DAPNET 2.0 Concept and Interface Definition

Thomas Gatzweiler, DL2IC
Philipp Thiel, DL6PT
Marvin Menzerath
Ralf Wilke, DH3WR

Last Change: August 6, 2018

Abstract

This is the concept and interface description of the version 2 of the DAPNET. It's purpose in comparison to the first version released is a more robust clustering and network interaction solution to cope with the special requirements of IP connections over HAMNET which means that all network connections have to be considered with a WAN character resulting in unreliable network connectivity. In terms of consistence of the database, "eventually consistence" is considered to be the most reachable. There are "always right" database nodes inside the so called HAMCLOUD. In case of database conflicts, the version inside the HAMCLOUD cluster is always to be considered right.

Contents

Chapter 1

Introduction

more text

1.1 Key Features

The version 2 will please the user/operator with the following key features:

- Reliable clustering of Node instances over unreliable WAN connections like HAMNET
- Transmitter telemetry realtime display on Website and App with Websockets
- Use of microservices instead of one big application. Easier to develop, maintain and update
- Load-Balancing and fail-over for user and transmitter interfaces
- Real-time on-air display of transmitters on map
- Third-Party API for Brandmeister, APRS, etc.
- Priority Queuing for transmitters
- Send calls to individual transmitters and/or transmitter groups
- Inflexible concept of transmitter groups is replaced by group tags on transmitters
- Improved Cluster status monitoring

1.2 Historic Background

write some history

1.3 Concept presentation

An overview of the DAPNET 2.0 concept is given in Fig. 1.1.

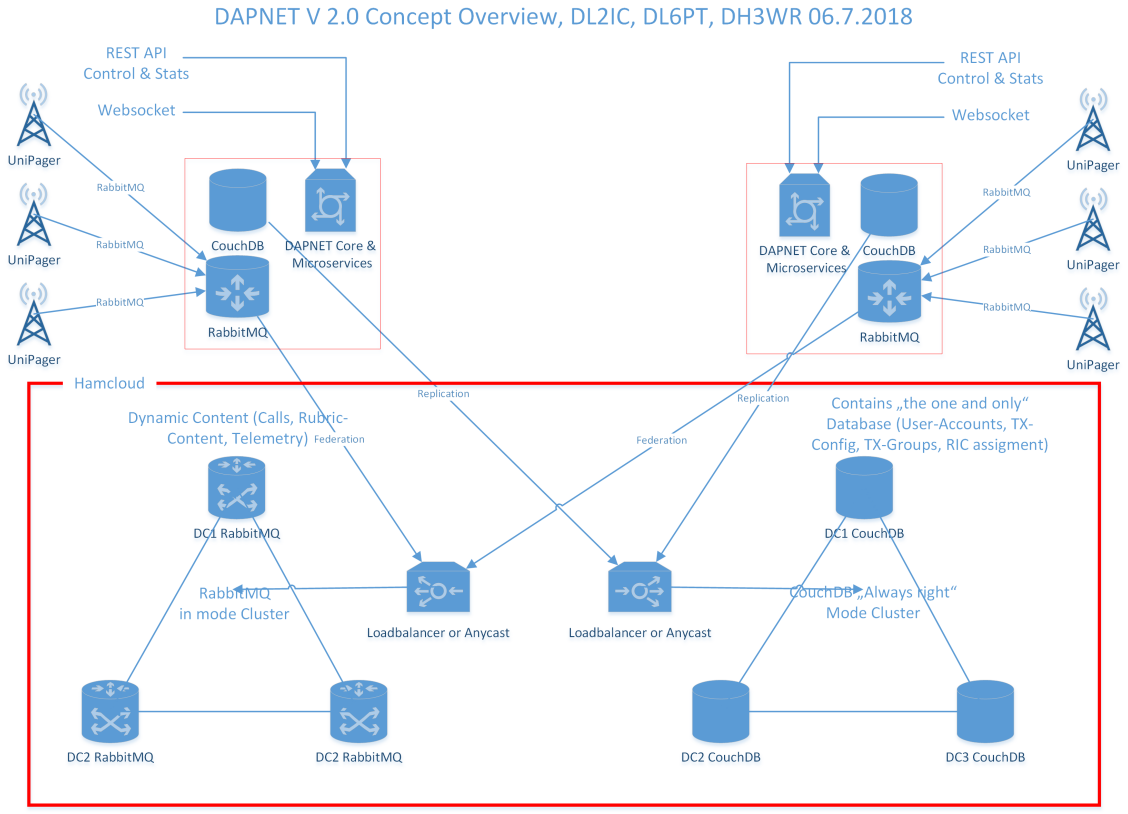

Figure 1.1: Overview of DAPNET Clutering and Network Structure

The details of a single node implementation are shown in Fig. 1.2.

1.4 Transmitter Software

1.4.1 Unipager

The default software for new transmitters is Unipager. It is developed and maintained by Thomas Gatzweiler, DL2IC. There is a debian based repository available. CI technology is used to assure automated compiling of new versions. Transmitters can be updated all at once if they subscribe to the SaltStack remote management program.

1.4.2 Backward compatibility to XOS slave protocol

The former amateur radio paging transmitters use the XOS slave protocol. It is defined here. There is a NTP like time sync sequence at the beginning of each connection establishment to assure the synchronicity of the transmitters for TDMA. In times of packet-radio, this approach was necessary. Nowadays NTP is used to sync the transmitter clocks; anyway it's still supported.
As DAPNET V2 introduces RabbitMQ and REST interfaces towards transmitters, there is a need for a backward compatibility module, which is also part of the DAPNET V2 package. We hope that after some month, all IP based transmitters have switched to the new interface implementation.

1.4.3 DAPNET-Proxy for AX.25 transmitters

For AX.25 only transmitters like PR430, there is still a demand to support thee XOS slave protocol over plain AX.25. There is a already working solution to pipe the TCP Data through a lot of intermediate programs towards a AX.25 device. The general data flow is shown in the DAPNET DokuWiki. Figure 1.3 in shown for reference only.

1.5 DAPNET Network

1.5.1 Overview and Concept

1.5.2 Used third-party Software

Used third-party Software is:

- RabbitMQ for Message delivery to transmitters and between nodes
- CouchDB as distributed database backend working on unreliable WAN connections
- NGINX as low resource high performance load balancing server for default Interface endpoints
- Docker for easy deployment and update purposes
- SaltStack for easy distributed updates and maintenance

1.5.3 HAMCLOUD Description

The HAMCLOUD is a virtual server combination of server central services on the HAMNET and provide short hop connectivity to deployed service on HAMNET towards the Internet. There are three data centers at Essen, Nürnberg and Aachen, which have high bandwidth interlinks over the DFN. There are address spaces for uni- and anycast services. How this concept is deployed is still tbd.

More information is here and here.

Define if uni- or anycast entry points will exist

Define if all 3 ham-cloud sites will have the same in-ternet incoming ports, and what is the Inter-net DNS concept

1.5.4 Transmitter Group Handling Concept

In the first Version of DAPNET, transmitters had to be member of one or more logical transmitter groups. Personal calls and rubric content could only be send to a transmitter group, which afterwards sent the data to be member transmitters. Changes in membership required the assigned owner of the group to do so.
In DAPNET V2, there will by just *virtual* transmitter groups by assigning one of more *tags* to a transmitter by its owner himself. Messages can be sent to either a single or group of individual transmitters and/or a single or group of tags. Each transmitter containing the tag will send out the message.

1.5.5 Rubric Handling Concept

1.5.6 Queuing Priority Concept

A main drawback of the original DAPNET implementation was the lack of priorities in the message queuing on a transmitter. With increasing popularity the load on the transmitters increased and the FIFO working principle led to personal calls being sent out several 10 minutes later than submitted.
To overcome this, a 5 class priority scheme is implemented in DAPNET. Messages to send out are queued

