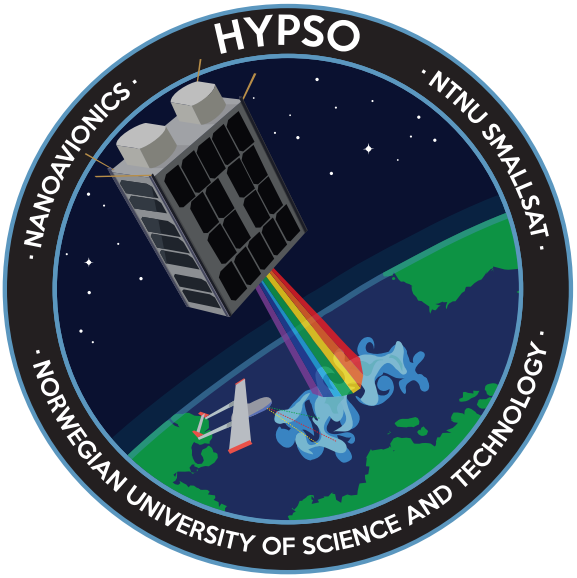
NTNU | Norwegian University of Science and Technology

Department of Engineering Cybernetics

Project Assignment

Open MCT Telemetry Data Visualisation System for NTNU HYPSO SmallSat Mission and Operations



Author:
Audun V. Nytrø
Supervisor:
Tor Arne Johansen
Co-supervisor:
Mariusz Eivind Grøtte

June, 2020

DRAFT

Compiled at 2020-06-27 13:31:21Z

1 Abstract

This report documents the design and implementation of an Open Mission Control Technologies (Open MCT) telemetry storage and visualisation system for the NTNU HYPSO SmallSat. Open MCT is a The system is based on an Express web server running on Node.js, which requests and stores telemetry from a telemetry database operated by NanoAvionics. The system was initially designed to decode Flight Computer and Electronic Power Supply module telemetry, but is expandable to accept other telemetry sources with minimal modifications to the Open MCT interface. It can store and subsequently display telemetry from these sources in near real-time in Open MCT. The stored telemetry data may also be exported or requested outside Open MCT by connecting to the Express-based telemetry web server using HTTP for historical telemetry data, or with WebSocket for subscribing to near real-time telemetry data updates.

The latest version of the documentation and source code for the system is available at https://github.com/NTNU-SmallSat-Lab/mct-depot.

Contents

List of Figures

List of Tables

List of Terms

backend is the component of a computer system that accesses and processes data, often displayed to an end user in a frontend. 12, 17

CubeSat is a type of miniaturised satellite with a volume that is made up of one or more ten-centimetre cubes, each with a mass of no more than 1.3 kilograms. 1

Depot or **MCT Depot**, is a service that processes, temporarily stores and provides data for Open MCT. 12, 14, 16, 17, 20, 29

ECMAScript 6 is the 2015 release of ECMAScript. ECMAScript is a JavaScript-inspired programming language widely used for writing application code for web browsers, and is increasingly being used for writing web services using Node.js. 12

ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code. 17

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. i, 4, 12

ezStruct is a Node.js package that provides a simple way to pack and unpack C struct strings. 14

frontend is the user-facing component of a computer system, which displays and provides methods for interacting with data provided by a backend. 10, 12, 17

hash is a value returned by a hash function, which maps data of arbitrary size down to fixed-size values. 22

Jest is a JavaScript library for creating, running and structuring software tests. 24

NanoAvionics is a satellite mission integrator, manufacturing high-performance multi-purpose CubeSat nanosatellites; provides various hardware, integration and launch services for HYPSO. i, 2, 7, 11, 12

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that lets developers run JavaScript code outside a web browser. i, 4, 12

packing is the process of converting a data object - such as a C struct - to a numerical representation that can be stored in a computer's memory. 14

PostgREST is a standalone web server that gives access to a PostgreSQL database over a RESTful HTTP API. 7, 14, 19

production server is the main server on which a website or web application is being hosted and accessed by users. 20

satellite bus is a general satellite model on which one can build a specialised satellite without having to start from scratch. A satellite bus commonly provides systems for basic functionality, such as attitude control, radio communications, attitude control, propulsion, power generation and distribution. 2

struct is shorthand for structure, which is a collection of variables of possibly different data types. 7, 8, 14

telemetry is the collection of measurements or other data at remote points and their automatic transmission to receiving equipment for monitoring. i, 1, 3

unpacking is the inverse process of packing, and converts a stored binary data to a more readily usable format. 7, 13–15

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. i, 4, 5, 16

List of Acronyms

API Application Programming Interface. 5

AUV Autonomous Underwater Vehicle. 2

DB Database. 22

EPS Electrical Power Supply. 22

FC Flight Computer. 22

FR Functional Requirement. 11, 22, 24

HTTP HyperText Transfer Protocol. i, 4, 5, 11, 14, 16, 19

HYPSO HYPer-spectral Smallsat for ocean Observation. i, 1, 2

JPL Jet Propulsion Laboratory. 3

JSON JavaScript Object Notation. 4, 5, 22

NA NanoAvionics. 19

NASA National Aeronautics and Space Administration. 3

NFR Non-Functional Requirement. 11, 12, 14, 22

UAV Unmanned Aerial Vehicle. 2

USV Unmanned Surface Vehicle. 2

2 Introduction

The HYPSO project at NTNU SmallSats is in need of a system for displaying and working with telemetry data from the satellite.

Current space telemetry visualisation systems are often large monoliths of closed-source software written exclusively for one system with limited potential for modification or reuse without extensive rewrites. One major exception to this is the recently emerging Open Mission Control Technologies framework from NASA and JPL, which provides an interesting opportunity for trying out a new and more flexible approach to telemetry visualisation.

The goal of this project assignment is to use this new framework to build a telemetry data visualisation system for the HYPSO satellite, and investigate if it is a good fit for the HYPSO team as a telemetry visualisation system.

This report will start with quick introductions to what telemetry, Open MCT and HYPSO are, followed by an overview of the requirements and inputs plus outputs for the system we need to implement to get data from the HYPSO satellite into Open MCT. After this a design specification draft for a possible initial version of the system will be documented and discussed, before covering to the process, design changes and final results of implementing a functional version of the planned system.

3 Background

3.1 Telemetry basics

Satellites and other spacecraft produce a fairly significant amount of data that needs to be processed on a day-to-day basis. The part of this that is sent to a ground station via a wireless communication link is what's commonly referred to as telemetry in the space industry. Some of this data may be unique to the mission of the spacecraft, such as image data from a camera or a measurement from a scientific instrument. However, in many cases the most important data for the ground-based operators is the telemetry data that lets them gauge the current status of the spacecraft and subsequently plan future missions and actions for it.

This is the kind of telemetry that will be the main focus of this project assignment. It usually consists of basic measurements done by the spacecraft on its hardware components, and allows ground station operators to view vital physical parameters for mission planning such as on-board power usage and generation, temperatures, position and attitude. Another common component of spacecraft telemetry is the state of its software, with important flags, variables and counters often being transmitted to let the operators know what it is doing. An example of a display showing this type of telemetry from a CubeSat is seen in Figure 1.

Satellite telemetry visualisation systems commonly start at the satellite and end at a ground station, as seen in the first step in Figure 2, where one or more operators can view telemetry from the satellite and potentially send commands back. The HYPSO mission at NTNU wants to extend this to make telemetry data more readily accessible to team members and potentially external users outside the ground station using Open MCT, as seen towards the right in Figure 2.

3.2 NTNU HYPSO

The HYPSO mission is primarily a science-oriented technology demonstrator. It will enable low-cost and high-performance hyperspectral imaging and autonomous on-board processing that fulfil science requirements in ocean colour remote sensing and oceanography.

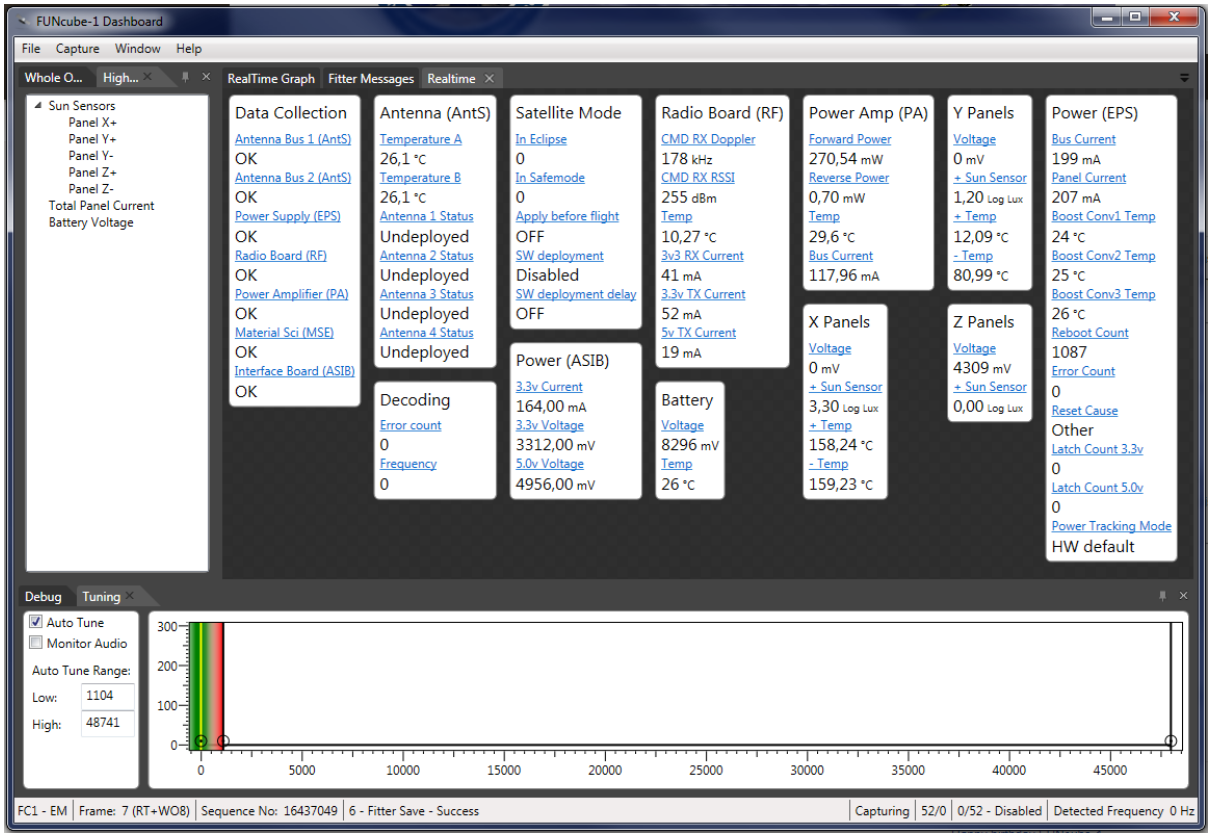
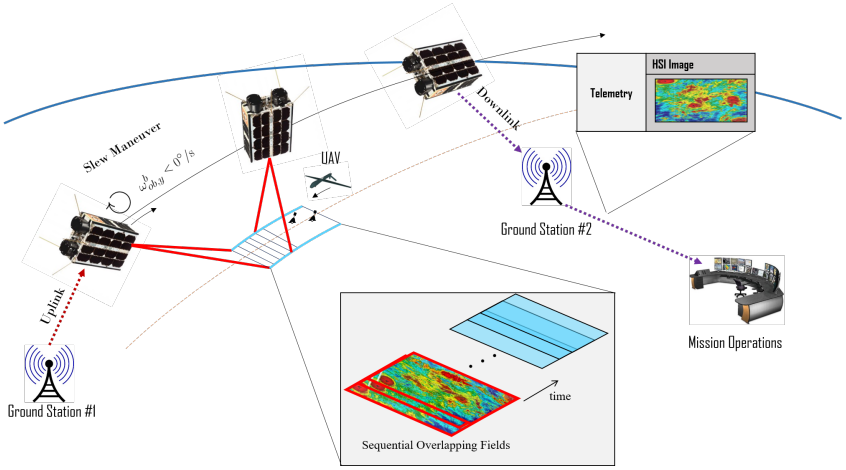


Figure 1: Telemetry display from the FUNcube-1 educational CubeSat [4]



HYPSO will be the first SmallSat launched by NTNU, with launch planned for Q4 2020 followed by a second mission later. These are part of a vision to provide a constellation of remote-sensing SmallSats, adding a space-based asset platform to the multi-agent architecture of UAVs, USVs, AUVs and buoys that have similar ocean characterisation and monitoring objectives.

The satellite bus and launch provider for the mission is NanoAvionics, which also provides various other services such as telemetry storage and telecommand tools through their Mission Control Software package.

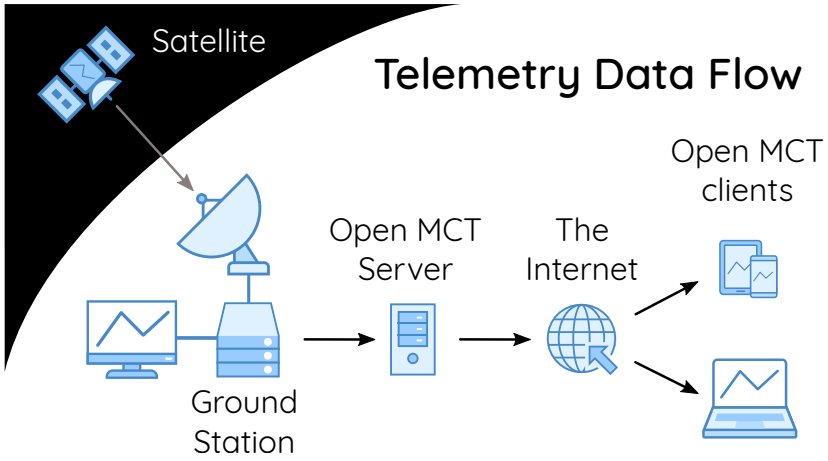


Figure 2: Data flow in the proposed Open MCT-based telemetry visualisation system

3.3 Open Mission Control Technologies

Open Mission Control Technologies (hereafter referred to as "Open MCT") is a new mission control framework for visualisation of various types of data inside a web browser, both on mobile and desktop devices. It is actively developed as open source software at NASA's Ames Research Center, in collaboration with the Jet Propulsion Laboratory.

It is very flexible and extensible, allowing for many different types of data to be integrated and easily accessible on one single website for mission planning and telemetry data analysis. It reduces the need for mission operators to switch between many different applications to view all necessary data. [5, 10]

Another large advantage of Open MCT as a telemetry visualisation system is that it allows end users to customise and share their own telemetry views without complex programming. This opens up for quick adaptation to unpredicted use cases that would commonly require significant time investment by programmers in more traditional telemetry visualisation systems; instead, Open MCT allows end users to access all its data how and when they want. This is enabled by the many plugins available by default in Open MCT [1], providing multiple ways to display data using graphs, lists and tables, in addition to providing data export functionality right out of the box. [9]

Plugins are also the main way for developers to add new functionality to Open MCT, such as adding a new telemetry source to display. Where this telemetry is provided from and over which protocols is not an explicit part of the Open MCT specification, but plugins are instead expected to register providers - special JavaScript functions, with specified inputs and outputs - for certain data types. These specify how Open MCT may request and receive that data, and how it can display it.

A high-level overview of the providers implemented as part of a typical Open MCT plugin for a custom telemetry source is the composition provider, object provider and telemetry provider. The composition provider defines where each value is located in the object tree on the left-hand side in Figure 4. The object provider essentially maps metadata for each value to a JavaScript object Open MCT can understand (see Source Code 3), which allows it to use values received from a telemetry provider - either in real-time or on request - to display a graph or other visualisation.

