Open Gateway API

Integrator documentation

Ing. Martin Horniak

bAvenir s.r.o.

martin.horniak@bavenir.eu

Version 0.5

March 2018

Intentionally left blank

[table of content goes here]

Version history

This change log will be recorded from version 1.0.

Introduction

This documentation is intended to explain how to install, configure and utilize the Open Gateway API (OGWAPI) in order to access remote IoT objects. It also describes what steps that need to be done in order to integrate a new IoT object into the system, so it can be accessed by other objects on the network.

First chapter discuss a general overview of Open Gateway API functionality and its place among other components of the system.

Second chapter goes through installation and configuration procedures. It also lists and describes configuration parameters of the software.

Third chapter serves as a tutorial on how to utilize some of the functionality the OGWAPI provides, namely the set of endpoints that are used by the local objects in order to exchange data with remote objects. It then describes all the endpoints of the OGWAPI.

Fourth chapter discuss integration of the objects into the network. In other words, it takes reader through the requirements that the implemented adapter has to meet in order to allow other objects on the network to access it.

The last chapter goes through frequently asked questions.

1 Open Gateway API overview

Open Gateway API, or further in this documentation OGWAPI, is a data gateway interconnecting two or more IoT ecosystems, that are behind a NAT or not included in public IP addressing scheme. OGWAPI in those infrastructures acts alike a chat client, that is able to exchange messages with other OGWAPIs in a P2P network. Embedded in these chat messages are HTTP/HTTPS requests, that are to be transferred across the network. At the other side, they are decoded by the other OGWAPI and turned back to HTTP/HTTPS requests, distributed in the remote private IoT ecosystem.

The OGWAPI provides multiple HTTPS interfaces for your local infrastructure, that should cover all (or at least most) of the functionalities a common IoT ecosystem provides:

- retrieving and setting of a property (e.g. brightness),
- starting or stopping of an action (e.g. raising curtains),
- transmitting an event (e.g. door were opened).

However, the OGWAPI is just one part in the whole system, that consists of two more layers and one web portal, that makes this exchange as much automated and secure as possible.

The layer that is closest to the OGWAPI is what we call an Agent. Agent acts like a router, distributing the incoming requests among the local nodes. Moreover it makes certain steps easier, like automatic registration of several devices at once, as well as a certain degree of auto configuration. Although the use of an Agent is not mandatory, it is highly recommended. When using an Agent in your local infrastructure, you will not need the whole set of OGWAPI endpoints, just a reduced set, mainly for data consumption (consult Agent documentation about which endpoints are necessary to be used).

The last layer is an Adapter. The Adapter is a translator that transforms common VICINITY language to a specific infrastructure, akin to a driver and you, as an integrator, have to use or code one.

One more part that closes the circle in the VICINITY system is the Neighbourhood manager Web UI, that lets you choose which devices are to be visible for which other devices and create partnerships akin to social network friendships.

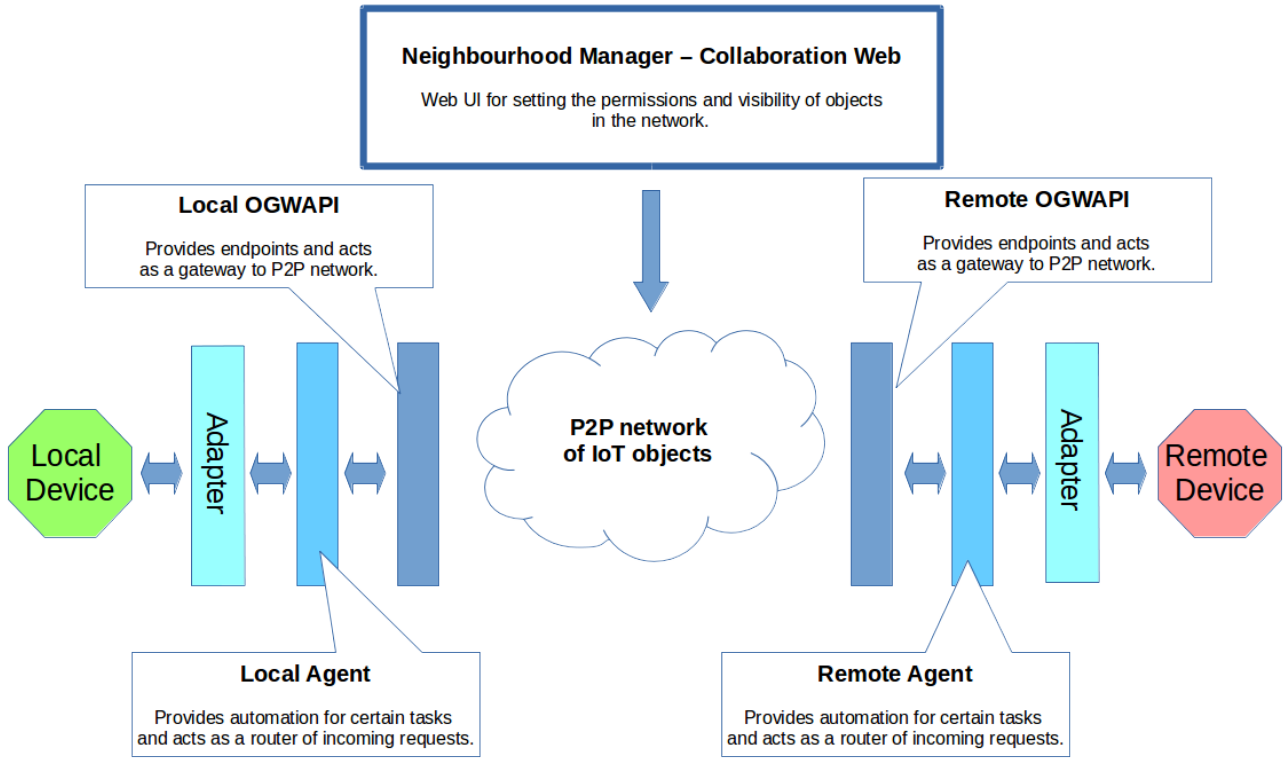

Illustration 1: Interaction diagram of the VICINITY system.

2 Open Gateway API installation and configuration

OGWAPI is a standalone Java application, composed of a single JAR file, that binds to a specific port on the machine where it runs. If you want to run the OGWAPI, following requirements must be met:

- In order to download the OGWAPI, you need to have git installed.
- In order to run the OGWAPI you need to have Java JRE 8 installed. It is programmed using plain Java OpenJDK, so you don't need to install Java from Oracle. Of course, if you already have Oracle's version of Java installed, it will run as well.
- OS requirements are not specified. It should work on Linux, Unix, MacOS and Windows.
- Overall HW requirements are dependent on how many devices are connected through the OGWAPI. You can count 30MiB for the OGWAPI itself (depending on configuration) and roughly 200KiB for every device that is connected through it. As a rule of thumb, it was demonstrated that OGWAPI can run on all versions of Raspberry Pi.
- If you'd like to run OGWAPI on one of the privileged ports (1 – 1024), you need to execute it as root (or other administrator equivalent of your particular user).
- The git repository comes with a pre-build executable JAR file, however you can always build your own from the provided source using Maven (see section Building from source codes).

2.1 Installation

Depending on the OS you are using, there are multiple approaches that can be taken while installing the OGWAPI software on your machines. For all OS types there is the possibility to build the OGWAPI from source codes. As the software is written in plain Java that is runnable on OpenJDK JVM, no big issues are expected. For users of Linux OS, there is also the ability to install the OGWAPI using two most popular installation managers - .deb and .rpm.