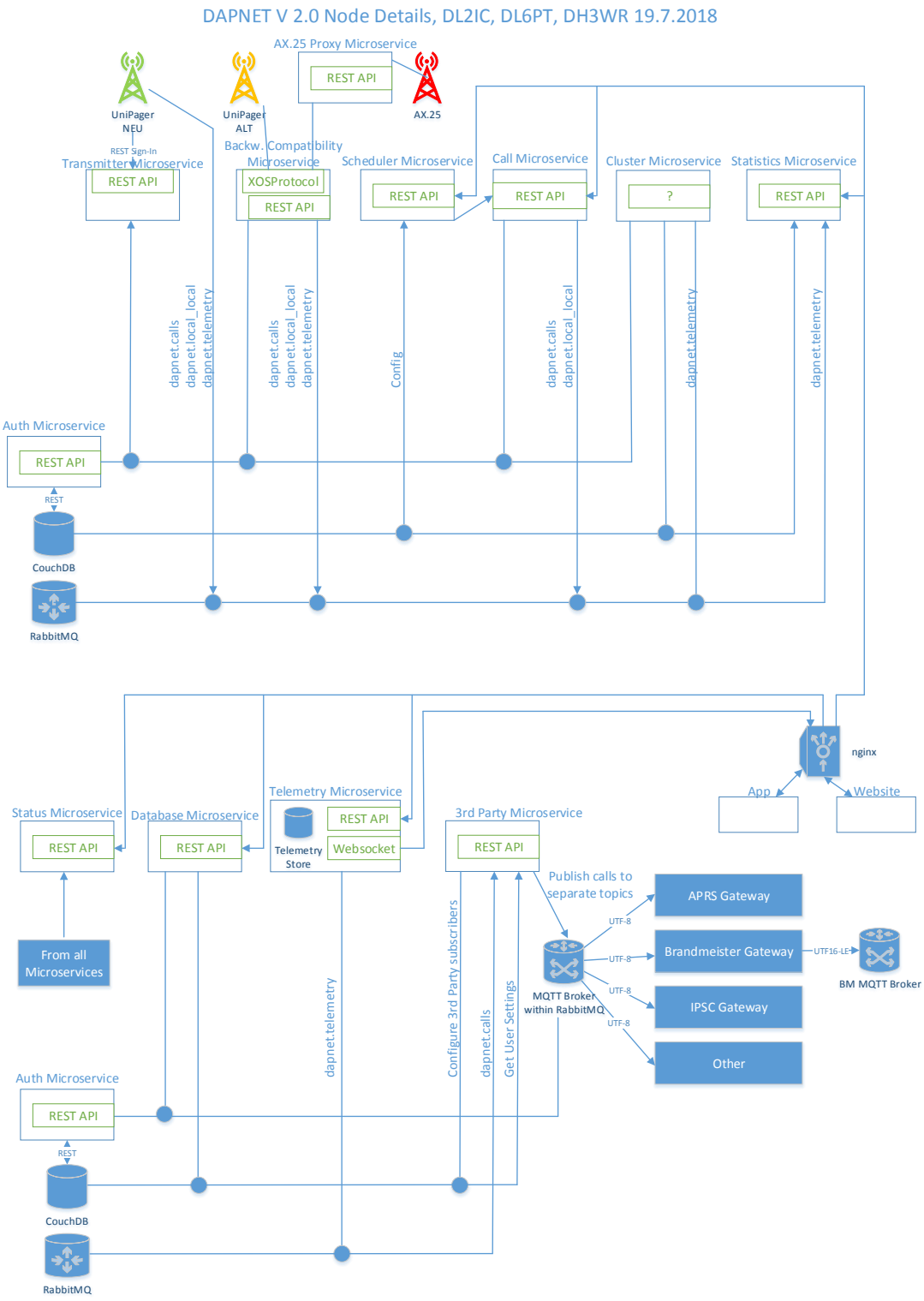

Figure 1.2: Node Details

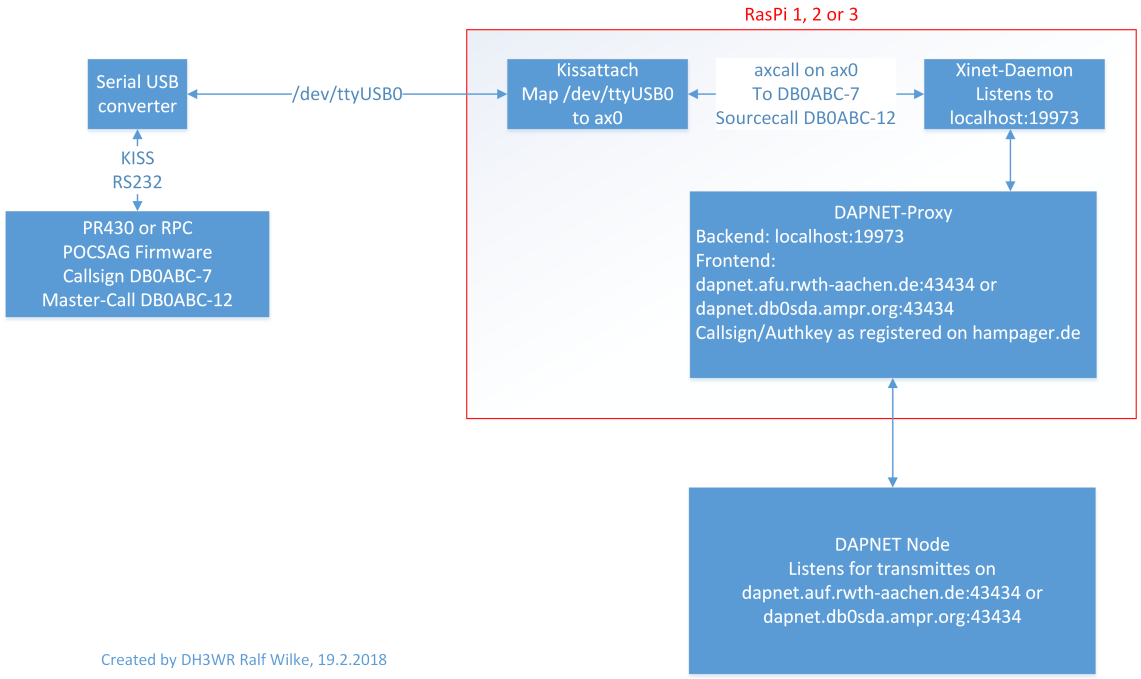

Figure 1.3: Data flow for AX.25 connections from DAPNET

Chapter 2

DAPNET Network Definition

2.1 Cluster Description

2.1.1 Real-time Message delivery with RabbitMQ

2.1.2 Distributed Database with CouchDB

2.1.3 Authentication Concept

2.1.4 Integration of new Nodes

2.2 Interface Overview and Purpose

2.2.1 RabbitMQ Exchange

There are 3 exchanges on each RabbitMQ instance available:

dapnet.calls Messages that are distributed to all nodes

dapnet.local_calls Messages coming only from the local node instance

dapnet.telemetry Messages containing telemetry from transmitters

Transmitters publish their telemetry data to the **dapnet.telemetry** exchange, while consume the data to be transmitted from a queue that is bound to the **dapnet.calls** and **dapnet.local_calls** exchanges.

The idea is to distinguish between *local* data coming from the local Core instance and data coming from the DAPNET network. This is necessary, as for example the calls to set the time on the pagers are generated by the local Core and not shall not distributed to other Cores and their connected transmitter to avoid duplicates.

dapnet.calls

This federated exchange receives calls from all Core instances. Personal calls are always published to this exchange, as they are unique and only published by the Core that receives the call via the Core REST API. Rubric content is also emitted here. The transmitter to receive the call is defined via the routing key.

dapnet.local_calls

The the local Core publishes special calls to this exchange, like the time set calls, the rubric names and repetitions of rubric content for the local connected transmitters.

In short, all calls that are generated by the Scheduler on a Core instance are published to this exchange. As the scheduler runs on every node, otherwise the calls would be transmitted several times by the same transmitter. This exchange is not federated with other RabbitMQ instances on other Cores.

dapnet.telemetry

On each Core instance, the Statistic, Status and Telemetry microservice described in section 2.2.4 is consuming the telemetry of all transmitters. The received data is stored and delivered via the Core REST API and the Websocket API in section 2.2.5 to connected websites or apps.

2.2.2 CouchDB Interface

The CouchDB interface is a REST interface defined in the CouchDB documentation. All communication with the CouchDB database are done by means of the interface. No user should be able to connect to the CouchDB REST interface, only the Core software components should be able to do so. The local node can access CouchDB with randomly created credentials which are automatically generated on the first startup of the node. For database replication, the other nodes are authenticated by their authentication key in the nodes database.

2.2.3 Core REST API

The Core REST API is the main interface for user interactions with the DAPNET network.

2.2.4 Statistic, Status and Telemetry REST API

2.2.5 Websocket for real-time updates on configuration, Statistics and Telemetry API

2.2.6 MQTT Fanout for third-party consumers

In order to allow third-party application to consume the data sent out by DAPNET transmitters in an easy and most generic way, there is an MQTT brocker on each Core. As the RabbitMQ instance already has a plugin to act as an MQTT broker, this solution is chosen.

To dynamically manage the third-party applications attached to DAPNET, there is a CouchDB-Database containing the existing third-party descriptive names, corresponding MQTT topic names and authentication credentials to be allowed to subscribe to the that specific MQTT topic.

It is a intention to not fan out every content on DAPNET to every third-party application but let the user decide if personal calls directed to her/him will be available on other third-party applications or not. The website will display opt-in checkboxes for each subscriber to enable or disable the message delivery for each third-party application. As we have had some issues in this topic in the past, this seems the best but still generic and dynamic solution.

The fan out consists of the source and destination callsign, the destination RIC and SubRIC and an array of callsign and geographic location of the transmitters, where this specific call is supposed to be sent out by DAPNET transmitters. The type of transmitter is also given. The reason to output also the transmitter and their location is to enable third-party applications to estimate the content's distribution geographic area and take adequate action for their own delivery or further processing. (Example: Regional Rubric content to Regional DMR Group SMS.)

The third-party applications can (if access is granted) only read from the topic. All Core instaces have read/write access to publish the data.

The MQTT topics are kept local on the Core instance and are never distributed between DAPNET-Cores.

2.3 Other Definitions

2.3.1 Scheduler

2.3.2 User Roles and Permissions

There are two types of users: Admins and Non-Admins. Admins are allowed to do everything. Non-Admins are just allowed to edit the entities that they own and send calls.

Make overview of data displayed to Non-Admin users from CouchDB in REST-Calls (see 6.1.

Chapter 3

Internal Programming Workflows

3.1 Sent calls

3.2 Add, edit, delete User

Show current users

1. Get current status via `GET /users` on Core URL
2. Handle updates via Websocket

Add and Edit User

1. If edit: Get current status via `GET /users/<username>` on Core URL
2. Show edit form and place data
3. On save button event, send `POST /users/<username>` on Core URL

The core will update the CouchDB and generate a RabbitMQ administration message to inform all other nodes. This information is transmitted by the Stats and Websocket Micro-Service to all connected websocket clients to get them updated. This will also happen for the website instance emitting the edit request, so its content is also updated.

Delete User

1. Ask "Are you sure?"
2. If yes, send `DELETE /users/<username>` on Core URL

The core will update the CouchDB and generate a RabbitMQ administration message to inform all other nodes. This information is transmitted by the Stats and Websocket Micro-Service to all connected websocket clients to get them updated. This will also happen for the website instance emitting the edit request, so its content is also updated.

3.3 Add, edit, delete Subscriber

3.4 Add, edit, delete Node (tbd)

3.5 Add, edit, delete Transmitter

3.6 Implementation of Transmitter Groups

3.7 Add, edit, delete Rubrics

Show current configuration

1. Get current status via `GET /rubrics` on Core URL
2. Handle updates via websocket

Add and Edit rubrics

1. If edit: Get current status via `GET /rubrics/<rubricname>` on Core URL
2. Show edit form and place data
3. On save button event, send `POST /users/<rubricname>` on Core URL

The core will update the CouchDB and generate a RabbitMQ administration message to inform all other nodes. This information is transmitted by the Stats and Websocket Micro-Service to all connected websocket clients to get them updated. This will also happen for the website instance emitting the edit request, so its content is also updated.

