(FP7 614100)

D3.3 Communication Management

August 30 – 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:	D3.3 Communication Management.docx
Document version:	1.1
Document owner:	Ferry Pramudianto (Fraunhofer FIT)
Work package:	WP3. Resource Abstraction and IoT Communication Infrastructure
Task:	Task 3.3 Communication Management
Deliverable type:	P
Document status:	☒ approved by the document owner for internal review ☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Ferry Pramudianto	3/7/2015	Introduction
0.2	José Ángel Carvajal Soto	9/7/2015	LinkSmart GlobalConnect
0.3	Alexandr Krylovskiy	20/7/2015	LinkSmart LocalConnect
0.4	Ferry Pramudianto	29/7/2015	Ready made for internal review
1.0	Ferry Pramudianto	12/8/2015	Improved according to the reviewers' comments
1.1	Marc Jentsch	31/8/2015	Final polishment

Internal review history:

Reviewed by	Date	Summary of comments
Jussi Kiljander	3/8/2015	Accepted with minor comments
Enrico Ferrera	5/8/2015	Accepted with minor comments

Index:

1. Executive summary

This deliverable describes the IMPReSS communication management, which is extended from the LinkSmart middleware. In this deliverable, we describe an overhaul of the communication management concept involving local network communication (communication which does not require public IP addresses), which is called LocalConnect and the GlobalConnect for enabling communication, which requires interaction over the internet and behind firewalls.

In the local network communication, IMPReSS relies on two well-known communication standards including MQTT for enabling publish subscribe communication and REST based communication for enabling pull based communication. In addition it includes a Catalog service enabling applications to find the appropriate communication endpoints. In contrast to the IoTResource discovery, the LocalConnect Catalog provides information how to access the device through the endpoints that are exposed by the IoTResources, meanwhile the IoTResource discovery provides semantic information to understand what the values refer to. IoTResource discovery takes the information provided by the LocalConnect Catalog to provide syntactic information so that the users do not need to access both of them when developing applications.

The GlobalConnect on the other hand, forms an overlay network over the internet and provides a tunneling service to send messages through its backbone network that can be easily exchanged using different technology. Currently it supports JXTA and 0MQ that leverage on P2P and centralized architecture respectively. The GlobalConnect acts as a router and hides the actual addresses of the service providers and make them seems as local services to ensure the privacy of the service providers.

2. Introduction

The current LinkSmart depends on x86 based gateways (PCs) to host the middleware components. Unfortunately, this requires the users to have a high initial investment for hosting the applications built on top of the LinkSmart middleware. Moreover, relying on PCs for automating appliances is very inefficient from the energy consumption perspective.

Alternatively, ARM architecture offers a sufficient computing power to run a home or personal gateway with much lesser costs. While the cheapest x86 based gateway would cost around 200 USD, ARM based computing platform could go as low as 30 USD. Moreover, ARM has been used widely for mobile devices because of its very efficient energy consumption and at the same time provides sufficient computing power for mobile devices. This feature is very beneficial for energy management applications where the overhead of the system must be significantly smaller than the energy saved in order to achieve a higher return of investment from ICT infrastructure.

Thus, in task 3.3, we extended the LinkSmart core architecture and components allowing the system to scale down or up easily for different types of gateway. For this purpose, we designed the communication manager with exchangeable backbone network as plugins. For the backbone network, we investigated lightweight communication protocols that have gained a lot of support from IoT community and would work for ARM and x86 architecture.

Initially, the LinkSmart backbone network relies on the P2P paradigm since it offers an ideal solution to overcome a single point of failure when using distributed resources to perform critical functions. LinkSmart also uses a P2P network to manage all communication inside the middleware omitting the need of a central component. The LinkSmart P2P implementation is based on JXTA, i.e. a set of open protocols that allow the connection and the exchange of information among communication devices. In LinkSmart, all communication to devices is routed through an overlay P2P network, leveraging on tunneling mechanisms, allowing services to communicate with each other even if they are behind firewalls or NATs. The LinkSmart middleware eases communication management, providing developers with mechanisms that allow for using Web Services instead of having to deal with the various underlying technologies directly. These features enable communication across heterogeneous application-domain resources, resulting in direct communication among all devices inside a LinkSmart network.

