Application - Data Access Interface specification

Authors

Peter Kaae Thomsen

Legal Information

Disclaimer

The contents of this document are subject to change without prior notice. OrbiWise makes no representation or warranty of any nature whatsoever (neither expressed nor implied) with respect to the matters addressed in this document, including but not limited to warranties of merchantability or fitness for a particular purpose, interpretability or interoperability or, against infringement of third party intellectual property rights, and in no event shall OrbiWise be liable to any party for any direct, indirect, incidental and or consequential damages and or loss whatsoever (including but not limited to monetary losses or loss of data), that might arise from the use of this document or the information in it.

OrbiWise and the OrbiWise logo are trademarks of the OrbiWise SA company.

All other names are the property of their respective owners.

Trademark List

All trademarks and registered trademarks are the property of their respective owners.

Release	Date	Modification description
0.1	22-11-2014	Started
0.2	12-02-2015	Updated to be aligned to mid February release
0.3	16-02-2015	Branched of into two documents. These are the “public” API.
0.4	26-02-2015	Added push mode interfaces.
0.8	16-04-2015	Back annotated with updated from DASS-RNSS interface specification [5].
0.9	24-04-2015	Added new “out-of-band” parameter to set initial spreading factor on slot rx2. Detail HEX and Base64 formats
0.10	26-06-2015	Fixed type in JOIN procedure key usage. Added new (optional) fields in the registration message Added new option to control confirmed/unconfirmed downlinks.
0.11	24-08-2015	Added clarification of downlink life cycle and downlink transmission status
0.12	12-12-2015	Added API for web-socket based push messages.
0.13	5-May-2016	Updated with clarification and updated API for customers and users. Added explanation on rights and login-forwarding. Added gateway API section.
0.14	1-Aug-2016	Added MQTT section

Table of Content

1 Introduction

1.1 Purpose

This document specifies the interface between the Data Access Sub-System and the OrbiWise Radio Network Sub-System.

1.2 Status of document

First release, aligned to OrbiWise external Release 1 mid February 2015.

1.3 Scope

1.4 Reference

[1] LoRaWAN V1.0
[2] RSS – Net interface spec v0.1
[3] Guidelines for 64-bit Global Identifier (EUI-64) General
https://standards.ieee.org/develop/regauth/tut/eui64.pdf
[4] Base64 encoding
https://tools.ietf.org/html/rfc4648
[5] DASS – RNSS interface specification v1.0

1.5 Definitions, Acronyms, and Abbreviations

DASS	Data Access Sub-System
RNSS	Radio Network Sub-System
APP	Application
CRUD	Create, Read, Update & Delete : REST protocol paradigm

2 Interface overview

This document details the interface seen by the customer applications towards the OrbiWise DASS (Data Access SS). The interface enables the customer to manage devices on the network as well as the operator to register customers.

Data send to and received from the devices are cached in persistent temporary storage inside the RNSS (in the RNSS proxy SS) ensuring that there are no real-time constraints on the applications to access the data.

Figure 1 show the context of the interface in the overall network.

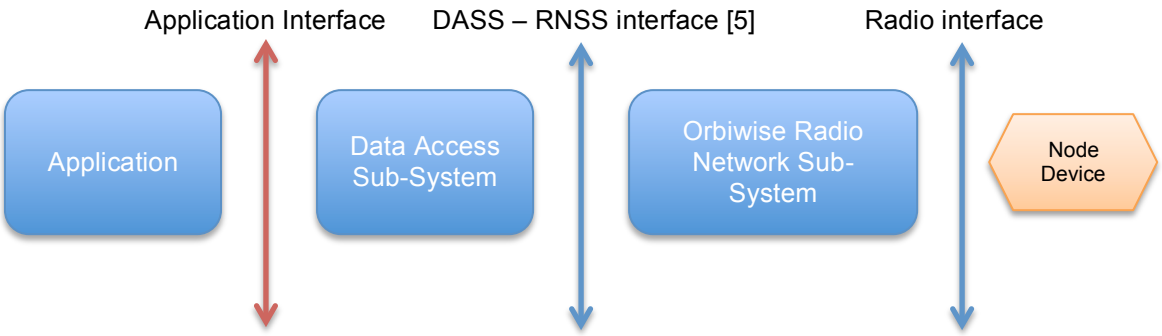


Figure 1. - OrbiWise Solution interfaces overview

Messages are REST based following the “CRUD” principle with transport over HTTPS. Authentication is done using the Basic authentication scheme of HTTP(S). Basic authentication uses the standard header field “Authorization” with the argument string: “Basic [username:password]$_{\text{base64 encoded}}$”.

Data exchanged in the message body is JSON format and will have “Content-Type” HTTP header directives with “application/json”. All messages are HTTP v1.1 compliant.

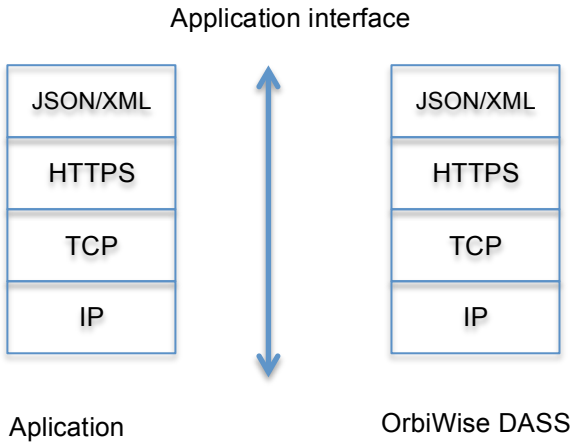


Figure 2. - Stack overview

The Application<->DASS interface is a host/client <-> host/client interface, meaning that both sides can initiate request and receive requests. After server start-up, the DASS will act as host only and the application will connect to the DASS as a client only. In this “pull-mode” the application must initiate all communication and must poll (GET) the DASS for updates.

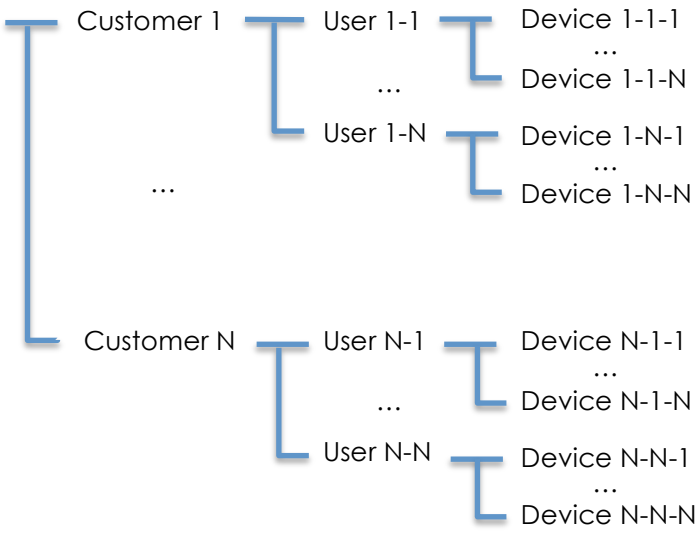
To avoid this polling the application must register its own host interface on the DASS after which the DASS will be able to act as a client too, and make unsolicited PUT/POST to the DASS with incoming payload and update of status. This is “push-mode”.

Push-mode and Pull-mode co-exist simultaneously, but once push-mode is started, only house-keeping operations will be performed in pull-mode and all “normal” data flow is performed in push-mode.

3 Customers and Users

The DASS organize access to devices by a hierarchy of customers and users. Customers and users are similar in that they can both be assigned the same rights and can both own devices, but differ only in their hierarchical organization.

Each device is associated with exactly one user or customer and a user belongs to exactly one customer.