Delete rubric

1. Ask "Are you sure?"
2. If yes, send `DELETE /users/<rubricname>` on Core URL

The core will update the CouchDB and generate a RabbitMQ administration message to inform all other nodes. This information is transmitted by the Stats and Websocket Micro-Service to all connected websocket clients to get them updated. This will also happen for the website instance emitting the edit request, so its content is also updated.

3.8 Add, edit, delete Rubrics content

3.9 Add, edit, delete, assign Rubrics to Transmitter/-Groups

3.10 Docker integration

DL2IC: Docker Integration beschreiben

3.11 Microservices

A DAPNET node consists of serveral isolated microservices with different responsibilities. Each microservice runs in a container and is automatically restarted if it should crash. Some microservices can be started in multiple instances to fully utilize multiple cores. The access to the microservices is proxied by a NGINX webserver which can also provide load balancing and caching.

REST endpoint		Microservice
*	/users/*	Database Service
*	/nodes/*	
*	/rubrics/*	
*	/subscribers/*	
*	/subscriber_group(s)/*	
GET	/transmitter/grouptags	
DELETE	/transmitter/<transmittername>	
PUT	/transmitter/<transmittername>	
*	/calls/*	Call Service
*	/rubrics/content/*	Rubric Service
GET	/transmitters	Transmitter Service
GET	/transmitters/:id	
POST	/transmitters/bootstrap	
POST	/transmitters/heartbeat	
POST	/cluster/discovery	Cluster Service
GET	/telemetry/*	Telemetry Service
WS	/telemetry/transmitters	Summary data of all TX
WS	/telemetry/transmitter/<TxName>	Details for TX <TxName>
WS	/telemetry/nodes	Summary data of all nodes
WS	/telemetry/node/<NodeName>	Details for Node <NodeName>
WS	/changes	Database Changes Service
GET	/status/*	Status Service
GET	/statistics	Statistics Service
GET	/rabbitmq/*	RabbitMQ Auth Service

3.11.1 Database Service

- Proxies calls to the CouchDB database
- Controls access to different database actions
- Removes private/admin only fields from documents

3.11.2 Call Service

- Generates and publishes calls to RabbitMQ
- Receives all calls from RabbitMQ
- Maintains a database of all calls

3.11.3 Rubric Service

- Publishes rubric content as calls to RabbitMQ
- Periodically publishes rubric names as calls to RabbitMQ

3.11.4 Transmitter Service

- Maintains a list of all transmitters and their current status

3.11.5 Cluster Service

- Maintains a list of known nodes and their current status
- Manages federation between RabbitMQ queues
- Manages replication between CouchDB databases

3.11.6 Telemetry Service

- Maintains the telemetry state of all transmitters
- Forwards telemetry updates via websocket

3.11.7 Database Changes Service

- Forwards database changes via websocket

3.11.8 Status Service

- Periodically checks all other services and connections

3.11.9 RabbitMQ Auth Service

- Provides authentication for RabbitMQ against the CouchDB users database

3.11.10 Time and Identification Service

- Sends periodic time and identification messages to RabbitMQ

3.12 Ports and Loadbalacing Concept

3.13 Periodic Tasks (Scheduler)

3.14 Plugin Interface

3.15 Transmitter Connection

Transmitter connections consist of two connections to a Node. A REST connection for initial announcement of a new transmitter, heartbeat messages and transmitter configuration and a RabbitMQ connection to receive the data to be transmitted.

The workflow for a transmitter connection is the following:

1. Announce new connecting transmitter via Core REST Interface (6.1.5).
2. Get as response the transmitter configuration or an error message (6.1.5).
3. Initiate RabbitMQ connection to get the data to be transmitted (6.2.1).

The authentication of the transmitter's REST calls consist of the transmitter name and its AuthKey, which is checked against the value in the CouchDB for this transmitter.

3.16 Transmitter connections

If a transmitter wants to connect to DAPNET, the first step is to sign-in and show its presence via the Core REST interface. This interface is also used for transmitter configuration like enabled timeslots and keep-alive polling.

3.16.1 Authentication of all HTTP-Requests in this context

All HTTP-requests issued from a transmitter have to send a valid HTTP authentication, which is checked against the CouchDB. It consists of the transmitter name and its AuthKey.

3.17 DAPNET-Proxy

Da es sich bei den Anfragen um POST-Requests mit JSON Body handelt, wäre es einfacher da den AuthKey mit dazu zu packen, so wie es auch schon in der Protokoll-Definition umgesetzt ist.

Chapter 4

External Usage Workflows

4.1 General Concept of REST and Websocket-Updates

4.2 Website and App

4.2.1 Authentication

4.2.2 Calls

4.2.3 Rubrics

4.2.4 Rubrics content

4.2.5 Transmitters and Telemetry

4.2.6 Nodes

4.2.7 Users

4.2.8 MQTT consumers

4.2.9 Scripts and automated Software for DAPNET-Input

Chapter 5

Setup and Installation

5.0.1 Accessible ports from HAMNET

5.1 Unipager

5.2 DAPNET-Proxy

5.3 DAPNET Core

5.4 Special issues for Core running in Hamcloud

5.4.1 Accessible ports from internet

To offer the endpoints to internet-based transmitters and users, the following port have to be accessible:

Type	Port	Application
TCP	80	HTTP Webinterface and Websocket
TCP	443	HTTPS Webinterface and Websocket
TCP	5672	RabbitMQ Client connection

5.4.2 Load balancing and high availability

Internet-based
To offer load balancing and high availability, the internet-based DNS record *hampager.de* would use DNS round-robin with the static internet IPs of the Hamcloud instances.

HAMNET/Hamcloud-based
The Hamcloud instances would offer an anycast IP to for transmitter and user connections. There is a special subnet of 44.0.0.0/8 IPs designated for this anycast approach. Besides, the Hamcloud DAPNET instances will have unicast IPs for administration and their inter-node-synchronization. To connect other nodes besides from the three hamcloud instances, the endpoint to be attached will also be distributed via anycast for maximal fail-over capability.

rework with content of discussion from 2.8.2018 on network structure

Chapter 6

Protocol Definitions

6.1 Microservices API

6.1.1 Preamble

All HTTP(s) communication should be compress with gzip to reduce network load. That's especially important for the answers to GET-calls of all entity's details.

See Microservices definition.

6.1.2 Database Service

GET /users

Returns all users with all details in JSON format.

Mapping to CouchDB:
`GET /users/_all_docs?include_docs=true` Filter couchDB output to produce just the output below:

Role **admin** or **support** example result:


```
{
  "total_rows": 2,
  "offset": 0,
  "rows": [
    {
      "_id": "dh3wr",
      "_rev": "1-09352254509c9ddf86e80fd83868d557",
      "email": "ralf@secret.com",
      "role": "user",
      "enabled": true,
      "created_on": "2018-07-08T11:50:02.168325Z",
      "created_by": "dl2ic"
    },
    {
      "_id": "dl2ic",
      "_rev": "1-c0a6ecb1a60b58254e808fc68d61ec00",
      "email": "mail@secret.de",
      "role": "admin",
      "enabled": true,
      "created_on": "2018-07-08T11:50:02.168325Z",
      "created_by": "dh3wr"
    }
  ]
}
```


Role **user** example result: `403 Forbidden`

GET /users?startkey="dh3wr"&endkey="dl2ic"

Just for role **admin** or **support**. Role **user** will get`403 Forbidden`.

Return all details of all users in alphabetical order between dh3wr and dl2ic in JSON format.
Mapping to CouchDB:
`GET /users/_all_docs?startkey="dh3wr"&endkey="dl2ic"&include_docs=true`

Output filtering is the same as for single user request regarding output content and requestor's role

GET /users?limit=<n>&skip=<m>

Skip the first <n> entries and output just <m> entries. Used for pagination. Can be combined with the filtering from section 6.1.2.

The `limit` and `skip` parameters are passed as they to the CouchDB REST API.

GET /users/<username>

Return details of <username> in JSON format.
Mapping to CouchDB :
`GET /users/<username>`

Role **user** will get `403 Forbidden`, if not asking for her/himself.

Role **user** example result asking for her/himself or role **admin** or **support**:

```
{
  "_id": "dh3wr",
  "_rev": "1-09352254509c9ddf86e80fd83868d557",
  "email": "ralf@secret.com",
  "role": "user",
  "enabled": true,
  "created_on": "2018-07-08T11:50:02.168325Z",
  "created_by": "dl2ic"
}
```

GET /users/_usernames

Return just an JSON array of all usernames. Used where selections have to be done on the website.
Mapping to CouchDB with filtering in microservice:
`GET /users/_all_docs?include_docs=false`

All roles example result:

```
{
  ["dh3wr","dl2ic"]
}
```

PUT /users - Add new user

Add the user <username>.

Role **admin** or **support** or editing the requestor's own entry are the only allowed roles.

User to API: Example content to add user dl6pt

```
{
  "_id" : "dl6pt",
  "password": "$2y$12$lqUueRVo94f439Tt7zqrZOHPfm6YoBzNawWLLIykF3nMip3L6mxLK",
  "email": "ralf@secret.com",
  "role": "admin",
  "enabled": true,
}
```

Mapping to CouchDB with adding information by microservice:
`PUT /users/<username>`

```
{
  "_id" : "dl6pt",
  "password": "$2y$12$lqUueRVo94f439Tt7zqrZOHPfm6YoBzNawWLLIykF3nMip3L6mxLK",
  "email": "ralf@secret.com",
  "admin": "admin",
  "enabled": true,
  "created_on": "2018-07-08T11:50:02.168325Z",
  "created_by": "dl2ic"
}
```

The `created_by` and `created_on` content has to be added by the microservice.

PUT /users - Edit existing user

Edit the existing user <username>. Just the changed values have to be sent. The `_id` and `_rev` must be sent always.

Role **admin** or **support** or editing the requestor's own entry are the only allowed roles. Other requests result in returning `403 Forbidden`.