In P2P each node must maintain knowledge about the peers to which it communicates to. While this approach is very resilient against node failures and would work perfectly for a network of PCs, it requires a lot of computing resources and energy to calculate the routes between peers. Thus, this approach does not fits well for a mixed network, where less powerful and battery operated nodes partake. Exchanging information between the peers leads to numerous transmission overhead, which affects the processing capacity and energy consumptions greatly.

In the IMPRESS project, we redesign the LinkSmart communication concept and divide it into two major components. The first major component addresses the local network communication, which is called LinkSmart LocalConnect, and the second major component addresses the communication between different networks or over the internet, which is called LinkSmart GlobalConnect. This separation between the two allows us to focus on the communication problems on different levels and scenarios.

In the LinkSmart LocalConnect, we provide a device and service cataloging, which can be used to register the available devices and their services. The GlobalConnect disseminates this information to the other LocalConnect, so that any application may only use the Catalog in its local network and still having the global overview of the available services. The GlobalConnect is also responsible to tunnel any communications to the remote devices and services through two kinds of network, a centralized network using MQTT or Zero MQ, and a P2P network enabled by JXTA.

3. LinkSmart LocalConnect

LinkSmart LocalConnect is a new stack of components, allowing to set up local smart environments consisting of a number of devices, applications and services, which can be discovered and communicated with using the publish/subscribe or request/response messaging.
The basic LocalConnect components are:

- Service Catalog, which is discoverable and allows other components to find services such as the Resource Catalog or the MQTT broker
- Resource Catalog, holding a collection of resources, grouped by devices
- Device Connector, the component that represents one or multiple devices and exposes its resources

Devices are in general physical devices that are mainly relevant from a deployment perspective.
Each device can have one or multiple resources.
A resource can be a sensor or an actuator.

3.1 Resource Catalog

Resource catalog is the registry of devices and resources they expose.
It manages the registry in its storage back-end and exposes a RESTful Web API, which can be used as an entry point for any application that want to discover devices and their resources, and find out how to talk to them.

3.1.1 Representation and data formats

The representation formats used by the Catalog API are based on JSON-LD, which allows for a clean and readable API and at the same time makes it possible to use Semantic Web (or rather Linked Data) if needed.
Loosing the restrictions for different protocols and formats specifications, we define higher-level objects (Collection, Registration, Resource) more strictly, than the lower-level (Protocol, Representations).
Collection is the highest-level resource of the Catalog API, describing a collection of resources.

Collection
```
{
  "@context": <string>
  "devices": {}
  "resources":[]
}
```

where:

- **context:** here and afterwards, the JSON-LD context
- **devices:** dictionary of **Registration** objects
- **resources:** array of **Resource** objects

Note: support paging of **resources** array is planned

Registration is an entry in the resource catalog, exposed as a "resource" with CRUD operations through the Catalog API.

Every registration is a JSON-LD document:

Registration
```
{
  "id": <string>,
  "type": <string>,
  "name": <string>,
  "description": <string>,
  "meta": {},
  "resources": [],
  "ttl": <number>,
  "created": <timestamp>,
  "updated": <timestamp>,
  "expires": <timestamp>
}
```

where:

- **id:** the URL (mapped to @id in JSON-LD context) uniquely identifying the registration in the network (or node in the rdf graph)
- **type:** type of the registration (mapped to @type in JSON-LD context) – useful in the semantic web context (rdfs:Class)
- **name**: a simple name of the registration
- **description:** a human-readable description of it
- **meta**: an object describing any meta-data related to the device (e.g., device vendor, location)
- **resources**: an array of **Resources** managed by this registration (e.g., sensors/actuators of a device)
- **ttl:** the TTL of the registration in seconds, submitted by the client (registration is expired if not updated within this interval). TTL is -1 for local registrations (they never expire).
- **created:** iso8601 timestamp showing when the registration was put in the catalog
- **updated:** iso8601 timestamp showing when it was updated last time
- **expires:** iso8601 timestamp showing when it will expire

Resource is a resource exposed by the registration in the catalog: for devices connected to the gateway this is a sensor or actuator.
Eventually, resource can be anything that has a representation and a protocol for accessing it (e.g., a simple switch, a virtual sensor)

Resource
```
{
  "id": <string>,
  "type": <string>,
  "name": <string>,
  "meta": {},
  "protocols": [],
  "representation": {}
  "device": <string>
}
```
where:

- **id:** the URL (mapped to @id in JSON-LD context) uniquely identifying the resource in the network (or node in the rdf graph)
- **type**: type of resource (mapped to @type in JSON-LD context) – useful in the semantic web context (rdfs:Class)
- **name:** simple name of the resource
- **meta:** an object describing any meta-data associated with the resource (e.g., units for sensor)
- **protocols:** an array of **Protocol** objects – protocols that can be used to access the resource

- **representation:** a dictionary of **Representations** – content-types the resource can be serialized into
- **device:** a link to the **Registration** of the device exposing this resource

Protocol describes the protocols for accessing and manipulating the resource state (protocol semantics).