3.4 Existing implementations

There are few current users of Open MCT which make their implementations available to the public, but it is used widely within NASA and JPL for both space-based and terrestrial applications, with some of the more recognisable names being the Mars 2020 rover Perseverance, Mars Cube One, ICESat 2 and the Cold Atom Laboratory. LightSail 2 is an example of a high-profile non-NASA/JPL external user of

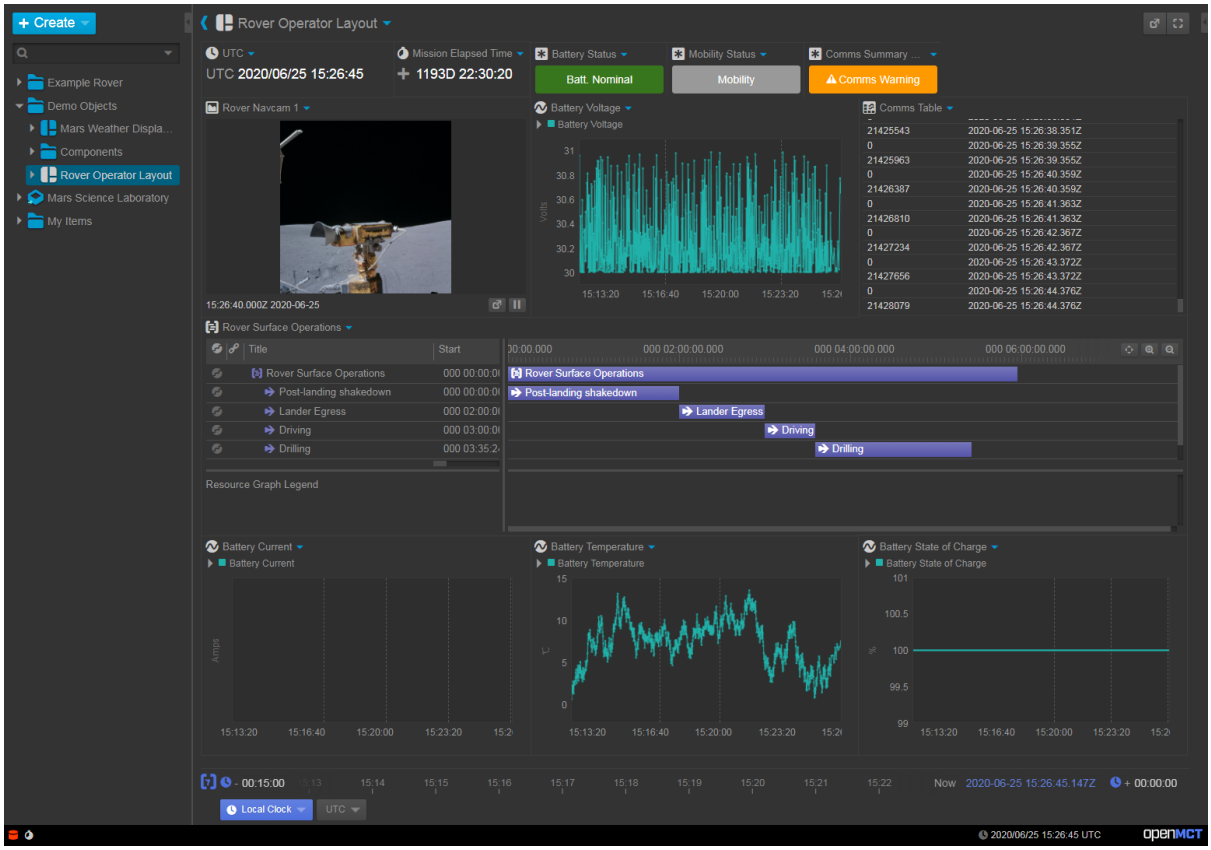


Figure 3: Sample Open MCT-based telemetry view

Open MCT. [2]

Some of the more well-documented recent Open MCT implementations will be briefly introduced below, with most detail being given to the Open MCT project's official tutorial which is the common starting point for all of them.

3.4.1 Open MCT tutorial

This is the current reference Open MCT implementation, and the easiest way to get started with the framework as the current documentation can be somewhat lacking or confusing to start with from scratch.

Node.js and Express is used to implement a telemetry web server that simulates a simple spacecraft. It provides telemetry for it as JSON over HTTP for historical data, and over WebSocket for realtime data.

In Open MCT a plugin (called the "dictionary plugin") is used to load the metadata specifying how the each data point in the telemetry should be displayed and acquired from the server. Two separate plugins provide the actual implementation of the HTTP and WebSocket clients that interface with the aforementioned web server.

3.4.2 CloudTurbine

CloudTurbine - a NASA-supported data streaming service - has made a prototype interface for Open MCT to test various aspects of how their service can be used with it.

Their prototype is a direct fork of the Open MCT tutorial repository. They have developed what they

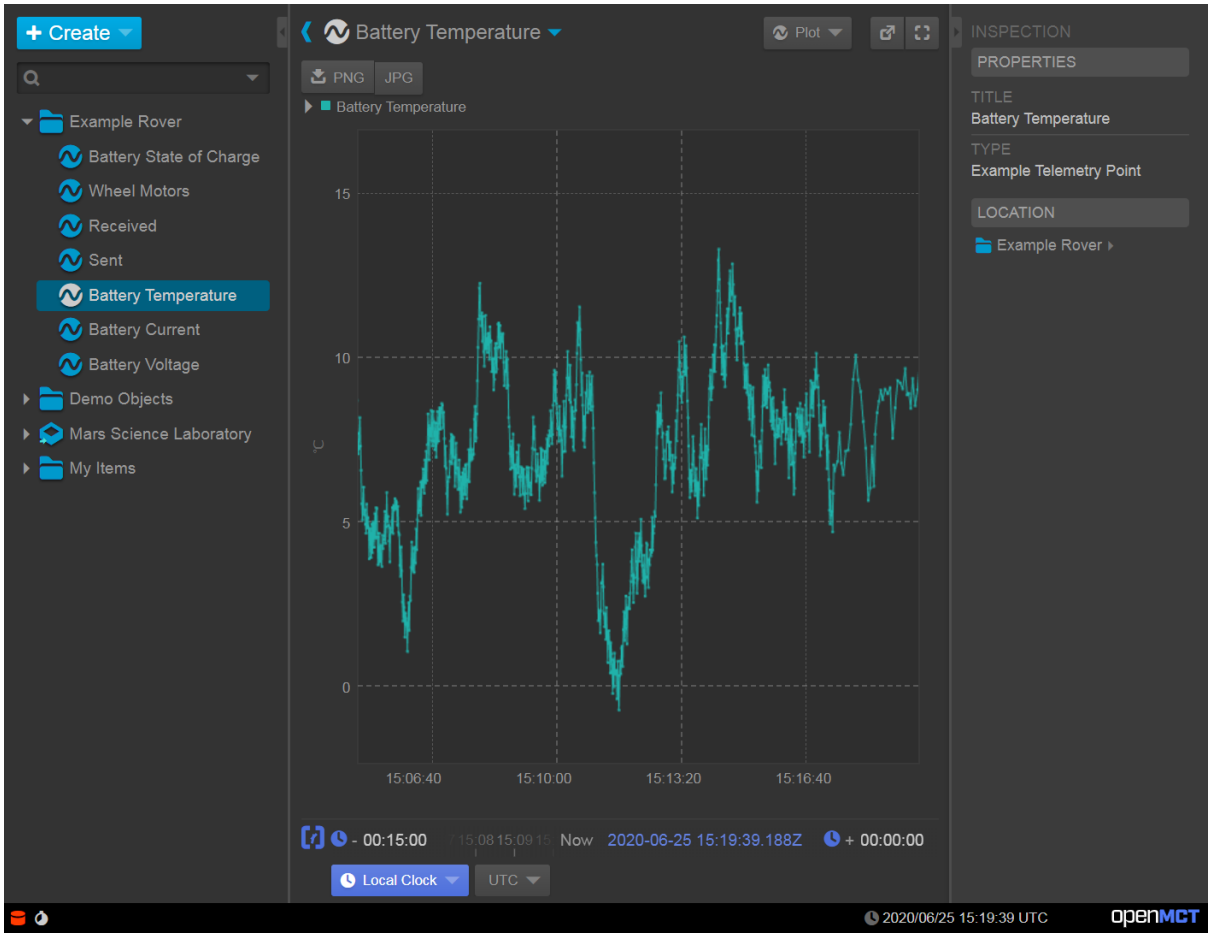


Figure 4: Sample Open MCT-based telemetry view, single value

call a "telemetry generator layer" in Express between the CloudTurbine service and Open MCT, with the generator layer querying their backend for new data and supplying it to Open MCT as JSON over a HTTP or WebSocket connection. [12]

3.4.3 ROSMCT

ROSMCT uses Open MCT to provide data and telemetry visualisation for Robot Operating System (ROS) robots. It consists of three distinct components, with a Rosbridge node (which provides a JSON API for ROS) running on the ROS instance to collect data from, a web server that collects data from Rosbridge, and finally an Open MCT plugin that allows the data to be displayed in Open MCT.

The web server in between Open MCT and the Rosbridge node does not have any local telemetry storage, and only provides access to realtime data from the ROS instance. [11]

3.4.4 Other implementations

A select few other interesting public uses of Open MCT that unfortunately couldn't get a full description in this report can be found in the list below.

1. https://bergie.iki.fi/blog/nasa-openmct-iot-dashboard/ - Uses Open MCT to display IoT device data

2. https://github.com/hudsonfoo/kerbal-openmct - Directly requests data from a Telemachus server instance inside KSP from an Open MCT plugin, no dedicated server/middleware

3. https://github.com/SDRPLab/yamcs-openmct-plugin - Directly requests data from a Yamcs server instance in an Open MCT plugin, no dedicated server/middleware

4 Problem and Scope

The assignment for this project was fairly open and to some degree exploratory. It has therefore been necessary to collect some information about what was wanted from a telemetry processing and display system for HYPSO, such as which types of telemetry it should handle. This information has been collected and summarised in the sections below, and will be referenced later to guide the design and implementation process.

4.1 Inputs

The initial main source of data for the system and Open MCT will be stored historical telemetry data from a PostgREST web server hosted by NanoAvionics. Received telemetry values can be retrieved from this service as C-style packed structs stored in hexadecimal strings, along with some metadata, inside a JSON object, as seen in Source Code 1. The telemetry returned from this server before HYPSO is launched comes from a ground-based test satellite. [7]

This server places fairly severe limitations on where it can be accessed from, with manually distributed API keys and IP address whitelisting being required before it can be used.

```
{
    "epsa_id": 18193,
    "epsa_init_ts": "2020-02-05T10:23:37.510788",
    "epsa_complete_ts": "2020-02-14T09:07:29.333605",
    "epsa_session_id": 3,
    "epsa_download": true,
    "epsa_file_id": 66,
    "epsa_gs_id": 1,
    "epsa_entry_nr": 88569,
    "epsa_entry_data":
    ↪  "\\x80febcd39891d741200c0...[414 characters trimmed]...f7f7f7f02000000"
}
```

Source Code 1: Sample EPS telemetry datum from NanoAvionics' PostgREST server (JSON)

The data string (as seen in `epsa_entry_data` in Source Code 1) needs to be unpacked using a provided C struct definition (see Source Code 2) before the data can be sorted or indexed by time, due to info about the time the telemetry applies to not being stored in the metadata sent with the string: It only contains the times for when the transmission associated with the telemetry datum was initiated and completed, not the time the telemetry data itself applies to. These times may be significantly different, since the HYPSO satellite does not have continuous contact with a ground station.

```
typedef struct
{
    double timestamp;
    uint32_t uptimeInS;
    uint32_t gs_wdt_time_left_s;
    uint32_t counter_wdt_gs;
    uint16_t mpptConverterVoltage[4];
    uint16_t curSolarPanels[8];
    uint16_t vBatt;
    uint16_t curSolar;
    uint16_t curBattIn;
    uint16_t curBattOut;
```

```
    uint16_t curOutput[18];
    uint16_t AOcurOutput[2];
    uint16_t outputConverterVoltage[8];
    uint8_t outputConverterState;
    uint32_t outputStatus;
    uint32_t outputFaultStatus;
    uint16_t outputOnDelta[18];
    uint16_t outputOffDelta[18];
    uint8_t outputFaultCnt[18];
    int8_t temp[14];
    uint8_t battState;
    uint8_t mpptMode;
    uint8_t batHeaterMode;
    uint8_t batHeaterState;
} eps_telemetry;
```

Source Code 2: EPS telemetry struct definition (C) from [6]

Other inputs, such as image data and processed payload telemetry data, will likely also need to be integrated at a later time to allow accessing them through Open MCT. This is due to the flexibility it offers with regards to navigating and displaying data, and the convenience of eventually having all the data output from the satellite accessible in one place. This means that the implementation will need to be fairly flexible with regards to what inputs it accepts; raw telemetry from NanoAvionics will not be the only data input to the system in the long-term, which needs to be accounted and planned for from the beginning.

4.2 Outputs

Open MCT needs a certain minimum of output data from the implemented system to work, in addition to custom JavaScript plugins to map these outputs to inputs Open MCT can understand. The implemented system must provide past telemetry values plus present telemetry values to these plugins with a reasonably short delay after they are received, and also provide value metadata so Open MCT can know how to display them.

```
{
    "identifier":
    {
        "namespace": "mctdepot.taxonomy",
        "key": "eps.vBatt"
    },
    "name": "Battery voltage",
    "type": "mctdepot.telemetry",
    "telemetry":
    {
        "values":
        [
            {
                "key": "value",
                "name": "Value",
                "hints": {
                    "range": 1
                },
                "units": "mV"
```

```
            },
            {
                "key": "utc",
                "source": "timestamp",
                "name": "Timestamp",
                "format": "utc",
                "hints": {
                    "domain": 1
                }
            }
        ]
    },
    "location": "mctdepot.taxonomy:eps.rootfolder"
}
```

Source Code 3: Open MCT telemetry value metadata sample for `vBatt`

A sample of the practical minimum amount of metadata Open MCT needs to display a visualisation for one telemetry value (here `vBatt`, the battery voltage, as stored in Source Code 2) is shown in Source Code 3. This illustrates that there is a fair bit of work that needs to be done to go from input to output in the required implementation, at least for telemetry from the NanoAvionics server.

Another important note regarding the implementation's output is that other groups within HYPSO have also expressed a desire for access to unpacked telemetry data, which means that there should be a well-documented way to request or export its output data outside Open MCT.

4.3 Requirements

A concise overview of the interpreted requirements the implementation should fulfil is shown below.

4.3.1 Functional Requirements

1.0 Should display telemetry from NanoAvionics (NA) telemetry web server

1.1 Should display historical telemetry data from NA

1.2 Should display near-realtime telemetry data from NA

2.0 Should display telemetry from other sources

3.0 Should deliver or export processed telemetry outside Open MCT

4.0 Should be modular and expandable

5.0 Tests should have 75% or better code coverage

4.3.2 Non-Functional Requirements

6.0 Integrating new telemetry sources should require minimal extra work

Note: "Minimal extra work" here implies that it should be well-documented how to add new telemetry sources, and that contact with new code and the need to understand implementation details of Open MCT should be minimised. The implementation should abstract away the connection to Open MCT from the user that is implementing the new telemetry source.

7.0 Accessing telemetry data from outside Open MCT should require minimal extra work

Note: This implies exports should be easy to acquire, and in a format that's easy to read and parse.

8.0 Response times should be below one second for all frontend actions

9.0 End-to-end delay should be less than one minute

Note: This means that the time between new telemetry being available from NanoAvionics to it showing up in Open MCT should be less than one minute.

10.0 Implementation should introduce a minimum of new programming languages, concepts and frameworks to the HYPSO project, to assist with code adoption and maintenance by existing developers

5 Design and Implementation

5.1 Introduction

With an overview of the necessary inputs and outputs of the system, and the requirements placed on its functionality and performance, it is possible to develop a draft for the design and implementation. This version will be referred to as version 0, or v0 for short.

From the requirements and restrictions on access to the NanoAvionics telemetry server, it is clear that the system will need to have local storage to store telemetry data to decouple Open MCT from this server. Giving every device that Open MCT should be available on direct access to the NanoAvionics server is not feasible, due to the time needed to manually approve the IP address for each device and issue new authentication keys. Their server is an entity that is to some degree unknown, and does not have any specific guarantees on availability or performance.

Since Open MCT will need to be hosted by a web server of some kind to be accessible in a web browser, implementing the required telemetry storage and parsing mechanisms on this web server would also aid with meeting the requirements for telemetry data availability outside Open MCT (see FR3.0 and NFR7.0) by providing it over HTTP.

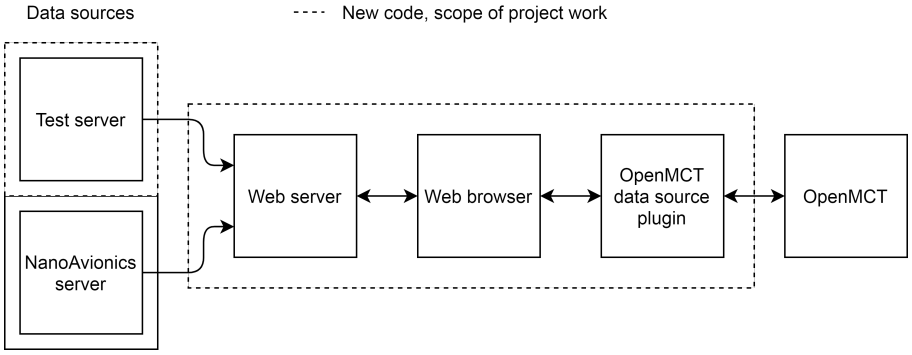


Figure 5: Early system block diagram

A simple high-level system block diagram for such a system is shown in Figure 5, showing each component and the connections between them. The test server is an alternate data source that provides similar output to the NanoAvionics server, to aid with testing and development, as the NA server may not be available at all times making an alternate telemetry data source an essential component.