User to API: Example content to edit user:
First get the user's revision as in 6.1.2.

Then generate `PUT` request with content:

```
{
  "_id" : "dl6pt",
  "_rev": "1-09352254509c9ddf86e80fd83868d557",
  "password": "$2y$12$lqUueRVo94f439Tt7zqrZOHPfm6YoBzNawWLLIykF3nMip3L6mxLK",
  "email": "ralf@secret.com",
  "role": "support",
  "enabled": true,
}
```

Mapping to CouchDB with adding information by microservice:
`PUT /users/<username>`

```
{
  "_id" : "dl6pt",
  "_rev": "1-09352254509c9ddf86e80fd83868d557",
  "password": "$2y$12$lqUueRVo94f439Tt7zqrZOHPfm6YoBzNawWLLIykF3nMip3L6mxLK",
  "email": "ralf@secret.com",
  "role": "support",
  "enabled": true,
  "created_on": "2018-07-08T11:50:02.168325Z",
  "created_by": "dl2ic"
}
```

DELETE /users/<username>?rev=

Delete user <username>. The Database Service has be make sure that all dependencies of a user account are deleted as well, for example transmitters subscribers or rubrics, that contain **just** this <username> as owner as the only one entry (left).

Role **admin** or **support** or deleting the requestor's own entry are the only allowed roles. Others get as return message `403 Forbidden`.

First get the user's revision as in 6.1.2.
User to API: `DELETE /users/<username>?rev=1-09352254509c9ddf86e80fd83868d557`

Mapping to CouchDB: direct forward of request

GET /nodes

Returns all nodes with all details in JSON format.

Mapping to CouchDB:
`GET /nodes/_all_docs?include_docs=true` Filter couchDB output to produce just the output below:

Role **admin** or **support** example result:

```
{
  "total_rows": 2,
  "offset": 0,
  "rows": [
    {
      "_id": "db0sda-dc1",
      "_rev": "1-cf7d2abfe193f476888be7108a0f548f",
      "auth_key": "8PL9eJXccQ6X9Yq"
      "coordinates": [34.123456, -23.123456],
      "description": "some words about that node",
      "hamcloud": true,
      "created_on": "2018-07-03T08:00:52.786458Z",
      "created_by": "dh3wr",
      "changed_on": "2018-07-03T08:00:52.786458Z",
      "changed_by": "dh3wr",
      "owners": ["dl1abc","dh3wr","dl2ic"],
      "avatar_picture": <couchdb attachment??>
    },
    {
      "_id": "db0sda-dc2",
      "_rev": "1-ee070b17db9c3c58658d10fdedad2f48",
      "auth_key": "73mxX4JLttzmVZ2"
      "coordinates": [34.123456, -23.123456],
      "description": "some words about that node",
      "hamcloud": true,
      "created_on": "2018-07-03T08:00:52.786458Z",
      "created_by":"dh3wr",
      "changed_on": "2018-07-03T08:00:52.786458Z",
      "changed_by":"dh3wr",
      "owners": ["dl1abc","dh3wr","dl2ic"],
      "avatar_picture": <couchdb attachment??>
    }
  ]
}
```

Role **user** example result. If the user is one of the owners of a node, display also the detail-Information, like in the second array entry. Here dh3wr is requesting.

```
{
  "total_rows": 2,
  "offset": 0,
  "rows": [

    {
      "_id": "db0sda-dc1",
      "coordinates": [34.123456, -23.123456],
      "description": "some words about that node",
      "hamcloud": true,
      "created_on": "2018-07-03T08:00:52.786458Z",
      "created_by":"dl2ic",
      "changed_on": "2018-07-03T08:00:52.786458Z",
      "changed_by":"dl2ic",
      "owners": ["dl1abc","dl2ic"],
      "avatar_picture": <couchdb attachment??>
    },
    {
      "_id": "db0sda-dc2",
      "_rev": "1-ee070b17db9c3c58658d10fdedad2f48",
      "auth_key": "73mxX4JLttzmVZ2"
      "coordinates": [34.123456, -23.123456],
      "description": "some words about that node",
      "hamcloud": true,
      "created_on": "2018-07-03T08:00:52.786458Z",
```

```
      "created_by":"dh3wr",
      "changed_on": "2018-07-03T08:00:52.786458Z",
      "changed_by":"dh3wr",
      "owners": ["dl1abc","dh3wr","dl2ic"],
      "avatar_picture": <couchdb attachment??>
    }
  ]
}
```

GET /nodes?startkey="db0sda-dc1"&endkey="db0sda-dc3"

Just for role **admin** or **support**. Role **user** will get`403 Forbidden`.

Return all details of all users in alphabetical order between db0sda-dc1 and db0sda-dc3 in JSON format.
Mapping to CouchDB:
`GET /nodes/_all_docs?startkey="db0sda-dc1"&endkey="db0sda-dc2"&include_docs=true`

Output filtering is the same as (for single user request) regarding output content and requestor's role

GET /users?limit=<n>&skip=<m>

Skip the first <n> entries and output just <m> entries. Used for pagination. Can be combined with the filtering from section 6.1.2.

The `limit` and `skip` parameters are passed as they to the CouchDB REST API.

GET /nodes/<nodename>

Return details of <nodename> in JSON format.
Mapping to CouchDB :
`GET /users/<nodename>`

Role **user** example result if not owner:

```
{
  "_id": "db0sda-dc2",
  "coordinates": [34.123456, -23.123456],
  "description": "some words about that node",
  "hamcloud": true,
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by":"dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by":"dh3wr",
  "owners": ["dl1abc","dh3wr","dl2ic"],
  "avatar_picture": <couchdb attachment??>
}
```

Role **user** example result if owner of node or role **admin** or **support**:

```
{
  "_id": "db0sda-dc2",
  "_rev": "1-ee070b17db9c3c58658d10fdedad2f48",
  "auth_key": "73mxX4JLttzmVZ2"
  "coordinates": [34.123456, -23.123456],
  "description": "some words about that node",
  "hamcloud": true,
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by":"dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by":"dh3wr",
  "owners": ["dl1abc","dh3wr","dl2ic"],
  "avatar_picture": <couchdb attachment??>
}
```

GET /nodes/_nodenames

Return just an JSON array of all nodenames. Used where selections have to be done on the website. For all roles example result:

```
{
  ["db0sda-dc1", "db0sda-dc2"]
}
```

GET /nodes/_nodenamedescription

Return just an JSON array of all nodenames and their description. Used where selections have to be done on the website. For all roles example result:

```
{
  [
    {
      "_id:␣"db0sda-dc1",
␣␣␣␣␣␣"description":␣"some words about that node"
␣␣␣␣},
␣␣␣␣{
␣␣␣␣␣␣"_id: "db0sda-dc2",
      "description": "some words about that node"
    }
  ]
}
```

PUT /nodes - Add new node

Role **user** and **support** get `403 Forbidden`.

Only role **admin** is allowed. Example POST message to send:

```
{
  "_id": "db0sda-dc2",
  "auth_key": "73mxX4JLttzmVZ2"
  "coordinates": [34.123456, -23.123456],
  "description": "some words about that node",
  "hamcloud": true,
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by":"dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by":"dh3wr",
  "owners": ["dl1abc","dh3wr","dl2ic"],
  "avatar_picture": <couchdb attachment??>
}
```

PUT /nodes - Edit existing node

Role **user** and **support** get returned `403 Forbidden`.

First get node like in section 6.1.2 to get the revision. Then send the PUT request with just the changed values. The `_id` and `_rev` must be sent always. Only role **admin** is allowed. Example POST message to send:

```
{
  "_id": "db0sda-dc2",
  "_rev": "1-ee070b17db9c3c58658d10fdedad2f48",
  "auth_key": "73mxX4JLttzmVZ2"
  "coordinates": [34.123456, -2.123456],
  "description": "New description with changed position"
}
```

DELETE /nodes/<nodename>?rev=

Delete node <nodename>. No dependency check necessary.

Role **admin** is the only allowed role. Others get as return message `403 Forbidden`.

First get the node's revision as in 6.1.2.
User to API: `DELETE /node/<node>?rev=1-09352254509c9ddf86e80fd83868d557`

Maybe the transmitter service should inform the connected transmitters to the now deleted node to do a switch-over?

GET /rubrics

Returns all rubrics with all setting details in JSON format. This does **not** include the content of the 10 rubric message slots.

Mapping to CouchDB:
`GET /rubrics/_all_docs?include_docs=true` Filter couchDB output to produce just the output below:

Role **admin** or **support** example result:

```
{
  "total_rows": 1,
  "offset": 0,
  "rows": [
    {
      "_id": "dx-kw",
      "_rev": "1-166c3257894d0aea8ee68c1861ca508a",
      "number": 4,
      "description": "DX Cluster Spots KW",
      "label": "DX KW",
      "transmitter_groups": [
        "dl-hh"
      ],
      "transmitters": [
        "db0abc"
      ],
      "cyclic_transmit": false,
      "cyclic_transmit_interval": 0,
      "owner": [
        "dh3wr",
        "dl1abc"
      ]
    }
  ]
}
```

Role **user** example result. All details but the `_rev` are also given out to role **user**.

```
{
  "total_rows": 1,
  "offset": 0,
  "rows": [
    {
      "_id": "dx-kw",
      "number": 4,
      "description": "DX Cluster Spots KW",
      "label": "DX KW",
      "transmitter_groups": [
        "dl-hh"
      ],
      "transmitters": [
        "db0abc"
      ],
      "cyclic_transmit": false,
      "cyclic_transmit_interval": 0,
      "owner": [
        "dh3wr",
        "dl1abc"
      ]
    }
  ]
}
```

GET /rubrics/_view/byNumber?startkey=<n>&endkey=<m>

Just get the output as in section 6.1.2, but just for rubric numbers between <n> and <m>. Check before passing the request to CouchDB, that neither <n> nor <m> are higher then 95.