Protocol
```
 {
  "type": <string>,
  "content-types": [],
  "endpoint": {},
  "methods": []
}
```

where

- **type:** type of the protocol (NOTE: this is not mapped to JSON-LD @type, for now). ATM we use two types of protocols: "REST" and "MQT""
- **content-types:** an array of Content-Type strings – keys in the **Representation** dictionary – indicates which representations this protocol support
- **endpoint:** an object describing the protocol endpoint. Defined for each protocol individually – everything that is a valid JSON is valid
  For HTTP, we use url: <string>, for MQTT – method: <broker-uri:topic>
- **methods:** an array of protocol verbs (E.g., "GET,POST,PUT,DELETE" for REST, "PUB,SUB" for MQTT)

Representation is a dictionary where keys are Content-Type strings (same as used in the Resource.method array)

Representation
```
{
<key>: {},
...
}
```

where <key> can be any Content-Type (though this is a convention only as of now – in fact, one can put there any <string>), and value is an object.
The object can be any valid JSON, e.g., a document describing JSON-Schema, a link to XML Schema, simple key:value or anything.

3.1.2 Catalog API

We use JSON-LD and Hydra vocabulary for it to eventually implement hypermedia-driven REST API, though focusing on making it work first (and not expecting every client understand Hydra later).

The API implements CRUD for **Registrations**, and additionally a read-only API for (LinkedData-enabled) **Resources**

- **/dc** - returns the Collection of all resources (catalog root) + filtering (see below)
  - supports: GET
- **/dc/:registration** - returns a specific registration (Registration.id)
  - supports: POST (create), GET (retrieve), PUT (update), DELETE (delete)

- **/dc/:registration/:resource** - returns a specific resource of a specific registration (Resource.id)
    - supports: GET

Filtering

Catalog API supports a simple filtering inspired by JSON schema:
/dc/:type/:path/:op/:value

where:

- **type** is one of [device,devices,resource,resources]:
    - **device, resource** returns a single, first matched registration/resource entry (no order guarantees), in the same format as **/dc/:registration** and **/dc/:registration/:resource**
    - **devices, resources** returns a (filtered) collection, in the same format as **/collection**
- **path** is a dot-separated path in json document similar to json-path
- **op** is one of [equals, prefix, suffix, contains] strings comparison operations
- **value** is the intended value/prefix/suffix/substring of the key identified by **path**

3.2 ServiceCatalog

The entry point of the Service Catalog API returns a collection of Services in the following format:

```
{
  "id": "<string>",
  "type": "ServiceCatalog",
  "@context": "<string>",
  "services": []
}
```

The id field is used throughout the API and describes the location (relative URL) of the returned resource, fulfilling the indentifiability REST interface constraint. For entry point, it equals to the path in the API endpoint URL, which is configurable and defaults to /sc.
The fields type, and @context are used to enable LinkedData support and can be ignored by clients using the plain JSON API described in this document.
The services array holds an array of Services

3.2.1 Service

Each service is represented in the following format:

```
{
    "id": <string>
    "type": "Service",
    "name": <string>,
    "description": <string>,
    "meta": {},
    "protocols": [
        {
            "type": <string>,
            "endpoint": {},
            "methods": [],
            "content-types": [ ]
        }
    ],
    "representation": { },
    "ttl": <int>,
    "created": <timestamp>,
    "updated": <timestamp>,
    "expires": <timestamp>
}
```
The id field is a relative URL providing a dereferenceable identifier of the service in the catalog. It is constructed as /path/ + service-id, where:

- **path** is the path in the catalog API endpoint URL
- **service-id** is a unique service identifier in the network and the convention is to construct it as hostname/servicename

The rest of the fields have following meanings:

- **name** is a short string describing a service (e.g., "MqttBroker")
- **description** is a human-readable description of a service (e.g., "Demo MQTT Broker")
- **meta** is any hashmap describing meta-information of a service
- **protocols** is an array of protocols supported by a service
- **type** is a short string describing the protocol (e.g., "MQTT", "REST")
- **endpoint** is an object describing the protocol endpoint (e.g., URL for a Web API)
- **methods** is an array of protocol verbs (e.g., "GET,POST,PUT,DELETE" for REST, "PUB,SUB" for MQTT)
- **content-types** is an array of strings representing MIME-types defined inrepresentation
- **representation** is a dictionary describing the MIME-types supported by a service
- **ttl** is an integer defining the Time-To-Live of a service registration
- **created**, **updated**, and **expires** are generated by the Device Catalog and describe TTL-related timestamps
- **page**, **per_page** and **total** are used for pagination

3.2.2 Protocols

Services can be exposed through different protocols, and below are conventions for describing some of them (format of entries in the protocols array).

MQTT

- type: MQTT
- endpoint: {"url": "scheme://address:port", "topic": "/topic"}
  - url describes the broker connection information as a URL (RFC 3986) using the following URI scheme
  - topic is an optional field describing the topic used by the service. If the described service itself is an MQTT broker, can be omitted
  - additional fields can be defined
- methods: ["PUB", "SUB"] - array of supported MQTT messaging directions. If the described service itself is an MQTT broker, this indicates whether it supports subscription, publishing, or both.
  - PUB - indicates that the service publishes data via MQTT
  - SUB - indicates that the service subscribes to data via MQTT
- content-types: ["application/json", ...] - array of of supported MIME-types (RFC 2046). Empty means payload agnostic

REST

- type: REST
- endpoint: { "url": "scheme://address:port"}
  - url describes the endpoint URL (RFC 3986)
  - additional fields can be defined
- methods: ["GET", "PUT", "POST", ...] - array of supported HTTP methods (RFC 2616)
- content-types: ["application/json", ...] - array of supported MIME-types (RFC 2046)

3.2.3 Example

A registration describing an MQTT Broker may look as follows:

```
    {
        "id": "/sc/server.example.com/MqttBroker",
        "type": "Service",
        "name": "MqttBroker",
        "description": "Demo MQTT Broker",
        "meta": {
            "apiVersion": "3.1",
            "serviceType": "_mqtt._tcp.server.example.com"
        },
        "protocols": [
            {
                "type": "MQTT",
                "endpoint": {
                    "url": "tcp://server.example.com:1883"
                },
                "methods": [
                    "PUB",
                    "SUB"
                ],
                "content-types": [ ]
            }
        ],
        "representation": { },
        "ttl": 120,
        "created": "2014-08-19T08:24:29.283372605+02:00",
        "updated": "2014-08-19T09:21:19.469528757+02:00",
        "expires": "2014-08-19T09:23:19.469528757+02:00"
    }
```

Note that representation and content-types fields are empty, because MQTT is a wire-level protocol.

3.2.4 REST API

The REST(ish) API of the Service Catalog includes CRUD to create/retrieve/update/delete service registrations and a simple filtering API to search through the catalog.
As described above, the path is a configuration parameter setting the path in the catalog API endpoint URL.

CRUD

- /path returns all registered services as ServiceCatalog
  - methods: GET
- /path/ endpoint for creating new entries in the catalog
  - methods: POST
- /path/<service-id> returns a specific service registration given its unique id as aService
  - methods: POST (create), GET (retrieve), PUT (update), DELETE (delete)
  - <id> id of the registration

Filtering

- /path/<type>/<path>/<op>/<value>
  - methods: GET
  - <type> is either **service** (returns a random matching entry) or **services** (returns all matching entries)
  - <path> is a dot-separated path in the Service similar to json-path
  - <op> is one of (**equals**, **prefix**, **suffix**, **contains**) string comparison operations
  - <value> is the intended value/prefix/suffix/substring of the key identified by<path>

Example

curl http://catalog.example.com/sc/services/meta.serviceType/prefix/_mqtt
will return all services in the catalog which meta.serviceType starts with **_mqtt** (as MQTT broker defined in the example above)

Pagination

The Services returned in services array in ServiceCatalog are paged using
the page andper_page parameters.
The results are then include the following additional entries:

- page is the current page (if not specified - the first page is returned)
- per_page is the number of entries per page (if not specified - the maximum allowed is returned)
- total is the total number of Services in the catalog