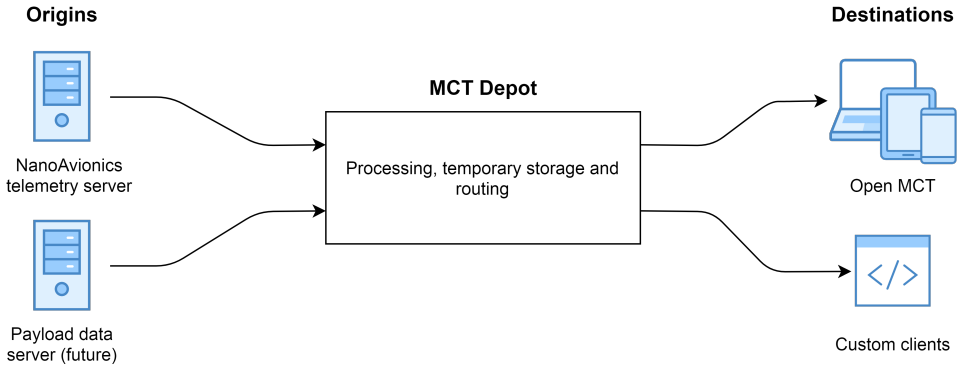


Figure 6: MCT Depot overview

To make it easier to distinguish between new and existing code, the system that is to be implemented

inside the dashed lines in Figure 5 has been given the name *MCT Depot*, or just *Depot* for short. The name comes from structure and function of the system resembling that of a train or bus depot, in that it processes and temporarily stores incoming telemetry data from various origins before routing it on to Open MCT or another destination, as seen in Figure 6.

5.2 Language and framework selection

To implement Depot as outlined above, it was as consequence of NFR10.0 decided to use JavaScript (in the form of ECMAScript 6) for the backend since it's already used in the Open MCT frontend. This avoids introducing more than one new programming language for the HYPSO project to support in relation to this system, and encourages code reuse between the frontend and backend.

5.3 Proposed system architecture

Seeing as Depot should be independent of the telemetry data source - it shouldn't matter to the destinations whether the telemetry is coming directly from NanoAvionics' server, a file-based backup, or another server providing some kind of relevant data - it was natural to split it into four more or less independent parts: The telemetry fetching system, the telemetry processing subsystem, the data management and storage system, and the telemetry serving system. The data management and storage system is the link between the otherwise independent telemetry fetching and telemetry serving systems.

It was decided to use Node.js running an Express-based web app to host and manage each of these subsystems. Express was chosen since it is one of the most used and well-documented web server frameworks for Node.js, and has already been used in multiple other successful implementations of Open MCT telemetry servers, including the current official Open MCT server implementation tutorial.

Based on the block diagram above a data flow diagram for the full system was created to get an idea of the types and amounts of data that would be going between each subsystem. This can be found in Figure 7.

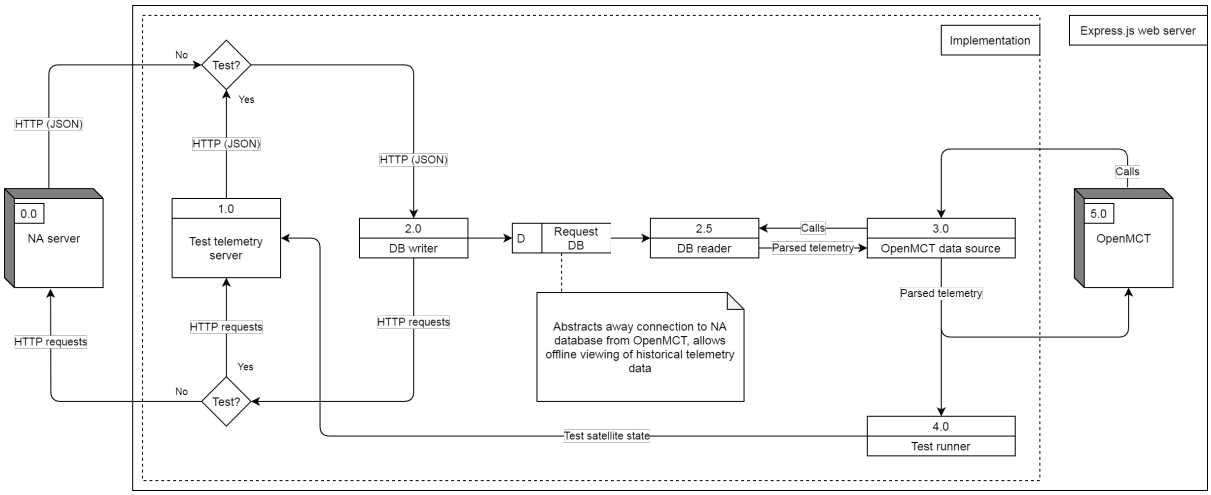


Figure 7: v0 Level 0 Data Flow Diagram

After this a complete class diagram with the full planned extent of modules, with descriptions of their local variables and methods was created. This will be referenced actively in the section below outlining the functionality and decisions behind the implementation of each module. This can be found in Figure 8, with cutouts for each subsystem found in figures 9 to 13.

Attempting to read Figure 8 directly is not recommended, and it is mostly included as a way to get a quick overview of the connections between the subsystems.

5.3.1 Shared modules

These provide simple but important utility functions, such as managing the current configuration of the system and allowing it to be imported/exported to a file, plus providing a system log.

The class diagram for these modules can be found in Figure 9.

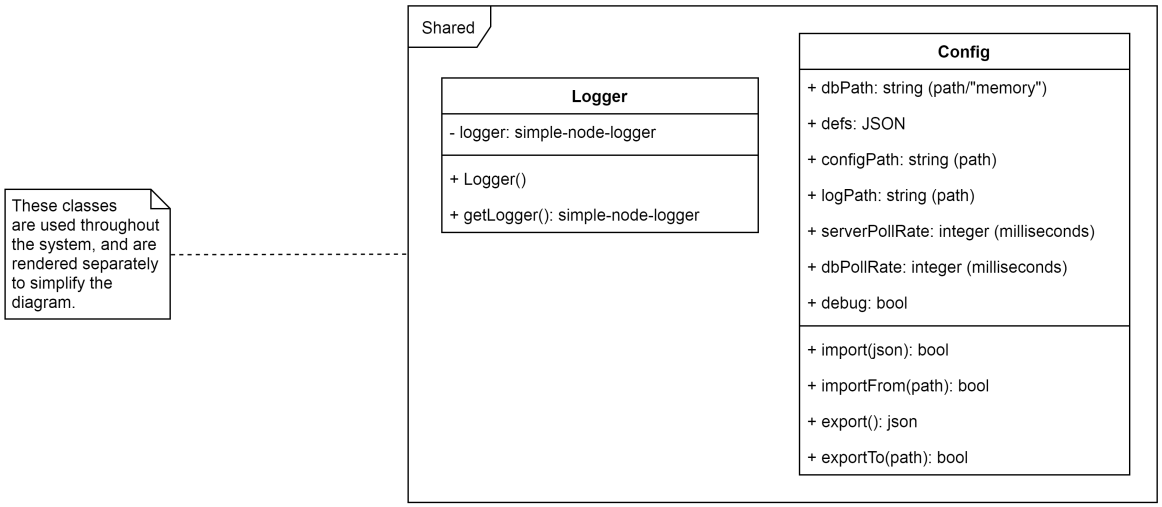


Figure 9: v0 Class Diagram: Shared Modules

The `Config` module provides a reasonable set of default settings, so that external configuration files only need to contain the settings to be changed or added if the defaults do not suit the intended use of the system.

5.3.2 Telemetry fetching subsystem

This subsystem provides an interface for getting telemetry data from external origins. It is implemented by the `TelemetryFetcher` class, which is realised for various specific external sources as `FileTelemetryProvider`, `TestTelemetryProvider`, and `NATelemetryProvider`. As the names imply, these get telemetry data from a file, a test spacecraft, and NanoAvionics' server respectively.

The default implementation of all of these are simple polling services that check their sources at a regular interval to see if there is any new data; if so this data is written to the connected `DbManager`. A connection to a `TelemetryDefinition` via `TelemetryParser` may be required if the telemetry type, timestamp or any metadata to be stored cannot be determined directly from the received data without it being unpacked and parsed first. These modules will be introduced in the next section.

The class diagram for these modules can be found in Figure 10.

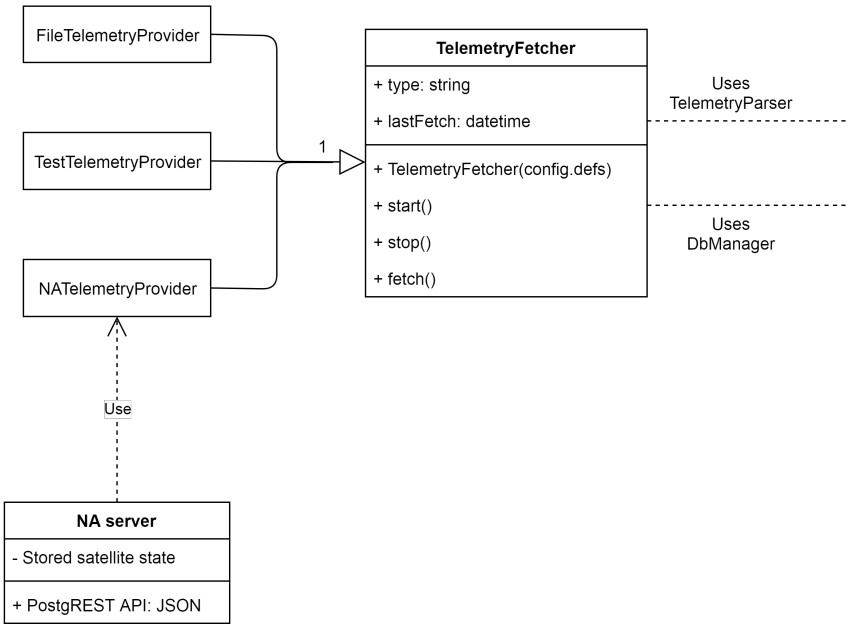


Figure 10: v0 Class Diagram: Telemetry Fetching

5.3.3 Telemetry processing subsystem

The top-level module here, `TelemetryParser`, provides methods for configuring, parsing and unpacking various types of telemetry data, with a general class `TelemetryDefinition` that has a special subclass `NATelemetryDefinition` that provides a quick way to set up the `TelemetryDefinition`s for each type of telemetry we get directly from NanoAvionics' server.

The data that is returned from NanoAvionics' using their PostgREST-based HTTP API is in the form of JSON with some basic metadata and, more importantly, a string that contains a packed C-style struct. In `NATelemetryDefinition` we use a third-party module called ezStruct that allows us to quickly convert this to JSON based directly on the struct header definitions we get in text files from NanoAvionics, speeding up the process of implementing and updating the telemetry definitions greatly (as required by NFR6.0). Updating or adding a new `TelemetryDefinition` for parsing telemetry data from NanoAvionics' should be as simple as downloading the struct header file and updating Depot's configuration to include it.

The class diagram for this subsystem can be found in Figure 11.

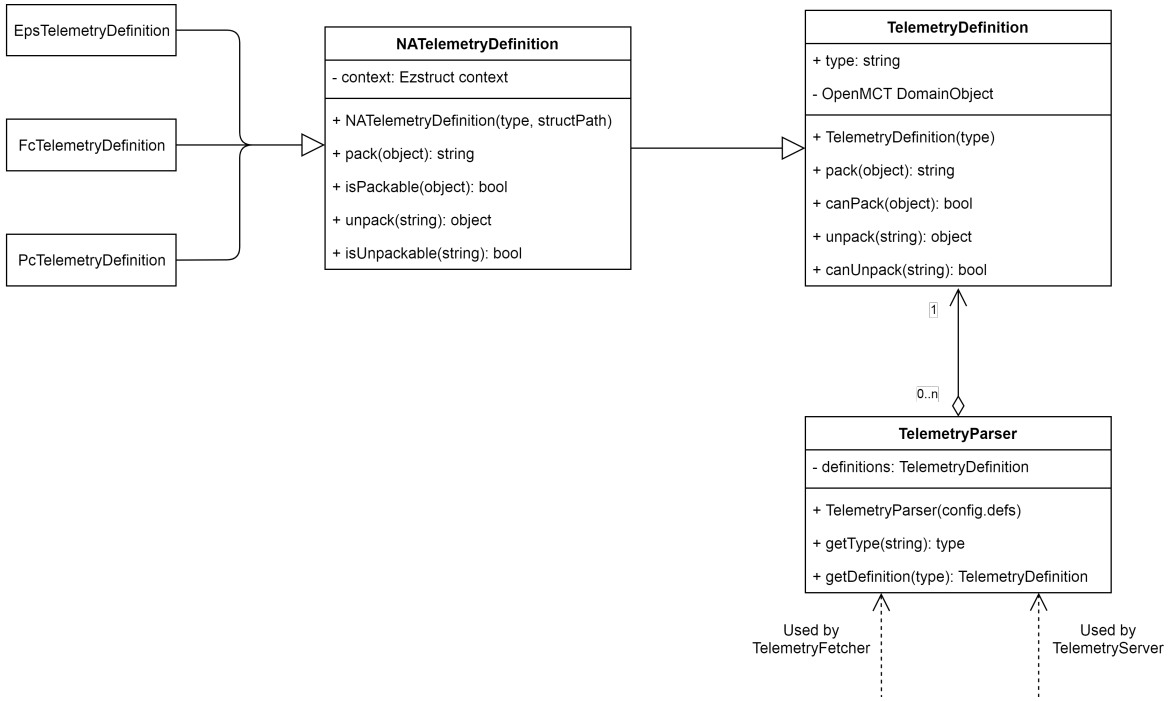


Figure 11: v0 Class Diagram: Telemetry Parsing

5.3.4 Data management and storage subsystem

The next major part is the `DbManager`, which handles storing data (usually from a `TelemetryFetcher`) and reading data (usually from a `TelemetryServer`). The interface here is fairly simple, as the proposed implementation just has a single table that can be indexed by timestamp and type (which maps to a `TelemetryDefinition` that can be used to unpack the data stored for that type).

The class diagram for this subsystem can be found in Figure 12.

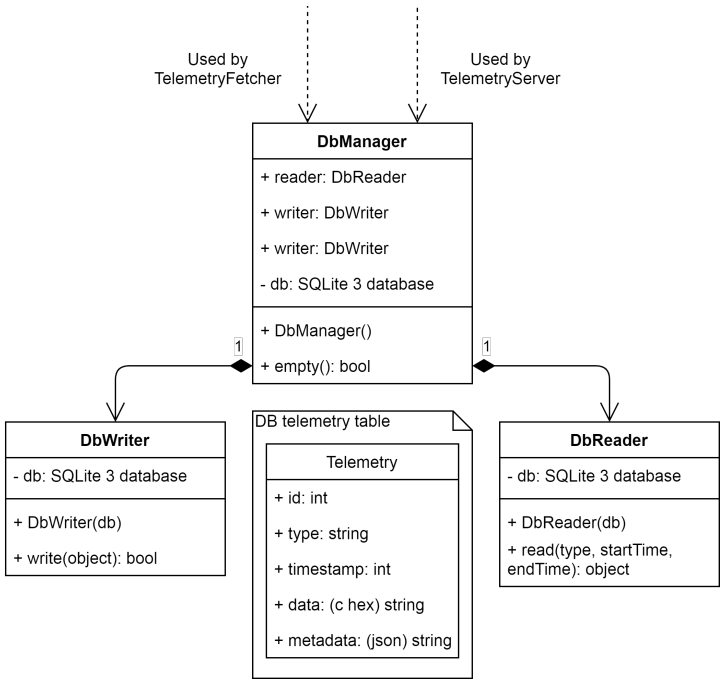


Figure 12: v0 Class Diagram: Database Management

To host the database it was decided to use SQLite; the fairly simple structure of the database used works well with it, and the large benefits it provides in portability (for "offline" work or quick backups) due to the database being stored in a single file made it a good candidate compared to other more complex database solutions.