GET /rubrics/_view/byTransmitter?startkey="db0abc"&endkey="db0abc"

Just get the output as in section 6.1.2, but just for rubric that contain *db0abc* in the `"transmitters"` array.

GET /rubrics/_view/byTransmitterGroup?startkey="dh-hh"&endkey="dh-hh"

Just get the output as in section 6.1.2, but just for rubric that contain *dl-hh* in the `"transmitter_groups"` array.

GET /rubrics/_view/withCyclicTransmit

Just get the output as in section 6.1.2, but just for rubric that have the cyclic transmit flag enabled.

GET /rubrics/<rubricname>

Return all setting details just of <rubricname> in JSON format. This does **not** include the content of the 10 rubric message slots. Output as in section 6.1.2.

Mapping to CouchDB :
`GET /rubrics/<rubricname>`

GET /rubrics/_rubricnames

Return just an JSON array of all rubricnames in a JSON Array. Allowed with all roles. Used where selections have to be done on the website.

For all roles example result:

```
{
  ["dl-hh", "dl-nw"]
}
```

GET /rubrics/_rubricnamesdescription

Return just an JSON array of all rubricnames and their description in a JSON Array. Allowed with all roles. Used where selections have to be done on the website.

For all roles example result:

```
{
  [
    {
      "_id: "dl-hh",
      "description": "Germany, Hamburg"
    },
    {
      "_id: "dl-by",
      "description": "Bavaria"
    }
  ]
}
```

PUT /rubrics - Add new rubric

Role **user** gets `403 Forbidden`.

Only role **admin** and **support** is allowed.
Example POST message to send:

```
{
  "_id": "dx-kw",
  "number": 4,
  "description": "DX Cluster Spots KW",
  "label": "DX KW",
  "transmitter_groups": [
    "dl-hh"
  ],
  "transmitters": [
    "db0abc"
  ],
  "cyclic_transmit": false,
  "cyclic_transmit_interval": 0,
  "owner": [
    "dh3wr",
    "dl1abc"
  ]
}
```

PUT /rubrics - Edit existing rubric

This is just about the rubrics details, and does **not** include the content of the 10 rubric message slots.

Role **admin** and **support** are allowed to do changes.
Role **user** gets returned `403 Forbidden`.

First get rubric like in section 6.1.2 to get the revision. Then send the PUT request with just the changed values. The `_id` and `_rev` must be sent always.

Example POST message to send:

```
{
  "_id": "dx-kw",
  "_rev": "1-166c3257894d0aea8ee68c1861ca508a",
  "description": "DX Cluster Spots KW"
}
```

DELETE /rubrics/<rubricname>?rev=

Delete rubric <rubricname>.
Also delete content of this rubric.

Noch heraus-finden, wie das genau geht.

Role **admin** and **support** are allowed.
Role **user** gets returned `403 Forbidden`.

First get the rubric's revision as in 6.1.2.
User to API: `DELETE /rubrics/<rubricname>?rev=1-09352254509c9ddf86e80fd83868d557`

GET /subscribers

Returns all nodes with all details in JSON format.

Mapping to CouchDB:
`GET /nodes/_all_docs?include_docs=true` Filter couchDB output to produce just the output below:

Role **admin** or **support** example result:

```
{
  "total_rows": 1,
  "offset": 0,
  "rows": [
    {
      "_id": "dl1abc",
      "_rev": "1-44182aeb25815b19babe1c0a6bb95e68",
      "description": "Peter",
      "pagers": [
        {
          "ric": 123456,
          "function": 3,
          "name": "Peters Alphapoc",
          "type": "alphaPpoc",
          "enabled": true
        }
      ],
      "third_party_services": [
        "APRS",
        "BM"
      ],
      "owner": [
        "dh3wr",
        "dl1abc"
      ],
      "groups": [
        "rwth-afu"
      ]
    }
  ]
}
```


Role **user** example result. If the user is one of the owners of a the subscribere, display also the detail-Information, like in the in the second array entry. Here dh3wr is requesting.


```
{
  "total_rows": 2,
  "offset": 0,
  "rows": [
    {
      "_id": "dl1abc",
      "description": "Peter",
      "pagers": [
        {
          "ric": 123456,
          "function": 3,
          "name": "Peters Alphapoc",
          "type": "alphaPpoc",
          "enabled": true
        }
      ],
      "owner": [
        "dl1abc"
      ]
    },
    {
      "_id": "dh3wr",
      "_rev": "1-44182aeb25815b19babe1c0a6bb95e68",
      "description": "Ralf",
      "pagers": [
        {
          "ric": 123456,
          "function": 3,
          "name": "Ralfs Skyper",
          "type": "skyper",
          "enabled": false
        }
      ],
      "third_party_services": [
        "APRS",
        "BM"
      ],
      "owner": [
```

```
        "dl1abc",
        "dh3wr"
      ],
      "groups": [
       "rwth-afu"
      ]
    }
  ]
}
```

GET /subscribers/<subscribername>

Return all setting details just of <subscribername> in JSON format. **??**.

Mapping to CouchDB :
`GET /subscribers/<subscribername>`

Filter output according to role as in section 6.1.2.

GET /subscribers/_subscribernames

Return just an JSON array of all subscribers. Used where selections have to be done on the website.
Mapping to CouchDB with filtering in microservice:
`GET /subscribers/_all_docs?include_docs=false`

All roles example result:

```
{
  ["dh3wr","dl2ic"]
}
```

GET /subscribers/_subscribernamesdetails

Return just an JSON array of all subscribers and their description. Used where selections have to be done on the website.
Mapping to CouchDB with filtering in microservice:
`GET /subscribers/_all_docs?include_docs=false`

All roles example result:

```
{
  [
    {
      "_id": "dh3wr,
      "description:␣"Ralf"
␣␣␣␣},
␣␣␣␣{
␣␣␣␣␣␣"_id":␣"dl2ic,
      "description:␣"Thomas"
␣␣␣␣}
␣␣]
}
```

PUT /subscribers - Add new subscriber

Role **user** gets `403 Forbidden`

Only role **admin** and **support** is allowed.
Example POST message to send:

```
{
  "_id": "dl1abc",
  "description": "Peter",
  "pagers": [
    {
      "ric": 123456,
```

```
      "function": 3,
      "name": "Peters Alphapoc",
      "type": "alphaPpoc",
      "enabled": true
    }
  ],
  third_party_services": [
    "APRS",
    "BM"
  ],
  "owner": [
    "dh3wr",
    "dl1abc"
  ],
  "groups": []
}
```

PUT /subscribers - Edit existing subscriber

Role **admin** and **support** are allowed to do changes.
Role **user** gets returned `403 Forbidden`, if not in owner array.

First get subscriber like in section **??** to get the revision. Then send the PUT request with just the changed values. The `_id` and `_rev` must be sent always.

Example POST message to send:

```
{
  "_id": "dl1abc",
  "description": "Peter",
  "pagers": [
    {
      "ric": 65432,
      "function": 2,
      "name": "Peters Alphapoc, heute mal anders",
      "type": "alphaPpoc",
      "enabled": true
    }
  ]
}
```

DELETE /subscribers/<subscribername>?rev=

Delete subscriber <subscribername>. If must be also deleted from any subscriber_group that is containing it. If it is the only one subscriber on a subscriber_group also delete that subscriber_group.

Role **admin** and **support** are allowed to do so.
Role **user** gets returned `403 Forbidden`, if not in owner array.

GET /subscriber_groups

Returns an array of existing subscribers_goups tags in JSON format. This is allowed for **all** roles.

```
{
  ["dl.OV-G01","dl.rwth-afu"]
}
```

GET /transmitters/_view/groups

Returns a JSON array of used transmitter groups tags from all known transmitters. Used for a suggestion of already existing transmitter group tags on the website.

DELETE /transmitters/<transmittername>?rev=

Delete the transmitter <transmittername>. Also delete the transmitter from explicit entries on rubrics.

First get transmitter revision as defined in section 6.1.5. Then send the request with the revision.

PUT /transmitters - Add new transmitter

Add a new the transmitter.

Allowed roles are **admin** and **support**. Role **user** will get `403 Forbidden` Example data to send:

```
{
  "_id": "db0wa",
  "usage": "widerange",
  "timeslots": [
    true,
    true,
    false,
    true,
    true,
    false,
    true,
    true,
    false,
    false,
    true,
    true,
    false,
    true,
    true,
    false
  ],
  "power": 20,
  "owners": [
    "dh3wr"
  ],
  "groups": [
    "dl.nw.koeln.aachen"
  ],
  "emergency_power": {
    "available": true,
    "infinite": false,
    "duration": 7200
  },
  "coordinates": [
    50.71613,
    6.165481
  ],

  "aprs_broadcast": false,
  "enabled": true,
  "auth_key": "Arj39135jAKS",
  "antenna": {
    "type": "omni",
    "gain": 0,
    "direction": 0,
    "agl": 1
  }
}
```

The MS has to add

```
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by": "dh3wr",
```

and add it to the REST PUT call.

PUT /transmitters - Edit existing transmitter

Edit an existing transmitter.

Allowed roles are **admin** and **support**. Role **user** will get `403 Forbidden` `_id` and `_rev` have to be send always. So first get the current revision of the transmitter with a `GET /transmitters/<transmittername>`.

Is an active connection reset to that transmitter necessary? If the Authkey changes, an already established connection will keep working? And what about timeslot changes? They have to be applied immediately to the transmitter.

Example data to send:

```
{

  "_id": "db0wa",
  "_rev": "3-212820d0a75061289c8fbe39192fde22",
  "usage": "widerange"
}
```

The MS has to add or change the keys

```
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by": "dh3wr",
```

and add them to the REST PUT call.

6.1.3 Call Service

GET /calls

Returns the last 100 calls with all details.

GET /calls?limit=<number>

Returns the last <number> of calls with all details. If <number> is higher than the available calls, just return all available calls.

GET /calls/_view/byDate

With GET parameters:

```
GET /calls/_view/byDate?startkey="<startddate">&endkey="<enddate>"
```

Returns the calls made within the specified time span with all details. If there are no calls stored in the specified time span, return empty JSON.

