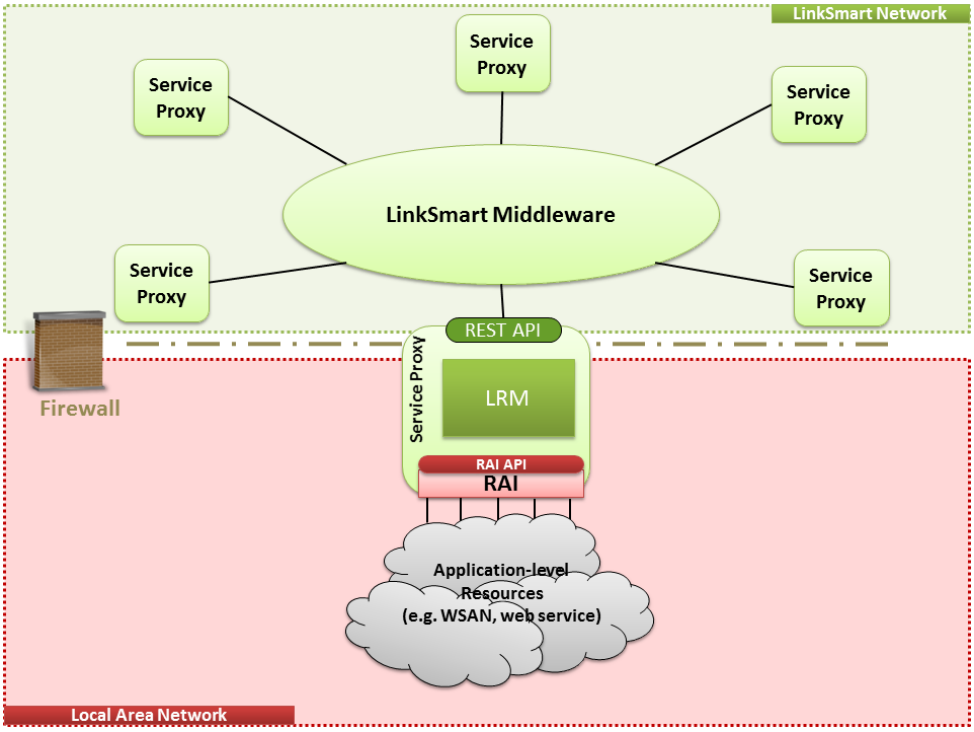

Figure 9: Architecture using RAI API

5.1 RAI Core APIs

The RAI core interface contains methods for:

- Receiving and handling events from the bottom layer.
- Serving requests and forwarding notifications to the upper layer.

Furthermore, as shown in Figure 10, this interface includes methods for retrieving a reference to a single *Device* object and for filtering groups of devices by *DeviceType* and *DeviceNetwork*. In addition, methods to interact directly with the devices managers are provided.

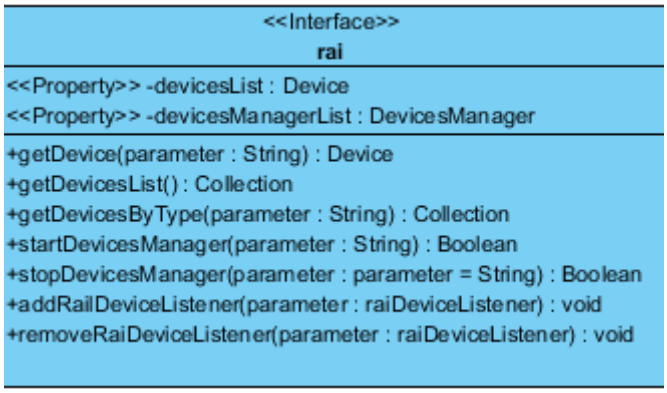

Figure 10 – Interface RAI core

For instance, in order to monitor and control available resources, the Service Proxy's LRM has to retrieve the addressed virtual Device through the getDevice method and then, it has to call the specific method of the resource (e.g. device.getTemperature) from the virtual Device object.

5.2 RAI core

The RAI core is a class implementing the RAI core interface. It provides an actual implementation of the methods useful to interact with the physical world and get values from physical devices. This component implements the *DeviceListener* interface, because it has to be aware of the current list of the devices available in the RAI instance, in order to expose them to the other components like the connectors.

5.3 RAI connectors components

A single RAI instance is able to abstract many different resources at the same time. It needs just to execute the specific DeviceManager(s) for the specific resources to abstract. An IMPReSS Service Proxy leverages on RAI APIs and can expose within the IMPReSS ecosystem the resources as "services" to be consumed. The Service Proxy can thus offer a number of services depending on the available resources that RAI can manage.

To generalize the vision of Service Proxy as a service aggregator, we can also consider an architecture as shown in Figure 11, which introduce the Connector concept.