2.1.1 Linux / Unix – installation from the Git repository

This is a step-by-step tutorial on how to download and install the OGWAPI from downloaded source codes on Debian-based Linux systems. It should be valid for other derivatives, with a very small deviations at most.

The installation consists of 6 steps, out of which 5 are mandatory:

1. Cloning the repository.

2. Building the OGWAPI from the source codes [optional].
3. Creating a system user for the OGWAPI.
4. Creating directories where the OGWAPI will run and where the logs will be stored.
5. Copying the binary file and the configuration directory into the destination directory.
6. Changing the owner and making the binary executable.

2.1.1.1 Cloning the repository

Start with cloning the git repository into a directory of your choice. Usually, this is not the same directory, as the one where you want to install your running instance of the OGWAPI (although it can be, of course):

```
$ cd /path/to/the/directory
$ git clone https://cpsgit.informatik.uni-kl.de/VICINITY/vicinity-open-gateway-api.git
```

You should now see the following directory structure. If you just need to install the OGWAPI (and not building it yourself), the important directories are in bold:

```
./
../
.classpath
config/       ←- Sample OGWAPI configuration file. Also, when you run it
       from an IDE, the file inside is the actual valid config.
Doc/          ←- Javadoc and this file.
.git/         ←- Git configuration.
log/          ←- When you run it from an IDE, logs are by default stored
       here. You can ignore it otherwise.
pom.xml       ←- Maven configuration.
.project      ←- Eclipse IDE project directory.
README.md     ←- Roughly the same information as here + changelog.
.settings/
src/          ←- Source files.
target/       ←- This is where you find pre-built JAR executable (or your
   own build).
Tools/    ←- Some tools we were using during development and testing.
```

Now, if you want to build the OGWAPI yourself, it is good time to jump to section Building from source codes. Otherwise, skip that section and continue with section about creating the dedicated system user.

2.1.1.2 Building the OGWAPI from source codes

The Git repository you have downloaded in the previous step already contains the latest build of the OGWAPI. However, you can always decide to play around and build your own binary.

In order to do this, you will need Maven, so necessary binaries are downloaded along the way. Moreover, building just for the sake of being able to build a software may be fun, but is not very useful. If you are reading this section, you probably also want to make some changes in the source code itself.

The entire OGWAPI was programmed using Eclipse IDE, and the Git repository you cloned contains Eclipse project, that can be easily imported. Javadoc is also in the directory tree, see the section where you cloned the repository to your machine.

Once you made and tested your changes from the Eclipse, you can make a runnable binary by using Maven like this:

```
$ cd /path/to/your/repository
$ mvn clean compile assembly:single
```

The result of your new build is in the ‘target’ directory in the repository directory tree – the only JAR file that should be there under normal circumstances.

2.1.1.3 Creating a dedicated system user

First of all, let’s make the installation a bit more secure. It is always a good idea to run various services with their own system user. This is especially true, when you just plan to start the service on your Raspberry Pi and leave it on – which is the normal course of action. There are two exceptions from this rule though:

- you have a GOOD reason to run the OGWAPI on port <1024, in which case you have to run it as root,
- you are planning to play around a little, in which case you will probably want to use your own system user to run the OGWAPI, for more convenience.

If none applies to you, you can create a dedicated system user named ‘ogwapi’ like this (for this you need to be root):

```
# adduser --system --group --no-create-home --shell /bin/sh ogwapi
```

2.1.1.4 Putting it all in the right place

Now it is necessary to put all files in the directory, where it is going to be run from. Lets say, you want to run the OGWAPI from /opt/ogwapi. You have to create the directory, then copy the necessary files into it and change permissions. Start with creating it:

```
# mkdir /opt/ogwapi
```

Now copy necessary files – when you are installing the OGWAPI from the git repositroy, only two files are needed to be copied – the binary JAR file and the configuration directory.

```
# cp -r /path/to/the/repository/config /opt/ogwapi/
```

The OGWAPI JAR file may have a cumbersome name, however it gets distributed like that in order to know which version you are using. You can rename it now to your liking.

```
# cp /path/to/the/repository/target/GatewayApi-0.5-jar-with-dependencies.jar
/opt/ogwapi/gateway.jar
```

Now make sure, that the running directory has the right owner and the JAR file is executable:

```
# chown -R ogwapi:ogwapi /opt/ogwapi
# chmod u+x /opt/ogwapi/gateway.jar
```

Also, you need to decide, where you want to store the logs that are produced by a running gateway. You can set this parameter later in the configuration file, but the directories have to be created first. In this case, let’s say that we want to keep them together.

The final directory tree should look like this:

```
root@themachine:/opt/ogwapi# ls -l
total 9132
drwxr-xr-x 4 ogwapi ogwapi    4096 feb 27 16:43 ./
drwxr-xr-x 5 root   root      4096 feb 27 13:39 ../
drwxr-xr-x 2 ogwapi ogwapi    4096 feb 27 13:41 config/
-rwxr--r-- 1 ogwapi ogwapi 9331984 feb 27 13:41 gateway.jar*
drwxr-xr-x 2 ogwapi ogwapi    4096 feb 27 13:52 log/
```

First part, the installation, is now done, you can head for the configuration section.

2.1.2 Linux – .deb installation package

To be done.

2.1.3 Linux - .rpm installation package

To be done.

2.1.4 MacOS

Detailed documentation is yet to be tested. However, as the software is written in plain Java, based on the OpenJDK JVM, the building from source code approach should be working, provided you execute equivalent steps as described in the Building the OGWAPI from the source codes.

2.1.5 Windows

Detailed documentation is yet to be tested. However, as the software is written in plain Java, based on the OpenJDK JVM, the building from source code approach should be working, provided you execute equivalent steps as described in the Building the OGWAPI from the source codes.

2.2 Configuration

If you installed the OGWAPI from cloned repository according to the tutorial in previous sections, you can find the configuration file in the installation directory of that particular instance (./config/GatewayConfig.xml). If you installed it via the packaging manager, you have a single instance of the OGWAPI and its configuration file is in /etc/ogwapi/GatewayConfig.xml.

2.2.1 Some notes on how the configuration is read

There are two ways of how the OGWAPI loads its configuration on start. In the first case, you specify the configuration file as the single non-option (as in without any option switch) argument in the command line when starting it. This is how the OGWAPI learns its path to configuration file when it is installed from the package manager (see starting scripts, if you are curious). In the second case, you don’t supply the OGWAPI with any argument and it tries to locate the configuration in the ./config/GatewayConfig.xml file, relative to the particular binary file (this is how we did it when we were installing the OGWAPI from the Git repository). Remember, in both cases the configuration is

loaded on start, which means that you MUST restart the OGWAPI whenever there is a change in configuration (otherwise the configuration change will not make it into the actual running instance).

The configuration file is, as you might have noticed, a single XML file, with one root element <configuration />. The parameters are set as additional nested elements, with no attributes. When a given parameter is not set, or is omitted, all parameters either have a default value that is used, or the OGWAPI will shut down on start, right after the configuration was loaded, if the parameter was so important that omitting it will prevent it to run normally.