GET /calls/_view/byIssuer

```
GET /calls/_view/byIssuer?key="dh3wr"
```

Returns all the calls issued from callsign dh3wr with all details. If there are no calls stored with in the specified time span, return empty JSON. (The microservice has to transfrom the request into *startkey="dh3wr"&endkey="dh3wr"* to the CouchDB GET request by itself.)

GET /calls/_view/byRecipient

```
GET /calls/_view/byRecipient?key="dh3wr"
```

Returns all the calls with recipient callsign dh3wr with all details. If there are no calls stored in the specified time span, return empty JSON. (The microservice has to transfrom the request into *startkey="dh3wr"&endkey="dh3wr"* to the CouchDB GET request by itself.)

Any combination of the given filter method

GET /calls/_view/pending

Return all details of pending calls, that are not transmitted by at least one transmitter.

GET /calls/_view/pending_all

Return all details of pending calls, that are not transmitted by all designated transmitters.

POST /call

Insert call to the system. Send in POST content:

```
{
  "subscriber": ["dh3wr",...],
  "subscriber\_groups": ["dl.ov-g01",...]
  "priority" : 1 to 5,
  "message": "This is an example call",
  "transmitter_groups": ["dl-all","on-all"]
}
```

6.1.4 Rubric Service

GET /news

Returns an array of all rubrics and their content in JSON format.

GET /news/_view/byRubric

```
GET /news/_view/byRubric?startkey="metar-dl"&endkey="metar-dl"
```

Returns just the content of <rubricname> content in JSON format.

GET /news/_view/byRubric/message_no>

Returns just the content of <rubricname> content and message number <message_no> in JSON format.

PUT /rubrics/content

Add content to rubric <rubricname> on the first message slot and move the existing message one to the end. The 10th. entry will be lost. An automated resend of all rubric content slots will be necessary.

PUT /rubrics/content/<rubricname>/<message_no>

Add or override the content of rubric <rubricname> on the message slot <message_no>. An automated resend of just this message slot will be necessary.

DELETE /rubrics/content/<rubricname>?rev=

Delete all content in rubric <rubricname>. The content will be still on Skypers that have received it before, but it will not be transmitted periodically any more. No dependency check necessary.

DELETE /rubrics/content/<rubricname>/<message_no>?rev=

Delete the content in rubric <rubricname> with message slot <message_no>. The content will be still on Skypers that have received it before, but it will not be transmitted periodically any more. No dependency check necessary.

6.1.5 Transmitter Service

GET /transmitters

Return all transmitters with all details in JSON format.

GET /transmitter/<transmittername>

Return all details just of transmitter <transmittername>.

GET /transmitters/transmitternames

Return an JSON array of all transmitter names. Used where selections have to be done on the website.

POST /transmitters/bootstrap

`POST /transmitter/bootstrap`

```
{
  "callsign": "db0avr",
  "auth_key": "<secret>",
  "software": {
    "name": "UniPager",
    "version": "1.0.2"
  }
}
```

Answers to the bootstrap REST call

`200 OK`

```
{
  "timeslots": [true, true, false, true, ...],
  "nodes": [
    {
      "host": "node1.ampr.org",
      "port": 4000,
      "reachable": true,
      "last_seen": "2018-07-03T07:43:52.783611Z",
      "response_time": 42
    }
  ]
}
```

`423 Locked`

```
{
  "error": "Transmitter temporarily disabled by config."
}
```

`423 Locked`

```
{
  "error": "Transmitter software type not allowed due to serious bug."
}
```

POST /transmitters/heartbeat

`POST /transmitter/heartbeat`

```
{
  "callsign": "db0avr",
  "auth_key": "<secret>",
  "ntp_synced": true
}
```

Answers to the heartbeat REST call `200 OK`

```
{
  "status": "ok"
}
```

If network wants to assign new timeslots without disconnecting (for dynamic timeslots)

`200 OK`

```
{
  "status": "ok",
  "timeslots": [true, true, false, ...],
  "valid_from": "2018-07-03T08:00:52.786458Z"
}
```

If network wants to initiate handover to other node

`503 Service unavailable`

```
{
  "error": "Node not available, switch to other node."
}
```

6.1.6 Cluster Service

POST /cluster/discovery

6.1.7 Telemetry Service

GET /telemetry/transmitters

Return the stored telemetry **summary** values for all transmitters.

GET /telemetry/transmitters/<transmittername>

Return all the stored telemetry values for transmitter <transmittername>.

GET /telemetry/nodes

Return the stored telemetry **summary** values for all nodes.

GET /telemetry/nodes/<nodesname>

Return all the stored telemetry values for node <nodename>.

WS /telemetry

See the section for Websocket API.

6.1.8 Database Changes Service

WS /changes

See the section for Websocket API on database changes.

6.1.9 Status Service

The purpose of the status service is to provide a short overview of the DAPNET network and the microservices.

GET /status/nodes

No authentication required.

Answer: `200 OK`

```
{
  "nodes": [
    {
      "host": "node1.ampr.org",
      "port": 4000,
      "reachable": true,
      "last_seen": "2018-07-03T07:43:52.783611Z",
      "response_time": 42
    }
  ],
  "connections": {
    "rabbitmq": true,
    "couchdb": true,
    "hamcloud": true,
  },
  "hamcloud_node": false,
  "general_health": true
}
```

What is "port"?

GET /status/node/<nodename>

No authentication required.

Answer: `200 OK`

```
{
  "host": "node1.ampr.org",
  "port": 4000,
  "reachable": true,
  "last_seen": "2018-07-03T07:43:52.783611Z",
  "response_time": 42,
  "connections": {
    "rabbitmq": true,
    "couchdb": true,
    "hamcloud": true,
  },
  "hamcloud_node": false,
  "general_health": true
}
```

What is "port"?

GET /status

Get status of this node. `200 OK`

```
{
  "good_health" : true,
  "version" : "1.2.3"
  "microservices\_running" : {
    "database" : true,
    "call" : true,
    "rubric" : true,
    "transmitter" : true,
    "cluster" : true,
    "telemetry" : true,
    "database-changes" : true,
    "statistics" : true,
    "rabbitmq" : true,
    "thirdparty" : true
  }
}
```

GET /status/<service_name>

List of valid values for *service_name*:

database-service
call-service
rubric-service
transmitter-service
cluster-service
telemetry-service
database-changes-service
statistics-service
rabbitmq-service

200 OK

```
<Status output from service itself>
```

6.1.10 Statistics Service

GET /statistics

No authentication required.

Answer: 200 OK

```
{
  "users" : 1234,
  "transmitters": {
    "personal": {
      "online": 13
      "total": 34
    },
    "widerage": {
      "online": 53,
      "total": 97
    }
  }
  "nodes": {
    "online": 10,
    "total": 19
  },
  "processed_calls": 1234,
  "processed_rubric_content_changes": 234
}
```

On the calls and rubric content changes: Always increasing counter link traffic on network device or reset at 00:00 am?

6.1.11 RabbitMQ Service

GET /rabbitmq/*

6.2 RabbitMQ

There are 3 exchanges available on each RabbitMQ instance:

dapnet.calls Messages shared between all nodes

dapnet.local_calls Messages coming from the local node instance

dapnet.telemetry Messages containing telemetry from transmitters

6.2.1 Transmitters

Valid Messages are:

dapnet.calls

The messages to transfer data to be transmitted by the transmitter have the following format.

For each transmission, there is a separate RabbitMQ message, as different receivers might need different text encoding. All encoding is already done, when this message is created. The transmitter does no character encoding at all. Both personal pagings and rubric related messages are transmitted with this protocol.

```
{
  "id": "016c25fd-70e0-56fe-9d1a-56e80fa20b82",
  "protocol": "pocsag",
  "priority": 3,
  "expires": "2018-07-03T08:00:52.786458Z",
  "message": {
    "ric": 12342, (max 21 Bits)
    "type": "alphanum", | "numeric"
    "speed": 1200,
    "function": 0 to 3,
    "data": "Lorem ipsum dolor sit amet"
  }
}
```

The selection of the transmitter is done by means of the routing key. Besides, the priority is also used in the RabbitMQ queuing to deliver higher priority messages first.

dapnet.local_calls

Same as for the the network originated calls in section 6.2.1.

6.2.2 Telemetry

On the telemetry exchange, all transmitters and nodes publish their telemetry messages. The format the same as in section 6.3 and 6.4.

6.2.3 MQTT API for third-party consumers

In order to allow third-party instances like , or others to get the emitted calls and rubric contents in a real time event driven way, there is an MQTT API. It is not implemented via a dedicated MQTT broker, but uses the existing RabbitMQ instance (https://www.rabbitmq.com/mqtt.html. There is no distribution of the messages via this MQTT broker; it is local only. So every node publishes the messages locally on its own. Each subscriber has an array of enabled third-party applications. This allow to define the user, if call directed to her/his subscriber shall be also sent to third-party services (see 6.7.4.

check with DL2IC

The currently existing MQTT topics are defined in the CouchDB (see section 6.7.7). This makes it possible to add more third-party services and authorized users during runtime without the need to update the software. The valid users to subscribe to the topic are also listed in the same CouchDB database.

The only permitted access for third-party consumers is read. So the subscribe request from a third-party MQTT-Client must use authentication which is checks against the CouchDB data. If correct, read access is granted. Core software has always write access to publish the calls group messages.

The transmitters who are supposed to send out the personal call or the rubric content are published with callsign, geographic location and type of transmitter (widerange or personal). With this generic concept, every third-party application can decide what to do with the content received.

The encoding of the data is UTF-8.