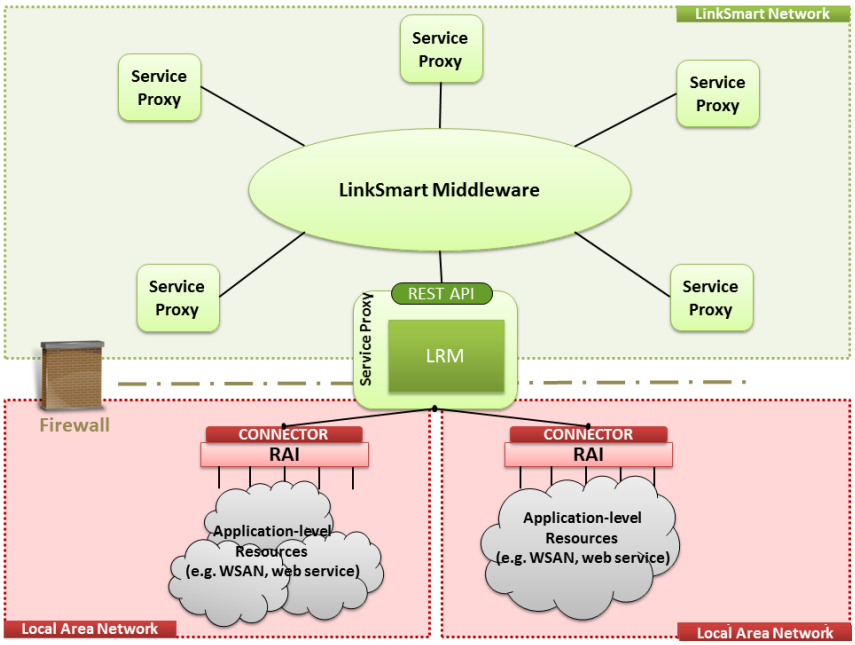

Figure 11: Architecture using RAI connectors

In this scenario, the Service Proxy works as aggregator of geographically-distributed application-level resources abstracted by geographically-distributed RAI instances. For instance, suppose to have different Wireless Sensors and Actuators Networks (WSAN), belonging to local area networks, that monitor and control different areas of a company. If we want to expose to the LinkSmart world the "MyCompany" service, made of services provided by the different WSANs, we can leverage on the RAI connector components. The connectors are additional components that are placed upon RAI API (as shown in Figure 12) and are in charge to expose application-level resources, using standard communication protocols like REST, SOAP, XMPP, MQTT, etc.