The structure of the configuration file tries to divide all the parameters into few bigger sets, in order to be easier to comprehend and manage. As its downside, it is impossible to throw the parameters around freely, they always have to be in their dedicated parent elements. The structure goes like this:

```
<configuration>
    <general>
    … general settings, like what engine to use to connect to the network etc.
    </general>

    <logging>
    … logging settings, where to find the log file and what is worth the
    record.
    </logging>

    <xmpp>
    … xmpp engine settings, what server to use and on what port is listens.
    </xmpp>

    <api>
    … settings for interface between the OGWAPI and Agent/Adapters, to what
    port and IP the  OGWAPI binds on start, how the Agent/Adapters  authenticate
and where to send the request arriving from the network.
    </api>

</configuration>
```

When talking about configuration of a certain parameter, we will use a shortened transcription for the sake of brevity. Lets say you need to change the parameter for log file:

```
<configuration>
    <logging>
        <file>some path to file</file>
    </logging>
</configuration>
```

This is quite long and hardly usable in text. We will therefore shorten it in text to *logging→file*. The *configuration* element is omitted, since all parameters need to be child elements of that particular element.

2.2.2 Basic configuration

The default configuration of most parameters offers functionality straight out of the box without spending too much time tweaking. Nevertheless, two thing should be configured/verified on new installations, especially when the OGWAPI was installed from the Git repository:

1. Location of log file and setting the desired log level.

Change the parameter *logging→file* to desired location, presumably the one you created during installation. If you installed the OGWAPI via a package manager, the log file is set to */var/log/ogwapi/%s-gateway.log*. Note that the %s character in the string is replaced by a time stamp of the moment, when the OGWAPI instance is started.

Also, you might want to adjust *logging→level* to fit your needs. Permitted levels are (in order from most quiet, to most talkative) OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.

2. IP address and port of the Agent.

In order to receive requests from the P2P network, it is necessary to set an IP address and a port of your local Agent, that will process these requests. If you don't have an Agent running, you can state the IP address and port of your Adapter, provided it can correctly process the requests (see the section Integration and adapter development). The parameters to change are *api→agentIP* and *api→agentPort.*

You can always play around with the other parameters. Their meaning and how they affect the system behaviour is (should be) explained in-line in the configuration file. If that is not the case, take a look at the section List of configuration parameters and their meaning.

2.2.3 List of configuration parameters and their meaning

Following is a list of configuration parameters and their meaning. Sample configuration file can be found within cloned Git repository, or in /etc/ogwapi/GatewayConfig.xml if you installed the OGWAPI via a package manager.

- **logging→file**

  Set a relative or absolute (recommended) path to log file. In order to differentiate among multiple log files, a '%s' can be added to arbitrary location in the string, which will be replaced by a time stamp during start. Each start of the Gateway will produce a new log file with a new time stamp.

- **logging→level**

Set a log level - messages with severity level lower than this setting will not be recorded. The list of levels in descending order is following:

SEVERE

WARNING

INFO

CONFIG

FINE

FINER

FINEST

It corresponds with the levels used by class java.util.logging.Logger, which the VICINITY Gateway utilizes for event logging. This value can also be set to OFF, which will disable the logging mechanism completely.

- **logging→consoleOutput**

  Setting this value to 'true' will cause the application to log its output to console, aside from logging it into file. This can be useful when debugging the software. Setting it to 'false' will suppress this behaviour, instead logging events solely to log file. This is the default behaviour.

- **xmpp→server**

  URL/IP address of XMPP server. If not set, the application exits.

- **xmpp→port**

  Defaults to 5222 if not set. Server port uses STARTTLS.

- **xmpp→security**

  Setting this parameter to true will enable encryption of communication. The policy is to try the strongest mechanisms first. Not having a valid certificate on server will always fail the connection.

  Setting it to false will disable the security all together (not recommended, only for debug purposes). Connecting to a server that accepts only TLS will fail the connection. Default is true.

- **xmpp→domain**

  XMPP domain that is served by the server. Defaults to vicinity.eu.

- **xmpp→debugging**

  Enables debugging of the XMPP communication between the Gateway and the server / other Gateways. Note that this is to be used in conjunction with the SMACK debugger, which is external tool.

See http://download.igniterealtime.org/smack/docs/latest/documentation/debugging.html

Default is false.

- **xmpp→messageTimeout**

  Number of seconds to consider request message as no longer relevant. After a request is sent from point A to point B, point A waits for response. If the response does not arrive until this timeout expires, point B is considered unreachable. If the response arrives after this happens, the response is ignored and discarded and a new request has to be sent.

  Default is 30 seconds.

- **api→port**

  Set the port on which the API will be served. If not explicitly set, it defaults to 8181. Be aware that running the software on privileged ports (<1024) needs root's privileges.

- **api→enableHttps**

  Set whether the API will be served via HTTP or HTTPS. Takes either true or false value. Default is true, however in some installations it might not be supported.

- **api→authRealm**

  Authentication realm for the RESTLET BEARER authentication schema. It is only taken into account if the authMethod is set to bearer.

  Defaults to vicinity.eu.

- **api→authMethod**

  Authentication method for objects logging into the Gateway API. Following methods are valid:

  basic - Basic HTTP authentication standard.

  digest - Digest HTTP authentication standard.

  bearer - Token authentication (JWT/OAuth)

  none - No authentication. Experimental, for debugging purposes only. Without authentication, there is no way of logging into the network other than hard coded credentials (or other kind of customization).

  Defaults to basic.

- **api→agentIP**

If the agent listens on a different interface (or a different machine), it is necessary to set its IP address with this parameter. Defaults to localhost.

- **api→agentPort**

  Port on which the adapter listens. Defaults to 9997.

2.3 Running the OGWAPI

Again the way the OGWAPI is run depends heavily on the way you installed in the first place. If you installed it via a package manager, the standard service call is available, as for other services on your machine.

```
# service ogwapi start | stop | restart
```

On the other hand, when you installed the OGWAPI manually from the Git repositories, enter the installation directory and issue the following command:

```
# su - ogwapi -c "nohup java -jar GatewayApi-0.5-jar-with-dependencies.jar &"
```

The su part will make sure the command is run as the ogwapi user (the space character between dash and user name is not a typo!). Then the actual start command follows in the quotes. Nohup is used as a way of making sure the OGWAPI will keep running even after you log out from the terminal.

2.4 Updating an existing installation of OGWAPI

If you installed the OGWAPI via the package manager, regular updates should be delivered via update mechanisms of the particular package manager depending on your settings. Please consult the documentation of your package manager on how to carry out the update if it is not done automatically.

If you installed the OGWAPI manually from the Git repository, it is recommended you make a regular updates by fetching the new version of the repository and manually replace the binary file.

In both cases, please take a note of the README file, as there can be valuable information about new configuration parameters or functionality.

3 Using Open Gateway API

The OGWAPI is enabling your IoT infrastructure to interconnect with other IoT infrastructures. Based on the Neighbourhood manager settings, permissions are always checked, whether or not two IoT objects are capable of mutual communication. Provided these permissions are verified and met, OGWAPI brings your IoT new functions for interoperability:

- retrieving a list of properties, events and actions supported by a remote IoT object
- changing properties or executing an action on a remote object
- fire an asynchronous event, that gets propagated to subscribed remote objects
- subscribing and receiving such event
- querying the P2P network for data.

3.1 Interfaces overview

OGWAPI provides HTTP REST endpoints, that your Adapter can connect to in order to utilize these functions. There are a few dozen endpoints the OGWAPI is providing, so in order to make the use of it a little easier, they have been divided into following groups, that are called interfaces:

- **authentication interface**

  Provides your Adapter with endpoints that can log it into the P2P network, or log it out. It is necessary to note, that you don’t need to explicitly log your adapter into the network. Since the OGWAPI tries to be in line with REST calls philosophy, the credentials are always sent as a part of the request header (see online documentation for various HTTP authentication mechanisms). Therefore, the Adapter/Agent gets logged in immediately when it makes its first call, no matter which endpoint is requested. Of course, there are reasons why it is a good idea to explicitly authenticate, like making the object available for the network without making the first request. It is therefore recommended to log your devices in during start up.