The format of the data published for **personal paging calls** is

```
{
  "pagingcall" : {
    "srccallsign" : "dl2ic",
    "dstcallsign" : "dh3wr",
    "dstric" : 12354,
    "dstfunction" : 0 .. 3,
    "priority" : 3,
    "message" : "DAPNET 2.0 rocks dear YL/OM"
    "transmitted_by" : [
      {
        "callsign" : "db0abc",
        "lat" : 12.123456,
        "long" : 32.123456,
        "type" : "personal" | "widerange"
      },
      {
        "callsign" : "db0def",
        "lat" : 12.123456,
        "long" : 32.123456,
        "type" : "personal" | "widerange"
      }
    ],
    "timestamp" : "2018-07-03T08:00:52.786458Z"
  }
}
```

The format of the data published for **rubric_content paging calls** is

```
{
  "rubricmessage" : {
    "message" : ""
    "transmitted_by" : [
      {
        "callsign" : "db0abc",
        "lat" : 12.123456,
        "long" : 32.123456,
        "type" : "PERONAL" | "WIDERANGE"
      },
      {
        "callsign" : "db0def",
        "lat" : 12.123456,
        "long" : 32.123456,
        "type" : "PERONAL" | "WIDERANGE"
      }
    ],
    "timestamp" : "2018-07-03T08:00:52.786458Z"
  }
}
```

6.3 Telemetry from Transmitters

Telemetry is sent from transmitters to the RabbitMQ exchange **dapnet.telemetry** as defined in section 6.2. It is also used in the same way on the websocket API to inform the website and the app about the telemetry in real-time in section 6.6.1 and 6.6.2.

This is sent every minute in complete. If there are changes, just a subset is sent. The name of the transmitter is used as routing key for the message.

```
{
  "onair": true,
  "node": {
    "name": "db0xyz",
    "ip": "44.42.23.8",
    "port": 1234,
    "connected": true,
    "connected_since": "2018-07-03T08:00:52.786458Z"
   },
```

```
"ntp": {
  "synced": true,
  "offset": 124,
  "server": ["134.130.4.1", "12.2.3.2"],
},
"messages": {
  "queued": [123, 123, 123, 123, 123, 123],
  "sent": [123, 123, 123 , 123, 123, 123]
},
"temperatures": {
  "unit": "C" | "F" | "K",
  "air_inlet": 12.2,
  "air_outlet": 14.2,
  "transmitter": 42.2,
  "power_amplifier": 45.2,
  "cpu": 93.2,
  "power_supply": 32.4,
  "custom": [
    {"value": 12.2, "description": "Aircon Inlet"},
    {"value": 16.2, "description": "Aircon Outlet"},
    {"value": 12.3, "description": "Fridge Next to Programmer"}
  ]
},
"power_supply": {
  "on_battery": false,
  "on_emergency_power": false,
  "dc_input_voltage": 12.4,
  "dc_input_current": 3.23
},
"rf_output" : {
  "fwd": 12.2,
  "refl" : 12.2,
  "vswr" : 1.2
},
"config": {
  "ip": "123.4.3.2",
  "timeslots" : [true, false,...,     false],
  "software": {
    name: "Unipager" | "MMDVM" | "DAPNET-Proxy",
    version: "v1.2.3", | "20180504" | "v2.3.4",
  },
}
"hardware": {
  "platform": "Raspberry Pi 3B+"
},
"rf_hardware": {
  "c9000": {
    "name" : "C9000 Compact",
    "<pa_dummy>" : {
      "output_power" : 123,
      "port" : "/dev/ttyUSB0"
    }
    "<rpc>": {
      "version" : "XOS/2.23pre"
    }
  },
  "raspager": {
    "name": Raspager",
    "modulation": 13,
    "power": 63,
    "external_pa": false,
    "version": "V2"
  },
  "audio": {
    "name" = "Audio",
    "transmitter": "GM1200" | "T7F" | "GM340" | "FREITEXT",
    "audio_level": 83,
    "tx_delay": 3
  },
  "rfm69": {
    "name" : "RFM69",
    "port": "/dev/ttyUSB0"
  },
```

```
    "mmdvm": {
      "name" : "MMDVM",
      "dapnet_exclusive": true
    }
  },
  "proxy" : {
    "status": "connected" | "connecting" | "disconnected"
  }
}
```

6.4 Telemetry from Nodes

Telemetry is sent from nodes to the RabbitMQ exchange **dapnet.telemetry** as defined in section 6.2. It is also used in the same way on the websocket API to inform the website and the app about the telemetry in real-time in section 6.6.3 and 6.6.4.

This is sent every minute in complete. If there are changes, just a subset is sent. The name of the nodes is used as routing key for the message.

```
{
  "good_health" : true,
  "microservices" : {
    "database" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "call" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "rubric" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "transmitter" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "cluster" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "telemetry" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "database-changes" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "statistics" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "rabbitmq" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "thirdparty" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
  },
  "connections" : {
    "transmitters" : 123,
    "third_party" : 3
  },
  "system" : {
    "free_disk_space_mb": 1234
    "cpu_utilization": 0.2
```

```
    "is_hamcloud" : false
  }
}
```

6.5 Statistic, Status and Telemetry REST API

The statistic and telemetry REST API provides up-to-date information regarding the transmitters and the network via REST. This can be used by e.g. grafana to draw nice graphes or nagios plugins.

6.5.1 Telemetry from Transmitters

`GET /telemetry/transmitters`

No authentication required. Here all stored telemetry from all transmitters is provided.

Answer: `200 OK` See 6.3

`GET /telemetry/transmitters/<transmittername>`

No authentication required. Here all stored telemetry from the specified transmitter is provided.

Answer: `200 OK`

See 6.3

`GET /telemetry/transmitters/<transmittername>/<section_of_telemetry>`

No authentication required. Here all stored telemetry within the telemetry section from the specified transmitter is provided. Possible sections are 2. Level JSON groups, see 6.3.

Examples: onair, telemetry, transmitter_configuration

Answer: `200 OK`

See 6.3

6.5.2 Telemetry from Nodes

`GET /telemetry/nodes`

No authentication required. Here all stored telemetry from all nodes is provided.

Answer: `200 OK` See 6.4

`GET /telemetry/nodes/<nodename>`

No authentication required. Here all stored telemetry from the specified node is provided.

Answer: `200 OK`

See 6.4

6.6 Websocket API

The idea is to provide an API for the website and the app to display real-time information without the need of polling. A websocket server is listing to websocket connections. Authentication is done by a custom JOSN handshake. The connection might be encrypted with SSL if using the Internet or plain if using HAMNET.

The data is taken from the **dapnet.telemetry** exchange from the RabbitMQ instance and further other sources if necessary.

Define if authentication is necessary of some endpoints?

There are 5 main endpoints in the websocket interface:

Endpoint		Microservice
WS	/telemetry/transmitters	Summary data of all TX
WS	/telemetry/transmitters/<TxName>	Details for TX <TxName>
WS	/telemetry/nodes	Summary data of all Nodes
WS	/telemetry/nodes/<NodeName>	Details for Node <NodeName>
WS	/changes	Database changes

6.6.1 Telemetry from Transmitters - Summary of all TX

URL: `ws://FQDN/telemetry/transmitters`

The data is the same as received from the **dapnet.telemetry** exchange from the RabbitMQ instance. It is defined in section 6.3.
The websocket-Server generates an array of JSON Objects which have the name of the transmitter obtained from the RabbitMQ routing key.
The current time slot is also sent in the summary and updated also by its own every time a time slot change happens.

```
{
  "transmitters": [
    "db0abc" : {
      "onair": true,
      "node": {
        "name": "db0xyz",
        "ip": "44.42.23.8",
        "port": 1234,
        "connected": true,
        "connected_since": "2018-07-03T08:00:52.786458Z"
       },
      "ntp": {
        "synced": true
      },
      "messages": {
        "queued": [123, 123, 123, 123, 123, 123],
        "sent": [123, 123, 123 , 123, 123, 123]
      },
      "config": {
        "ip": "123.4.3.2",
        "timeslots" : [true, false,..., false],
      "software": {
        name: "Unipager" | "MMDVM" | "DAPNET-Proxy",
        version: "v1.2.3", | "20180504" | "v2.3.4"
      },
      "proxy" : {
        "status": "connected" | "connecting" | "disconnected"
      }
    },
    "db0xyz" : {
      "onair": true,
      "node": {
    ....
```

```
    }
  ],
  "current_timeslot" : 12
}
```

6.6.2 Telemetry from Transmitters - Details of Transmitter

URL: `ws://FQDN/telemetry/transmitters/<transmittername>`

The data is the same as received from the **dapnet.telemetry** exchange from the RabbitMQ instance. It is defined in section 6.3.
The websocket-Server gives out all the telemetry data from a certain transmitter. The name of the transmitter obtained from the RabbitMQ routing key.

```
{
  "onair": true,
  "node": {
  "ip": "44.42.23.8",
    "port": 1234,
    "connected": true,
    "connected_since": "2018-07-03T08:00:52.786458Z"
  },
  "ntp": {
    "synced": true,
    "offset": 124,
    "server": ["134.130.4.1", "12.2.3.2"],
  },
  "messages": {
    "queued": [123, 123, 123, 123, 123, 123],
    "sent": [123, 123, 123 , 123, 123, 123]
  },
  "temperatures": {
    "unit": "C" | "F" | "K",
    "air_inlet": 12.2,
    "air_outlet": 14.2,
    "transmitter": 42.2,
    "power_amplifier": 45.2,
    "cpu": 93.2,
    "power_supply": 32.4,
    "custom": [
      {"value": 12.2, "description": "Aircon Inlet"},
      {"value": 16.2, "description": "Aircon Outlet"},
      {"value": 12.3, "description": "Fridge Next to Programmer"}
    ]
  },
  "power_supply": {
    "on_battery": false,
    "on_emergency_power": false,
    "dc_input_voltage": 12.4,
    "dc_input_current": 3.23
  },
  "rf_output" : {
    "fwd": 12.2,
    "refl" : 12.2,
    "vswr" : 1.2
  },
  "config": {
    "ip": "123.4.3.2",
    "timeslots" : [true, false,...,     false],
    "software": {
      name: "Unipager" | "MMDVM" | "DAPNET-Proxy",
      version: "v1.2.3", | "20180504" | "v2.3.4",
    },
  }
  "hardware": {
    "platform": "Raspberry Pi 3B+"
  },
  "rf_hardware": {
    "c9000": {
```

```
      "name" : "C9000 Compact",
      "<pa_dummy>" : {
        "output_power" : 123,
        "port" : "/dev/ttyUSB0"
      }
      "<rpc>": {
        "version" : "XOS/2.23pre"
      }
    },
    "raspager": {
      "name": Raspager",
      "modulation": 13,
      "power": 63,
      "external_pa": false,
      "version": "V2"
    },
    "audio": {
      "name" = "Audio",
      "transmitter": "GM1200" | "T7F" | "GM340" | "FREITEXT",
      "audio_level": 83,
      "tx_delay": 3
    },
    "rfm69": {
      "name" : "RFM69",
      "port": "/dev/ttyUSB0"
    },
    "mmdvm": {
      "name" : "MMDVM",
      "dapnet_exclusive": true
    }
  },
  "proxy" : {
    "status": "connected" | "connecting" | "disconnected"
  }
}
```