The customer and user hierarchical organization allows the operator to create customer accounts for customers and assign the rights to create and manage their own users. The customer level can also be used to simply organize users into different groups - the organization does not impose any particular policies.

3.1 Rights

Customers and users can be assigned a number of rights. The most important rights (a full list of rights incl. system administration rights are specified in separate annex) are:

Administration rights:

- Customer administrator
- (User) administrator
- Device administrator
- Gateway administrator

General rights:

- Can own gateways
- Can manage gateways
- Access to network gateways metadata and location

Users and customers can be assigned any of the above rights individually and it is possible to have a user account with full set of administration rights, while the customer account that the user belong to have no rights at all. The purpose of customers and users is for hierarchical organization only.

Users or customer accounts (referred to as just accounts in the following) with “customer administrator” rights are considered as the highest order of administrator. Administrators of this type typically only belong to the network operator. Customer administrators can create and delete other customer accounts, and can login forwarding (see section 3.2) to any customer account and user

account to manage these accounts (i.e. register / delete devices, change rights, change account settings, etc.). A customer administrator can see a list of all customers on the system.

Accounts with (user) administrator rights (generally referred to only as administrators) have can manage all users that belong to the same customer. I.e. if the customer account itself has (user) administrator rights it can manage its own users (add / delete / modify / register devices etc.). Similarly user accounts with administrator rights can manage all the users that belong to the same customer as the account itself. Administrators cannot access any information that belongs to other customers. An (user) administrator can see a list of all the users that belong to the same customer.

Accounts with device administrator rights (referred to as the “can_register” rights in the user provisioning API) can add / delete / modify devices on its own account. Accounts without this right cannot manage the its own devices (it can still access the devices data though) and must rely on an administrator to perform the device provisioning.

Accounts with gateways administrator rights can assign gateways to its account. Gateways generally belong to the network and are managed by the network administration tools and not in the DASS. However it is possible for customers to see status information about certain gateways (e.g. gateway that are located on the customers premises). Some gateway parameters such as name and position can also be managed by DASS accounts with explicit rights for this. The gateway administrator rights only give rights to add gateways to its own account but when combined with the (user) administrator and customer administrator rights, gateways administrators can assign gateway to other users using login forwarding (see section 3.2).

In addition to the administration rights there are several additional finer granularity rights properties that can assigned.

The “can own gateways” right must be set on user accounts that is assigned gateways. Accounts that do not have this right will see all gateway related API as non-existent and cannot have gateway assigned to it. Only gateway administrators can assign the “can own gateways” right.

The “can manage gateways” right is an additional right that can be assigned to accounts with the “can own gateways”. This right allows modification of some gateway parameters such as the position of the gateway, and the assigned name.

Accounts with “can access network gateway info” will receive in all uplink payload message an additional object with information about the gateways that received the uplink. Information such as RSSI and SNR and the gateway ID is available with each uplink message. Further, accounts with this right can query the location of the network gateway based on the ID.

More details on the various rights details is provided in the REST API sections later in the document.

3.2 Administration, Login Forwarding and Rights Promotion

The credentials in the HTTP header identify a user or customer. The userids (the userid field is used the account name for both Users and Customers) must be unique and it is therefore recommended to enforce a policy of using e.g. email addresses as accounts.
The user ID is always provided in the “Authorization” field in the HTTP head together with the password encoded in the standard “basic authentication scheme”.

Users with administration or customer administration rights can “masquerade” as another user or customer respectively, by using login forwarding.

Login forwarding means that the administrator user can use its account credentials to login in as another user or customer. Appending the administrator username with the slash separator followed by

the target user/customer account userid does this. I.e. the Authorization header field would look like this:

Authorization: adminuser-userid/targetuser-userid:adminuser-password|$_{\text{base64 encoded}}$

The DASS will interpret the credentials above as if the REST command is for the target-user, but with the difference that the rights of the target-user is temporarily elevated (promoted) to the rights of the admin-user doing the login.

This means that for a user the e.g. doesn't have rights to register devices, an administrator can perform the registration for the user, but using login-forwarding and use the device registration commands as if the target-user had rights to do it.

4 Message exchange description

Three classes of commands are available; device management, device payloads commands and customer management. A short summary of the commands is shown below. All node devices are identified by their DevEUI (device extended Unique Identifier), hence for any device to be registered to the network it must have valid DevEUI. Devices are always registered uniquely to one customer profile.

4.1.1 Device management (add, remove and get status):

Get list of registered nodes or just one node. Returns basic registration status and downlink fcnt.

```
GET /rest/nodes
GET /rest/nodes/{deveui}
```

Add a new node to the DASS/RNSS.

```
POST /rest/nodes
```

Remove a node from the DASS/RNSS

```
DELETE /rest/nodes/{deveui}
```

Request (PUT) a status update (i.e. LoraWAN DevStatusReq MAC procedure) from the node, and read (GET) the device MAC status. The status include battery level and link margin of the node device.

```
PUT /rest/nodes/{deveui}/status
GET /rest/nodes/{deveui}/status
```

4.1.2 Payload management

Note all payloads are encrypted and must be encrypted/decrypted by the either the DASS or the applications.

Get all the received uplink payloads or just the latest. Each payload is returned with varies parameters such as receive timestamp, RSSI, SNR, SF and unique ID that is used to delete the payload when read.

```
GET /rest/nodes/{deveui}/payloads/ul
GET /rest/nodes/{deveui}/payloads/ul/latest
```

Delete uplink payload from the temporary storage. If payloads are not deleted after reading, next read will return the same payload. The ID is the ID returned with the payloads.

```
DELETE /rest/nodes/{deveui}/payloads/ul/{id}
```

Send a (encrypted) payload (downlink) to a node. The port and fcnt is provided as part of the URL. The payload is given directly as base64 encoded data in the HTTP message body:

```
POST /rest/nodes/{deveui}/payloads/dl?fcnt={xx}&port={yy}
```

The fcnt to use for the encryption and that must be provided as part of the send command can be read from the RNSS using the following command:

```
GET /rest/nodes/{deveui}
```

This returns (with other parameters) the last used fcnt. The next fcnt to use for the downlink is calculated simply by adding +1 to the returned fcnt.

Get status of outstanding downlink message. The ID is the ID returned in the send (POST) command.

```
GET /rest/nodes/{deveui}/payloads/dl/{id}
```

4.1.3 Push mode call-back messages

To avoid polling for new payloads, status for on-going downlink and node status, the user application can register to receive pushed callback messages when the status changes. Callback message are live messages that are sent immediately when the event happens.

Two push schemes are defined.

- No retry: If there are payloads already stored in the persistent temporary storage before the push mode is started, the application must manually read out these messages using the pull interface. Already existing payloads and status are not pushed.
- With retry: continuously push undelivered payloads to flush the persistent temporary payload storage.

When a new uplink payload has arrived the RNSS will send:

```
POST /rest/callback/payloads/ul
```

The message body will contain a JSON object with the node DevEUI, the actual payload, timestamp and all needed parameters to decrypt it.

In the downlink direction once a downlink message has completed and the delivery status is know (i.e. the payload we received or not) the DASS will send:

```
PUT /rest/callback/payloads/dl
```

The message body contain a JSON object with the node DevEUI, ID of the downlink payload and status of the delivery.

When there is an update on the node (such as an update on the FCNT, change in registration state, etc.) the DASS will send:

```
PUT /rest/callback/nodeinfo
```

The message body contain a JSON object with the node information.

When a node has updated it device status with battery status and link margin (either requested from the DASS or unsolicited by the RNSS) the RNSS will send:

```
PUT /rest/callback/status
```

The message body contain a JSON object with the node status.