- **consumption interface**

  Endpoints of this group are listing, getting and setting values of a remote object’s properties. Also, they execute long term actions on these objects.

- **exposing interface**

  Provides you with the ability to create a new feed and fire an event that gets distributed to other subscribed objects. Also, through endpoints of this groups, you can subscribe to an event channel.

- **discovery interface**

It is very likely that your IoT infrastructure possesses many objects. In order to avoid setting up each one of your light bulbs manually, you can register automatically several objects at once. This automatic discovery can be also done periodically, if your infrastracture is prone to frequent changes. In order to do this, it is necessary to compare your currently available devices with the ones that are already registered in the system. Discovery interface polls communication servers, which of your objects are already registered, so your automatic registration system can avoid re-registering them. Discovery should always precede a registration.

- **registry interface**

Provides endpoints to automatically register newly discovered objects.

- **query interface**

  Provides a simple interface for intelligent querying the P2P network for specific data.

It is necessary to repeat, that some of the endpoints require to send a payload (in general, all the endpoints that are not utilizing GET or DELETE methods). On the other hand, the rest of endpoints return some kind of a payload. Both payloads are usually in a form of JSON and its precise structure is driven by semantic vocabulary [link a document where it is explained].

3.2 Testing and debugging

As the communication with the OGWAPI is done via HTTP, it is possible to use simple REST client (Postman, Insomnia, etc…) for sending testing requests in order to see how the OGWAPI behaves and what is sent back. For more information about using these tools, please consult the online documentation for the tool of your choice. For more information about the endpoints, please read on.

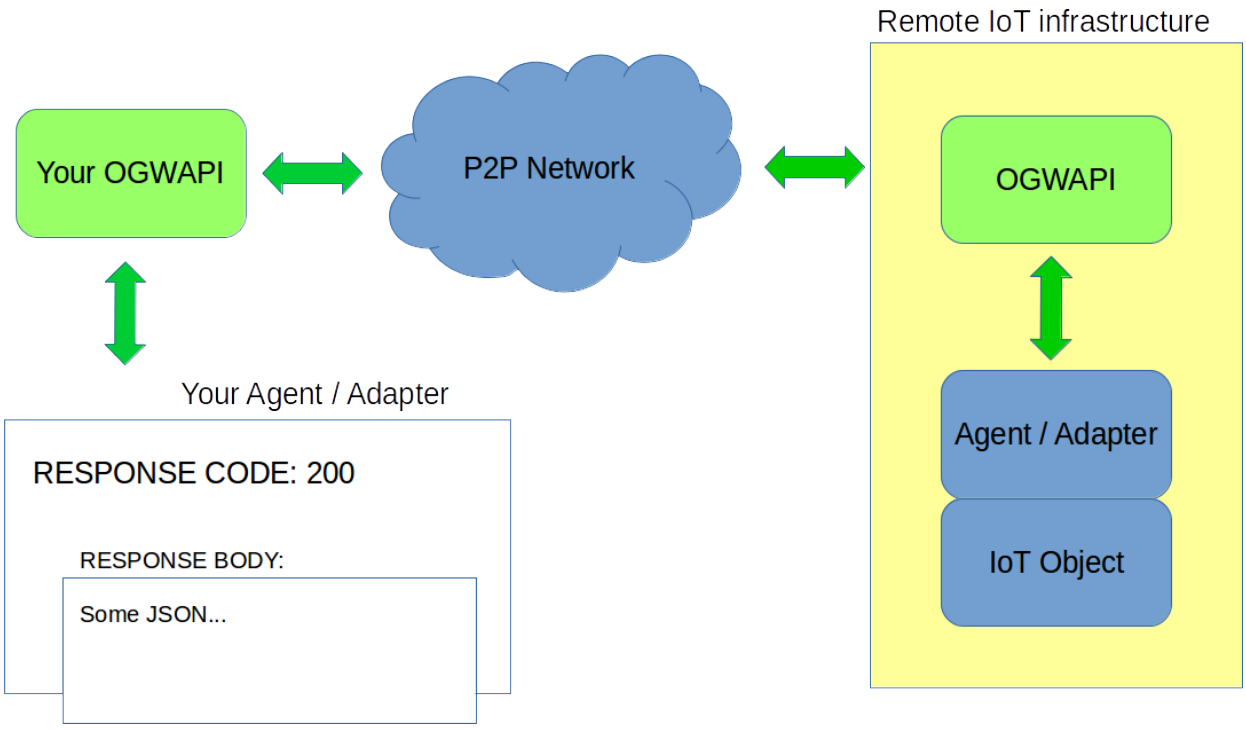

Illustration 2: Normal communication between two IoT infrastructures.

3.2.1 Error propagation

One of the last things before you get your hands dirty with integrating your infrastructure, is understanding the error propagation. From the general overview section we can see, that using the OGWAPI divides the communication path to three logical sections, where a communication error can occur:

- the part between your infrastructure and your Gateway,
- the P2P network between two Gateways,
- the part between remote Gateway and remote infrastructure.

First part will respond as a regular HTTP service, both when communication is working flawlessly and when error condition occurs. The code in the response will provide you with more information about what could have happened. A basic knowledge of HTTP response codes is in place, but in general, code 2XX means everything is OK, 3XX that the OGWAPI could not be reached (check IP, port, and whether it runs), 4XX means you entered wrong credentials and 5XX

that the OGWAPI could not digest what you fed it with (reasons may vary and you have to see the logs for actual reasons).