Using some kind of object-relational mapping framework - such as Sequelize - was investigated but eventually dropped, due to the low level of complexity in the proposed database schema and the queries required. There was also a lack of demand for many of the features that these frameworks come with, such as support for generating database migrations to preserve data if the database schema is updated. The database functions more as a cache or buffer for processed telemetry data than a primary source of it, with new copies of the raw telemetry data being available and backed up in other locations (albeit with higher latency and request complexity until it is imported into the database again).

5.3.5 Telemetry serving subsystem

This subsystem hosts and manages one or more HTTP/WebSocket servers for getting data for a specified `TelemetryDefinition` from a `DbManager`. The HTTP servers (implemented by `HistoryServer`) are used for allowing a service to request historical data for a period, while the WebSocket servers (implemented by `RealtimeServer`) allow a service to subscribe to real-time data updates without having to continuously poll a HTTP server.

The class diagram for this subsystem can be found in Figure 13.

5.3.6 Open MCT client plugins

These allow Open MCT to get and subscribe to telemetry data from Depot through the telemetry serving subsystem described above, and allows mapping this data to various user-defined visualisations.

The preliminary class diagram for this subsystem can be found in Figure 13.

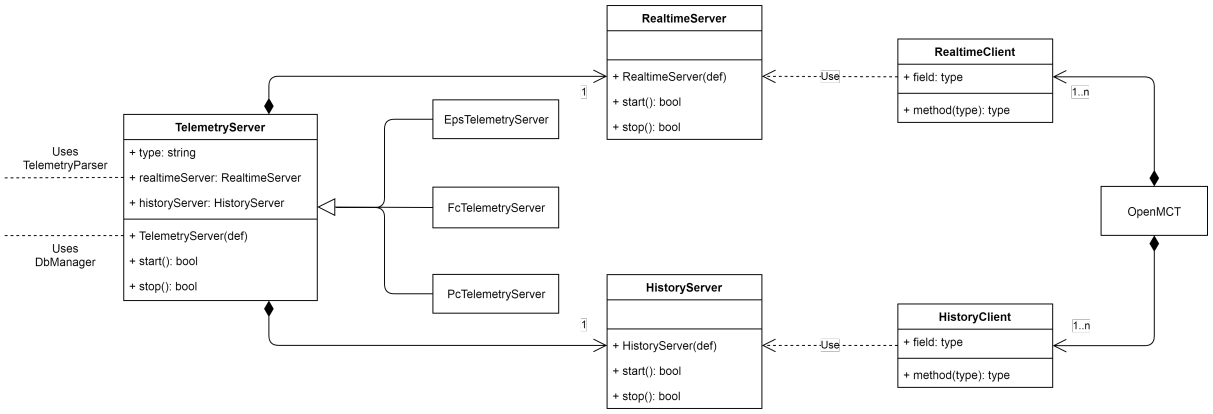


Figure 13: v0 Class Diagram: Telemetry Server and Client

5.4 Code standards

It was decided to keep the code style and syntax similar to what's already encouraged and used by the Open MCT project in their contribution guidelines [3] for both the backend and new frontend code. A simplified version of Open MCT's ESLint configuration file is used to enforce this code style throughout the project, where the simplifications are the removal of configuration specific to external dependencies not used by Depot.

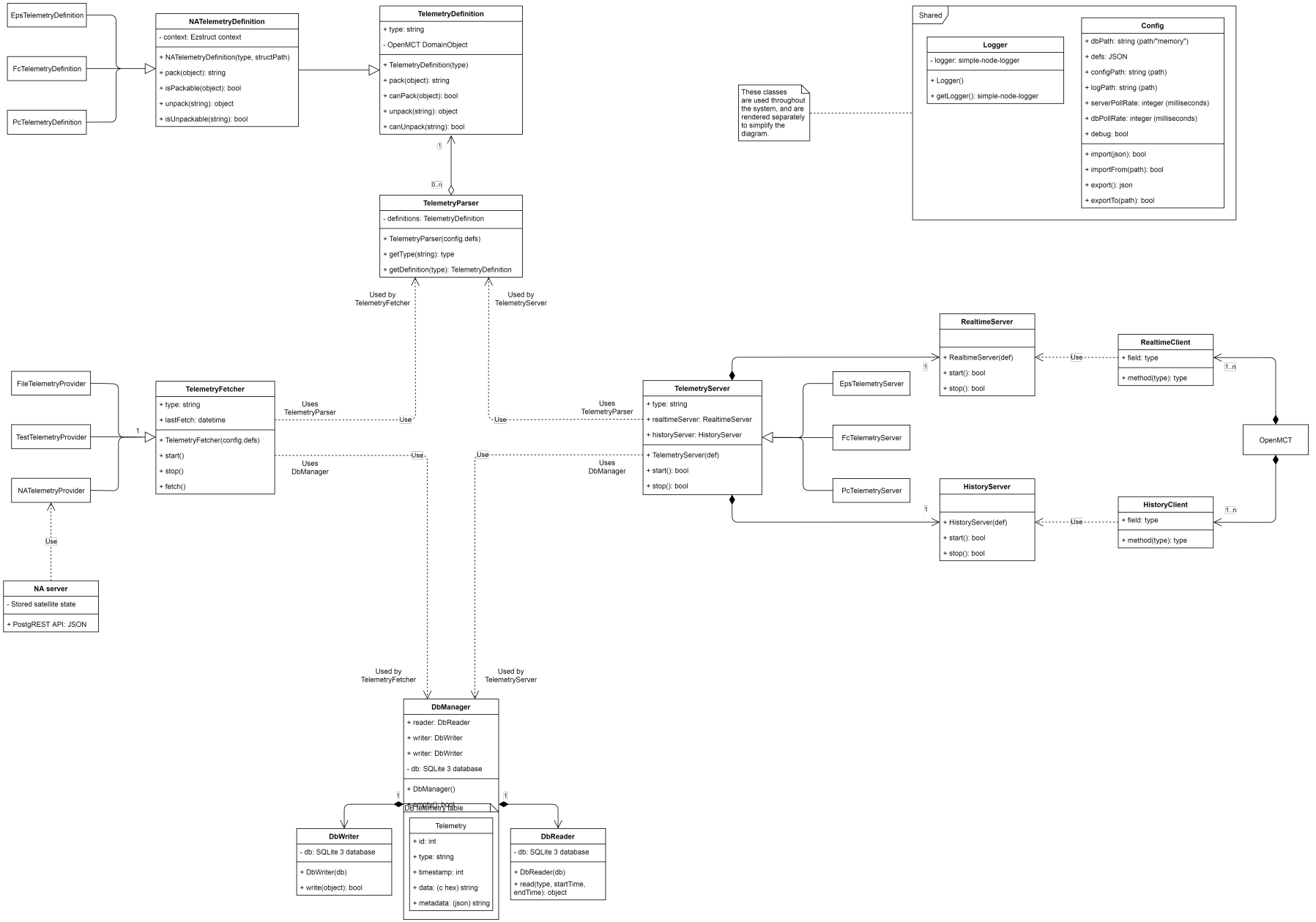


Figure 8: v0 Class Diagram Overview

6 Results and Discussion

The latest version of the source code for the implemented system is available at https://github.com/NTNU-SmallSat-Lab/mct-depot. Release version 1.0 is the subject of the results and discussion in the rest of this report unless otherwise is specified. The documentation for version 1.0 is also included in appendix C.

Figure 14: Sample Open MCT view

The v1.0 release allows viewing of historical and realtime telemetry data for the HYPSO satellite Electrical Power Supply and Flight Computer, with a custom sample view of this data shown in Figure 14. This data may currently be loaded from a JSON file exported from the NA PostgREST API, and in theory by connecting the server directly to the PostgREST API, although this has only been tested against a mock server due to time and computer lab access constraints under the COVID-19 pandemic.

6.1 Design change summary

All of the modules in the system required some level of change or adaptation during the development process from the initial design proposal from the previous section. Certain classes (such those for the logger and payload controller telemetry) were removed from version 1.0 due to insufficient time or test data, while others (such as the `DbPoller` and `DbHasher`) were added to account for unplanned changes in the implementation.

The telemetry serving and client subsystems in particular were not described in much detail in the design draft, and have had multiple new classes added, such as a new HTTP server-client pair for providing system configuration and telemetry metadata to Open MCT.

The database-related classes also saw multiple changes due to the initial specification for them being too simple or slow in practice.

A more detailed description of the changes and the reasoning behind them may be found below.

6.2 Final system architecture

An overview of the class diagram for the implemented system may be found in Figure 15. Attempting to read this directly is again not recommended due to its size, and it is mostly included as a way to get a quick overview of the connections between the subsystems.

6.2.1 Shared modules

The `Config` class did not see any major changes except for the removal of the methods for exporting the configuration, as these were not found to be particularly useful in practice for version 1.0.

The `Logger` class was removed due to it being found to not be required for version 1.0. It will be more relevant to implement once Depot is ready to run on a production server.

The updated class diagram for these modules can be found in Figure 16.

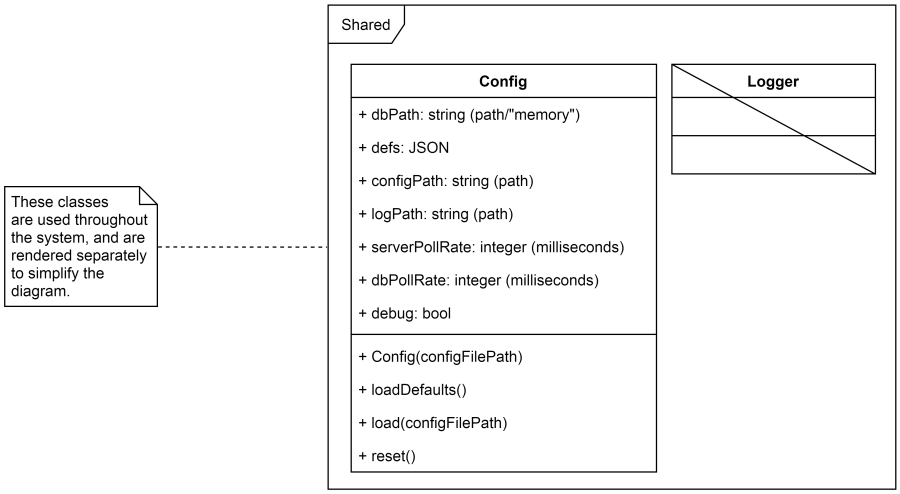


Figure 16: v1.0 Class Diagram: Shared Modules

6.2.2 Telemetry fetching subsystem

The main changes in the `TelemetryFetcher` class is the `fetch()` method being split into `run()` (which calls the following methods at regular intervals), `fetch()` (which gets and parses data from a given telemetry source) and `store()` (which stores the parsed data if it isn't in the database already). This was done to decouple the running and storing steps from the fetching process, to make it easier to only redefine the methods required when adding a new telemetry source.

The updated class diagram for these modules can be found in Figure 17.

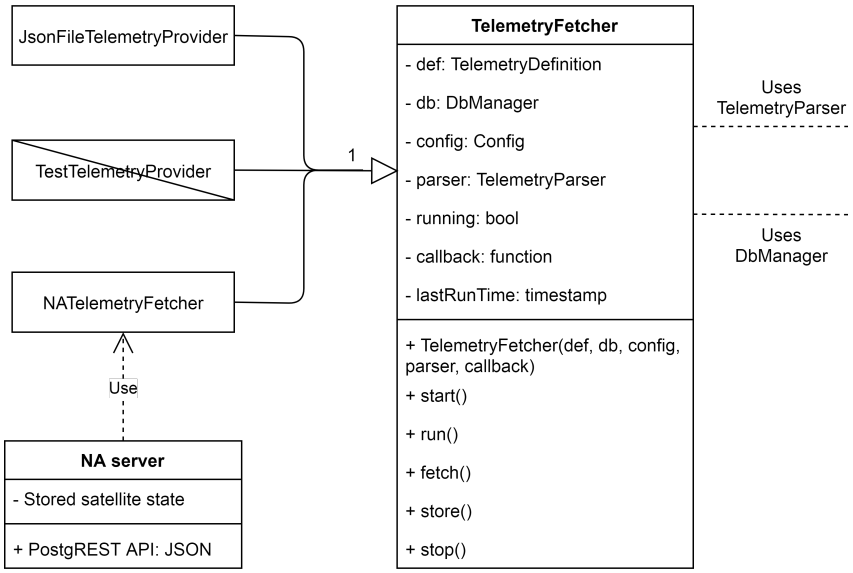


Figure 17: v1.0 Class Diagram: Telemetry Fetching

6.2.3 Telemetry processing subsystem

The main change is the addition of a new `parse(telemetryString)` method, which takes telemetry data directly from a fetcher and parses it to a valid telemetry object; this operation has no inverse, as the original telemetry string is included in this object and directly stored in the database in case it is required later, either due to changes in the definition requiring it to be parsed again or if another service than Open MCT needs access to a full copy of it. The field is also hashed and used to quickly check if an incoming telemetry sample already exists in the database.

Another new method is the `GetMctMetadata()` method, which returns a list of telemetry value metadata ready for use in Open MCT.

The updated class diagram for this subsystem can be found in Figure 18.

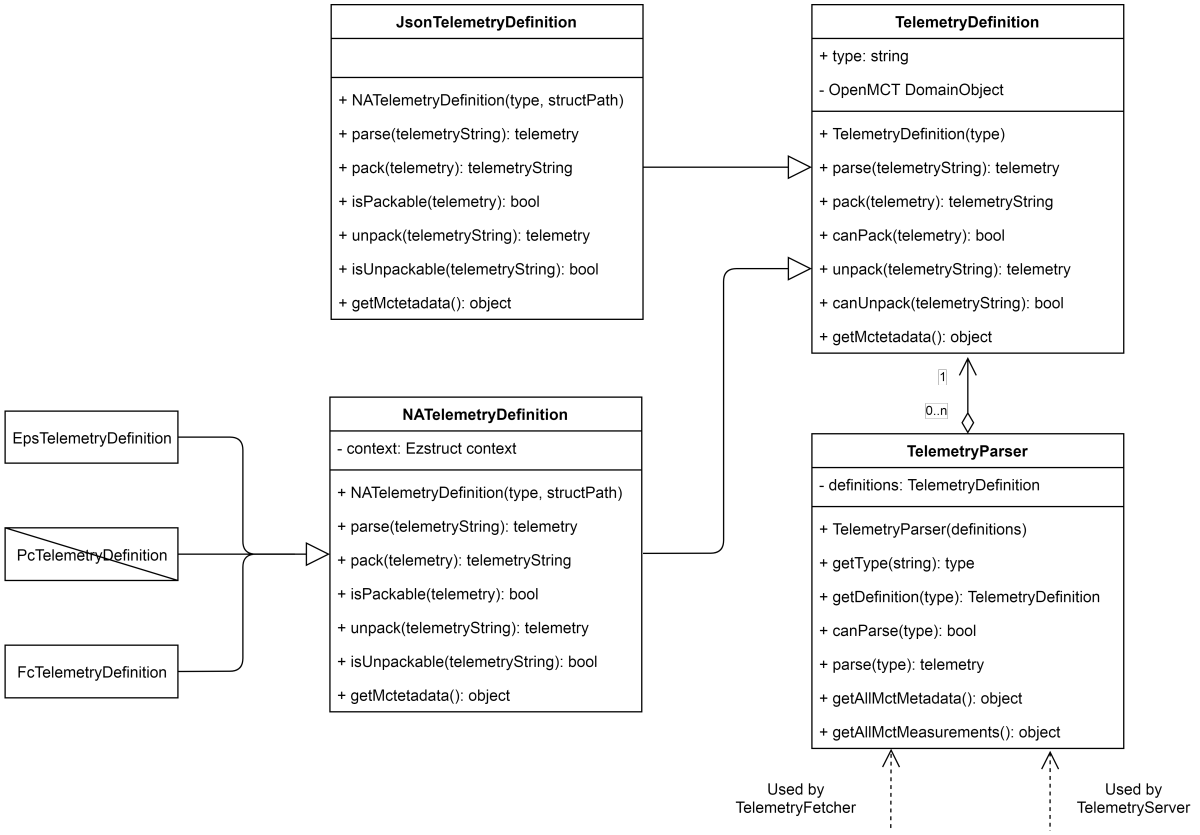


Figure 18: v1.0 Class Diagram: Telemetry Parsing

6.2.4 Data management and storage subsystem

The data management and storage subsystem needed fairly large changes to meet with the requirements (specifically FR1.2, NFR8.0 and NFR9.0) due to very poor performance of the initially proposed architecture.