When a node attempt to join the network with the JOIN procedure, the DASS will send the message to the user application with the join request for the application to authenticate, generate keys, and encrypt the join accept message:

```
PUT /rest/callback/join
```

The user application can register for push call-back message by sending a registration request to the DASS:

```
PUT /rest/pushmode/start
```

The message body must contain a JSON object with the URL of the user application host interface where the DASS will send the push callback messages.

When the users application no longer want to receive push (e.g. if the server is being restarted) it must send to the DASS:

```
PUT /rest/pushmode/stop
```

4.1.4 Customer registration messages

Get list of registered customers:

```
GET /rest/customers
```

Add a new customer. A JSON payload with credentials and rights of the customer must be provided in the message body.

```
POST /rest/customers
```

Delete a customer and all users and devices associated with the customer

```
DELETE /rest/customers/customer-id
```

4.1.5 User registration messages

Get list of users for a customer:

```
GET /rest/users
```

Add a new user to a customer. A JSON payload with credentials and rights of the user is provided to the message body.

```
POST /rest/users
```

Delete a user and all associated devices

```
DELETE /rest/customers/user-id
```

4.2 Formats

4.2.1 HEX Notation

All keys, EUIs and the DevAddr(*) are always represented in HEX notation in all JSON message and in the URL path (in the case of the DevEUI).

The ordering of bytes follows the recommendations from the LoraWAN specification.

Three notations are supported (with varies lengths for either DevAddr, DevEUI/AppEUI and keys):

“AABBCCDD00112233”

“0xAABBCCDD00112233”

“AA-BB-CC-DD-00-11-22-33”

All HEX notations are case insensitive.

Note, for DevAddr it is also possible to specify the address and a decimal number representation of the HEX value. In JSON notation the hex valies are represented as strings with quotes (“”) whereas the decimal number is directly represented as a number (i.e. without quotes).

4.2.2 Base64 payload notation

All payload data provided in either JSON (or directly in the message body for a downlink payload message) is always encoded in standard Base64 representation.

The Base64 encoding scheme, encodes binary data into an ASCII text string with each character representing 6-bit of the binary message. This allows inserting the binary payload into a standard string in the JSON message. The Base64 standard encoding scheme is defined in [4].

4.3 Device management - detailed description

This section describes the details of each message.

4.3.1 Registers a Node

Register a new node on the DASS/RNSS.

URL: `https://host[:port]/rest/nodes`

Method: POST
Direction: Application->DASS

The message body must contain a JSON object with some of the following fields:

```
{
  "lora_device_class": 0,         // 0: class A, 1: class B, 2: class C
  "deveui": "0981336439373734",   // 8-byte identifier in hex
  "appeui": "hex",                // 8-byte identifier in hex
  "nwkskey": "hex",               // 16-byte network session key in hex
  "appskey": "hex",               // 16-byte application session key in hex
  "appkey": "hex",                // 16-byte application key in hex
  "devaddr":122,                  // 32-bit device address in decimal, or
                                  // 32-bit hex string, e.g. "0000007a"
  "lora_fcnt_32bit":true,         // true: 32-bit fcnt, false: 16-bit fcnt
  "lora_rx_delay1":1,             // RxDelay1 parameter in seconds
  "lora_rx_delay2":2,             // RxDelay2 parameter in seconds
  "lora_major":0,                 // Lora Major number. Must be zero.
  "lora_rx2_sf":"sf12",           // (optional) Initial spreading factor
                                  // of RX slot 2. If not present sf12 is used
  "comment":"text comment",       // optional meta-data comment string
  "expiry_time_uplink": 168,      // ul payload expiry time in hours
  "expiry_time_downlink": 168,    // dl payload expiry time in hours

  "device_properties": "static,outdoor",// (optional) comma separated string
                                  // with properties. see list of
                                  // properties in table below.
  "qos_class": 0,                 // Assigned QoS profile. Valid values are
                                  // 0 to 3 (normally) where the value
                                  // is the index of the QoS profiled
                                  // by the network operator.
  "redundant_uplink_cnt": 0,      // (optional, default:0)
                                  // 0: let network decide, 1: use one
                                  // transmission per uplink (no redundancy),
                                  // 2-8: use 2-8 transmissions per uplink
  "max_allowed_dutycycle": 0.1,   // (optional) maximum allowed dutycycle
                                  // in percent.
  "expected_avr_dutycycle": 0.01,  // (optional) expected dutycycle of
                                  // device
  "options": 0                    // This field should only be used when
                                  // specifically requested by the operator.
                                  // can be omitted or set to zero.
}
```

The presence of the `appkey`, `appskey`, `nwkskey` and `devaddr` depends on the type of device registration used and the level of security required. The following combinations are supported.

Registration scheme	AppKey	DevAddr	NwkSKey	AppSKey	
JOIN					Node must JOIN. Join request will be pushed to application. Payload will pass through the DASS encrypted and must be encrypted/decrypted by the application. The NwkSKey must be provided to the DASS/RNSS as part of the push JOIN procedure.
	X				Node must JOIN. The JOIN request will be managed by the DASS. Subsequent payloads will be encrypted/decrypted in the DASS and the application can parse raw payloads. Note. An AppEUI can be provided as part of the registration, but the value will be overwritten by the actual value provided from the node during the JOIN procedure.
Personalised		X	X		The device has pre-generated keys. The network manages all MAC related encryption but pass the encrypted application payloads directly to and from the application and the application must perform encryption/decryption.
					The device has pre-generated keys. The network manages all MAC related encryption, and the DASS will manage all the

		X	X	X	application payload encryption, meaning the user application can provide and receive unencrypted payloads.

All field must be present in the JSON object (in accordance with the above table), but the `appeui` and `comment` fields can be set to empty strings.

The `comment` and `appeui` fields are not used by the RNSS but will be stored in the internal RNSS database to ease administration and debugging.

The DevEUI must be unique per device, the DevAddr can be assigned to multiple devices, but the DevAddr + NwkSKey pair must be unique.

The `expiry_time_uplink` and `expiry_time_downlink` is the time in hours an up- or downlink payload will stay in the persistent temporary storage if the DASS or application does not delete it.

The `device_properties` field is a string with comma separated properties that best describe the device. These properties are used by the network to improve the overall quality of server in the entire system. The following properties are currently supported.

Property name	Description
“static”	The device is installed in a fixed position and does not move.
“mobile”	The device can move around
“indoor”	The device is generally used indoors
“outdoor”	The device is generally used outdoors.

Return values:

Status value	**Meaning**	**Description**
200	OK	Registration was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Error in registration	One or more of the parameters are invalid. Possible causes for this error are: devaddr + nwkskey pair already used, missing parameters in the message.
406	Not Acceptable	Some of the parameters are not valid (e.g. wrong key length, etc).
409	Already registered	The device (based on the DevEUI) was already registered on the network. No values have been updated. If a device needs to be registered with new values, the device must be deleted first before being added again.

4.3.2 Get Node Info

Get information on a registered node.

URL: `https://host[:port]/rest/nodes/{deveui}`
Method: GET
Direction: Application->DASS

The command return a message with a JSON (or XML) payload with the following field:

```
{
    "device_status": 3,          // 0: registered but never seen,
                                 // 1: seen, but with MIC error,
                                 // 2: JOIN’ed (for JOIN devices only),
                                 // 3: successfully received uplink
```

```
        "last_reception": "timestamp", // time when node was last seen
        "dl_fcnt": 45,                 // last used downlink fcnt
        "device_class": 0,             // 0: class A, 1: class B, 2: class C
        "registration_status": 1,      // 0: pending RNSS registration,
                                       // 1: registered, 2: pending RNSS
                                       // deregistration
        "deveui": "deveui in hex",     // DevEUI of node in HEX
        "expiry_time_uplink": 168,     // ul payload expiry time in hours
        "expiry_time_downlink": 168    // dl payload expiry time in hours
    }
```

The timestamp is in ISO 8601 format “yyyy-mm-ddThh:mm:ss.SSSZ” format (where SSS is in milliseconds) and aligned with UTC time.

