DAPNET 2.0 Concept and Interface Definition

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

July 8, 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

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.

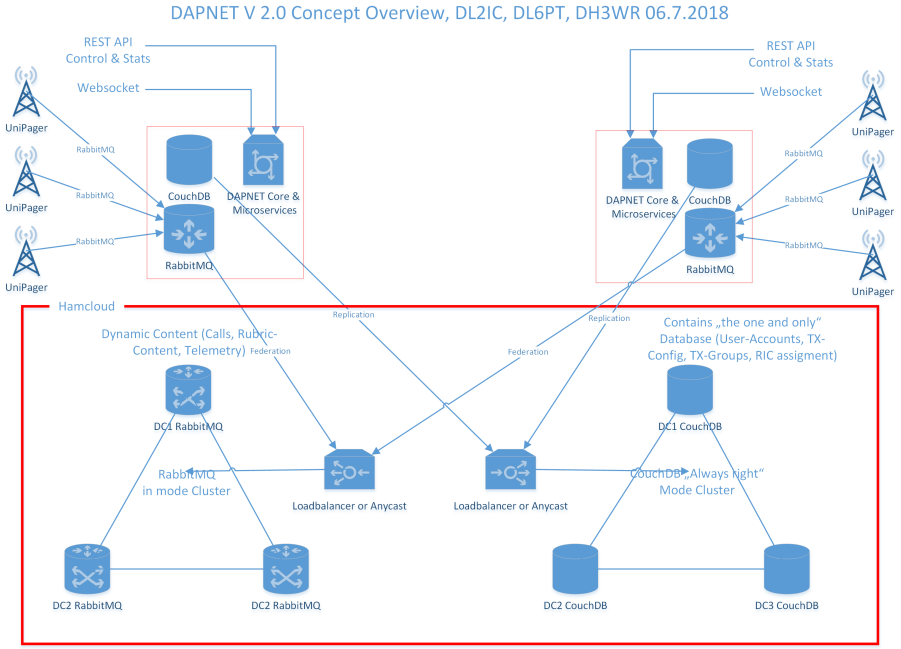

Figure 1.1: Overview of DAPNET Clutering and Network Structure

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

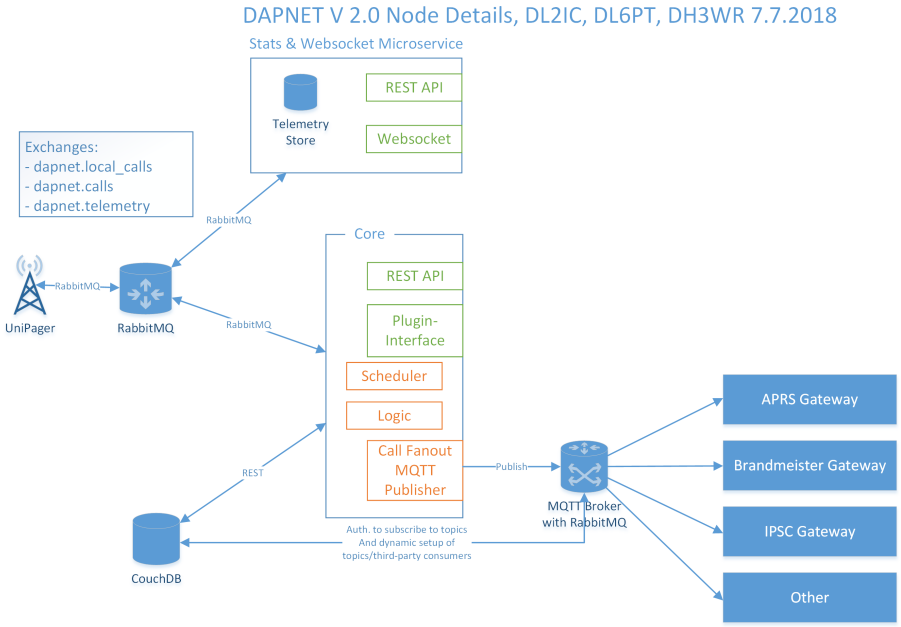

Figure 1.2: Node Details

1.4 Transmitter Software

1.4.1 Unipager

1.4.2 DAPNET-Proxy

1.5 DAPNET Network

1.5.1 Overview and Concept

1.5.2 Used third-party Software

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 https://www.swiss-artg.ch/fileadmin/Dokumente/HAMNET/HamCloud_-_Angebotene_Dienste_in_der_HamCloud.pdf and here http://hamnetdb.net/?m=as&q=hamcloud.