Especially the process of checking if a telemetry point was already stored in the database when running a `TelemetryFetcher` was very slow, resulting in system startup times with large JSON inputs containing approximately 1000 telemetry points being on the order of multiple minutes (or worse). This could also lead to unacceptable performance with regards to the requirements on the response time of the system to incoming new data, since the system per now checks all inputs against the database to verify that they are new.

The solution to this was the implementation of a new column on the database table containing a SHA-1 hash (chosen for its speed and fairly short length, see [8]) of each stored telemetry point, generated using a new class called `DbHasher`. This column is used to create another index for the `Telemetry` table, eliminating the need to load and iterate over every single point in the database to check if a point already exists.

The performance improvements from this change were fairly substantial, with `TelemetryFetcher` in many cases running multiple orders of magnitude faster. This was most noticeable with the EPS telemetry test dataset, which both had more points (18193 points vs. the 6719 points the FC dataset) and about twice the byte length per telemetry point.

A comparison of the before/after performance of the data management and storage subsystem benchmarked using a `JsonFileTelemetryFetcher` at system startup can be found in figures 19 to 21, with "DB init" referring to the case where the target database is empty.

The raw data used to create the performance graphs may be found in appendix A.

The updated class diagram for this subsystem can be found in Figure 22.

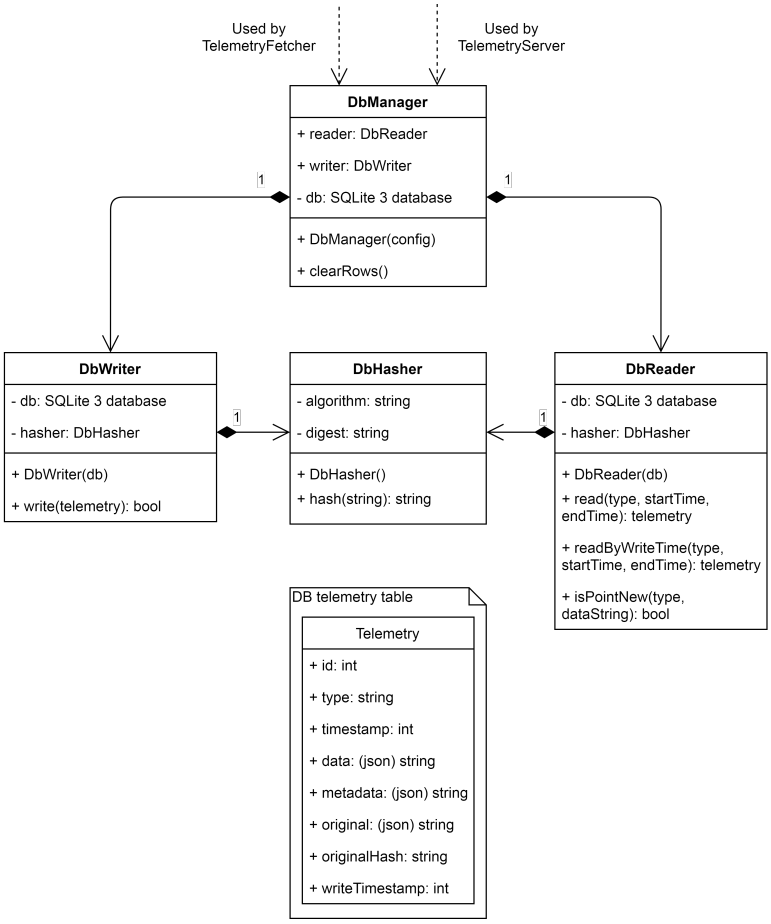


Figure 22: v1.0 Class Diagram: Database Management

6.2.5 Telemetry serving subsystem

The updated class diagram for this subsystem can be found in Figure 23. The main change here is the addition of the `ConfigServer` class, which provides system configuration from a `Config` instance and telemetry value metadata from `GetAllMctMetadata()` on a `TelemetryParser`.

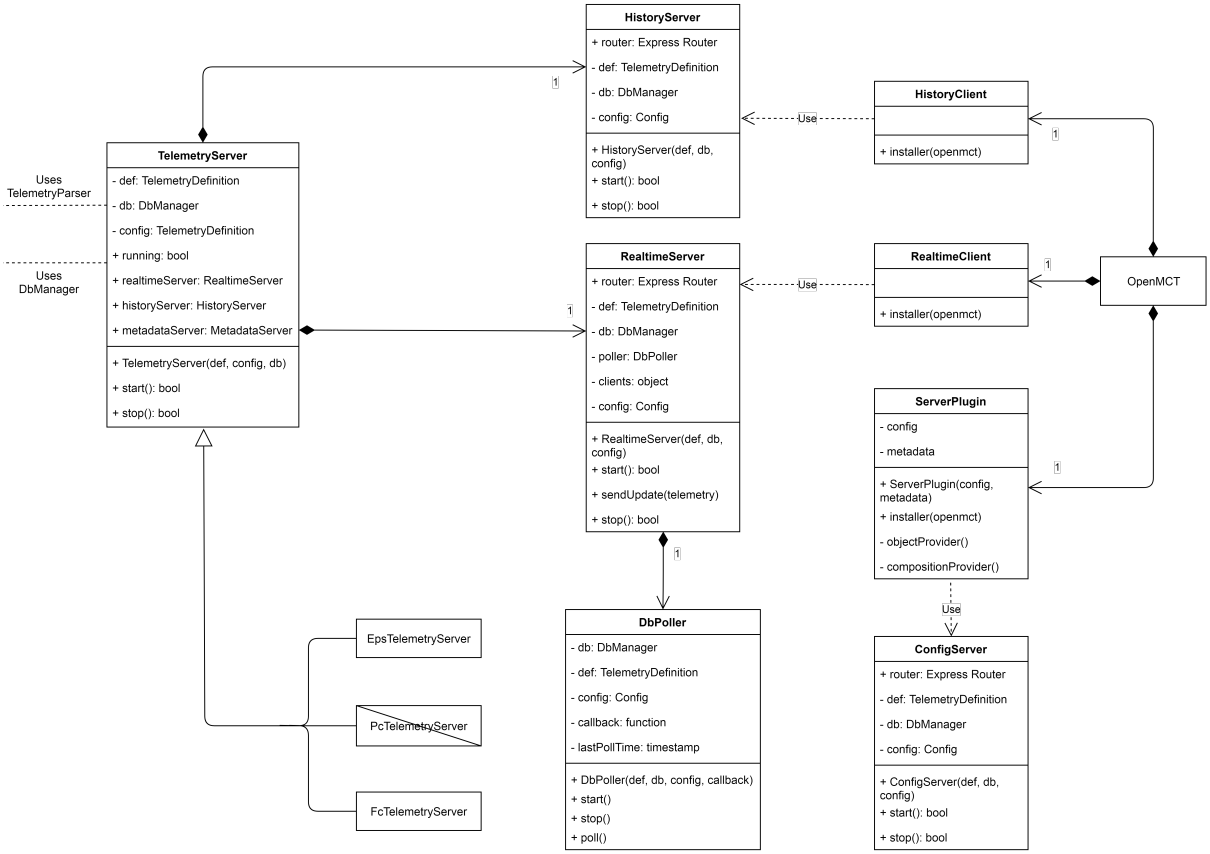


Figure 23: v1.0 Class Diagram: Telemetry Server and Client

6.2.6 Open MCT client plugins

These were underspecified in the initial system design proposal due to much of the implementation details being unknown, as with the telemetry serving subsystem that they connect to.

The final configuration has three plugins, with `ServerPlugin` being responsible for loading telemetry metadata from a `ConfigServer` and populating the Open MCT object tree (shown on the left-hand side in Figure 14). After this `RealtimeClient` and `HistoryClient` are loaded, which as the names imply connect to a `RealtimeServer` and `HistoryServer` respectively to provide telemetry data to Open MCT.

The updated class diagram for this subsystem can be found in Figure 23.

6.3 Test coverage

The test coverage requirements specified in FR5.0 were not quite met, but version 1.0 still has above 50% unit test coverage for most files and metrics. The main hole in test coverage is for the Open MCT client plugins, due to them being finished last and functioning significantly different to all other parts of the system. Making good tests for them requires a significant level of insight into how Open MCT works.

A short summary of the system's test coverage may be found in table 1, with a full Jest test coverage report available in appendix B.

Table 1: System Test Coverage Summary

File or directory	Statements	Branches	Functions	Lines
All files	68.4%	50.4%	56.1%	68.2%
Plugins	15.9%	45.0%	4.76%	16.1%
Telemetry definitions	77.0%	62.8%	62.16%	76.4%
Web server	61.1%	11.1%	66.7%	61.1%
Database	69.9%	31.3%	69.6%	70.3%
Telemetry services	75.5%	75.0%	60.4%	75.1%

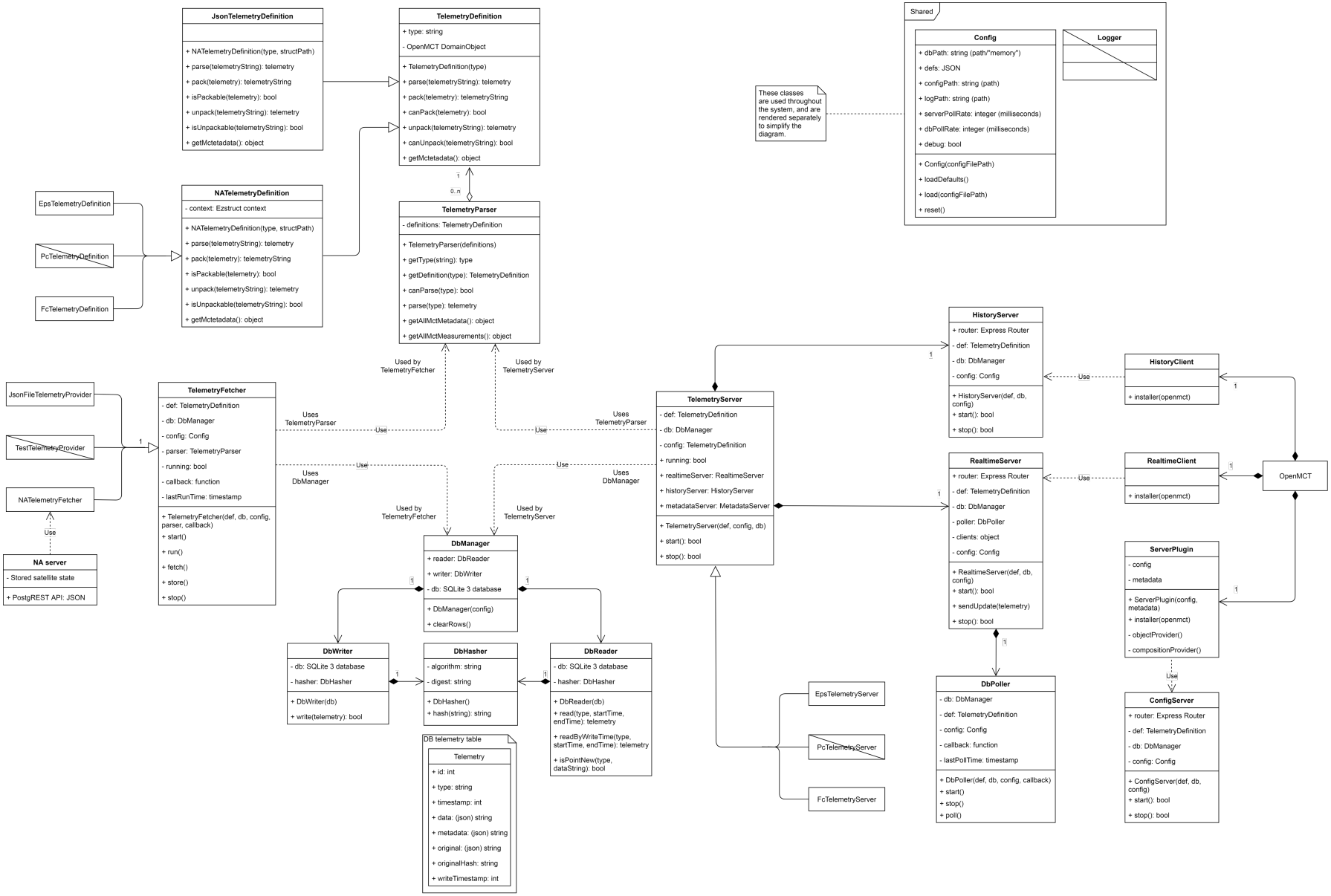


Figure 15: v1.0 Class Diagram Overview

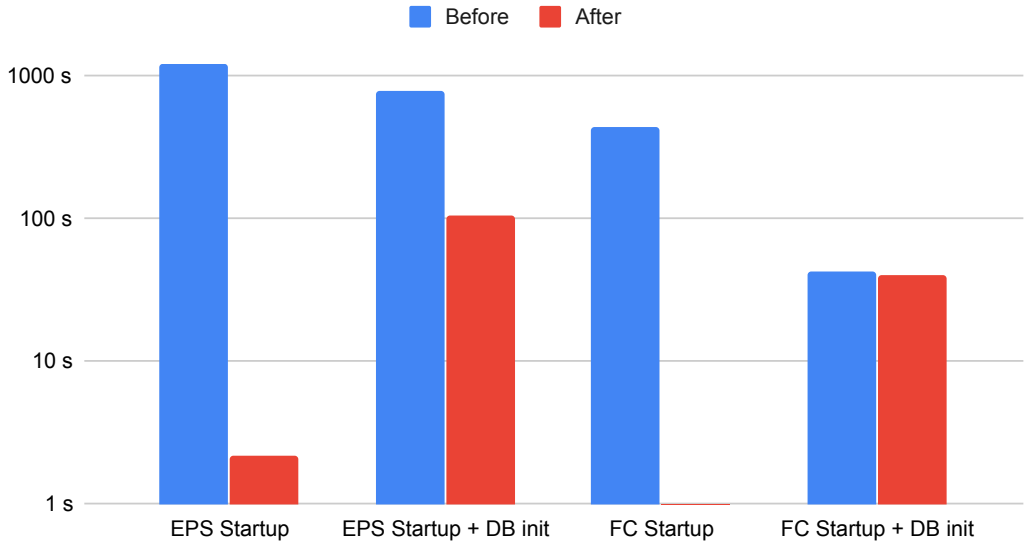


Figure 19: System startup: Total load time

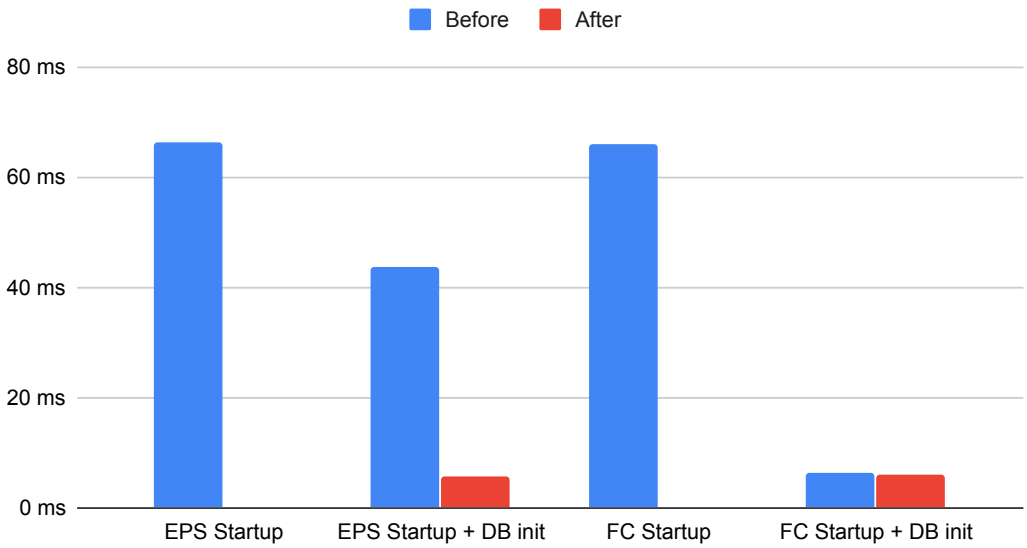


Figure 20: System startup: Load time per telemetry point

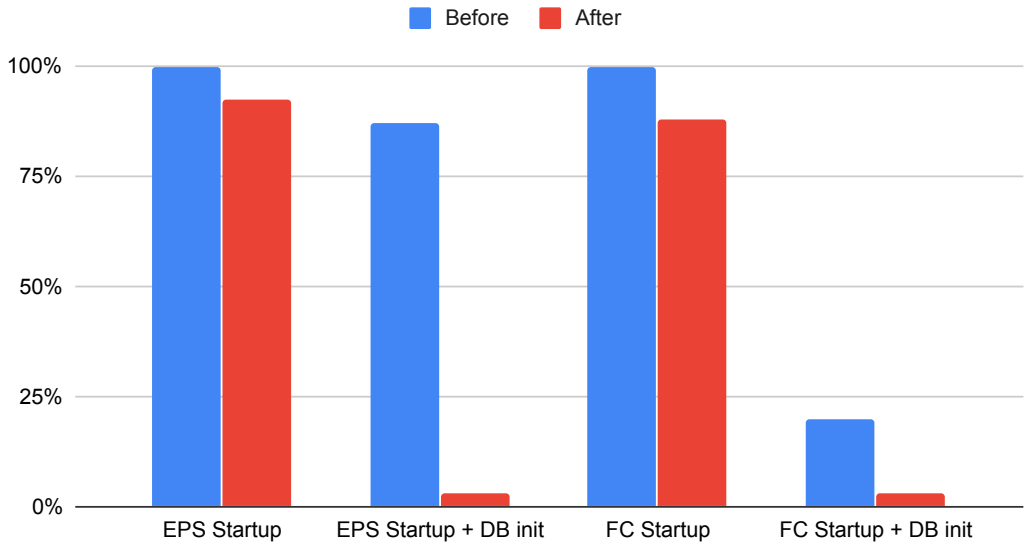


Figure 21: System startup: Percentage of load time spent in `isPointNew()`

7 Conclusion

The proposed and subsequently implemented system (named MCT Depot) seems to meet the functional and non-functional requirements specified in Section 4 fairly well, and should work in practice as a good basis for a mission planning and satellite troubleshooting tool. However, further work and user testing is needed to verify its performance as a practical telemetry visualisation system in real-world situations, and to make it ready for practical use.