Return values:

Status value	Meaning	Description
200	OK	Request ok, the above JSON (or XML) message is received in the message body
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.3.3 Get List of Nodes

Get list of registered nodes with the node info for each node.

URL: https://host[:port]/rest/nodes
Method: GET
Direction: Application->DASS

The request returns an array of node info objects (see description of object in section 4.3.2):

```
[
    {"device_status": 3, "last_reception": "timestamp", "dl_fcnt": 45,
    "device_class": 0, "registration_status": 1, "deveui": "deveui in hex",
    "appeui": "HEX"},

    {"device_status": 3, "last_reception": "timestamp", "dl_fcnt": 45,
    "device_class": 0, "registration_status": 1, "deveui": "deveui in hex",
    "appeui":"HEX "},

    ...
]
```

Return values:

Status value	Meaning	Description
200	OK	Request ok, the above JSON (or XML) message is received in the message body
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.3.4 Delete Node from RNSS

Deregister a node from the RNSS.

Note that the successful return of this command means that the request has been accepted and is valid. The actual node de-registration and deletion from internal databases can take several seconds

for the RNSS to complete. There is therefor a chance that, if the same device is being registered again immediately after the deregistration, the re-registration may fail.
To see that a device have been completely deleted a query on the device with the Get Node Info command (see section 4.3.2) can be used and should return with a error 404.

URL: https://host[:port]/rest/nodes/{deveui}
Method: DELETE
Direction: Application->DASS

Return values:

Status value	Meaning	Description
200	OK	Delete was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.3.5 Request Node Status Procedure

Request the RNSS to perform the LoraWAN DevStatusReq MAC procedure (see section 5.5 “End-Device Status” in [1]). The procedure will be performed by the RNSS on the tail of the next device uplink. The result will be available from the device on the uplink again after that. Hence the procedure can take as long as the duration of two normally scheduled uplink periods from a device, and the results may not be available for hours or even days depending on the uplink period of the device.

URL: https://host[:port]/rest/nodes/{deveui}/status
Method: PUT
Direction: Application->DASS

Return values:

Status value	Meaning	Description
200	OK	Request was accepted. A DevStatusReq will be sent to the node on the next opportunity.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.3.6 Read Node Status Procedure Result

Read the result from a completed DevStatusReq MAC procedure (see previous section).

URL: https://host[:port]/rest/nodes/{deveui}/status
Method: GET
Direction: Application->DASS

The following JSON (or XML) object is returned:

```
{
    "battery_status": 0,   // value from LoraWAN specification,
                           // 0: connected to power source,
                           // 1..254: battery level 1 being minimum and
                           // 254 being maximum, 255: not possible to measure
    "margin_status": 0,    // receive margin of device in dB to its
                           // sensitivity level
    "timestamp_status": "2015-02-06T10:43:23.331Z",// GMT time of last report
    "req_status": 2        // 0: never updated, 1: update requested,
```

```
                                    // 2: request pending, 3: result ready
    }
```

The values can be considered valid only when the `req_status` field has a value of 3 (meaning result ready). The timestamp is in ISO 8601 format “yyyy-mm-ddThh:mm:ss.SSSZ” format (where SSS is in milliseconds) and aligned with UTC.

Return values:

Status value	Meaning	Description
200	OK	Request was OK. The above explained JSON object is returned in the message body.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.4 Payload Management – Detailed Description

4.4.1 Receive all Pending Uplink Payloads from Device

Receive an array with all received payloads for a device. The payloads are stored in the persistent temporary storage inside the RNSS. The application should delete the payload once they have been safely “consumed”. Unread payloads will be automatically deleted by the RNSS after an expiry period.

URL: https://host[:port]/rest/nodes/{deveui}/payloads/ul
Method: GET
Direction: Application->DASS

A JSON (or XML) message body is returned with an array of all payloads

```
[
   {
       "dataFrame": "AB==",   // raw (encrypted) payload in base64 format
       "port": 1,             // MAC port the message was receive on
       "timestamp": "2015-02-11T10:33:00.578Z",   // time of reception in GMT
       "fcnt": 138,           // uplink fcnt (needed for decryption)
       "rssi": -111,          // RSSI from gateway
       "snr": -6,             // SNR from gateway
       "sf_used": "8",        // used spreading factor
       "id": 278998,          // unique identifier (64-bit) of payload.
                              // needed to delete the payload

        "gtw_info": [         // see note below.
          {gtw_id: "0000000012340000", rssi: -100, snr: 5 },
          {gtw_id: "0000000012350000", rssi: -90, snr: 15 }, …
        ]
   },
    ...
]
```

The `gtw_info` field is only present for user or customer accounts that have the “can access network gateway info” right. It contains a list of all gateways that receive the uplink message, and the signal quality data associated with it. The gateway ID can be used to query the gateway location through the gateway API.

Return values:

Status value	Meaning	Description
200	OK	Request ok. The above list of payloads is return in the message body
204	No Payload	Request ok, but there is currently no payload pending for this devices (i.e. the persistent temporary storage is empty).
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.4.2 Receive latest payload from device

Devices where only the latest value is relevant can be query for just the latest received payload.

URL: https://host[:port]/rest/nodes/{deveui}/payloads/ul/latest
Method: GET
Direction: Application->DASS

A JSON message body is returned with a single payload object (i.e. no array). The payload object is described in the previous section.

Return values:

Status value	Meaning	Description
200	OK	Request ok. Single JSON object with payload and parameters (see section 4.4.1).
204	No Payload	Request ok, but there is currently no payload pending for this devices (i.e. the persistent temporary storage is empty).
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.4.3 Delete Uplink Payload

Payloads stay in the RNSS persistent temporary storage until they are deleted by the application or until the payload expiry period is reached. Once a payload has been read and provided to its final destination is should be deleted.
The payload is deleted based on its unique id that is provided to the application in the same message as the payload itself.

URL: https://host[:port]/rest/nodes/{deveui}/payloads/ul/{id}
Method: DELETE
Direction: Application->DASS

Return value:

Status value	Meaning	Description
200	OK	Payload successfully deleted
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.4.4 Send Downlink Payload to Node Device

Downlink payloads are sent to the DASS for it to transmit them to the Node device on next opportunity. Payloads are provided either encrypted or un-encrypted depending on the chosen key scheme (see 4.3.1).

Note: Devices that does not use the default application payload scheme (option in upcoming LoraWAN specification) does not need to provide the FCNT, but must encrypt the payload in a self-containing way that does not rely on the MAC frame FCNT. This will be detailed further in a later release.

URL:
https://host[:port]/rest/nodes/{deveui}/payloads/dl?port={xx}[&fcnt={yy}][&confirmed=true]
Method: POST
Direction: Application->DASS

The payload itself is put directly in the request message body in base64 format.

Downlink message are by default transmitted as confirmed messages, but it is possible to control the usage of confirmed vs unconfirmed downlink message types. Setting &confirmed=false as a URL option will force transmission of the downlink as unconfirmed.

As it may take long time before the payload can be transmitted it is not possible to get delivery status on the payload immediately. Instead the DASS must query the packet status, and a unique ID is therefore assigned to the downlink payload and returned on the POST for the DASS to identify the payload for later query.