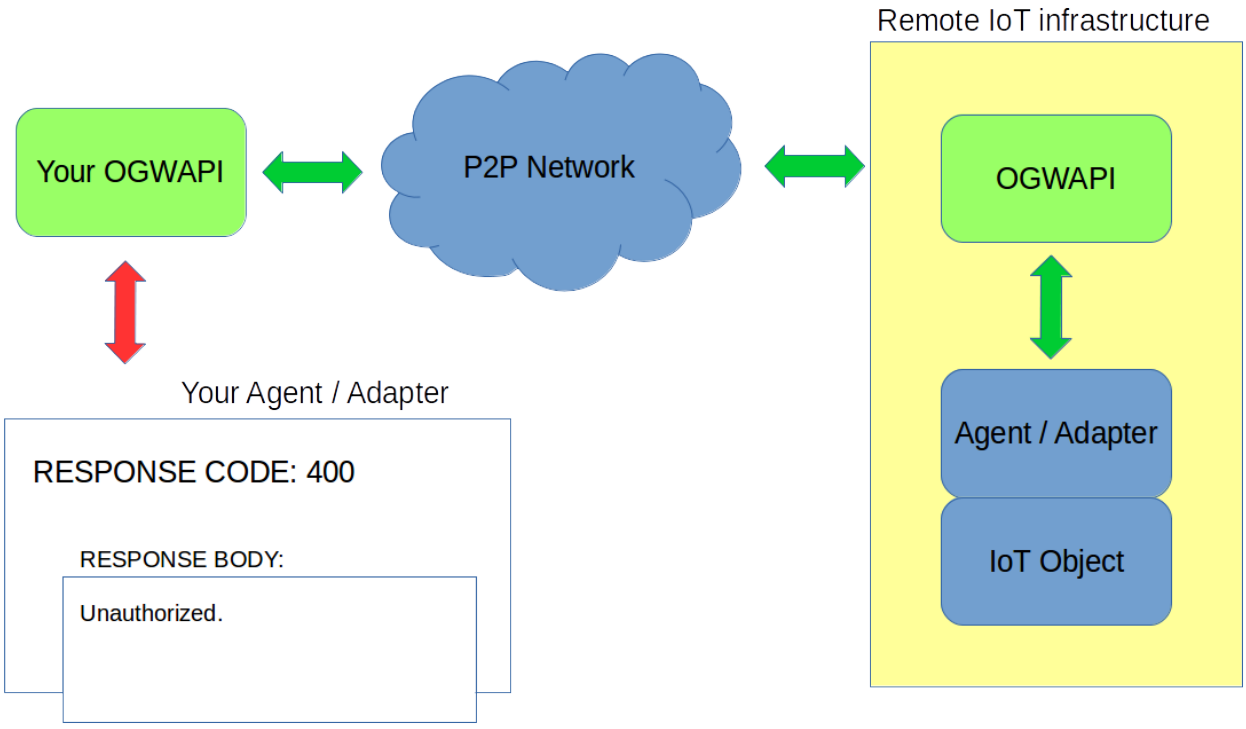

Illustration 3: Example of error originating within your local infrastructure.

When an error occurs in the second part, in the P2P network, it is very likely that the request from your side will get a time out exception. In some cases you also get 5XX. Again, seeing the logs will make the image a little bit clearer. Sometimes these errors are caused by misconfiguration of the OGWAPI, sometimes maybe a restart is in place, however most of the times the error is out of your reach and if the situation will not resolve itself in a few hours, please contact the support.

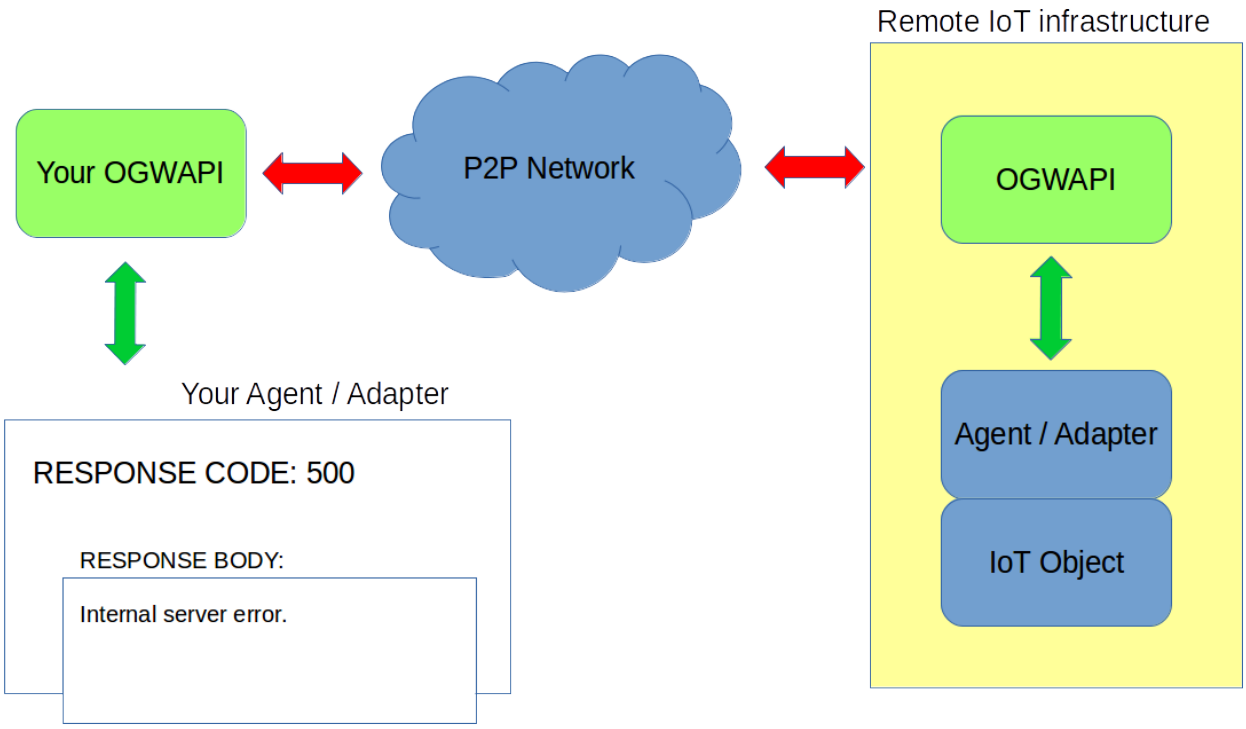

Illustration 4: Example of an error in the P2P network.

The last origin of an error condition can be in the remote IoT infrastructure. In this case, when your request reaches the remote OGWAPI, it also tries to reach the remote Agent/Adapter via HTTP. Error code from this communication will propagate back to you in the **response body. T**he emphasis is in place, because the response code of your request will in this case be 2XX – all good, however in the response body, you may find e.g. 500 Internal server error instead of the JSON you have been expecting. That will tell you that in this case, your request successfully travelled all the way to remote OGWAPI, but unfortunately, the remote OGWAPI could not establish connection with the object you called, because the Adapter misbehaved and crashed.

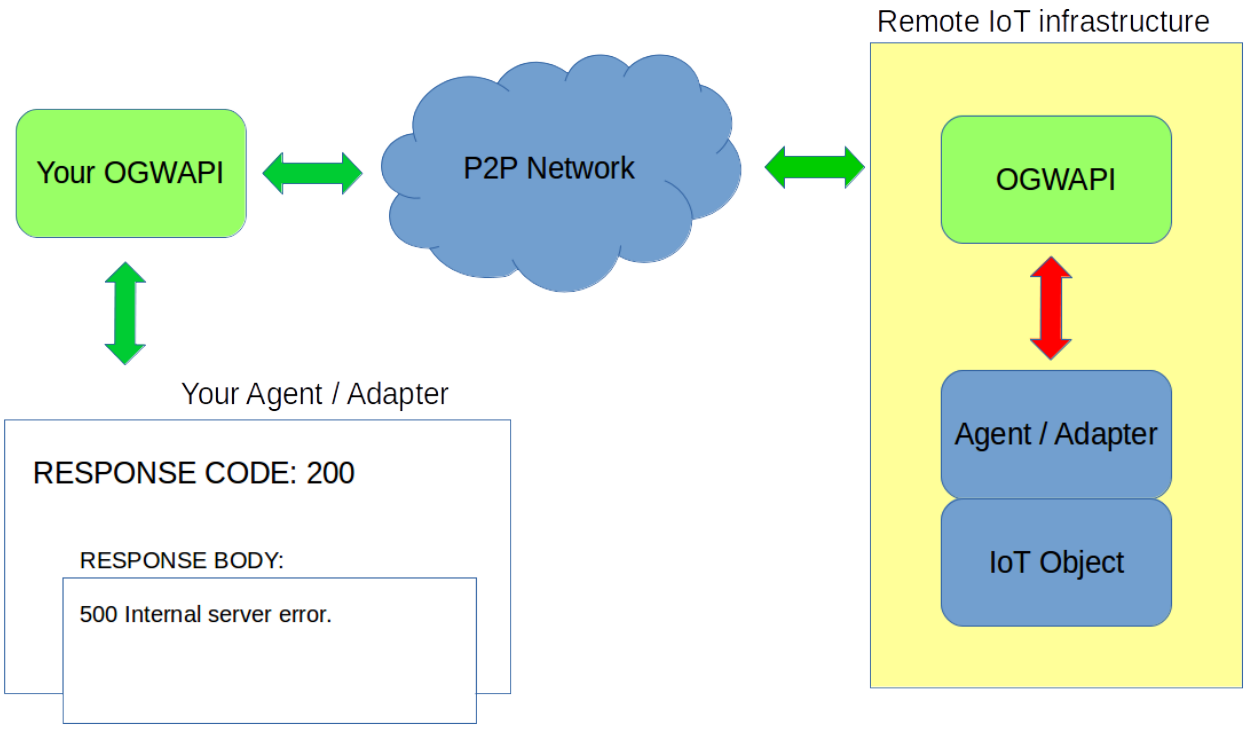

Illustration 5: Example of an error in the remote IoT infrastructure.