The process of implementing it could have been streamlined by doing further research about the exact input and output requirements Open MCT has before finalising the design specification draft and class diagrams. Lack of detailed knowledge here was the factor that led to most of the unforeseen design changes and delays in implementing the first functional version of Depot. Part of this is likely due to it being somewhat difficult to go from the Open MCT API specification - the main up-to-date source of documentation as of June 2020 - to a working Open MCT-based data visualisation system.

8 Future Work

The system can and should be expanded upon before it can be used as a practical telemetry visualisation system. A list of proposed potential changes and improvements to the v1.0 release of the system may be found in the list below.

- Create custom views with relevant telemetry data for different real-world situations and users
- Set up and document production server
- Write documentation for using Open MCT
- Further development of NanoAvionics server connection
- Add remaining NanoAvionics telemetry definitions
- Add more data sources
- Improve handling of NanoAvionics array telemetry values
- Improve test coverage
- Code refactoring and cleanup
- Add server-side data storage for Open MCT views and data changes
- Add processing, storage and display of calculated/derived telemetry values

A more detailed list may also be found in the issue tracker on the project's GitHub repository.

Bibliography

[1] Henry, Andrew: Open MCT plugins. (2020). – URL https://nasa.github.io/openmct/plugins/

[2] Henry, Andrew: Who is using Open MCT. (2020). – URL https://nasa.github.io/openmct/whos-using-open-mct/

[3] Henry, Andrew ; Woeltjen, Vic: Contributing to Open MCT. (2020). – URL https://github.com/nasa/openmct/blob/master/CONTRIBUTING.md

[4] Limebear, Richard W.: The AMSAT-UK FUNcube Handbook. (2013). – URL https://funcubetest2.files.wordpress.com/2010/11/funcube-handbook-en_v13.pdf

[5] Motroc, Gabriela: Getting started with NASA's Open MCT framework: Familiarity with JavaScript is required. (2018). – URL https://jaxenter.com/nasa-open-mct-framework-interview-152243.html

[6] NanoAvionics: NA-EPS-001 - Electrical Power Supply Software. (2020)

[7] NanoAvionics: NA-PostgREST API-001 - PostgREST API Usage. (2020)

[8] Thompson, Chris: What is the fastest node.js hashing algorithm. (2017). – URL https://medium.com/@chris_72272/what-is-the-fastest-node-js-hashing-algorithm-c15c1a0e164e

[9] Trimble, Jay: A New Architecture for Visualization: Open Mission Control Technologies. (2017). – URL http://www.esa-esaw2017.eu/sites/default/files/ESAW%202017%20Visualization_Trimble%203%20.pdf

[10] Trimble, Jay ; Henry, Andrew: Building a Community of Open Source Contributors. (2018). – URL https://ntrs.nasa.gov/search.jsp?R=20180003354

[11] Watkins, Dex: ROSMCT (Robot Operating System Mission Control Technologies). (2019). – URL https://github.com/rosmod/rosmct

[12] Wilson, John P.: CT/Open MCT: Modular Streaming Data Display. (2017). – URL https://www.cloudturbine.com/wordpress/wp-content/uploads/2017/12/CT-and-Open-MCT-white-paper.pdf

Appendix

A Database performance test data

```
C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct>npm run start

> hypso-openmct@0.0.1 start C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct
> node server/run.js

server config available at http://localhost:8471/config
Starting fetcher for fc
Stored 0 new points in database out of 6719 loaded points in 443211.0252530575 ms
Used 442355.55814397335 ms on checking if new (65.83651706265417 ms/check on average), 0.998069842444475% of total time
Starting server(s) for fc
fc realtime available at ws://localhost:8471/fc/realtime
fc history available at http://localhost:8471/fc/history
fc metadata available at http://localhost:8471/fc/metadata
Starting fetcher for eps
Stored 0 new points in database out of 18193 loaded points in 1207896.1295369864 ms
Used 1206128.682959199 ms on checking if new (66.29630533497493 ms/check on average), 0.998536756154302% of total time
Starting server(s) for eps
eps realtime available at ws://localhost:8471/eps/realtime
eps history available at http://localhost:8471/eps/history
eps metadata available at http://localhost:8471/eps/metadata
OpenMCT available at http://localhost:8471
```

Figure 24: Old TelemetryFetcher initial load performance, existing database

```
C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct>npm run start

> hypso-openmct@0.0.1 start C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct
> node server/run.js

server config available at http://localhost:8471/config
Starting fetcher for fc
Stored 0 new points in database out of 6719 loaded points in 975.8694709539413 ms
Used 856.709468960762 ms on checking if new (0.12750550215221937 ms/check on average), 0.8778935036499329% of total time
Starting server(s) for fc
fc realtime available at ws://localhost:8471/fc/realtime
fc history available at http://localhost:8471/fc/history
fc metadata available at http://localhost:8471/fc/metadata
Starting fetcher for eps
Stored 0 new points in database out of 18193 loaded points in 2200.4794130325317 ms
Used 2034.0959947109222 ms on checking if new (0.11180651870010017 ms/check on average), 0.9243876505564246% of total time
Starting server(s) for eps
eps realtime available at ws://localhost:8471/eps/realtime
eps history available at http://localhost:8471/eps/history
eps metadata available at http://localhost:8471/eps/metadata
OpenMCT available at http://localhost:8471
```

Figure 25: New TelemetryFetcher initial load performance, existing database

```
C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct>npm run start

> hypso-openmct@0.0.1 start C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct
> node server/run.js

server config available at http://localhost:8471/config
Starting fetcher for fc
Stored 6719 new points in database out of 6719 loaded points in 42019.04043900967 ms
Used 8400.175191640854 ms on checking if new (1.2502121136539446 ms/check on average), 0.19991354166770295% of total time
Starting server(s) for fc
fc realtime available at ws://localhost:8471/fc/realtime
fc history available at http://localhost:8471/fc/history
fc metadata available at http://localhost:8471/fc/metadata
Starting fetcher for eps
Stored 18193 new points in database out of 18193 loaded points in 795428.4941670895 ms
Used 690794.1506624222 ms on checking if new (37.970326535613815 ms/check on average), 0.8684553743397988% of total time
Starting server(s) for eps
eps realtime available at ws://localhost:8471/eps/realtime
eps history available at http://localhost:8471/eps/history
eps metadata available at http://localhost:8471/eps/metadata
OpenMCT available at http://localhost:8471
```

Figure 26: Old TelemetryFetcher initial load performance, new database

```
C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct>npm run start

> hypso-openmct@0.0.1 start C:\Users\AUDUNVN\Desktop\git repos\hypso-openmct
> node server/run.js

server config available at http://localhost:8471/config
Starting fetcher for fc
Stored 6719 new points in database out of 6719 loaded points in 39798.381542921066 ms
Used 1214.3263865709305 ms on checking if new (0.18073022571378636 ms/check on average), 0.030511953991428642% of total time
Starting server(s) for fc
fc realtime available at ws://localhost:8471/fc/realtime
fc history available at http://localhost:8471/fc/history
fc metadata available at http://localhost:8471/fc/metadata
Starting fetcher for eps
Stored 18193 new points in database out of 18193 loaded points in 104157.0530539751 ms
Used 3067.5075364112854 ms on checking if new (0.16860921983242375 ms/check on average), 0.02945079038307349% of total time
Starting server(s) for eps
eps realtime available at ws://localhost:8471/eps/realtime
eps history available at http://localhost:8471/eps/history
eps metadata available at http://localhost:8471/eps/metadata
OpenMCT available at http://localhost:8471
```

Figure 27: New TelemetryFetcher initial load performance, new database

B Jest test coverage report

The report shown in Figure 28 was generated using `npm run test`.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	68.4	50.43	56.12	68.21	
client/plugins	15.87	45	4.76	16.13	
HistoryClient.js	20	75	0	20	5,7,10,11,13,14,19,26
RealtimeClient.js	12.5	75	0	12.5	5,7,10,11,13,15,17,18,20,23-25,30,37
ServerPlugin.js	16.22	25	8.33	16.67	4,9-12,15,22,23,26,28,29,32,33,36,46,54,58-61,63,79,81,82,84,90,92,94,100,107
defs	76.97	62.79	62.16	76.35	
JsonTelemetryDefinition.js	46.67	100	50	46.67	13,21,23,25,27,31,35,49
NATelemetryDefinition.js	80.65	54.29	75	79.78	97,112,119,141,142,144,145,149,150,153,154,157,158,161,162,165,166,226
TelemetryDefinition.js	40	100	14.29	40	13,17,21,25,29,33
TelemetryParser.js	91.18	100	83.33	91.18	66,72,82
server	61.11	11.11	66.67	61.11	
Server.js	55.32	11.11	50	55.32	46,50,52,54,55,58,60,62,64,65,67,68,71,72,74,75,78,79,81,82,85
StaticServer.js	100	100	100	100	
server/db	69.9	31.25	69.57	70.3	
DbHasher.js	100	100	100	100	
DbManager.js	70	50	66.67	70	15,57,59,60,62,63
DbPoller.js	76.92	50	66.67	76	44,48,50-53
DbReader.js	63.64	0	66.67	65.63	91,95,109,111,112,114-117,123,127
DbWriter.js	61.11	50	66.67	61.11	20,21,25,26,30,31,50
server/telemetry	75.51	75	60.42	75.13	
ConfigServer.js	84.62	100	60	84.62	30,31
HistoryServer.js	84.62	100	57.14	84	51,52,55,56
JsonFileTelemetryFetcher.js	100	100	100	100	
MetadataServer.js	83.33	100	60	83.33	29,30
MockNATelemetryServer.js	85.71	100	66.67	85.71	26,46,47
RealtimeServer.js	86.49	75	75	86.11	71-73,77,78
TelemetryFetcher.js	81.63	70	57.14	81.25	17,55,66,86,90,92-95
TelemetryServer.js	14.81	100	0	14.81	11-13,15,17-19,21,26,28-30,35,37-39,44,46-48,50-52
shared	100	100	100	100	
Config.js	100	100	100	100	

Test Suites: 3 failed, 17 passed, 20 total
Tests: 59 passed, 59 total
Snapshots: 0 total
Time: 4.216s
Ran all test suites.

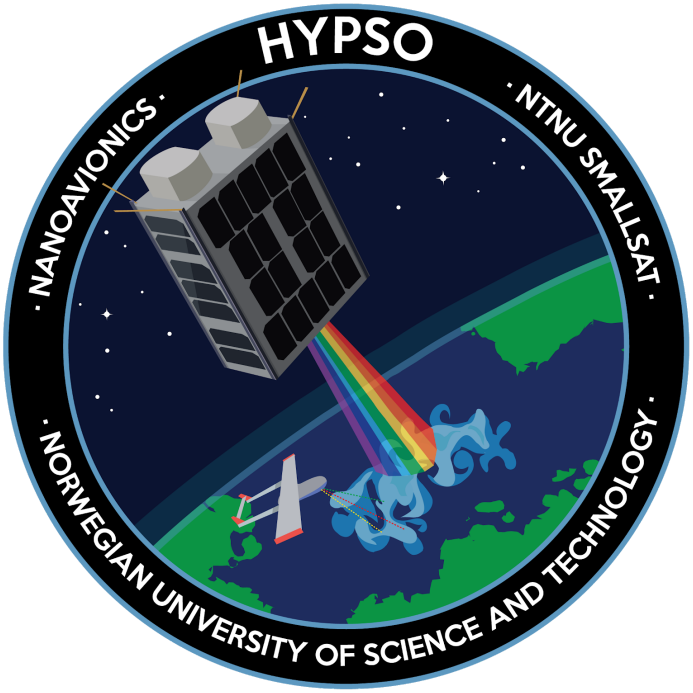
Figure 28: Test coverage report, v1.0

C HYPSO-DR-015 Open MCT Integration

This is the documentation for the 1.0 release of the system. The latest version may be found at https://github.com/NTNU-SmallSat-Lab/mct-depot/blob/master/docs/HYPSO-DR-015%20Open%20MCT%20Integration.pdf.

Open MCT Integration

HYPSO-DR-015



Prepared by: HYPSO Project Team
Reference: HYPSO-DR-015
Revision: 3
Date of issue: Date
Status: Preliminary
Document Type: Design Report

Table Of Contents

Table 1: Table of Changes

Rev.	Summary of Changes	Author(s)	Effective Date
0.1	*Initial Draft*	*J L Garrett*	
		R Birkeland	
0.2	*Document outline, title change*	Audun V. Nytrø	
0.3	*Documented system architecture and plan ahead*	Audun V. Nytrø	2020-03-27
0.4	*Adding new class diagrams and documentation for v1.0 release*	Audun V. Nytrø	2020-06-20

1 Overview

The HYPSO Mission will primarily be a science-oriented technology demonstrator. It will enable low-cost & high-performance hyperspectral imaging and autonomous onboard processing that fulfill science requirements in ocean color remote sensing and oceanography. NTNU SmallSat is prospected to be the first SmallSat developed at NTNU with launch planned for Q4 2020 followed by a second mission later. Furthermore, vision of a constellation of remote-sensing focused SmallSat will constitute a space-asset platform added to the multi-agent architecture of UAVs, USVs, AUVs and buoys that have similar ocean characterization objectives.

1.1 Purpose

The purpose of the HYPSO-DR-015 Open MCT Integration Design Report is to provide an overview of the planned system for viewing and browsing historical and live telemetry data using NASA's Open MCT software package.

1.2 Scope

This document covers the web server that fetches, unpacks and locally stores satellite telemetry data, which also hosts the Open MCT frontend. It also covers the code required for Open MCT to use the data provided by this server.

It does not cover any external users of the data used by the telemetry web server, such as the ground station software suite, but does describe how these users can interface with the telemetry web server to get historical and real-time data for use outside Open MCT.

1.3 Summary

This document describes the implementation details, usage and status of the system for displaying and analyzing telemetry data from the HYPSO spacecraft in Open MCT.

An Express-based web server running on Node.js with SQLite is currently being developed for hosting this system, which will allow users to get real-time and historical telemetry data in Open MCT, or using a separate HTTP or WebSockets client.

The document currently consists of the following:

- Chapter 2: System description, which contains implementation details and explanations for how the overall system is intended to work
- Chapter 3: System usage, which contains usage instructions and recommendations

- Chapter 4: Current status, which explains what work remains and the plan ahead for getting that work done

1.5 Applicable Documents

The following table lists the applicable documents for this document and work.