The post return the following JSON object:

```
{
    "id": 252,                  // unique ID to query payload later
    "data": "ABC=",             // the payload data sent
    "fcnt": 10,                 // the used fcnt
    "port": 1,                  // the used port
    "transmissionStatus": 0     // see definition below
}
```

The meaning of the transmission status is the following:

Transmission status	Description
0	Payload pending transmission (i.e. not sent yet)
1	Payload has been sent, but reception status unknown
2	Payload has been sent and acknowledged by the device
3	Payload has been sent, and NOT acknowledged by the device => most likely the device did not receive the downlink payload.
4	An error has been discovered on the payload. The possible reasons for this error are: fcnt collision, payload size too big.

The downlink packet life cycle is shown in figure 3. The transmission status get updated live to reflect the status of the packet. If the DASS has registered for push, it will receive a downlink payload status callback message (see section 4.5.2) on every change of the status.

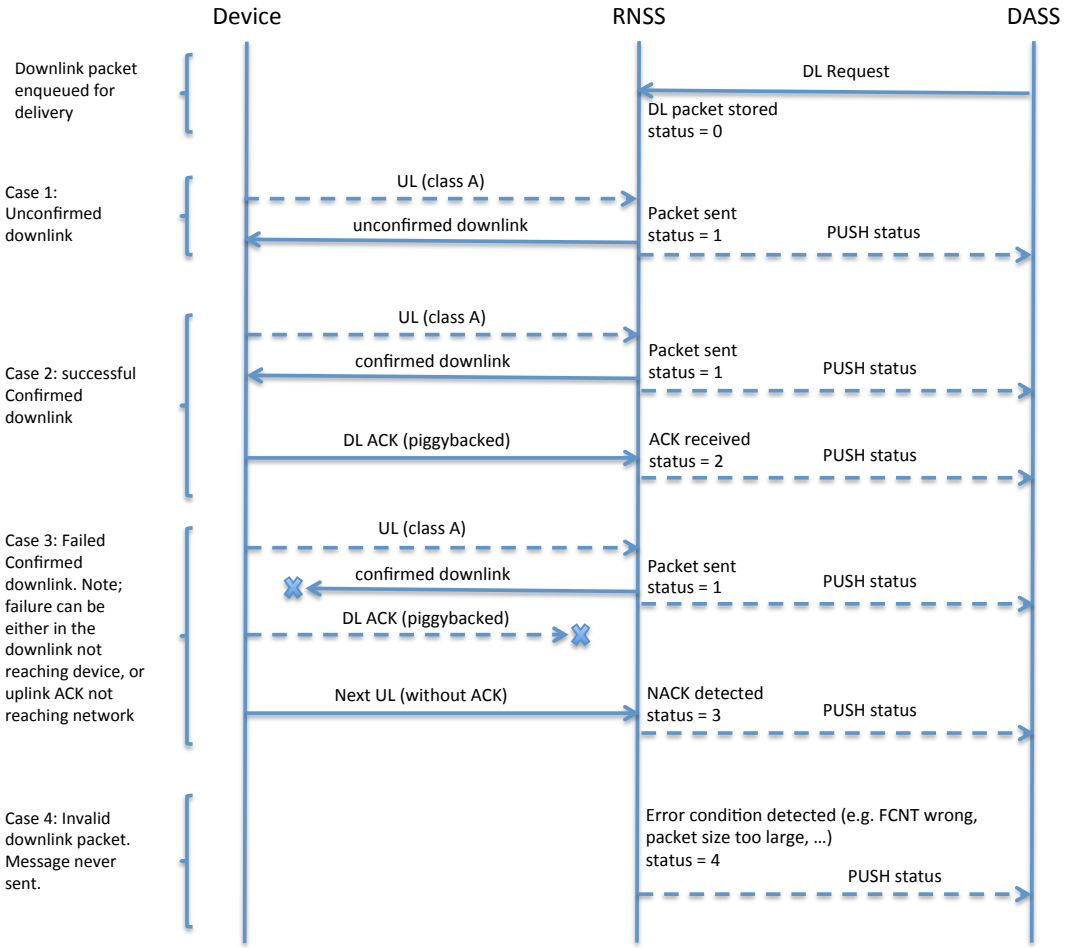


Figure 3: Downlink packet life-cycle

Return values:

Status value	Meaning	Description
200	OK	Payload successfully scheduled for transmission, and message body will contain the above JSON (or XML) object.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device	The DevEUI is not known

4.4.5 Check Downlink Payload Status

After scheduling a payload for transmission to a node device, the status of the packets successful delivery to the device can be queried. The payload status is queried by the ID returned from the POST command (see section 4.4.4).

URL: `https://host[:port]/rest/nodes/{deveui}/payloads/dl/{id}`
Method: GET
Direction: Application->DASS

The command returns the same JSON object as that returned by the payload POST commands (see section 4.4.4).

After the payload transmission has concluded (either successfully or not) the application should delete the payload packet from the DASS using the delete downlink payload command (see section 4.4.6). If the application does not delete the payload it will be automatically delete when the expiry period has been reached.

Return values:

Status value	Meaning	Description
200	OK	The downlink status and payload has been successfully found and returned in the message body. See section 4.4.4 for the details on the JSON (or XML) object.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device and/or packet ID	The DevEUI is not known or the packet ID does not exist.

4.4.6 Delete Downlink Payload

Downlink payloads stay in the RNSS persistent temporary storage even after having been sent to allow the application to query the status of the payload transmission. When the application has seen the completion of the downlink the payload should be deleted using this command.

URL: `https://host[:port]/rest/nodes/{deveui}/payloads/dl/{id}`
Method: DELETE
Direction: DASS->RNSS

Return value:

Status value	Meaning	Description
200	OK	The payload has been deleted.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown device and/or packet ID	The DevEUI is not known or the packet ID does not exist.

4.5 Push Mode Callbacks – Detailed Description

4.5.1 Receive Uplink Payload Callback

When a new uplink payload arrives the RNSS will store the message in the persistent temporary storage. If the application has registered push-mode callback the DASS will then immediately send the payload to the application with a POST. When the application has finished the request (i.e. answered 200 OK) the RNSS will automatically delete the payload from the persistent temporary storage again.

Note, the RNSS may delay the storage into the persistent temporary storage database by a configurable duration (e.g. 500ms) and if the application has confirmed the reception of the payload before this duration they payload will never be stored. This behaviour is transparent to the USER.

Push mode can be setup separately for each user profile.

The DASS push the following message on the payload arrival:

URL: `https://dash-host[:port][url-prefix]/rest/callback/payloads/ul`

Method: POST
Direction: DASS->Application

The message body contain the following JSON object:

```
{
    "deveui": "hex",        // DevEUI of source node
    "dataFrame": "AB==",    // raw (encrypted) payload in base64 format
    "port": 1,              // MAC port the message was receive on
    "timestamp": "2015-02-11T10:33:00.578Z",   // time of reception in GMT
    "fcnt": 138,            // uplink fcnt (needed for decryption)
    "rssi": -111,           // RSSI from gateway
    "snr": -6,              // SNR from gateway
    "sf_used": "8",         // used spreading factor
    "id": 278998,           // unique identifier (64-bit) of payload.
   "live": true,            // indicate if the message is live, or
                            // resent from the temporary storage
   "decrypted":false       // set true if the DASS decrypted the payload,
                            //  false if the message is still encrypted.

    "gtw_info": [           // see note below.
       {gtw_id: "0000000012340000", rssi: -100, snr: 5 },
       {gtw_id: "0000000012350000", rssi: -90, snr: 15 }, …
    ]
}
```

The `gtw_info` field is only present for user or customer accounts that have the “can access network gateway info” right. It contains a list of all gateways that receive the uplink message, and the signal quality data associated with it. The gateway ID can be used to query the gateway location through the gateway API.

