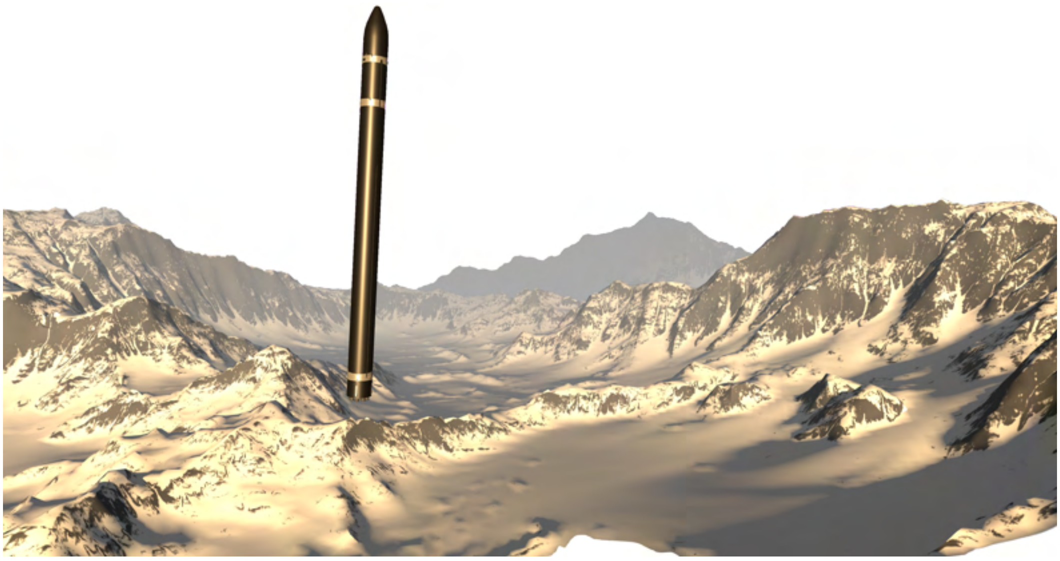
Development of a Web-Based SCADA System for Rocket Launches and Tests



Thesis for obtaining the academic degree
B.Sc.
at the School of Engineering and Design of the Technical University Munich

Supervisor Felix Ebert, M.Sc.
Chair of Space Propulsion

Examiner Prof. Dr. Chiara Manfletti
Chair of Space Propulsion

Submitted by Antonio Steiger
antonio.steiger@tum.de

Submission date Munich, 04.10.2022

Declaration of Authorship

I hereby declare that the thesis submitted is my own, unaided work. All direct or indirect sources used are acknowledged as references.

Munich, 04.10.2022

Abstract

Even though the ability to monitor and control a system as well as acquire data about it is required across a wide range of fields, vastly differing terminology is used. This thesis introduces the better known term SCADA to develop a modernized system with this ability for the field of rocket and propulsion research. To achieve this, signal chains for the most common components are laid out and tested across hard- and software domains. Capabilities like video monitoring with latencies below 180 ms, geographical maps for positional data, and three-dimensional vehicle rendering in real-time are demonstrated by developing customizable, web-based software. With a focus on ease of use and high dependability, industry 4.0 protocols were employed to automatically detect available measurement channels and distributed, redundant execution of the software was ensured.

Contents

Acronyms

S

T

U

W

Y

Glossary

A

Adapter - The term adapter is introduced to refer to computer programs that convert communication interfaces into protocols supported by web browsers. Adapters either translate onto a WebSocket or serve a website to accomplish this.... 34, 40, 49–53, 57, 58, 60–65, 67, 68, 70–73, 75–77, 123, 139

C

CSS - Cascading Stylesheets (CSS) are used to style graphical elements on websites. 42

D

DAQ - A Data Acquisition Device (DAQ) performs any combination of signal sampling, conditioning and conversion, stores and forwards acquired data and acts as a control source using it............. 18, 24, 25, 27–32, 36–39, 48, 49, 61, 62, 71, 72, 75, 76

Docker - Docker is a deployment tool for software. It supplies a consistent environment for software to be installed and run in across multiple platforms. This environment is called Docker container.. 43, 50–57, 64