Table 2: Applicable Documents

ID	Author	Title
[AD01]	ECSS Secretariat	ECSS-M-ST-10C: Space management: Project planning and implementation (tailored)
[AD02]	Norwegian Space Agency	NSC-Payload-PAQA (tailored)
[AD03]		
[AD04]		
[AD05]		

1.6 Referenced Documents

The documents listed in Table 3 have been used as reference in creation of this document.

Table 3: Referenced Documents

ID	Author	Title
[RD01]	Mariusz Grøtte	Telemetry Format Spreadsheet https://docs.google.com/spreadsheets/d/1DCzhNcp5J5GWeUGTtZC0nzXlwu4XpLFEQB6uAN2jq2Y/edit#gid=0
[RD02]		
[RD03]		
[RD04]		
[RD05]		

2 System description

2.1 Quick Introduction to Open MCT

Open MCT - short for Open Mission Control Technologies - is a new mission control framework developed and used by NASA for visualization of various types of data inside a web browser, both on mobile and desktop devices.

It is very flexible and extensible, allowing for many different types of data to be integrated and easily accessible on one single website for mission planning and telemetry data analysis. It reduces the need for mission operators to switch between many different applications to view all necessary data.

More information can be found on Open MCT's official website: https://nasa.github.io/openmct/.

2.2 Overview

At its core, the system consists of a web server that gets data from NanoAvionics' server, stores it locally and hosts the Open MCT frontend along with code that allows Open MCT to understand the telemetry data format used by NanoAvionics. A summary and early version of this structure and its connections can be seen in the high-level system block diagram shown in figure 2.1.

The data is stored in a local database to abstract away the connection to NanoAvionics, allowing us to manipulate and access the data as we want without having to depend on their server being available, and to avoid constantly doing requests to it for data we've downloaded before. This reduces our load on their server, and allows us to easily support multiple instances of Open MCT (or other services that need telemetry data) running and accessing telemetry data simultaneously if required.

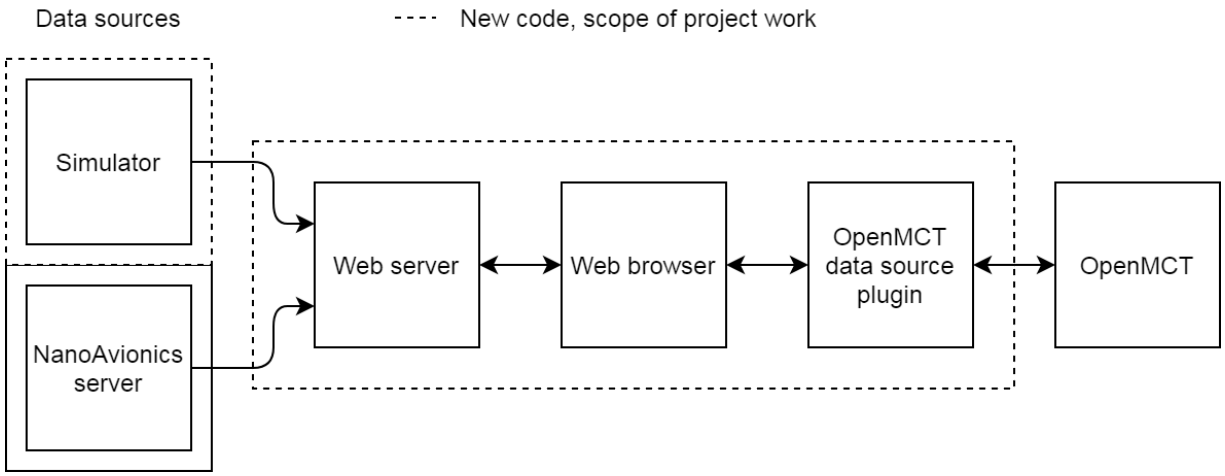


Figure 2.2 High-level system block diagram

2.3 Software architecture

To implement the system outlined above, it was decided to use JavaScript (in the form of ECMAScript 6) for the backend since it's already used in the Open MCT frontend, to avoid introducing more than one extra programming language for the HYPSO project to support in relation to this system, and to encourage code sharing between the frontend and backend.

Seeing as we want the system to be independent of the data source - whether it's directly to NanoAvionics' server, a file-based backup, or another server providing some kind of relevant data, it was natural to split the project into three more or less independent parts: The telemetry fetching system, the data management and storage system, and the telemetry serving system. The data management and storage system is the link between the otherwise independent telemetry fetching and telemetry serving systems.

Seeing as using JavaScript was wanted for the backend, it was decided to use Node.js running an Express-based web app to host and manage each of these subsystems. Express was chosen since it is one of the most used and well-documented web server frameworks for Node.js, and has already been used in multiple other successful implementations of Open MCT telemetry servers including the current official Open MCT server tutorial.

Based on the block diagram above a data flow diagram for the full system was created to get an idea of the types and amounts of data that would be going between each subsystem. This can be found in Figure 2.3.1.

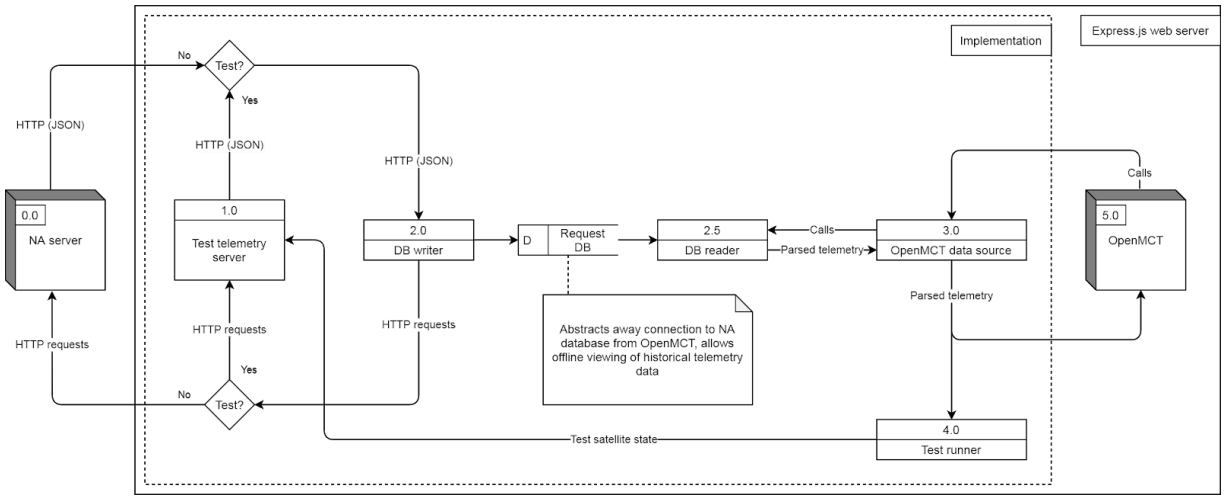


Figure 2.3.1 Level 0 Data Flow Diagram for the full system

After this a full class diagram with the full extent of modules and descriptions of their local variables and external methods was created; this will be referenced actively in the section below outlining the functionality and decisions behind the implementation of each module. This can be found in Figure 2.3.2, with cutouts for each subsystem found in Figures 2.3.3 to 2.3.7.

Attempting to read Figure 2.3.2 directly is not recommended, and it is mostly included as a way to get a quick overview of the connections between the subsystems.

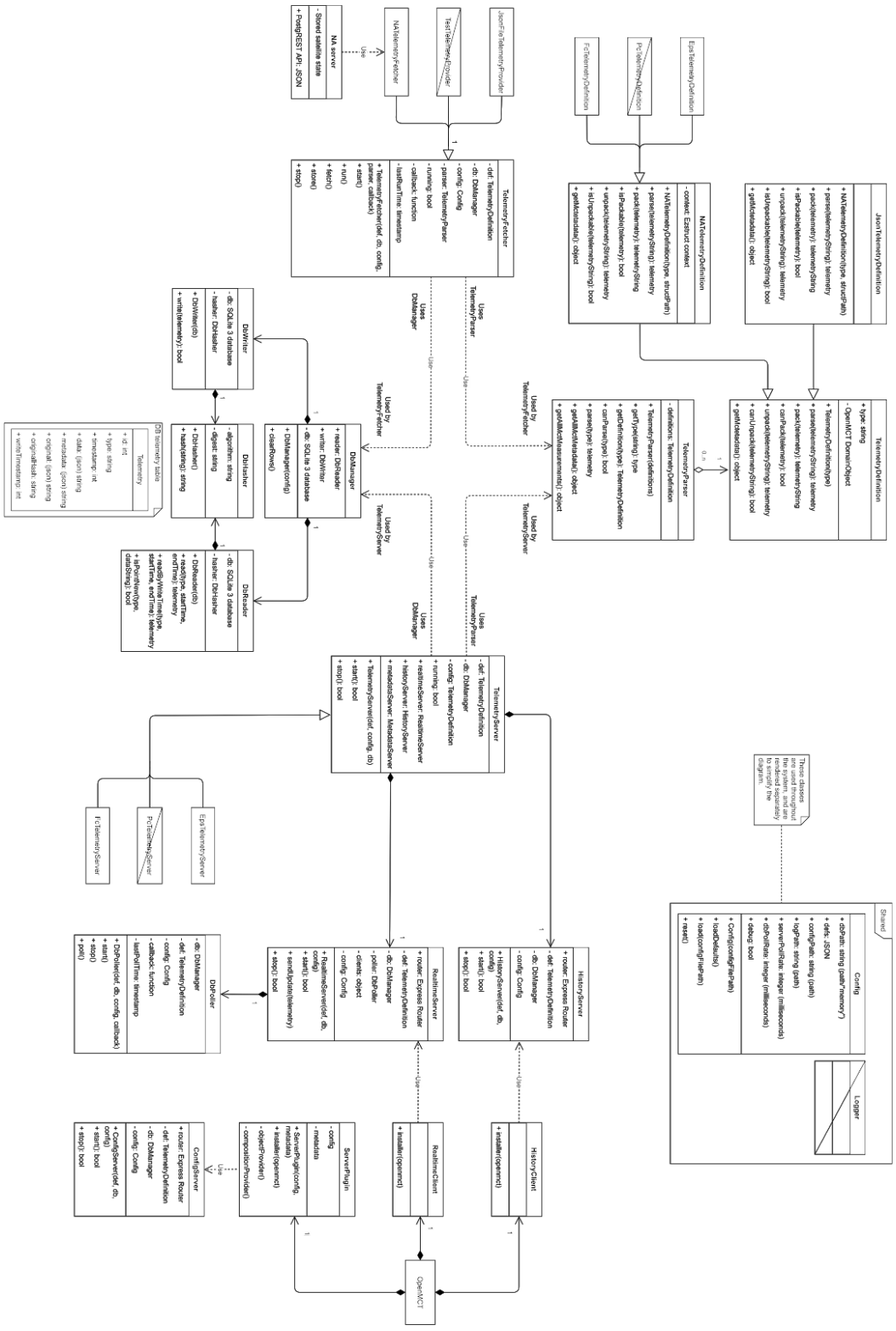


Figure 2.3.2 Class Diagram for full system

2.3.1 Shared modules

These provide simple but important utility functions, such as managing the current configuration of the system and allowing it to be imported/exported to a file, plus providing a system log.

The class diagram for these modules can be found in Figure 2.3.3.

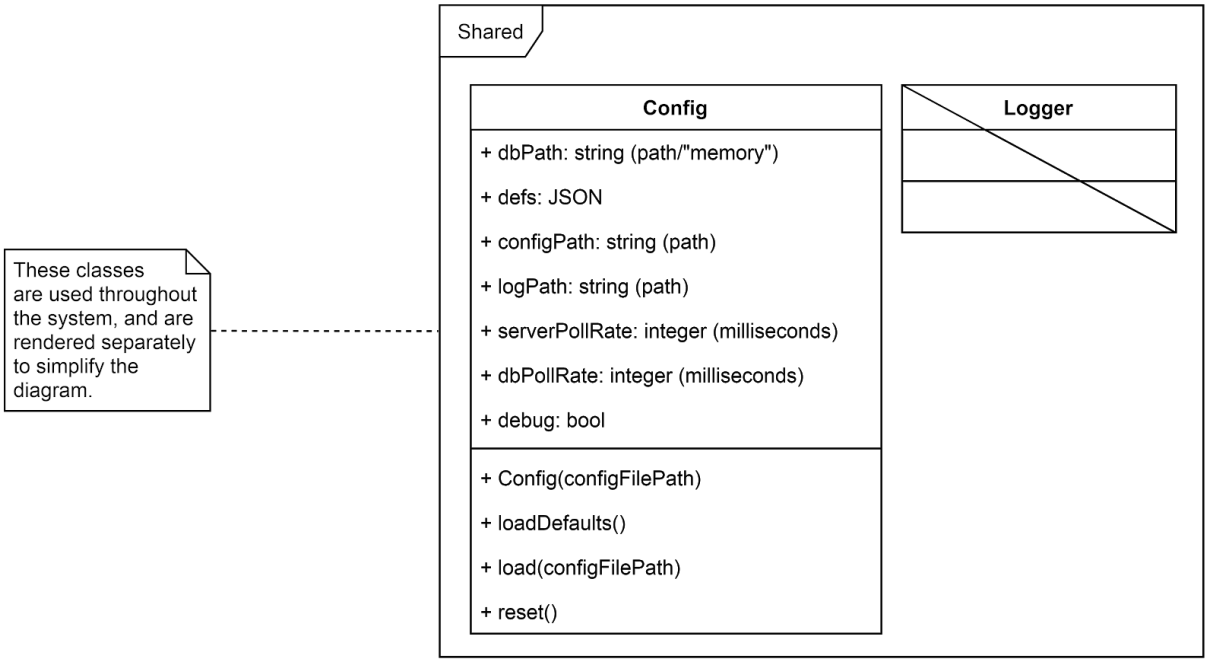


Figure 2.3.3 Class Diagram for shared modules

2.3.2 Telemetry fetching subsystem

This subsystem provides an interface for getting telemetry data from external sources implemented by the module `TelemetryFetcher`, which is realized for various specific external sources as `FileTelemetryProvider`, `TestTelemetryProvider`, and `NATelemetryProvider`. As the names imply, these get telemetry data from a file, a test spacecraft, and NanoAvionics' servers respectively.

The default implementation of all of these are simple polling services that check their sources at a regular interval to see if theres any new data; if so this data is written to the connected `DbManager`. A connection to a `TelemetryDefinition` via `TelemetryParser` may be required if the telemetry type, timestamp or any metadata to be stored cannot be determined directly from the received data without it being unpacked and parsed first. These modules will be introduced in the next section.

The class diagram for these modules can be found in Figure 2.3.4.

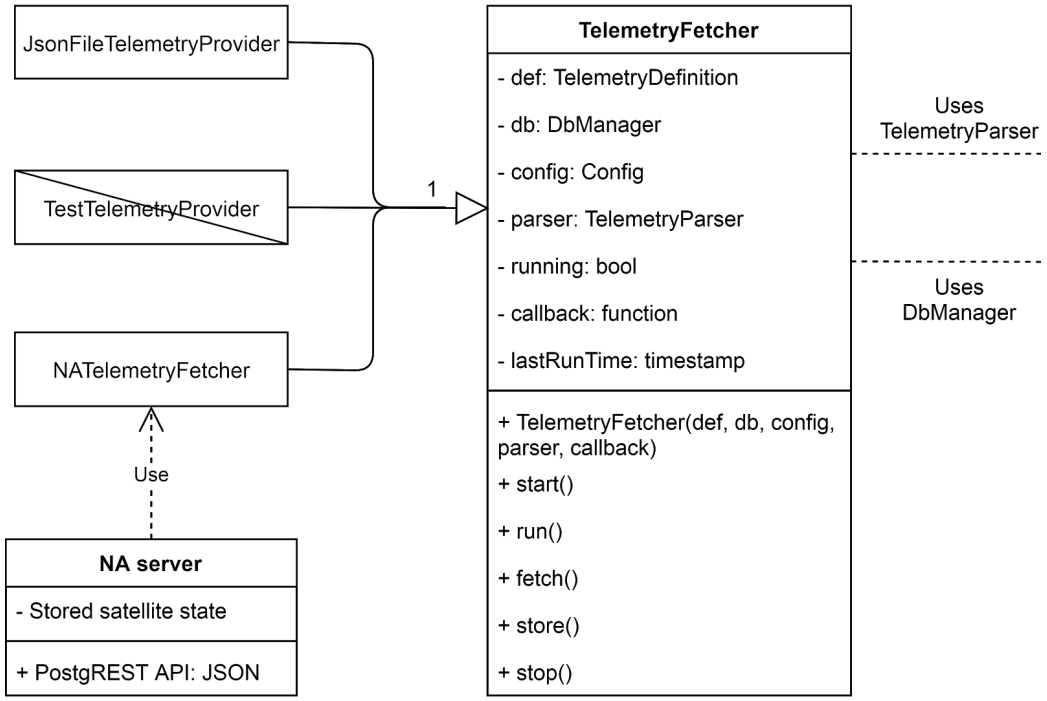


Figure 2.3.4 Class Diagram for telemetry fetching section

2.3.3 Data management and storage subsystem

This subsystem consists of two main modules; the `TelemetryParser` and `DbManager`.