Example

```
curl http://catalog.example.com/sc?page=1&per_page=10  curl
http://catalog.example.com/sc/services/meta.serviceType/prefix/_mqtt?page=
1&per_page=10
```

Versioning

API version is included as a parameter to the MIME type of request/response:

```
   application/ld+json;version=0.1
```

3.3 Device Connector

A Device Connector provides integration of heterogenous devices in the LinkSmart middleware, implementing the functionality of the Device Integration Layer.
Due to the diversity of available IoT devices and possible integration scenarios, it is rather a concept than a single component. It is expected to have different implementations of the Device Connector fulfilling the described here functional specification.

A Device Connector must implement the following functionality:

- Manage devices and their resources in the Resource Catalog:
  - *Publish* registrations to (the) remote Resource Catalog(s)
  - Continuously *update* these registrations (keepalive)
  - *Remove* the registrations on devices failures and graceful shutdwon of the Device Connector
  - Optionally, expose local read-only Resource Catalog API with managed resources
- Provide communication with devices over the network via standardized protocols
  - Expose APIs of devices/resources via standardized APIs and protocols (HTTP/REST, MQTT, etc)
  - Implement native APIs/protocols of devices internally (if needed)

3.3.1 The Device Gateway (DGW)

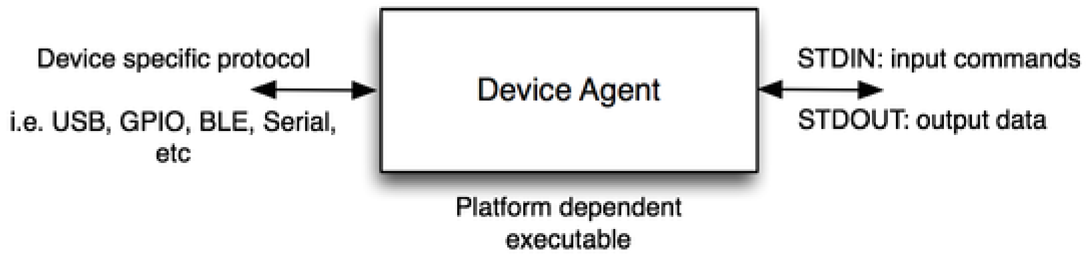
The Device Gateway (DGW) provides an implementation of the Device Connector offering a simple integration of various IoT devices in LinkSmart and rapid prototyping. Implementing the Device Connector functionality, it and acts as a gateway between the low-level hardware access protocol and a TCP/IP network. Furthermore, it also includes a local Resource Catalog, which can be used to discover the devices registered on the DGW.
The main goals behind the DGW design are:

- Pluggable devices support
- Natively compiled for major platforms and architectures, including arm linux

- No modification/re-compilation/re-deployment of the DGW for a new device
- Exposure of device capabilities as network services by declaration/configuration

3.3.2 Device Agent

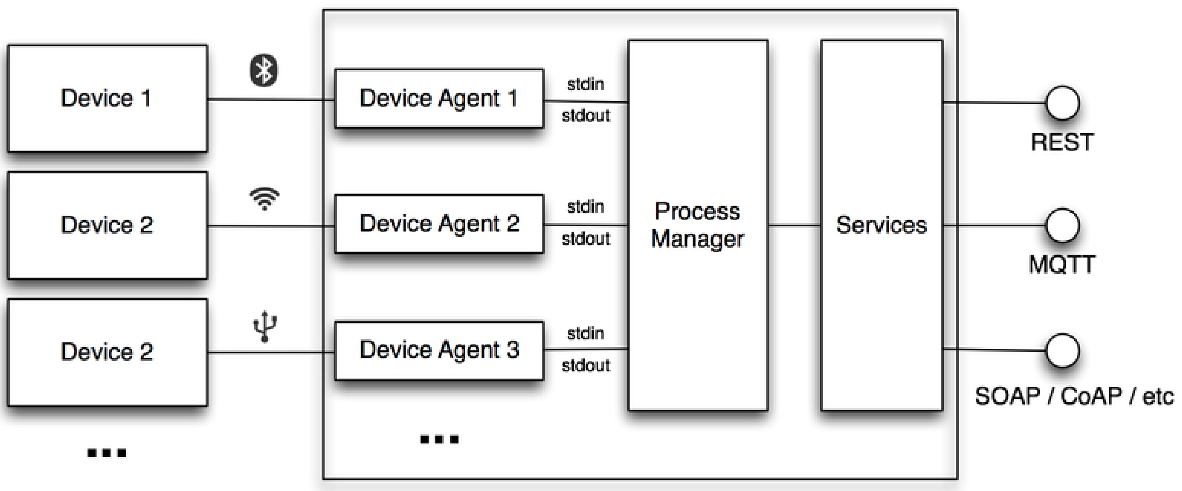
One of the core concepts of the DGW is the Device Agent - an executable (preferably platform independent, but in most cases they are dependent due to the different libraries for accessing the hardware), which implements the low-level communication with the actual device using its interface and protocol (e.g., talking via GPIO to temperature sensor, reading data via USB or detecting beacons via BLE). This is the left side of the communication in the Figure below.