The application must acknowledge the POST request by returning on of the following return values:

Status value	Meaning	Description
200	OK	The application has taken the payload and the DASS can delete it from the persistent temporary storage
202	Accepted	The application received the payload but will not process it. The DASS must leave the payload in the persistent temporary storage so it can be read out later via the “pull” interface using the GET.
Any other value	ERROR	The DASS will keep the payload in the persistent storage. The behaviour on any error is tbd. Current implementation ignores error and will not retry but will continue to push new messages. Error behaviour will be detailed in later release.

4.5.2 Downlink Payload Status Callback

When a downlink payload transmission has completed (successfully or unsuccessfully) the DASS inform the application (if the application has registered for push mode) with message with the downlink delivery status.

URL: `https://application-host[:port][url-prefix]/rest/callback/payloads/dl`
Method: PUT
Direction: DASS->Application

The message body contains the following JSON object:

```
{
    "deveui": "hex",            // DevEUI of the receiving node
```

```
    "id": 252,                  // unique ID of the dl payload
    "data": "ABC=",             // the payload data sent
    "fcnt": 10,                 // the used downlink fcnt
    "port": 1,                  // the used port
    "transmissionStatus": 0     // see table in section 4.4.5
}
```

The status of the downlink payload can be read in the `transmissionStatus` field, see table in section 4.4.5 for a definition of the status values.

The application must acknowledge the PUT request by returning on of the following return values:

Status value	Meaning	Description
200	OK	The application has read the downlink payload status and the DASS can delete the payload
202	Accepted	The application received the payload status but will not process it. The DASS must leave the payload in the persistent temporary storage so its status can be read out later via the “pull” interface using the GET.
Any other value	ERROR	The DASS will keep the payload in the persistent storage. The behaviour on any error is tbd. Current implementation ignores error and will not retry but will continue to push new messages. Error behaviour will be detailed in later release.

4.5.3 Node Info Update Callback

When node info is updated (e.g. the downlink FCNT is changed) and the DASS has registered for push message, the DASS will send a message with the node information to the DASS.

URL: `https://application-host[:port][url-prefix]/rest/callback/nodeinfo`
Method: PUT
Direction: DASS->Application

The message contains the following JSON object:

```
{
    "deveui": "hex",            // DevEUI of source node
    "device_status": 3,         // status of device, see section 4.3.2
    "last_reception": "timestamp", // time when node was last seen in GMT
    "dl_fcnt": 45,              // last used downlink fcnt
    "device_class": 0,          // 0: class A, 1: class B, 2: class C
    "registration_status": 1,   // registration status, see section 4.3.2
    "expiry_time_uplink": 168,  // ul payload expiry time in hours
    "expiry_time_downlink": 168 // dl payload expiry time in hours
}
```

The application must acknowledge the PUT request by returning on of the following return values:

Status value	Meaning	Description
200	OK	The info was received by the DASS
Any other value	ERROR	The behaviour on any error is tbd. Current implementation ignores error and will not retry but will continue to push new messages. Error behaviour will be detailed in later release.

4.5.4 Node Status Update Callback

When the device status from a node has been received by the RNSS and the DASS has registered for push mode, the RNSS send the device status update to the DASS.

URL: `https://application-host[:port][url-prefix]/rest/callback/status`
Method: PUT
Direction: DASS->Application

The message body contains the following JSON object:

```
{
    "deveui": "hex",       // DevEUI of source node
    "battery_status": 0,   // value from LoraWAN specification,
                           // 0: connected to power source,
                           // 1..254: battery level 1 being minimum and
                           // 254 being maximum, 255: not possible to measure
    "margin_status": 0,    // receive margin of device in dB to its
                           // sensitivity level
    "timestamp_status": "2015-02-06T10:43:23.331Z", // UTC time last report
    "req_status": 2        // 0: never updated, 1: update requested,
                           // 2: request pending, 3: result ready
}
```

The DASS must acknowledge the PUT request by returning on of the following return values:

Status value	Meaning	Description
200	OK	The info was received by the DASS
Any other value	ERROR	The behaviour on any error is tbd. Current implementation ignores error and will not retry but will continue to push new messages. Error behaviour will be detailed in later release.

4.5.5 Join Callback

When a device is registering on the DASS using the JOIN procedure and the AppKey is NOT provided to the DASS, the DASS will forward the JOIN request to the application for it to authenticate, generate keys and encrypt the join accept message.

Note, there is no pull mode version of the join callback, hence push modem must be enabled at all times for join procedure device to be able to join the network, unless the AppKey has been registered on the DASS in which case the DASS will manage the JOIN procedure autonomously.

The join procedure has real time constraints and the application must answer back to the DASS within <3 seconds (tdc).

URL: `https://application-host[:port][url-prefix]/rest/callback/join`
Method: PUT
Direction: DASS->Application

The message body contain the following JSON object:

```
{
    "deveui": "hex",       // DevEUI (8 bytes) of source node
    "appeui": "hex",       // AppEUI (8 bytes) of source node
    "join_request": "base64",    // raw join request message from node
    "join_accept": "base64"      // unencrypted join accept message to
                                 // be encrypted
}
```

The DASS (or application) must perform the MIC check on the raw join request message using the AppKey (that must be know by the DASS or application). If the MIC is valid the DASS can extract the DevNonce from the join request message, the AppNonce, NetID and DevAddr from the unencrypted

join accept message and use these values together with the AppKey to create the two session keys, NwkSKey and AppSKey.

The AppKey is then used to calculate the MIC of the join accept message and finally used to encrypt the join accept message.

The details of join request and join accept frame formats are specified in [1] section 6.2.3 “Join procedure”.

The encoded join accept message and the generated Network session key (NwkSKey) is then passed back to the RNSS in the message body of the answer to the request. The answer JSON object is as follows:

```
{
    "deveui": "hex",              // DevEUI (8 bytes) of source node
    "join_accept": "base64",      // encrypted join accept message to send
                                  // back to the joining node
    "nwkskey": "hex"              // Generate network session key (16-bytes)
}
```

Return values:

Status value	Meaning	Description
200	OK	The join request MIC is ok and a valid join accept message and nwkskey is available in the JSON message body.
403	Forbidden	The DevEUI is not known and this node is not allowed to join
406	Not acceptable	The DevEUI is known, but the MIC check failed.

4.5.6 Push Mode Start

When the application want to receive push messages from the DASS it must first implement a HTTPS host interface that can be reached from the DASS and identified by a hostname or IP address. The application can then register the interface on the DASS and start push mode, which will cause the DASS to push any new payload or update in status directly to the application host interface. The application must implement the paths specified in previous sections for each of the different types of push messages.

To register and start the push mode service the following message is used:

URL: `https://dash-host[:port] /rest/pushmode/start`
Method: PUT
Direction: application->DASS

The message body must contain a JSON object with the following parameters:

```
{
    "host": "hostname",    // Hostname or IP address
                           // of DASS HTTPS host interface
    "port”: 1234,          // port number of DASS HTTPS host interface
    "path_prefix": "/abc", // path prefix
    "auth_string": ”string”, // see below
    "retry_policy": 0      // to be detailed
}
```

The host name must be prefixed by “http://” or https://.

The DASS will use the `auth_string` directly as the argument of the “Authorization” http header tag. I.e. the `auth_string` must contain the complete argument (e.g. “Basic base64”).

The DASS will push messages to the host and port specified in the JSON object. The `path_prefix` is added in front of the /rest/… path allowing the application host interface to be implemented in a sub-path on a shared host.

The `retry_policy` is defined as:

Retry value	Description
0	No retry. Payloads that could not be delivered by push (connection to application not possible, or payloads were already in persistent temporary storage prior to starting push) are not retransmitted. The payloads will remain in the persistent temporary storage until either read out by GET or until they expire and gets purged.
1	Continuously attempt to push undelivered payloads towards the application. In this mode the DASS will continue to push messages to the application until the persistent temporary payload storage has been emptied. Only payloads that have been explicitly marked as accepted (return value 202) will stay in the persistent payload storage for pull (GET) read out.