Domain object - OpenMCT defines all visualization types like plots and the data points they visualize as domain objects. OpenMCT's website includes a domain object browser on the left by default.. 46, 48

E

Equipment configuration file - The equipment configuration file is used to describe measurement and control capabilities of the equipment used. Together with the UI configuration file, it defines all information necessary to be able to use the SCADA software for a particular application, like an engine test................ 46–48, 54, 56, 75, 139

I

I/O plane - Term defined as part of this thesis in Section 1.2. The I/O plane includes all components that either condition, sample and convert signals, have the ability to control or both.. 19, 21–30, 48–50, 57, 75

J

JSON - JavaScript Object Notation (JSON) is a dictionary-like file format that can be used to hierarchically describe data structures in various programming languages. In this way, data structures can be made persistent outside of the runtime of a program. Various JSON online editors with syntax highlighting and hierarchical visualization features are freely available... 47, 48, 54, 70

N

Netdata - Netdata is a web application visualizing computer performance in real time. It is open-source and can be deployed in a Docker container............ 56, 64, 73, 139

“Node.js” - "Node.js" is a JavaScript runtime environment. It enables the use of JavaScript in a server environment. 36–38, 43, 49, 53, 58, 61, 64

O

OPC UA - The Open Platform Communications Unified Architecture (OPC UA) is a communications standard including the definition of an information model. This model semantically describes the data available throughout the network, enabling machine to machine communication. 38, 61, 62, 71, 72, 75, 76, 123

OpenMCT - Open Mission Control Technologies (OpenMCT) is a software for creating web-based data visualizations for telemetry producing systems. 44–49, 55–65, 67, 69–76, 134

OSI - The Open Systems Interconnections (OSI) model structures communication networks into seven functionally distinct parts. 25, 36, 37, 42

P

Primary data storage - In this thesis, primary data storage refers to the storage of complete, full resolution data. Using primary data storage, video would be stored in full resolution and measurements would be stored at full precision and sampling rate, for instance. 27, 28, 75

R

Repository - A software repository is a collection of code organized into a directory structure and managed with a version control tool, like Git. 41, 44, 45, 52, 56, 60, 65

S

SASS - Syntactically Awesome Stylesheets (SASS) are an extension of Cascading Stylesheets (CSS) offering additional functionality. 42, 46

SCADA - The term holistically refers to all architectures and technologies necessary to achieve Supervisory Control and Data Acquisition (SCADA) of any sytem like machines or processes. 18–31, 33–35, 38–41, 43, 46, 68, 71, 75–77

Secondary data storage - Secondary data storage refers to the storage of reduced data sets. Using secondary data storage, only every tenth acquired measurement value would be stored, for instance. 29, 49, 75

Socket - In programming, (network) sockets are used as an abstraction layer to enable processes to communicate over a network. For example, TCP sockets can be used to communicate with test equipment in the SCADA system. 35, 57, 58, 117

SSH - Secure Shell (SSH) is a protocol for remote, console based access to computers. 54

SysML - The Systems Modeling Language (SysML) is a graphical modeling language based on the Unified Modeling Language (UML). It can be used to visualize the structure and interconnections of complex systems. 34, 35, 39, 40, 60–62, 64, 75, 77

T

Target plane - Term defined as part of this thesis in Section 1.2. The target plane contains all components of the system that need to be supervised and or controlled. A rocket and all its supporting infrastructure could be called a target system from the perspective of a SCADA system. 19, 21, 22, 24, 27, 32, 41, 46, 66, 75

TCP - The Transmission Control Protocol (TCP) is an OSI-layer 4 protocol with error correcting and load managing features. It can be used for communication over an IP network. 34, 35, 37, 42, 48, 49, 57, 58, 68, 71, 123

U

UART - Universal Asynchronous Receiver Transmitters (UARTs) are circuits enabling serial communication; Commonly seen on microcontroller devices. 33–35, 66, 71

UI configuration file - OpenMCT users can save created user interface (UI) layouts in a JSON file for later reuse. This file is called UI configuration file throughout this thesis. It is used to ascribe meaning to data points and control capabilities described in the equipment configuration file. 48, 75