3.1 Object discovery and registration

Having an Agent connected into the OGWAPI as an additional layer brings an advantage of automatic discovery and registration of objects in your infrastructure. However, if you don't have it added in the processing chain, discovery and registration of devices must either be done manually one-by-one, or the automated process has to be performed by you. In order to leverage this functionality you have to perform the following steps:

1. Retrieve the objects from central servers, that are currently registered with your current infrastructure (you need the agent ID for doing this, so don't forget to register one in the neighbourhood manager). Naturally, in the beginning there will be none. To do this, use GET /api/agents/{agid}/objects.
2. Perform a discovery on your local infrastructure. There is no simple and generic way for all IoT infrastructures. You as an integrator will probably know your infrastructure the most and it is very likely that it possesses some sort of API that allows you to do this. Your task in this step is to list devices that are there on your network.

3. Compare these two sets. In other words, subtract the set of objects you received from our central servers from the set you obtained during the discovery in your infrastructure. This way you obtain a set of new objects that are to be registered. Create a JSON of Tds from the resulting set and do POST /api/agents/{agid}/objects.

The set of new devices will be registered on our servers and you should receive their freshly generated credentials in the response.

3.2 Data consumption

Data consumption is, along with object exposing, the functionality that will be used by your infrastructure most of the time. It is this functionality that lets you read and set properties of remote objects and to run actions on them. Needless to say that proper permissions need to be set in the Neighbourhood Manager Web first. The instructions in this section are just a logical sequence that has to be done, for the particular endpoint description please read the section Complete description of OGWAPI endpoints.

Properties are scalar values of an object in IoT ecosystem. As an example we can take a smart light bulb, which properties can include brightness and a colour of light it emits. These values can be read and set when correctly integrated into the system via OGWAPI, Agent and Adapter. Correct procedure would be to poll an object for a list of its properties and then to read or set a particular one:

1. GET /api/objects/{light bulb oid}/properties – returns a list of properties (optional, but recommended)
2. GET /api/objects/{light bulb oid}/properties/{e.g. brightness pid} – returns a value of a given property
3. PUT /api/objects/{light bulb oid}/properties/{e.g. brightness pid} – sets a value of a given property.

An **action** is something that the remote object can physically or virtually perform, can not be described by a simple scalar value and can take a long time to finish. Example of such action would be operating motorized window curtain. Issuing a command to perform an action will create an instance of **task** on the remote OGWAPI. The task is a representation of that particular action being currently executing.

Lets say we take the motorized window curtain, that has available actions ‘raise’ or ‘lower’ and a property ‘status’, that represents whether the curtain is currently raised or lowered. We poll it for ‘status’ property and if it is not in a desired position, we issue an action command to (let say) raise

it. The actual raising of the curtain is a task, that is being executed and its current status can be checked by the issuing object. The possible states of a task can be:

- Pending – somebody else also issued an action and our task is waiting in a queue for the previous to be finished.
- Running – our task is being executed.
- Failed – the execution can't proceed.
- Finished – our task is done.

Based on this, we can postulate a sequence of endpoints that has to be called in order to do this job:

1. GET /api/objects/{curtain oid}/actions – retrieves a list of actions (optional but recommended)
2. GET /api/objects/{curtain oid}/properties – retrieves a list properties (optional but recommended)
3. GET /api/objects/{curtain oid}/properties/{status pid} – gets the current status of the curtain, whether it is lowered or raised (optional but recommended)
4. GET /api/objects/{curtain oid}/actions/{raise action aid} – issues a command to raise the curtains – this returns a task ID
5. GET /api/objects/{curtain oid}/actions/{raise action aid}/tasks/{tid from step 4} – checks the status of the task – call this periodically to check whether it is done or not.
6. DELETE /api/objects/{curtain oid}/actions/{raise action aid}/tasks/{tid from step 4} – cancels the task if desired (optional…).

3.3 Exposing your IoT objects

For now we have been discussing how to poll a remote object for data, set its property or run an action. But what if you would like to subscribe to receive updates about some unscheduled changes in properties or other occurrences where periodical polling of that remote device is impractical? The event mechanism is a built in functionality of OGWAPI that is achieving exactly this. Before such mechanism can work, two things have to be ensured:

- the remote device has to be able to generate such event and has the subscription channel active and,
- the receiving device is subscribed to this channel and is capable of processing such information.

Lets start with the assumption that your device (object) can generate such events and you have an Adapter that can send this event to an OGWAPI when it happens. In order to activate the event channel, all your Agent/Adapter needs to do is call the endpoint

POST /api/events/{eid}

on your local OGWAPI. From that moment, the channel is active and remote objects can subscribe to it. At any time conditions are met to fire your event, create a JSON with what happened and send it to

PUT /api/events/{eid}

on your local gateway and it will get distributed among all the objects that are subscribed to it.

On the other hand, the remote site that wishes to receive these events needs to subscribe for their reception and be prepared to receive an event once it is sent. The recommended sequence is:

1. GET /api/objects/{oid}/events – retrieves a list of events supported by the remote object (optional but recommended)
2. GET /api/objects/{oid}/events/{eid} – retrieves a status of the channel (optional)
3. POST /api/objects/{oid}/events/{eid} – subscribes to the event channel

The Agent/Adapter on receiving side then needs to implement one endpoint, where the OGWAPI will connect in case an event is received. This is used as a callback – a remote object generates an event, it is sent to PUT /api/events/{eid} on an OGWAPI on its side, then distributed to subscribed objects, each of which needs to implement PUT /api/objects/{oid}/events/{eid}.

3.4 Search and querying the network

A SPARQL query can be used to poll the network of your befriended objects for certain data. This functionality is yet to be implemented.

3.5 Complete description of Open Gateway API endpoints

This section describes each endpoint provided by the OGWAPI software. Again, the endpoints are, for the sake of clarity, divided into groups according to the interface they are providing.

3.5.1 Authentication interface

Login

Description:

Logs the object into the P2P network. No parameters are required, however based on the authentication settings in the OGWAPI configuration file, the proper credentials are to be sent in the request header (consult online documentation for each authentication mechanism). Please note, that in compliance with the philosophy of REST calls, it is not necessary to explicitly use this endpoint to login, if not needed. The credentials are sent in each call anyway, so even if the Adapter is not online and it calls an endpoint on an OGWAPI, the credentials are verified and is automatically logged in.

Method:

GET

Endpoint:

`http://<gateway IP>:<port>/api/objects/login`

Parameters / payload:

None.

Returns:

200 and ‘Login successful’ in the body if the login credentials were valid, otherwise 400 Unauthorized.

Logout

Description:

Logs the object out of the P2P network, making it unreachable by other objects in the network.

Method:

GET

Endpoint:

`http://<gateway IP>:<port>/api/objects/logout`

Parameters / payload:

None.

Returns:

200 in all cases.