The DASS will return one of the following values:

Status value	Meaning	Description
200	OK	The push mode registration was successful. Push mode is now started.
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
400	Bad request	The JSON object is not valid

4.5.7 Push Mode Stop

When the application no longer can receive push messages (e.g. if the application is stopped or restarted) it should deregister from the DASS and the DASS will stop sending push message.

URL: `https://dash-host[:port] /rest/pushmode/stop`
Method: PUT
Direction: application->DASS

Status value	Meaning	Description
200	OK	Push mode is stopped
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.6 Push-Mode – Push via Web Socket

The REST based on callback push mode described in the previous sections require that the application that registers for push has an public IP address can be accessed by the DASS. For applications running behind NAT or firewall this may not be possible.

To support push-mode for applications with these constraints a second push-mode, push via Web-Socket, is available. A Web Socket (RFC6455) is standard Internet protocol that is used between web-browsers and the web-servers to establish a bi-directional link to exchange data.

As it is very commonly used in “normal” Internet web pages chances are good that varies firewall and corporate access policies will allow access of this kind.

A web socket is established from the application with a connection to the server. The connection is setup with standard web-socket header (which is basically HTTP). As for all request to the DASS, the

basic authentication scheme is used, and the “Authorization” header field must be provided as described in section 2.

Once the connection has been setup, the connection will remain open until the application closes the connection again. It is up to the application to re-establish the connection if the connection fails.

Web sockets are message base meaning that the receiver side always receives a block of data at the same size as it was sent (unlike stream based connections like TCP message boundaries are not guaranteed).

Push via web socket is setup per user (as for Push with REST callback) but unlike the latter it is possible to have more that one web socket push connection per user.

Currently the web sockets are only used for pushing data from DASS to application, but later versions may support also sending request from application to DAS.

4.6.1 Setup Web Socket Connection

The application establishes the connection using a standard web socket connector.

URL: `wss://dash-host[:port] /websocket/connect`

Once established successfully the DASS will start to push messages the application. Each message is contained in JSON object.

The following messages are used. Please refer to the listed sections for the details of the message and for the details for the inner JSON message body.

`{ "payload_ul": {…} }`	see section 4.5.1
`{ "payload_dl": {…} }`	see section 4.5.2
`{ "nodeinfo": {…} }`	see section 4.5.3
`{ "status": {…} }`	see section 4.5.4
`{ "join": {…} }`	see section 4.5.5
`{ "joined": {…} }`	see tbd.

4.7 Push-Mode – Push via MQTT

[WARNING: this section may be subject to change and should be considered preliminary]

An application can connect to the DASS using the MQTT(s) protocol. To connect, the application must provide the username and password from the owning account.

URL: mqtts://dass-host[:port]
Credentials mandatory: account username/password

Once connected the application must subscribe to channels to receive the push information. The available channels are:

```
username/payload_ul
username/payload_dl
username/nodeinfo
username/status
```

```
username/join      (* not currently available
username/joined    (* not currently available
```

The first part of the channel name must be the username that was used to make the connection. Attempting to set another username will result in an authorization error on the subscription.

Once a subscription has been made the DASS will send message to the application using the above listed channels. The payloads are in the same JSON format as their REST push versions. The REST message is encapsulated in an outer object with the name of the message type. I.e.:

Channel	JSON Message	
username/payload_ul	{ "payload_ul": {…} }	see section 4.5.1
username/payload_dl	{ "payload_dl": {…} }	see section 4.5.2
username/nodeinfo	{ "nodeinfo": {…} }	see section 4.5.3
username/status	{ "join": {…} }	see section 4.5.5
username/join	{ "joined": {…} }	see tbd

NOTE, currently only message from DASS to application is supported via MQTT. Downlink and payload management commands from application to DASS will support in later version.

4.8 Customer registration

4.8.1 Register new customer

Add a new customer profile to the DASS. Customers can only be added when credentials from an customer profile with administrator rights are used.

URL: `https://dash-host[:port] /rest/customers`
Method: POST
Direction: application->DASS

```
{
    "userid": "customer-id",      // name or id of customer
    "password": "passwd",         // password
    "customer_admin": false,      // can added and delete customers
    "administrator": false,       // administrator rights attribute
    "can_register": true,         // can administer device
    "gtw_admin": false,                 // can assign gateway ownership rights
    "can_access_gtw_info": true,        // will receive info about
                                        // gateways that received messages
                                        // and is allowed to query position
                                        // of general gateways

    "can_own_gtw": true,                // can own gateways
    "can_add_gtw": false,               // can add/remove gateways
    "can_mng_gtw": false,               // can manage setting on gateway
}
```

This command can only be issued from a user or customer that has the customer administration right.

return values

Status value	Meaning	Description
200	OK	Customer was added successfully
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	Forbidden	The user/customer used does not have customer administration rights.
409		Customer ID already exists.

4.8.2 Get list of all customers

Get a list of all customers registered on the DASS. This command can only be executed from an account with customer administrator rights. Customer profiles without administrator rights will only show own profile.

URL: `https://dash-host[:port] /rest/customers`
Method: GET
Direction: application->DASS

A JSON payload is returned with the following object:

```
[{
    "userid": "customer-id",     // name or id of customer
    "is_customer": true,
    "customer_admin": true,      // can added and delete customers
    "administrator": true,       // administrator rights attribute
    "can_register": true,        // can administer device
    "gtw_admin": true,           // can assign gateway ownership rights
    "can_access_gtw_info": true,
    "can_own_gtw": true,  // can own gateways
    "can_add_gtw": true,  // can add/remove gateways
    "can_mng_gtw": true,  // can manage setting on gateway
}, ... ]
```

Note only the fields `customerid`, `is_customer`, `administrator` and `can_register` are always present in the object. The other fields are only there if the value is true. The `is_customer` field is used to signify that this account is a customer account and not a user account.

Return values

Status value	Meaning	Description
200	OK	Command was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account does not have customer administration rights

4.8.3 Delete customer

Delete a customer from the DASS. Deleting a customer will delete all users and all devices registered on the users of that customer. Only a account with customer administrator rights can delete a customer profile.

URL: `https://dash-host[:port] /rest/customers/customer-id`
Method: DELETE
Direction: application->DASS

Status value	Meaning	Description
200	OK	Customer was deleted successfully
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.8.4 Get customers info

Get information about a customer account. The id of the customer is provided directly in the URL. This API can also be used to identify the customer that a user belongs to.

URL: `https://dash-host[:port] /rest/customers/{customer- or user-id}`
Method: GET
Direction: application->DASS

A JSON payload is returned with the following object:

```
{
    "userid": "customer-id",     // name or id of customer
    "is_customer": true,
    "customer_admin": true,      // can added and delete customers
    "administrator": true,       // administrator rights attribute
    "can_register": true,        // can administer device
    "gtw_admin": true,           // can assign gateway ownership rights
    "can_access_gtw_info": true,
    "can_own_gtw": true,  // can own gateways
    "can_add_gtw": true,  // can add/remove gateways
    "can_mng_gtw": true,  // can manage setting on gateway
}
```

Note only the fields `customerid`, `is_customer`, `administrator` and `can_register` are always present in the object. The other fields are only there if the value is true. The `is_customer` field is used to signify that this account is a customer account and not a user account.

Return values

Status value	Meaning	Description
200	OK	Command was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account does not have customer administration rights

4.8.5 Modify Customer Info

Modify a customer account.