The first of these, `TelemetryParser`, provides methods for configuring, parsing and unpacking various types of telemetry data, with a general class `TelemetryDefinition` that has a special subclass `NATelemetryDefinition` that provides a quick way to set up the `TelemetryDefinitions` for each type of telemetry we get directly from NanoAvionics' server.

The data that arrives from NanoAvionics' from their PostgREST-based HTTP API is in the form of JSON with some basic metadata and, more importantly, a string that contains a packed C-style struct. In `NATelemetryDefinition` we use a third-party module called EzStruct that allows us to quickly convert this to JSON based directly on the struct header definitions we get in text files from NanoAvionics, speeding up the process of implementing and updating the telemetry definitions greatly. Updating or adding a new `TelemetryDefinition` for parsing telemetry data from NanoAvionics' is usually as simple as downloading the struct header text file and updating the configuration to include it.

The class diagram for this subsystem can be found in Figure 2.3.5.

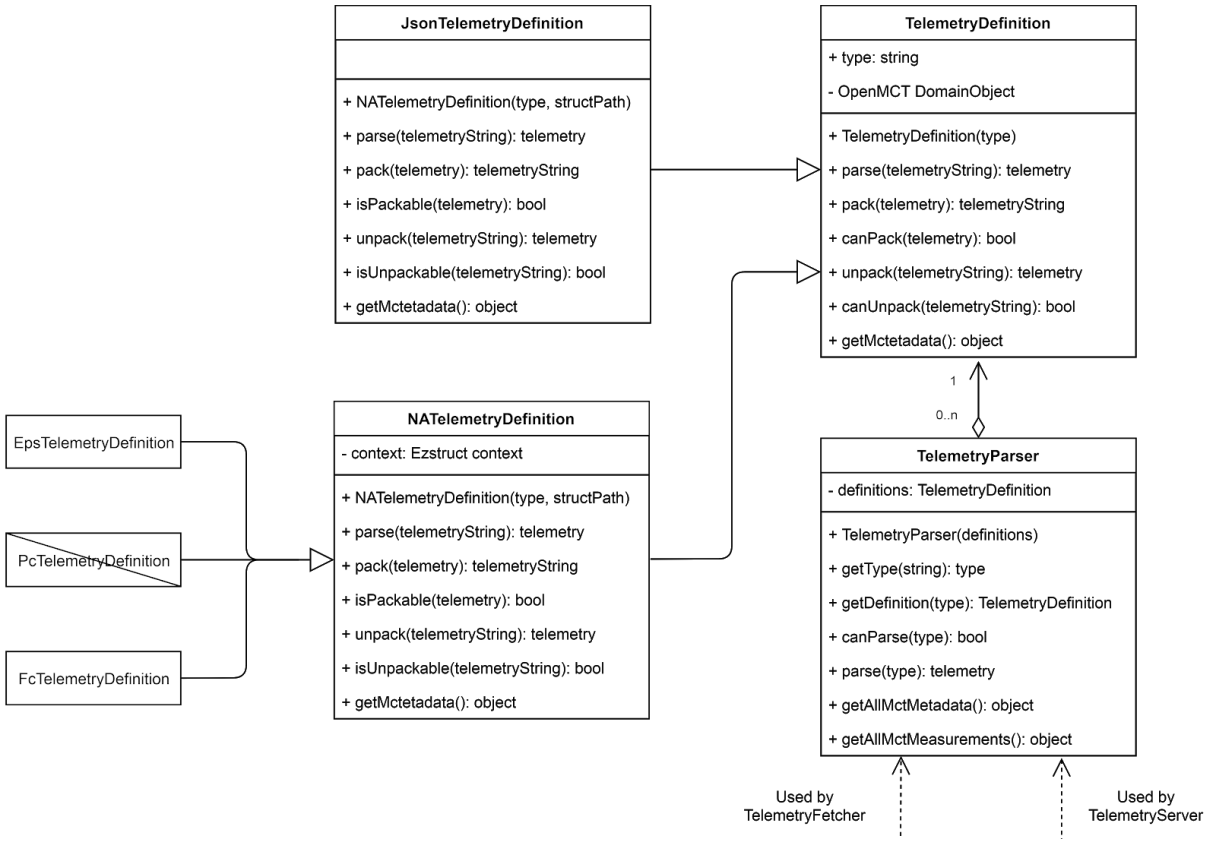


Figure 2.3.5 Class Diagram for telemetry parsing section

The next major part is the `DbManager`, which handles storing data (usually from a `TelemetryFetcher`) and reading data (usually from a `TelemetryServer`). The interface here is fairly simple, as the current implementation just has a single table that can be indexed by timestamp and type (which maps to a `TelemetryDefinition` that can be used to unpack the data stored for that type).

The class diagram for this subsystem can be found in Figure 2.3.6.

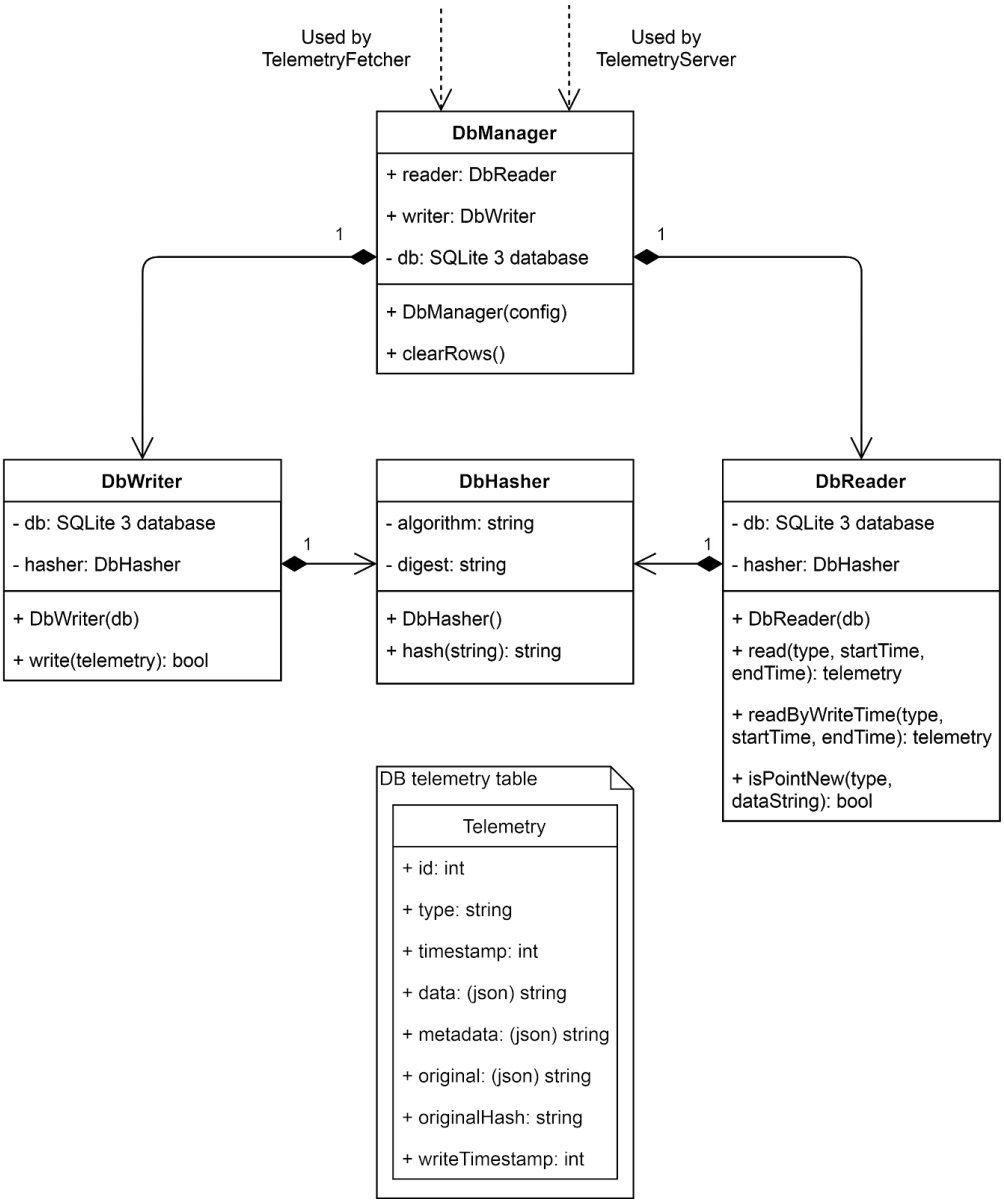


Figure 2.3.6 Class Diagram for database management section

To host the database it was decided to use SQLite; the fairly simple structure of the database used works well with it, and the large benefits it provides in portability (for “offline” work or backups) due to the database being stored in a single file made it a good candidate compared to other more complex database solutions.

2.3.4 Telemetry serving subsystem

This subsystem hosts and manages one or more HTTP/WebSocket servers for getting data for a specified `TelemetryDefinition` from a `DbManager`. The HTTP servers (implemented by

`HistoryServer`) are used for allowing a service to request historical data for a period, while the WebSocket servers (implemented by `RealtimeServer`) allow a service to subscribe to real-time data updates without having to continuously poll a HTTP server.

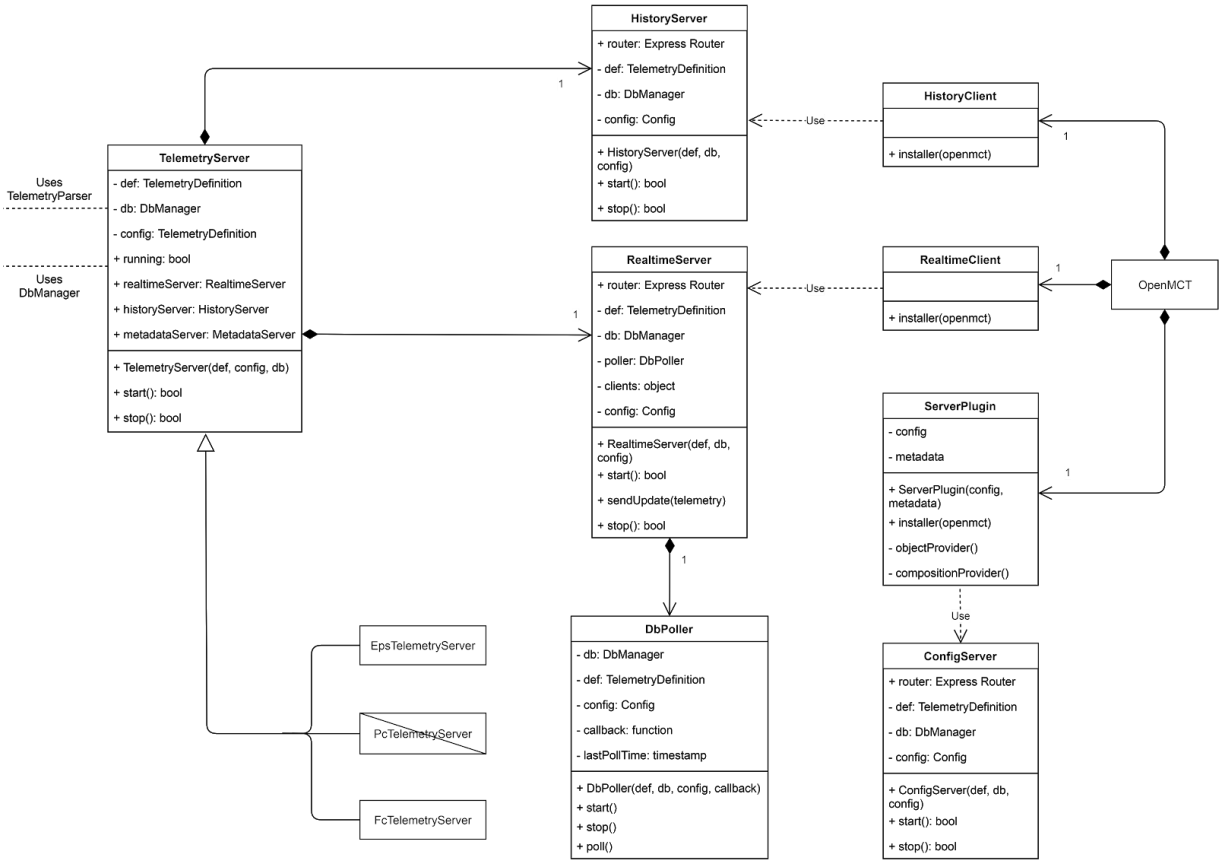


Figure 2.3.7 Class Diagram for telemetry server and Open MCT connection

2.3.5 Open MCT client plugins

These allow Open MCT to get and subscribe to telemetry data using the interface described above in section 2.3.4, and allows mapping this data to various user-defined visualizations.

2.4 Code standards

It was decided to keep the code style and syntax similar to what's already encouraged and used by the Open MCT project for both the backend and new frontend code required. A somewhat simplified version of Open MCT's ESLint configuration file is used to enforce this code style across this project; the simplifications are mostly the removal of configuration specific to external dependencies not used in this project.

2.5 Third-party modules

The main external modules to note besides Open MCT are EzStruct, better-sqlite3, Express and Jest with SuperTest.

3 System usage

THIS SECTION IS IN ACTIVE DEVELOPMENT, AND WILL EXPERIENCE MAJOR CHANGES AS THE SYSTEM DESIGN DESCRIBED ABOVE IS STILL IN THE PROCESS OF BEING FINALIZED AND IMPLEMENTED.

3.1 Setup and configuration

3.1.1 Server setup

The production server setup is still a work in progress.

A test/development server can be started by cloning the repository, navigating to it with a terminal and running npm install followed by npm run start to start the server. npm run test can be used instead to get a test coverage report.

3.1.2 Configuration options

Configuration options for the system are by default loaded from config.json inside the /state directory. See the included file for a sample on how to set up or change the system configuration - it uses all currently available configuration parameters.

3.1.3 Data definitions

NanoAvionics data definitions are added by copying the C struct definition for it to the structs folder, and referencing it in the configuration options. This has already been done for the EPS and FC general telemetry definitions.

3.1.4 Data sources

The data source for a telemetry definition is set in the definition for it inside config.json. Currently only one data source is supported per telemetry definition.

3.2 Operation

3.2.1 Using Open MCT

This section is still in development.

Navigating pre-configured views, creating new views, links to relevant Open MCT documentation

3.2.2 Monitoring system status

Not yet implemented in v1.0.

Status indicators and system health - last time data received from local telemetry server, last time data received from external telemetry server, last time data received from spacecraft

3.3 Expansion options

3.3.1 Adding new data sources

Documentation on how to set up new fetcher for data source, setting up definition for parsing and storing the data, setting up Open MCT plugin to read data

3.3.2 Telemetry API

The server provides telemetry data and metadata over HTTP and WebSockets.

Currently the easiest way to familiarise yourself with how these work is by taking a look at the requests sent by Open MCT to the server or by checking out the RealtimeServer, HistoryServer and ConfigServer test files.

4 Current status

See also the issue tracker for this project at https://github.com/AudunVN/hypso-openmct/issues for up-to-date and more detailed descriptions of the status and remaining work required for each subsystem.

Table 4: Module Status Summary

Module	Status
Shared modules	Initial version finished
Telemetry fetching	Initial version finished, needs additional testing against NanoAvionics' live PostgREST telemetry server
Data management and storage	Initial version finished
Telemetry serving	Initial version finished
Open MCT client	Initial version finished
Documentation	Work in progress
Server setup	Work in progress

4.1 Plan

The documentation, test coverage and server setup currently have no specific time frame, but will likely be done at a later time in 2020 to be ready for launch.

5 List of abbreviations

Table 5: List of Abbreviations

Abbreviation	Description
DB	Database
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation: Common data-interchange format for web services
NA	NanoAvionics
NASA	National Aeronautics and Space Administration
Open MCT	Open Mission Control Technologies: https://nasa.github.io/openmct/about-open-mct/