V

Vis/Int plane - Term defined as part of this thesis in Section 1.2. The Vis/Int Plane includes all components of a SCADA system that enable **vis**ualization and **int**eraction for humans. 19, 21, 22, 25–27, 29, 30, 32, 33, 38, 75

“Vue.js” - Vue.js is a JavaScript framework for the creating reactive user interfaces. 42

W

WARR - The Scientific Student Initiative for Rocketry and Spaceflight (ger. "Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt"; WARR) builds rockets, satellites, rovers and much more.. 14–17, 20, 22, 24, 26, 28, 31–34, 39, 41, 43, 44, 46, 58, 59, 66, 75, 76

Web application - A web application (also called web-based software) runs in the web browser. It always runs on the additional layer of the browser and not directly on a computer, like native software. 41, 42

Web stack - The collection of all code components, across the front- and the backend, used to realize a web application is called the web stack. 42, 45, 49, 58

WebSocket - WebSockets are a type of network socket usable within the constraints of web-based software. They enable persistent, bidirectional communication to endpoints outside of the web browser environment.. 42, 43, 49, 50, 53, 55, 58, 61, 62, 64, 67, 75, 117, 123

Y

YAML - "YAML Ain’t Markup Language" (YAML) is a hierarchically structured file format that can be used to describe data structures in various programming languages. In this way, data structures can be made persistent outside of the runtime of a program. Here, it is used to configure distributed execution of SCADA software components by creating a file called "docker-compose.yaml" for Docker. 51, 54

List of Tables

List of Figures

1. Introduction

Three-dimensional holographic displays and similarly seamless, futuristic User Interfaces (UIs) are commonly used in science fiction as a means to embellish stories. While some of today's software used to test, monitor and control aerospace systems resembles a step in this direction (see Fig. 1a), some software used in the field are facing obstacles in this regard.

Before this thesis was begun, it became increasingly clear that the software used for monitoring, data acquisition and control at the Scientific Workgroup for Rocketry and Spaceflight (ger. WARR) and at the Chair of Space Propulsion and Mobility (SPM) is beginning to show its age in a similar way. Both the Chair of Space Propulsion (SPM) and WARR are expecting to move to more dynamic and operational test scenarios in the future, testing propulsive landing maneuvers, thrust vectoring and more. The software currently in use neither offers a lot of potential for new developments fitting those applications, nor is it dependable or intuitive.

This thesis was written to investigate what next-generation data acquisition, monitoring and control systems could look like at WARR and at the SPM. It will detail and test approaches across the hard- and software domain that are necessary to achieve modern features. Some of these features include three-dimensional terrain and vehicle visualization in real time, redundant software execution, low latency live video and customizable UIs. To start, the drawbacks and problems of current systems have to be understood. They are laid out in the following section.

1.1. Problems and Drawbacks of Current Systems

Both WARR and the SPM are currently using LabView implementations to monitor and control static test stands. During tests, data is transferred to the monitoring computer and stored there. Both implementations exhibit problems that are hindering their innovation alongside the research they are used for. The most prevalent problems will be detailed in the following.

Modern Rocket Development Comes with New Data Types
Reusability of rockets as well as live mission broadcasts are becoming increasingly common in the launcher industry [1],[2]. Correspondingly, rocketry student teams are starting to live stream launches on the Internet and the first propulsive landing competitions are coming along. Both SPM and WARR are expecting to increase research in the propulsive landing of rockets and live streams are starting to become a necessity at WARR.
Dynamic test stands for these applications entail a range of new data types that the software used by both parties is not compatible with: Positional data, acceleration, orientation, live video and more. In addition, data is likely to arrive over radio (telemetry). This was previously not required for static test stand operations. Furthermore, these rather operational scenar-

ios have different software requirements compared to static testing. In places, the current software implementations are not suited for the strict procedures necessary to safely conduct dynamic tests and launches.
It is clear that the LabView setups for both the chair and WARR need to be extended. Prior to this, an investigation into the feasibility of the modifications is necessary. Especially the question of how to integrate telemetry into LabView needs to be answered. However, this leads to the next problem.