URL: `https://dash-host[:port] /rest/customers/{user-id}`
Method: PUT
Direction: application->DASS

A JSON payload is returned with the following object:

```
{
    "password": "passwd",
    "customer_admin": false,
    "administrator": false,
    "can_register": true,
    "gtw_admin": false,
    "can_access_gtw_info": true,
    "can_own_gtw": true,
    "can_add_gtw": false,
    "can_mng_gtw": false
}
```

The message need only set the fields that are to be modified. E.g. if the user wish to change the password but leave all rights unchanged, the message JSON should include only the `password` field.

Return values

Status value	Meaning	Description

200	OK	Command was successful
400	Bad request	Something wrong with the JSON message
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account does not have customer administration rights

4.9 User Registration

4.9.1 Register new user

Add a new users profile to the DASS. Users can be added by users or customers with administrator rights. If the user is added by a customer account, the user will be added to the customers. If a user is created from another user, the new user will belong to the same customer as the user that create the account.

URL: `https://dash-host[:port] /rest/users`
Method: POST
Direction: application->DASS

```
{
    "userid": "user-id",          // name or id of user
    "password": "passwd",         // password
    "customer_admin": false,      // can added and delete customers
    "administrator": false,       // administrator rights attribute
    "can_register": true,         // can administer device
    "gtw_admin": false,                 // can assign gateway ownership rights
    "can_access_gtw_info": true,        // will receive info about
                                        // gateways that received messages
                                        // and is allowed to query position
                                        // of general gateways

    "can_own_gtw": true,                // can own gateways
    "can_add_gtw": false,               // can add/remove gateways
    "can_mng_gtw": false,               // can manage setting on gateway
}
```

Return values:

Status value	Meaning	Description
200	OK	User was added successfully
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	
409		User ID already exists.

4.9.2 Get list of all users

Get a list of all users registered on the customer profile. This command can only be executed from the customer profile or from a user profile with administrator rights.

URL: `https://dash-host[:port] /rest/users`
Method: GET
Direction: application->DASS

A JSON payload is returned with the following object:

```
[{
    "userid": "user-id",   // name or id of user
```

```
    "customer_admin": true,      // can added and delete customers
    "administrator": true,       // administrator rights attribute
    "can_register": true,        // can administer device
    "gtw_admin": true,           // can assign gateway ownership rights
    "can_access_gtw_info": true,
    "can_own_gtw": true,  // can own gateways
    "can_add_gtw": true,  // can add/remove gateways
    "can_mng_gtw": true,  // can manage setting on gateway
    "associate": true            // can have devices
}, ... ]
```

Note only the fields `customerid`, `administrator` and `can_register` are always present in the object. The other fields are only there if the value is true.

Return values

Status value	Meaning	Description
200	OK	Command was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.9.3 Delete user

Delete a user from a customer profile. Deleting a user will delete all devices associated with the user. Only the owning customer profile or another user with administrator rights can delete a user.

URL: `https://dash-host[:port] /rest/user/user-id`
Method: DELETE
Direction: application->DASS

Status value	Meaning	Description
200	OK	User was added successfully
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.9.4 Get User Info

Get information about a user account. The id of the user is provided directly in the URL. A user can always get info about it own account, and users with administer rights can query info about all users that belong to the same customer. Accounts with customer administrator rights can query any user.

URL: `https://dash-host[:port] /rest/users/{user-id}`
Method: GET
Direction: application->DASS

A JSON payload is returned with the following object:

```
{
    "userid": "customer-id",     // name or id of customer
    "customer_admin": true,      // can added and delete customers
    "administrator": true,       // administrator rights attribute
    "can_register": true,        // can administer device
    "gtw_admin": true,           // can assign gateway ownership rights
    "can_access_gtw_info": true,
    "can_own_gtw": true,  // can own gateways
    "can_add_gtw": true,  // can add/remove gateways
    "can_mng_gtw": true,  // can manage setting on gateway
}
```

Note only the fields `customerid`, `administrator` and `can_register` are always present in the object. The other fields are only there if the value is true. The `is_customer` field is used to signify that this account is a customer account and not a user account.

Return values

Status value	Meaning	Description
200	OK	Command was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account does not have customer administration rights

4.9.5 Modify User Info

Modify a user account.

URL: `https://dash-host[:port] /rest/users/{user-id}`
Method: PUT
Direction: application->DASS

A JSON payload is returned with the following object:

```
{
    "password": "passwd",
    "customer_admin": false,
    "administrator": false,
    "can_register": true,
    "gtw_admin": false,
    "can_access_gtw_info": true,
    "can_own_gtw": true,
    "can_add_gtw": false,
    "can_mng_gtw": false
}
```

The message need only set the fields that are to be modified. E.g. if the user wish to change the password but leave all rights unchanged, the message JSON should include only the `password` field.

Return values

Status value	Meaning	Description
200	OK	Command was successful
400	Bad request	Something wrong with the JSON message
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account does not have administration rights to modify target user

4.10 Gateway API

4.10.1 Get List with Information about Gateways

Get list with information about all gateways associated to account

URL: `https://dash-host[:port] /rest/gateways`
Method: GET
Direction: application->DASS

A JSON payload with the following content is returned :

```
[
    {
        "id": "0000000008050313",
        "name": "313 Labo",
        "address": {},
        "latitude": 46.21025,
        "longitude": 2.922363,
        "antenna_gain": 3,
        "position_valid": true,
        "status": "OK",
        "backhaul_type": "Wired",
        "backhaul_cell_operator": "",
        "backhaul_cell_rssi": "N/A"
    },
    …
]
```

Return values

Status value	Meaning	Description
200	OK	Command was successful
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

4.10.2 Get Information about Gateways

Get information about gateway. If the gateway is associated to the account, the full list of information is returned. If the account has `can_access_gtw_info` right, this command can be used to query any gateway in the network, however, only the location (latitude / longitude) information is provided.

URL: `https://dash-host[:port] /rest/gateways/{gateway-id}`
Method: GET
Direction: application->DASS

The following JSON message is returned for devices that is associated to the account:

```
{
    "id": "0000000008050313",
    "name": "313 Labo",
    "address": {},
    "latitude": 46.21025,
    "longitude": 2.922363,
    "antenna_gain": 3,
    "position_valid": true,
    "status": "OK",
    "backhaul_type": "Wired",
    "backhaul_cell_operator": "",
    "backhaul_cell_rssi": "N/A"
}
```

For gateways that are not associated, but for accounts with `can_access_gtw_info` rights, the following is returned:

```
{
```

```
        "id": "0000000008050313",
        "latitude": 46.21025,
        "longitude": 2.922363
    }
```

Return values

Status value	Meaning	Description
200	OK	Command was successful
400	Bad request	Something wrong with the JSON message
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
404	Unknown gateway	

4.10.3 Associate gateways with user / customer

Associate a gateway to an account. Once associated status of this gateway can be queried from the account.

URL: `https://dash-host[:port] /rest/gateways/{gateway-id}`
Method: POST
Direction: application->DASS

This command take an optional JSON body with properties of the association. Currently no properties are defined but will be added in later releases.

```
{
}
```

Return values

Status value	Meaning	Description
200	OK	Command was successful
400	Bad request	Something wrong with the JSON message
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.
403	forbitten	Account doesn't have rights to associate the gateway

4.10.4 Un-associate gateways

Remove (un-associate) a gateway to an account.

URL: `https://dash-host[:port] /rest/gateways/{gateway-id}`
Method: DELETE
Direction: application->DASS

Return values

Status value	Meaning	Description
200	OK	Command was successful
400	Bad request	Something wrong with the JSON message
401	Unauthorized	The username/password used in the basic authentication scheme is invalid or not present.

403	forbitten	Account doesn’t have rights to associate the gateway
404	Unknown gateway	

/end