The right side of it describes how the Device Agent communicates with the DGW managing its execution: it is done using the most universal and ubiquitous interface - standard system input and output streams (stdin, stdout correspondingly). The standard system error stream (stderr) is used for logging purposes (data written into stderr will be forwarded to DGW for output in the debug console).

3.3.3 Process Manager

The Device Agent executable is not aware of the DGW and its interfaces. It is a standalone program that can be executed manually from a command line. In the DGW context, this program is executed and managed by the DGW's Process Manager, as depicted in the Figure below (a high-level overview of the DGW architecture):



The core of the system is the Process Manager that reads devices/resources configuration and executes corresponding device agents and redirects the system streams (stdin/stdout/stderr).Process Manager supports 3 types of agent execution: task, timer and service.

- task execution means the Device Agent is executed only once per request coming fromServices component. The last value is cached for a given TTL period.

- timer execution is similar to the task execution, but initiated by the Process Managerperiodically (using interval value from the configuration). The value is cached in between the executions.
- service execution means the Device Agent is running as a process and producing output intostdout constantly. The last value is cached as well.

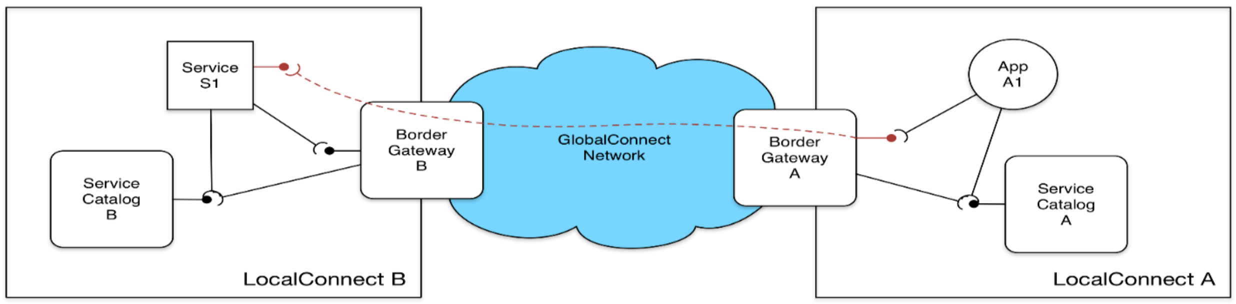
3.3.4 Services

The Services component is responsible for establishing communication with the devices connected to the DGW by applications and services over the network via standardized protocols. In current implementation, REST and MQTT protocols are supported.
The Services component creates a corresponding RESTful endpoint for each devices/resources configured with REST protocol and establishes a MQTT publication connection for devices/resources configured to use MQTT protocol. This component passes data from POST/PUT/DELETE HTTP requests to the corresponding Device Agents by writing it to their stdin and returning the (cached) value received from stdout to the HTTP GET requests. For resources configured to use MQTT, all messages written to stdout by the corresponding Device Agents are published to the configured MQTT broker.

4. LinkSmart GlobalConnect

The basic functionality of the GlobalConnect is to provide a Tunneling Service than enables transparent communication of applications and services beyond the boundaries of a private (routable) network. It helps to connects remote LocalConnect environments over the Internet.
A Border Gateway providing Tunneling Service can be used to expose a local network service (Tunneled Service) to a Global Overlay Network. A Tunneled Service gets tunneled Endpoint on every Border Gateway connected to the GlobalConnect Network. Applications from another private network communicate with a Tunneled Service through the local Border Gateway of that network using the corresponding Tunneled Endpoint.