"Use, But Do Not Touch" Philosophy

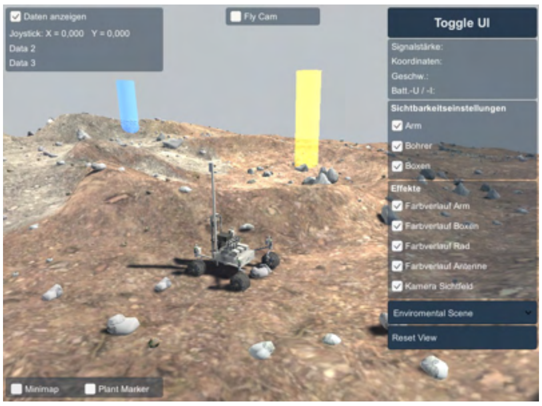
Both LabView implementations have not been actively maintained and extended in years. A term to encapsulate this could be "use, but do not touch" philosophy. It refers to the issue that the test stand software is regularly used by test engineers, but neither well understood nor expanded or improved. This problem can be equally observed at WARR and the SPM and mostly stems from the departure of the few developers acquainted with the software. This is especially a problem in a student initiative like WARR, in which the member base is rather volatile.
Developing software like this in-house has several advantages. First and foremost, customizability. With control over the software's code base, new features can always be added and modifications can be performed on any level. The customizability, however, can lead to a significant amount of non-reusable code. Projects like Telestion developed at TU Würzburg were created to combat this exact problem [3]. Secondly, dependence on third party development is reduced to a minimum. Lastly, as a consequence of customizability, more intuitive user interfaces fitting the exact application can be created. This offers potential for reduction of user errors.
Without gathering and maintaining know-how in-house, innovating the testing software is essentially unfeasible. In some aspects, a reliable and proven software is to be preferred over a less reliable, but innovative one. Nevertheless, the industry and many other student teams are not standing still.

The Aerospace Industry and Student Teams Have Moved Ahead

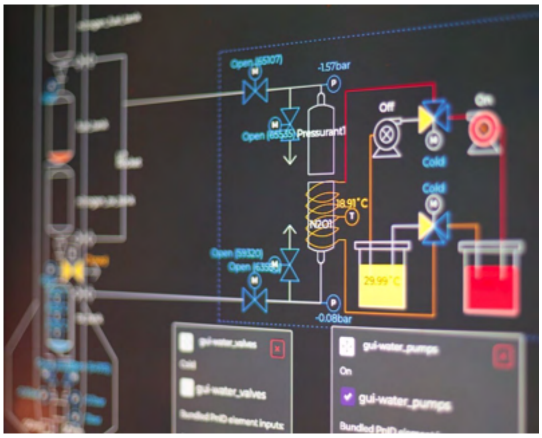
Aerospace companies are actively developing contemporary and capable software to control, test or monitor their systems. SpaceX, for instance, is using web-based control software for their space capsule, Dragon (see Fig. 1a) [4]. Another example is RocketLab's control room software (see Fig.1b).
Not only the industry is working on software technologies like this. Student aerospace teams worldwide are pursuing similar efforts as well, see Fig. 1c and 1d.

a: Software interface in SpaceX's Dragon capsule. [5]

b: RocketLab's control room. Adapted from [6]



c: A rover monitoring software using 3D terrain models developed by the student team STAR of TU Dresden. [7]



d: The control software used in the student rocketry team of TU Vienna. [8]

Fig. 1: Examples for recent advancements in aerospace software in the industry as well as in student teams.