6.6.3 Telemetry from Nodes - Summary of all Nodes

URL: `ws://FQDN/telemetry/nodes`

The websocket-Server generates an array of JSON Objects which have the name of the node obtained from the RabbitMQ routing key.

```
{
  "nodes" : [
    "db0sda" : {
      "good_health" : true,
      "connections" : {
        "transmitters" : 123,
        "third_party" : 3
      },
      "system" : {
        "is_hamcloud" : false
      }
    },
    "hamcloud1" : {
      "good_health" : true,
      "connections" : {
        "transmitters" : 658,
        "third_party" : 25
      },
      "system" : {
        "is_hamcloud" : true
      }
    },
    ....
  ]
}
```

6.6.4 Telemetry from Transmitters - Details of Node

URL: `ws://FQDN/telemetry/nodes/<nodename>`

The data is the same as received from the **dapnet.telemetry** exchange from the RabbitMQ instance. It is defined in section 6.3.
The websocket-Server gives out all the telemetry data from a certain node. The name of the transmitter obtained from the RabbitMQ routing key.

```
{
  "good_health" : true,
  "microservices" : {
    "database" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "call" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "rubric" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "transmitter" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "cluster" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "telemetry" : {
      "ok" : true,
      "version" : "1.2.3"
    },
    "database-changes" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "statistics" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "rabbitmq" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
    "thirdparty" :  {
      "ok" : true,
      "version" : "1.2.3"
    },
  },
  "connections" : {
    "transmitters" : 123,
    "third_party" : 3
  },
  "system" : {
    "free_disk_space_mb": 1234
    "cpu_utilization": 0.2
    "is_hamcloud" : false
  }
}
```

6.6.5 Database Changes

URL: `ws://FQDN/changes`

To inform the website or the app about changes in the CouchDB database, the websocket microservice keeps a connection to the local CouchDB API and receives a stream of updated to the database. As there may be data in the changes that are confidential, the stream is parsed and sent out in a reduced form to the websocket client. Further information: http://docs.couchdb.org/en/2.0.0/api/database/changes.html

The format of the updates is:

define/review format

Transmitter related

New transmitter added

```
{
  "type": "transmitter",
  "action" : "added",
  "name": "db0abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Existing transmitter changed

```
{
  "type": "transmitter",
  "action" : "changed",
  "name": "db0abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Transmitter deleted

```
{
  "type": "transmitter",
  "action" : "deleted",
  "name": "db0abc"
```

User related

New User added

```
{
  "type": "user",
  "action" : "added",
  "name": "db1abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Existing user changed

```
{
  "type": "user",
  "action" : "changed",
  "name": "db1abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

User deleted

```
{
  "type": "user",
  "action" : "deleted",
  "name": "db1abc"
}
```

Rubric related

New Rubric added

```
{
  "type": "rubric",
  "action" : "added",
  "id": "...",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Existing rubric changed

```
{
  "type": "user",
  "action" : "changed",
  "id": "...",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Rubric deleted

```
{
  "type": "user",
  "action" : "deleted",
  "id": "..."
}
```

Rubric content related

Check against CouchDB structure

New Rubric content added

```
{
  "type": "rubric_content",
  "action" : "added",
  "id": "...??",
  "data" : {
  (Complete Data dump of all ten rubric messages as stored in CouchDB)
  }
}
```

Existing rubric changed

```
{
  "type": "rubric_content",
  "action" : "changed",
  "id": "...",
  "data" : {
  (Complete Data dump of all ten rubric messages as stored in CouchDB)
  }
}
```

Rubric content deleted

```
{
  "type": "rubric_content",
  "action" : "deleted",
  "id": "..."
  "data" : {
  (Complete Data dump of all ten rubric messages as stored in CouchDB, some may be empty)
  }
}
```

Node related

New node added

```
{
  "type": "node",
  "action" : "added",
  "name": "db0abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Existing node changed

```
{
  "type": "node",
  "action" : "changed",
  "name": "db0abc",
  "data" : {
  (Data from CouchDB Change feed in processed way)
  }
}
```

Node deleted

```
{
  "type": "node",
  "action" : "deleted",
  "name": "db1abc"
}
```

6.7 CouchDB Documents and Structure

als Tabelle darstellen

6.7.1 Users

Table 6.1: CouchDB: Users

Key	Value-Type	Valid Value Range	Example
_id	string		dl1abc
password	string	bcrypt hash	—
email	string		dl1abc@darc.de
role	string	"admin"|"support"|"user"	true
enabled	boolean		true
created_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_by	string	valid user name	dh3wr
email_valid	boolean		true
avatar_picture	couchdb_attachment		

```
{
  "_id": "dl1abc",
  "password": "<bcrypt hash>",
  "email": "dl1abc@darc.de",
  "role": "admin",
  "enabled": true,
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by": "dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by":"dh3wr",
  "email_valid": true
  "avatar_picture": <couchdb attachment>
}
```

Wofür genau braucht man email_valid? Um ab

6.7.2 Nodes

Table 6.2: CouchDB: Nodes

Key	Value-Type	Valid Value Range	Example
_id	STRING	N/A	db0abc
coordinates	[number; 2]	[lat, lon]	[34.123456, 6.23144]
description	string	whatever	Aachen, Germany
hamcloud	boolean	true/false	true
created_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_by	string	valid user name	dh3wr
changed_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_by	string	valid user name	dh3wr
owners	[string]	N/A	["dl1abc","dh3wr","dl2ic"]
avatar_picture	couchdb_attachment		

```
{
  "_id": "db0abc",
  "auth_key": "super_secret_key",
  "coordinates": [34.123456, -23.123456],
  "description": "some words about that node",
  "hamcloud": true,
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by":"dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by":"dh3wr",
  "owners": ["dl1abc","dh3wr","dl2ic"],
  "avatar_picture": <couchdb attachment??>
}
```

6.7.3 Transmitters

Tabelle weiter machen

Table 6.3: CouchDB: Transmitters

Key	Value-Type	Valid Value Range	Example
_id	string	N/A	db0abc
auth_key	string	N/A	asd2FD3q3rF
enabled	boolean	true/false	true
usage	string	PERSONAL | WIDERANGE	WIDERANGE
coordinates	[number; 2]	[lat, lon]	[34.123456, 6.23144]
power	number	0.001 ...	12.3
created_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_by	string	valid user name	dh3wr
changed_on	string	ISO8601	2018-07-08T11:50:02.168325Z
changed_by	string	valid user name	dh3wr
owners	ARRAY of STRING	N/A	["dl1abc","dh3wr","dl2ic"]
avatar_picture	couchdb_attachment		

```
{
  "_id": "db0abc",
  "auth_key": "hdjaskhdlj",
  "enabled": true,
  "usage": "personal" | "widerange",
  "coordinates": [34.123456, -23.123456],
  "power": 12.3,
  "antenna": {
    "agl": 23.4,
    "gain": 2.34,
    "type": "omni" | "directional",
    "direction": 123.2,
```

```
    "cable_loss": 4.2
  }
  "owners" : ["dl1abc","dh3wr","dl2ic"],
  "groups" : ["dl-hh", "dl-all"],
  "emergency_power": {
    "available": false,
    "infinite": false,
    "duration": 23*60*60 // seconds
  },
  "created_on": "2018-07-03T08:00:52.786458Z",
  "created_by": "dh3wr",
  "changed_on": "2018-07-03T08:00:52.786458Z",
  "changed_by": "dh3wr",
  "aprs_broadcast": false,
  "antenna_pattern" : <couchDB attachment>,
  "avatar_picture" : <couchDB attachment>
}
```

6.7.4 Subscribers

If type is "Skyper", function is always 3. Keep this in mind

check if [] is valid JSON

```
{
  "_id" : "dl1abc",
  "description" : "Peter",
  "pagers" : [
    {
      "ric": 123456,
      "function": 0 .. 3,
      "name": "Peters Alphapoc",
      "type" : "UNKNOWN" | "Skyper" | "AlphaPoc" | "QUIX" | "Swissphone" | "SCALL_XT" | "Birdy"
      "enabled" : true
    },
    ...
  ],
  "third_party_services" : ["APRS", "BM"],
  "owner": ["dh3wr", "dl1abc"],
  "groups" : ["rwth-afu"]
}
```

6.7.5 Rubrics

```
{
  "_id": "wx-dl-hh"
  "number": 14,
  "description": "Wetter DL-HH",
  "label": "WX DL-HH",
  "transmitter_groups": ["dl-hh","dl-ns"],
  "transmitters": ["db0abc"],
  "cyclic_transmit": true,
  "cyclic_transmit_interval": 3600, // seconds
  "owner" : ["dh3wr", "dl1abc"]
}
```

6.7.6 Rubric's content

<UUID> of rubric (as defined in **??**)

```
{
  "_id" : "<UUID>",
  "rubric": "wx-dl-hh",
  "content": [
    "message1",
    ..,
    "message10"
  ],
}
```

6.7.7 MQTT services and subscribers

```
{
  "_id": "APRS",
  "topic": "aprs"
  "subscribers": [
    {
      "name": "example",
      "password": "<bcrypt hash>"
    },
    ...
  ]
}
```