4.1 Scenario



Service S1 in LocalConnect environment B uses the Tunneling Service (REST endpoint discovered in the local Service Catalog) provided by the Border Gateway B (BG B) to be tunneled beyond its LocalConnect environment.
Upon receiving the tunneling request, BG B does the following:

- Registers S1 in the GlobalConnect network and generates a unique identifier for it (Virtual Address)
- Starts advertising S1 in the GlobalConnect network

Border Gateway A (BG A) in LocalConnect environment A receives the S1 advertisement in the Global Connect network and does the following:

- Creates a local Tunneled Endpoint for S1
- Publishes the information about S1 with the Tunneled Endpoint in local Service Catalog A

Application A1 running in LocalConnect environment A discovers the Tunneled Endpoint of S1 in the local Service Catalog A and starts communicating with S1 through the BG A.

The communication proceeds as follows:

- A1 sends a request to the BG A (Tunneled Endpoint) over the private (tcp/ip) network
- BG A forwards the application request to the BG B over the GlobalConnect network
- BG B receives the request from BG A and forwards it to S1 over the private (tcp/ip) network
- S1 processes the application request and sends a response back to BG B over the private (tcp/ip) network
- BG B forwards the S1 response to BG A over the GlobalConnect network
- BG A receives the response from BG B and forwards it to A1 over the private (tcp/ip) network

4.2 Definitions

- LocalConnect environment: a deployment of LinkSmart LocalConnect running in a private network (at least Service Catalog)
- GlobalConnect network: an overlay network connecting multiple LocalConnect environments

- Tunneled Service: a network service running in LocalConnect environment (registered in local Service Catalog) tunneled in the GlobalConnect network
- Tunneling Service: a LinkSmart GlobalConnect service running in LocalConnect environment that can be used to expose local services in other LocalConnect environments (turning them into Tunneled Services) using the GlobalConnect network
- Tunneled Endpoint: a local (tcp/ip) endpoint of a remote Tunnelled Service in a LocalConnect environment
- Border Gateway: a host providing the following functionality:
  - running the Tunneling Service in LocalConnect environment
  - providing Tunneled Endpoints for Tunneled Services from remote LocalConnect environments

4.3 Deploying LinkSmart GlobalConnect

LinkSmart GC distribution is equipped with Apache Karaf that is an OSGi run-time container, that support different start modes and its console provides a full Unix-like environment.
Open a command line console and change the directory to "LINKSMART_HOME".
To start the server, run the following command in Windows:

```
bin\karaf.bat
```

respectively on Unix:

```
bin/karaf
```

upon successful startup, a welcome message would appear and the LinkSmart components are installed and activated by the container. After it finishes installing the components, one can view the the log from LinkSmart components:

```
karaf@root()> log:display
or
karaf@root()> log:tail
```

to see the (OSGi) bundles status:

```
karaf@root()> bundle:list
```

To stop Server from the console, enter D in the console:

```
^D
```

Alternatively, you can also run the following command:

```
system:shutdown
```

or simply

```
shutdown -f
```

4.3.1 LinkSmart GC Features (Deployment Modes)

LinkSmart GC provides different deployment modes with the help of Karaf's feature concept.
Following are the currently available features:

- **linksmart-gc**
  start all LinkSmart GC components (Tunneling, Zmq & Http backbones, NetworkManager, NetworkManageer_Rest etc)
- **gc-http-tunneling**
  start jetty server, network manager & Tunneling servlet
- **gc-backbone-http**
  start backbone router & Http implementation for Backbone interface
- **gc-backbone-zmq**
  start backone router & ZMQ based implementation of Backbone interface
- **gc-network-manager**
  start netwokr manager and associated components like identity-manager and backbone-router
- **gc-network-manager-rest**
  provide HTTP based (REST) interface for Network Manager for registration, de-registration, and service discovery etc

The feature:install command installs a feature as follows:

```
karaf@root()> feature:install linksmart-gc
```

The feature:uninstall command uninstalls a feature as shown below:

```
karaf@root()> feature:uninstall linksmart-gc
```

4.4 Tunnelling services using LinkSmart GlobalConnect

By default, the NetworkManager REST API is exposed during LinkSmart GC bootstrapping.
In case it's not started, the following command can be executed in Karaf console to install the NetworkManager REST API:

```
karaf@root()> feature:install gc-network-manager-rest
```