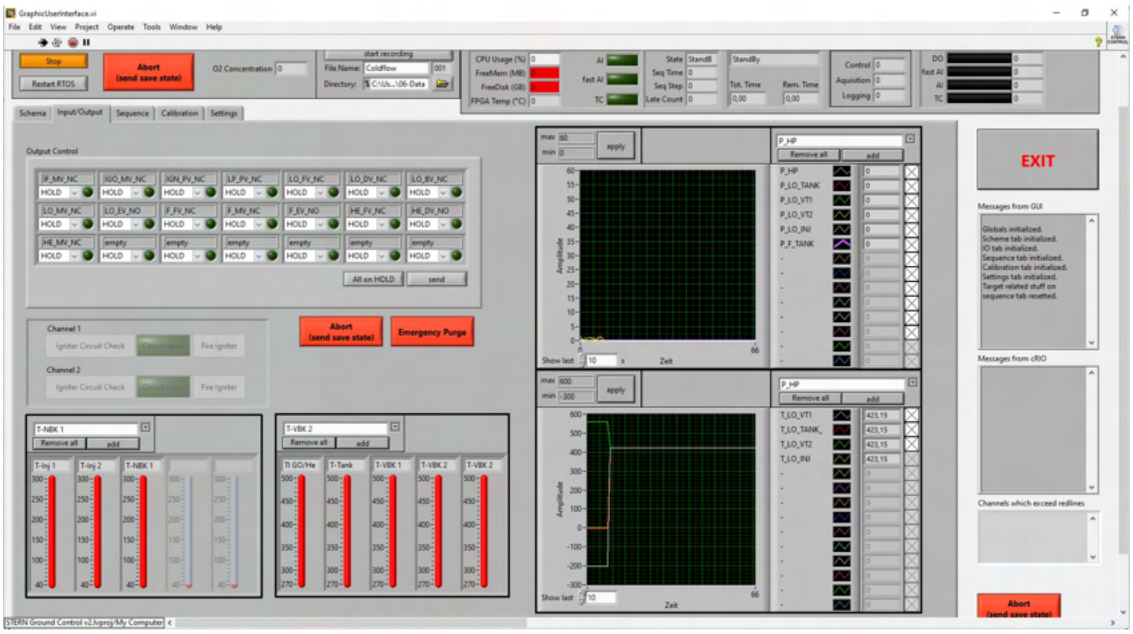


Fig. 2: A screen capture showing the user interface of WARR's current LabView implementation.

As portrayed in Fig. 1, some of the monitoring and control software used across aerospace has the ability to be tailored exactly to the application at hand. Visual elements resembling their physical counterparts are employed to increase intuitiveness. The LabView implementation used at WARR, depicted in Fig. 2, is lacking in this regard.

Unreliable Software
The attitude towards the LabView implementation used at WARR Rocketry is generally negative. This stems from many issues, some of them being:

- Starting the software is a complicated process that regularly fails.
- Operating the software often means memorizing an exact sequence of actions.
- Operator trainees have to be taught to follow unintuitive procedures.
- The software does not work as intended. For example: Changing the sampling rate for sensors to 50 kHz does visually seem to work, but it does not.
- The software is not reliable enough for some testing environments; It has to be restarted sporadically.

Especially the last point has caused some precarious situations in the past: During the last test of a hybrid rocket engine at WARR in October 2021, a significant amount of helium was lost to the atmosphere due to an unknown valve switching error. Both human error as well as a software bug are deemed equally likely. As a student initiative, losing helium does not only represent a loss of physical resources, but a significant financial loss to WARR.
An example of how this could have been avoided is a "blacklist" functionality for valve state combinations. Users could be warned prior to switching a valve that would cause fluids to be expelled. While an implementation of this is technically feasible in LabView, as previously mentioned, WARR does not have the required know-how.

1.2. Web-Based SCADA Systems — Terminology and Taxonomy

The ability to monitor and control a system as well as acquire and store data about it is required over a wide range of applications in many different fields. There is no consensus as to what to call systems that provide this ability. Terms like DACS, Data Acquisition Device (DAQ), DAS or DAU are used interchangeably or with significantly varying meanings and scopes. For example, when using the term DAQ, one party might intend to describe just the device handling signal sampling. Another party however, might refer to an entire system of components additionally enabling visualization or data storage. Because the term DAQ is quite common, it will be defined separately in Section 2.1. To avoid any misunderstandings, this section serves to define clear terminology for the scope of this thesis. Instead of using any of the terms mentioned above, this thesis will use the term Supervisory Control and Data Acquisition (SCADA).