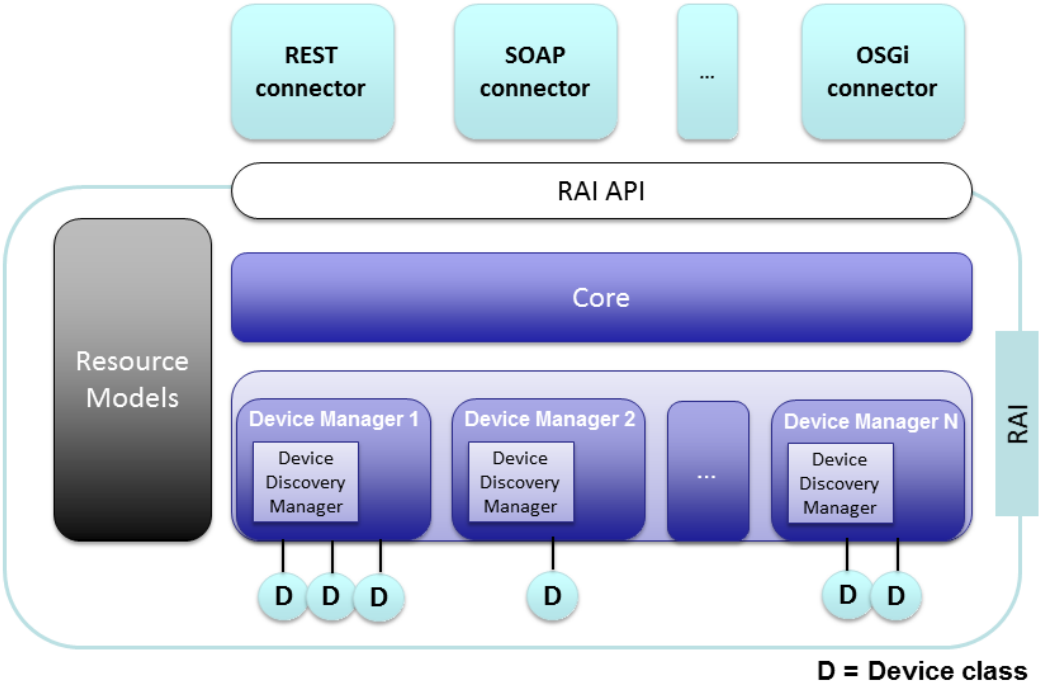

Figure 12: RAI architecture with Connectors

In this way, geographically-separated resources, abstracted and virtualized by RAI, are able to communicate with a unique Service Proxy that is responsible to aggregate all the available functionalities, exposing these as a unique complex service. For example, suppose to install, within a two-floors building, a network of switches that control the first floor and another one for the second floor. Both network are within the local area network of the building but are physically installed in two separated areas. At LinkSmart level, the whole switching system could be seen as a unique switching service for the building. In order to satisfy this requirement, the switching service can be exposed as a unique service through a Service Proxy that aggregate the two different switching service (one at the first floor and the other one at the second floor) leveraging on the RAI virtualization operated on the two different physical networks. In this way, within the LANs can be used the desired communication protocol (through the use of connectors) and use the Service Proxy as a gateway for the IMPReSS IoT platform. For security reasons, the connectors should be placed just within local area networks, protected by firewalls, otherwise the resources could be controlled directly by malicious agents, which bypass LinkSmart middleware and mixed-criticality managers (i.e. Resource Managers module [3]) developed within IMPReSS platform.

At this stage of development, it has been implemented just a REST connector. In the next section are reported the currently available REST API exposed by the RAI REST Connector.

5.4 RAI REST API

In this section are described all the REST resources currently exposed by the RAI REST Connector.

The table below shows a list of REST resources, including a resource description and an example of usage. Currently, the RAI REST Connector returns JSON documents to avoid performance issues, but it will be always possible to modify these resources to return both JSON and XML documents.

Resource	Description	Output example
/devices?deviceType={}&networkType={}	This resource retrieves the detailed list of the devices adapted by the RAI, parsed as a JSON vector. This resource filters device basing on the network and the device type. Parameters: • **networkType**: the network type to which the device belong (optional) • **deviceType**: the device type to which the device belong (optional)	HTTP GET devices/ [{ "id": "3300", "raiId": "bc71ffa1-20ae-4c2c-8e54-1afb80d3088f", "type": "rai:Thermometer", "networkType": "rai:Xively", "latitude": 43.44984, "longitude": -3.83006,

		"updatedAt": "2014-05-14 06:38:08", } …]

/devices/{deviceId}	This resource retrieves details about a specific device, parsed as a JSON.
	Parameters:
	• **deviceID**: is the id assigned by the RAI
		HTTP GET devices/bc71ffa1-20ae-4c2c-8e54-1afb80d3088f
		{
		"id": "3300",
		"raiId": "bc71ffa1-20ae-4c2c-8e54-1afb80d3088f",
		"type": "rai:Thermometer",
		"networkType": "rai:Xively",
		"latitude": 43.44984,
		"longitude": -3.83006,
		"updatedAt": "2014-05-14 06:38:08",
		"temperature": 32
		}

/devicemanagers	This resource retrieves device managers list. The result is a JSON vector containing a list of devices manager available.	HTTP GET /getalldevicemanagers
		[
		{
		"id": "7cfe96f3-6b69-442e-861a-989add208bc5",
		"status": "STARTED",
		"networkType": "rai:Xively"
		}
		……
		]

/devicemanagers/{deviceManagerId}	This resource retrieves device manager details. The result is a JSON containing details about the selected device manager.
	Parameters:
	• **deviceManagerId:** the device manager id assigned by the RAI
		HTTP GET /getalldevicemanagers/7cfe96f3-6b69-442e-861a-989add208bc5
		{
		"id": "7cfe96f3-6b69-442e-861a-989add208bc5",
		"status": "STARTED",
		"networkType": "rai:Xively"
		}

/devicemanagers/{deviceManagerId}	This resource has the same URL of the previous one, but it accepts POST. This can be used to change status to a specific device manager. This resource could be used to change any devices manager field.
	Possible result are:
	• **200 OK**: everything is correct
	• **503 SERVICE_UNAVAILABLE**: change device manager status fails
	• **404 NOT_FOUND**: the required device manager id does not exist
	Parameters:
	• **deviceManagerId**: is the device manager id assigned by the RAI
		HTTP POST /devicemanagers/{deviceManagerId}
		
		{
		"status": STARTED | STOPPED
		}

6 Summary & Conclusion

In this deliverable, the current Resource Adaptation Interface architecture and features have been outlined. Furthermore, it is indicated how the RAI interacts with the IMPReSS platform: a resource from the physical world is abstracted by the Resource Adaptation Interface and virtualized within the platform as a provider of services by the Service Proxy. Finally, two possible ways to use RAI are proposed: the first one uses RAI as a library, the second one as a separated local network component through the addition of RAI Connectors. Both RAI API and REST Connector API are specified and described.

This document presented a first version of RAI, for next release the APIs are going to be extended with features useful for network management (e.g. method for sensor latency measuring, sensors status, etc.) and event management (e.g. a sensor provide a measure asynchronously). Furthermore, ontology description are going to be investigated for resource models useful for interfaces definition.

7 Bibliography

[1] L. Atzori, A. Iera e G. Morabito, «The Internet of Things: A survey,» *Computer networks 54(15),* pp. 2787-2805, 2010.

[2] «Xively,» 2014. [Online].

[3] IMPReSS, «IMPReSS project - D4.2 Device and Subsystem Resource Management,» 2014.