Define if uni- or anycast entry points will exist

1.5.4 Rubric Handling Concept

1.5.5 Queuing Priority Concept

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 and Web-socket 6.5.

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 Microservices

A DAPNET node consists of serveral isolated microservices with different responsibilities. Each microservice runs in 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_groups/*	
DELETE	/transmitters/:id	
PUT	/transmitters	
*	/calls/*	Call Service
*	/rubrics/:id/*	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	
WS	/changes	Database Changes Service
GET	/status/*	Status Service
GET	/statistics	Statistics Service
GET	/rabbitmq/*	RabbitMQ Auth Service

3.10.1 Database Service

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

3.10.2 Call Service

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

3.10.3 Rubric Service

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

3.10.4 Transmitter Service

- Maintains a list of all transmitters and their current status

3.10.5 Cluster Service

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

3.10.6 Telemetry Service

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

3.10.7 Database Changes Service

- Forwards database changes via websocket

3.10.8 Status Service

- Periodically checks all other services and connections

3.10.9 RabbitMQ Auth Service

- Provides authentication for RabbitMQ against the CouchDB users database

3.10.10 Time and Identification Service

- Sends periodic time and identification messages to RabbitMQ

3.11 Ports and Loadbalacing Concept

3.12 Periodic Tasks (Scheduler)

3.13 Plugin Interface

3.14 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 (6.1.2).
2. Get as response the transmitter configuration or an error message (6.1.3).
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.

3.15 Transmitter connections

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

3.15.1 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.

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.

3.16 DAPNET-Proxy

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

5.1 Unipager

5.2 DAPNET-Proxy

5.3 DAPNET Core

5.4 Special issues for Core running in HAMCLOUD

Chapter 6

Protocol Definitions

6.1 Core REST API

define all the website/app REST calls to interact with DAPNET

6.1.1 Management and User interactions

Calls

Users

Subscribers

Transmitters

Existing Transmitter Groups

Rubrics

Rubric Content

Nodes

Third-Party services and subscribers

6.1.2 Transmitter sign-on, configuration and heartbeat

`POST /transmitter/bootstrap`

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

6.1.3 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."
}
```

6.1.4 Transmitter 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."
}
```

6.2 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,
  "message": {
    "ric": 12342,
    "subric": 0 to 3,
    "type": "ALPHANUM",
    "speed": 1200,
    "function": 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 publish their telemetry messages. The format the same as in section 6.3.

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 6.6.4.

check with DL2IC

The currently existing MQTT topics are defined in the CouchDB (see section 6.6.8). 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" : 12344,
    "dstsubric" : 0 ... 3,
    "priority" : 3,
    "message" : "DAPNET 2.0 rocks dear YL/OM"
    "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" : <TIMESTAMP>
  }
}
```

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" : <TIMESTAMP>
  }
}
```

6.3 Telemetry

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 6.5.1.

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": "<timestamp-format>"
   },
```

```
"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": {
    "<pa_dummy>": {
      "output_power": 123,
      "port": "/dev/ttyUSB0"
    }
    "<rpc>": {
      "version" : "XOS/2.23pre"
    }
  },
  "raspager": {
    "modulation": 13,
    "power": 63,
    "external_pa": false,
    "version": "V2"
  },
  "audio": {
    "transmitter": "GM1200" | "T7F" | "GM340" | "FREITEXT",
    "audio_level": 83,
    "tx_delay": 3
  },
  "rfm69": {
    "port": "/dev/ttyUSB0"
  },
  "mmdvm_dualhs": {
   "dapnet_exclusive": true
  }
},
```

```
  "proxy" : {
    "status": "connected" | "connecting" | "disconnected"
  }
}
```

6.4 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.

6.4.1 Statistics

Statistics are given in JSON with number values only to may parsing easier.

`GET /stats`

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?

6.4.2 Status

`GET /status`

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
}
```

6.4.3 Telemetry

`GET /telemetry`

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

Answer: `200 OK` See 6.3

`GET /telemetry/<transmittername>`

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

Answer: `200 OK`

See 6.3

`GET /telemetry/<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 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.

6.5.1 Telemetry

URL: `ws://FQDN/telemetry`.

The data is the same as received from the **dapnet.telemetry** exchange from the RabbitMQ instance. It is defined in section 6.3.