3.5.2 Consumption interface

Get a list of properties of a remote object

Description:

Retrieves a list of properties that the remote IoT object has.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/properties
```

Parameters / payload:

{oid} – Object ID of the remote object.

Returns:

List of properties that are provided.

Get a value of a property

Description:

Retrieves a value of a given property from a remote object.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/properties/{pid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{pid} – Property ID.

Returns:

A value of a given property.

Set a new value to a property

Description:

Sets a new value of a property on a remote object.

Method:

PUT

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/properties/{pid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{pid} – Property ID.

A JSON with a new value must be sent as a payload.

Returns:

Nothing.

Get a list of actions of the remote object

Description:

Retrieves a list of actions a remote object is providing.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/actions
```

Parameters / payload:

{oid} – Object ID of the remote object.

Returns:

Nothing.

Execute an action

Description:

Start execution of an action. The particular execution is called a task, has its own ID (that gets returned) and has multiple states, in which it can be – see the next endpoint.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/actions/{aid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{aid} – Action ID.

Returns:

A Task ID of the the particular execution.

Retrieve the status or a value of a given task

Description:

When an action is executed, it can take a while, like opening a door or running an algorithm. This particular execution, or run, is called a task. This endpoint returns a status of an action, that has been executed before.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/actions/{aid}/tasks/{tid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{aid} – Action ID.

{tid} – Task ID of the current execution.

Returns:

A status of the task and a resulting value, if there is some associated. The valid states are pending (the task has not yet been executed, but is queued), running (the task is in progress), failed (there was a problem during execution), finished (the task is done).

Cancel a task in progress

Description:

Cancels a task, that is in progress as a result of action being executed.

Method:

DELETE

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/actions/{aid}/tasks/{tid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{aid} – Action ID.

{tid} – Task ID of the current execution.

Returns:

Nothing.

An endpoint to call when a task is finished

Description:

When a task has finished its execution, the Agent/Adapter in charge of executing the task should call this endpoint to let the local OGWAPI know, that the action is completed. Since there is only one task at a time the Adapter is able to execute, there is no need to specify to OGWAPI which task is done – the queueing of the tasks is necessary for ‘the outside world’.

Method:

PUT

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/actions/{aid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

{aid} – Action ID.

Returns:

Nothing.

3.5.3 Discovery interface

Get a list of all visible objects

Description:

Retrieves a list of all IoT objects that are visible to that particular Agent/Adapter based on the permissions set in Neighbourhood Manager Web interface. This includes both your own and foreign devices. In order to make it into the list, it is necessary for the object to be online.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects
```

Parameters / payload:

None.

Returns:

JSON with a list of OIDs (object Ids).

Retrieve a thing description

Description:

Retrieves a semantic thing description of the object with a particular OID.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}
```

Parameters / payload:

{oid} – Object ID of the remote object.

Returns:

A thing description of an object, that should also include what actions, properties, events, etc. it supports.

Retrieve a list of objects connected to this agent

Description:

Retrieves a list of objects that are connected to this particular Agent (identified by his AGID – Agent ID). Note that it is not yet supported to retrieve such a list from a remote Agent, so you can poll the network only for the local one. It is necessary to call this endpoint before an automatic registration is attempted, so you know which objects need to be registered.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/agents/{agid}/objects
```

Parameters / payload:

{agid} – Agent ID of the local Agent.

Returns:

A JSON with a list of objects that are already registered under the given Agent, and the system – network and servers – are keeping track of.

3.5.4 Exposing interface

Activate an event channel

Description:

Used by an Agent/Adapter, that is capable of generating events and is willing to send these events to subscribed objects. A call to this endpoint activates the channel – from that moment, other objects in the network are able to subscribe for receiving those messages.

Method:

POST

Endpoint:

```
http://<gateway IP>:<port>/api/events/{eid}
```

Parameters / payload:

{eid} – Event ID.

Returns:

Nothing.

Send an event to subscribed objects

Description:

Used by an Agent/Adapter that is capable of generating events, to send an event to all subscribed objects on the network.

Method:

PUT

Endpoint:

```
http://<gateway IP>:<port>/api/events/{eid}
```

Parameters / payload:

{eid} – Event ID.

JSON containing the event, created according to semantic rules.

Returns:

Nothing.

De-activates the event channel

Description:

Used by an Agent/Adapter that is capable of generating events to de-activate an event channel. This will prohibit any other new objects to subscribe to that channel, and the objects that are already subscribed are notified and removed from subscription list.

Method:

DELETE

Endpoint:

```
http://<gateway IP>:<port>/api/events/{eid}
```

Parameters / payload:

{eid} – Event ID.

Returns:

Nothing.

List events generated by an object

Description:

This is used to check what events is a remote object capable of generating, and what channels are active.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/events
```

Parameters / payload:

{oid} – Object ID of the remote object.

Returns:

A JSON with a list of events that can be generated by a given object and which of them are active. JSON is generated by a rules given by the semantic dictionary.

Retrieve a current status of an event channel

Description:

Retrieves a status of a particular event channel – whether it is active or deactivated.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/events/{eid}
```

Parameters / payload:

{oid} – Object ID.

{eid} – Event ID.

Returns:

A JSON with a status of event channel. The JSON is generated according to the semantic dictionary.

Subscribe for event reception

Description:

Subscribes the object to an event reception. From that moment, the object calling this endpoint will start receiving events, until the channel is deactivated or the subscription is cancelled.

Method:

POST

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/events/{eid}
```

Parameters / payload:

{oid} – Object ID.

{eid} – Event ID.

Returns:

Nothing.

Unsubscribe from the event reception

Description:

By calling this endpoint, an object can cancel its own subscription to event channel.

Method:

DELETE

Endpoint:

```
http://<gateway IP>:<port>/api/objects/{oid}/events/{eid}
```

Parameters / payload:

{oid} – Object ID.

{eid} – Event ID.

Returns:

Nothing.

3.5.5 Registry interface

Register a set of new objects

Description:

Agent can use this endpoint to register a set of new devices (objects).

Method:

POST

Endpoint:

```
http://<gateway IP>:<port>/api/agents/{agid}/objects
```

Parameters / payload:

{agid} – Agent ID.

As a payload, a JSON with device Tds must be sent. The content of this JSON must be in line with semantic dictionary.

Returns:

JSON with new generated OIDs and credentials. The JSON will be in line with semantic dictionary.

Update existing set of objects

Description:

Agent can use this endpoint to update Tds of already registered objects.

Method:

PUT

Endpoint:

```
http://<gateway IP>:<port>/api/agents/{agid}/objects
```

Parameters / payload:

{agid} – Agent ID.

As a payload, a JSON with device Tds must be sent. The content of this JSON must be in line with semantic dictionary.

Returns:

JSON with new generated OIDs and credentials. The JSON will be in line with semantic dictionary.

Delete a set of objects

Description:

An agent can delete a set of objects, that are registered through it.

Method:

POST

Endpoint:

```
http://<gateway IP>:<port>/api/agents/{agid}/objects/delete
```

Parameters / payload:

{agid} – Agent ID.

As a payload, a JSON with device Ids must be sent. The content of this JSON must be in line with semantic dictionary.

Returns:

Nothing.

3.5.6 Query interface

Simple search

Description:

The network can be polled for specific data by using this endpoint.

Method:

GET

Endpoint:

```
http://<gateway IP>:<port>/api/sparql
```

Parameters / payload:

To be done.

Returns:

Search results.

Complex search

Description:

The network can be polled for specific data by using this endpoint.

Method:

POST

Endpoint:

```
http://<gateway IP>:<port>/api/sparql
```

Parameters / payload:

To be done.