The NM REST API support four HTTP methods for the following functionality:

1. Register new service (POST)
2. Query register services based on parametrized search criteria (GET)
3. Update existing service registration (PUT)
4. Remove service registration (DELETE)

4.4.1 New Service Registration

For registering a service into the Network Manager, a POST request is needed with a JSON payload. More information about the payload please see NetworkManager REST API. Below an example weather service is registered with Network Manager by providing its description and service ID.

```
curl -X POST -H "Content-Type:application/json; charset=UTF-8"
http://localhost:8082/NetworkManager -d
'{"Endpoint":"http://localhost:8082/WeatherService",
"BackboneName":eu.linksmart.gc.network.backbone.protocol.http.HttpImpl",
"Attributes":{ "DESCRIPTION":"WeatherService",
"SID":"eu.linksmart.gc.example.weatherservice"} }'
```

The response will be similar to this one below:

```
{
  "Endpoint":"http://localhost:8082/HttpTunneling/0/1.1.1.1",
  "VirtualAddress": "1.1.1.1",
  "Attributes": {
    "DESCRIPTION": "WeatherService",
    "SID": "eu.linksmart.gc.example.weatherservice"
  }
}
```

4.4.2 Querying for Service Registration

To query the Network Manager for service registration by its description, GET method is used with query string attributes, as shown below:

```
curl -H "accept:application/json"
'http://localhost:8082/NetworkManager?description="WeatherService"'
```

The response would be a JSON similar to the one below. NOTE: 1.1.1.1 is similar to some real virtual address "0.0.0.6765991535227307528".

```
[
  {
    "Endpoint":"http://localhost:8082/HttpTunneling/0/1.1.1.1",
    "VirtualAddress":"1.1.1.1",
    "Attributes": {
      "DESCRIPTION": "WeatherService",
      "SID": "eu.linksmart.gc.example.weatherservice"
    }
  }
]
```

4.4.3 Accessing the Service

The EndPoint received in a response is actually a virtual endpoint that is used to access the service through GC Tunneling. It is assumed that service is deployed somewhere and accessible by a

NetworkManager that was used to register service with. Consumer can now invoke the supported HTTP methods of a service. An example for GET method is given below:

```
curl http://localhost:8082/HttpTunneling/0/1.1.1.1/service-
path?param=value
```

4.4.4 Updating Service Registration

HTTP POST request is used to update an existing service registration. The JSON payload required for this POST is described in NetworkManager REST API. Below weather service is updated for new description.

```
curl -X PUT -H "Content-Type:application/json; charset=UTF-8"
http://localhost:8082/NetworkManager -d
'{"Endpoint":"http://localhost:8082/WeatherService",
"BackboneName":eu.linksmart.gc.network.backbone.protocol.http.HttpImpl",
"VirtualAddress":"1.1.1.1", "Attributes":{
"DESCRIPTION":"WeatherService2"} }'
```

The response will be similar to this one below:

```
{
  "Endpoint":"http://localhost:8082/HttpTunneling/0/1.1.1.2",
  "VirtualAddress": "1.1.1.2",
  "Attributes": {
    "DESCRIPTION": "WeatherService2"
  }
}
```

4.4.5 Removing Service Registration

Finally to remove a registered service a DELETE request need to be done with the assigned VirtualAddress as is shown below:

```
curl -X DELETE 'http://localhost:8082/NetworkManager/1.1.1.2'
```

5. Conclusion

The IMPReSS communication management enables application to find MQTT and REST endpoints that must be offered by the Resource Adaptation Interface (RAI). This allows applications to be developed independently from the IoT resources in the environment. MQTT and REST could be used by services to communicate within a local routable network. However, as the network grows involving the internet connectivity, services may not be able to communicate directly, because of the firewalls in between. IMPReSS GlobalConnect was created to help developers making their services accessible through an overlay network which offer transparency as well as security.

To use LocalConnect and GlobalConnect, developers must register their services to the LocalConnect by invoking a REST API. This knowledge about local services is exchanged between the LocalConnect and can be accessible through the tunneling service provided by the GlobalConnect. The LocalConnect expect a periodic registration from the services to be sure that the services are still alive.

The application developers then could query the catalog to find the endpoints of the services offered by the RAI. The remote devices will appear with local addresses since any request to them from the applications are going to be routed by the GlobalConnect.