6.5.2 Database Changes

URL: `ws://FQDN/database_changes`.

To inform the website or the app about changes in the CouchDB database, the websocket mirocservice 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:

6.5.3 Telemetry

6.5.4 Administration

If there is any update on the CouchDB, the other CouchDB instances are notified and updated automatically, but there is no trigger event to notify the websocket service about the change. As changes should be display immediately on the website or app, they have to be announced via websocket. In order to generate a trigger for changes in the CouchDB, the RabbitMQ exchange **dapnet.administration** is used and filled. All websocket microservices consume this exchange.

Transmitter related

New transmitter added

```
{
  "type": "transmitter",
  "action" : "added",
  "name": "db0abc",
  "data" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, remove
  }
}
```

Existing transmitter changed

```
{
  "type": "transmitter",
  "action" : "changed",
  "name": "db0abc",
  "data" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, remove
  }
}
```

Transmitter deleted

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

User related

New User added

```
{
  "type": "user",
  "action" : "added",
  "name": "db1abc",
  "data" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, just
  }
}
```

Existing user changed

```
{
  "type": "user",
  "action" : "changed",
  "name": "db1abc",
  "data" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, just
  }
}
```

User deleted

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

Rubric related

New Rubric added

```
{
  "type": "rubric",
  "action" : "added",
  "id": "...",
  "data" : {
  (Complete Data dump as stored in CouchDB)
  }
}
```

Existing rubric changed

```
{
  "type": "user",
  "action" : "changed",
  "id": "...",
  "data" : {
  (Complete Data dump as stored in CouchDB)
  }
}
```

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" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, just p
  }
}
```

Existing node changed

```
{
  "type": "node",
  "action" : "changed",
  "name": "db0abc",
  "data" : {
  (Complete Data dump as stored in CouchDB. If websocket client is authenticated as non-admin, just p
  }
}
```

Node deleted

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

6.6 CouchDB Documents and Structure

als Tabelle darstellen

6.6.1 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
admin	boolean		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",
  "admin": true,
  "enabled": true,
  "created_on": <DATETIME>,
  "created_by":"dh3wr",
  "changed_on": <DATETIME>,
  "changed_by":"dh3wr",
  "email_valid": true
  "avatar_picture": <couchdb attachment>
}
```

Wofür genau braucht man email_valid?

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]
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		

6.6.2 Nodes

```
{
  "_id": "db0abc",
  "auth_key": "super_secret_key",
  "coordinates": [34.123456, -23.123456],
  "hamcloud": true,
  "created_on": <DATETIME>,
  "created_by":"dh3wr",
  "changed_on": <DATETIME>,
  "changed_by":"dh3wr",
  "owners": ["dl1abc","dh3wr","dl2ic"],
  "avatar_picture": <couchdb attachment??>
}
```

6.6.3 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": <DATETIME>,
  "created_by":"dh3wr",
  "changed_on": <DATETIME>,
  "changed_by":"dh3wr",
  "aprs_broadcast": false,
  "antenna_pattern" : <couchDB attachment>,
  "avatar_picture" : <couchDB attachment>
}
```

6.6.4 Subscribers

check if [] is valid JSON

```
{
  "_id" : "dl1abc",
  "description" : "Peter",
  "pagers" : [
    {
      "ric": 123456,
      "subric": 0 .. 3,
      "uuid": "0023-1233-aefe-1234-3423-9812",
      "name": "Peters Alphapoc",
      "type" : "UNKNOWN" | "Skyper" | "AlphaPoc" | "QUIX" | "Swissphone" | "SCALL_XT" | "Birdy"
      "enabled" : true
    },
    ...
  ],
  "third_party_services" : ["APRS", "BM"],
  "owner": ["dh3wr", "dl1abc"]
}
```

6.6.5 Subscriber Groups

```
{
  "_id" : "ov-G01",
  "description": "Ortverband Aachen",
  "subscribers": ["dl1abc", "dh3wr"],
  "owner": ["dh3wr", "dl1abc"],
}
```

6.6.6 Rubrics List

```
{
  "_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": 60*60, // seconds
  "owner" : ["dh3wr", "dl1abc"]
}
```

6.6.7 Rubric's content

<UUID> of rubric (as defined in 6.6.6)

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

```
    ..,
    "message10"
  ],
}
```

6.6.8 MQTT services and subscribers

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