Returns:

Search results.

4 Integration and adapter development

Thus far, only the ways of accessing the remote befriended objects and their properties or actions was described. However we did not discuss what is necessary to be done in order to provide the same type of functionality to other remote objects. In other words, we did not cover what endpoints need to be implemented on your side in order to be reachable.

When, for example, a remote object calls an endpoint on its local Agent/OGWAPI, requesting a temperature property of your object, and that request is transmitted over the network into your local OGWAPI, the request is sent into your local Agent, that then distributes it to appropriate object. In order to do this, your local Agent needs to have an endpoint that your local instance of OGWAPI will call, when the request arrives, and where the value of the property will be provided. We call this a mirroring and we will use this term to describe what endpoint needs to be implemented to provide which functionality for some remote endpoint.

These are in contrary to whole OGWAPI just a few end points in consumption and exposing interface. It is fair to say, that you don’t need to implement all of the endpoints that the OGWAPI could use on your Agent/Adapter and some of them are not even possible to be implemented in any way.

4.1 Consumption interface integration

Consumption interface (as stated in previous descriptions) handles properties and actions. In order too be able to provide a property to other objects on the network, your Agent/Adapter needs to implement endpoints for

- providing a property from your device,
- setting a property on your device.

Similarly, when speaking about actions, your Agent/Adapter needs to implement endpoints for

- starting an execution of an action,
- retrieving a status of ongoing execution,
- cancelling an ongoing execution.

Moreover, after an action is finished, don’t forget to call PUT /api/objects/{oid}/actions/{aid}, so the OGWAPI will know that the action is done (and can start execution of a next task in the queue).

4.1.1 Properties

Providing a property

Description:

Endpoint should provide property value in line with the correct TD semantics.

Mirrors this OGWAPI endpoint:

`GET /api/objects/{oid}/properties/{pid}`

Method:

GET

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/properties/{pid}`

Parameters / payload:

{oid} – Object ID of your object – the one that holds the value of the property in question.

{pid} – The property in question.

Returns:

{

“brightness”: 65

}

Setting a property

Description:

Endpoint should set the property value to that it did receive (which should be in line with the correct TD semantics).

Mirrors this OGWAPI endpoint:

`PUT /api/objects/{oid}/properties/{pid}`

Method:

PUT

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/properties/{pid}`

Parameters / payload:

{oid} – Object ID of your object – the one that holds the value of the property in question.

{pid} – the property in question.

As a payload a JSON with the new property value should be sent, e.g.:

```
{
   “brightness”: 65
}
```

Returns:

Nothing.

4.1.2 Actions

Starting an execution

Description:

This endpoint is called when a request for starting an action arrives. Parameters can be sent in the payload.

Mirrors this OGWAPI endpoint:

`POST /api/objects/{oid}/actions/{aid}`

Method:

POST

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/actions/{aid}`

Parameters / payload:

{oid} – Object ID of your object – the one that can execute an action.

{aid} – The action in question.

Returns:

Nothing.

Retrieve a status of ongoing action

Description:

This endpoint when called should return a status of ongoing action on a particular device. Available statuses are:

- running,
- failed,
- finished.

You might have noticed that there was another status when discussing the consumption interface of OGWAPI – pending. This is however used only by OGWAPI when some external object is trying to execute an action, while another action is already being executed on your object. These concurrency issues are solved on the level of OGWAPI.

Mirrors this OGWAPI endpoint:

`GET /api/objects/{oid}/actions/{aid}/tasks/{tid}`

Method:

GET

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/actions/{aid}`

Parameters / payload:

{oid} – Object ID of your object – the one that can execute an action.

{aid} – The action in question.

Returns:

```
{
   “status”: “running”
}
```

Cancelling an ongoing execution

Description:

This endpoint will be called when a request for cancelling an ongoing execution arrives from some external object.

Mirrors this OGWAPI endpoint:

`DELETE /api/objects/{oid}/actions/{aid}/tasks/{tid}`

Method:

DELETE

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/actions/{aid}`

Parameters / payload:

{oid} – Object ID of your object – the one that can execute an action.

{aid} – The action in question.

Returns:

Nothing.

4.2 Exposing interface integration

When utilizing the exposing interface, we are used to think about the data chain as a distribution of events. In the section dedicated to that interface, we discussed the easy ways of how to enable the channel (if the device is the one that generates the event), how to send it and how to subscribe for reception of such events.

However, there is one part, that was not discussed yet. As the event mechanism is an asynchronous one, the object that wants to receive an event it is subscribed for, never knows when the event actually arrives. In order to facilitate a transmission of event to its final destination, the Agent / Adapter needs to implement an endpoint the OGWAPI will call, when an event arrives. Again, it is a mirror of the one that has been used to send the event in the first place.

Receive an event

Description:

This endpoint will be called when an event that the object is subscribed for arrives.

Mirrors this OGWAPI endpoint:

`PUT /api/objects/{oid}/events/{eid}`

Method:

PUT

Endpoint:

`http://<agentIP>:<agentPort>/agent/objects/{oid}/events/{eid}`

Parameters / payload:

{oid} – Object ID of your object – the one that is subscribed for event reception.

{eid} – The event ID.

Returns:

Nothing.

5 Frequently asked questions

These are the most frequent questions we were being asked while integrating various scenarios using the OGWAPI. We are trying hard to integrate answers to these questions somewhere into the documentation context, some questions however does not really fit into any section and is therefore recommended to read this part even if you don’t really have any questions, as it can give you some more insight into the OGWAPIs way of operation.

Q: Is it always necessary to use an Agent (internal or external) in order to receive data from OGWAPI?

A: No, the Agent is not mandatory part of the communication chain, although using it brings many advantages. However if your hardware is tight on resources or capabilities, you can wire your Adapter straight to OGWAPI. See the sections Overview, and Configuration.

Q: [Linux] When I try to run the OGWAPI as a user dedicated user, I get:

```
Error: Could not find or load main class [some packages].App
```

A: The directory, where you have your instance of OGWAPI, or any other parent directory higher in the directory tree is not executable for the dedicated user. Keep in mind that in order to ‘visit’ or ‘go through’ a directory on its way to the instance from the root directory when executing, the user needs to have ‘x’ permission on each of them. This is normal behaviour reserved not just for OGWAPI but for other applications as well.

Q: Is it possible to run multiple instances of OGWAPI on one machine, each bound to a different interface?

A: Yes, you can run multiple instances of OGWAPI on one machine. However keep in mind, that each one need to have its own installation directory and configuration file. If you installed your OGWAPI from a .deb or .rpm package, the other instances have to be installed from the cloned Git repository, as there can’t be more packages of the same name installed on one system with the package managers. Naturally, each instance then needs to be configured in its own configuration file to use at least different port. Regarding the binding to different interfaces, there is a parameter in the configuration file that allows you to bind that particular instance of OGWAPI to a single IP address and port on the machine. This way you can have multiple OGWAPIs listening on different IP addresses and ports, while running on the same machine.

Q: Is it possible to run multiple instances of Adapter connecting into one common OGWAPI?

A: Absolutely. However you need to have Agent enabled, either external or internal. The Agent behaves (not only) as a router in such cases.

Q: Can I specify the OGWAPI to use a different configuration file, other than set by default?

A: Yes. When you run your instance of the OGWAPI, just add the absolute path to your alternative configuration file as a command line argument (no option switch is required). If no specific path is added, the OGWAPI uses the ‘GatewayConfig.xml’ stored in the ‘config’ subdirectory relative to the binary file.