The term SCADA stems from the field of industrial automation. It is mostly used to describe systems that enable monitoring and control of technical processes. In this thesis, the SCADA approach will be applied and introduced to the field of rocket and propulsion research. It is the term of choice for two reasons. Firstly, it is one of the more widely known terms of the few that were previously mentioned and resembles a fairly well-defined concept. Secondly, the requirements for monitoring and control in both fields bear a lot of resemblance. In fact, the automation pyramid (see Fig. 3) and terminology from industrial automation is applicable to typical systems used in rocket and propulsion research.

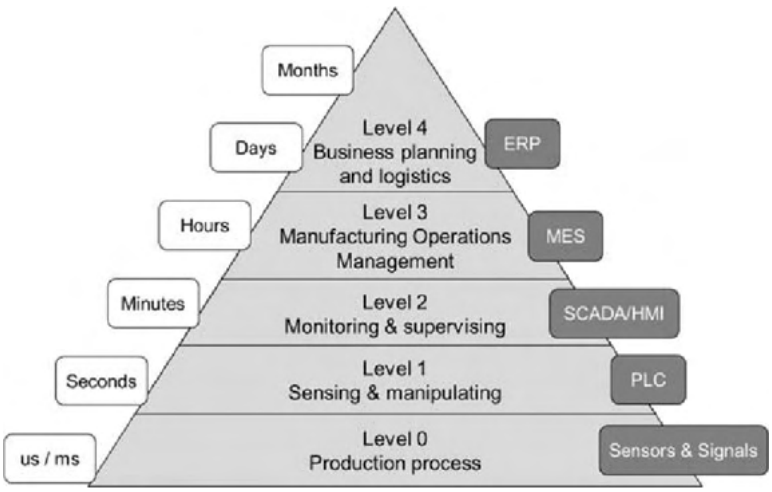


Fig. 3: The automation pyramid; Although it is typically used to model manufacturing processes, the structure of data acquisition and control in rocketry applications is reflected in its lower three levels. [9, Fig. 2]

To start, both fields use a wide range of mostly analog sensors and actuators (pyramid level 0), like pressure transmitters. Continuing, the sensor data needs to be conditioned and converted to be used by local controllers (pyramid level 1) for *automatic* control. For example, in the field of rocketry, level one might handle fast valve switching sequences for the startup of an engine. Lastly, humans need to intervene or guide from time to time and supervise the automated system (pyramid level 2). The dry-check of a rocket engine's fluid system for instance, is regularly performed by hand, sending control commands manually, one valve at a time. Requirements for monitoring in both fields have similarities as well. For instance, engineers have to monitor the fueling process of a rocket in a similar way to how they monitor a production plant. Similarities end at level two of the automation pyramid, however; Levels three and four do not correspond well to rocket systems in a scientific environment. Going forward, the first three levels of the automation pyramid depicted in Fig. 3 will be named individually to better identify their domain throughout this thesis.

- **Target plane (level zero):** Components that produce data to supervise and or need to be controlled. Sensors and actuators are situated here.
- **I/O plane (level one):** Components responsible for any combination of sampling, conditioning, conversion and control.
- **Vis/Int plane (level two):** Short for visualization and interaction. Components responsible for producing visualizations and accepting interactions from humans.

Together with the communication between them, the three planes frame the SCADA system that will be developed in this thesis. Fig. 4 visualizes this concept.

Components of the target plane exhibit at least one of the following properties:

- **Data Source:** Component produces data.
- **Control Sink:** Component accepts control.

A data source could be an analog pressure transmitter, for instance. Actuators like servos and valves are types of control sinks. The term actuator is not considered to apply to all control sinks here, as there exist devices like signal lights that need to be controlled, but do not actuate in any physical way. Notably, devices can be both data sources and control sinks simultaneously.

Data can rarely be visualized directly and control sinks can rarely be controlled directly. Usually any combination of sampling, conversion or conditioning is necessary. As such, to visualize analog voltages on a computer they must first be digitized with an analog to digital converter (ADC). Components that exhibit this behaviour are part of the I/O plane of the SCADA system. Because most sensors used in rocketry are analog, I/O systems play a central role in test setups and usually make up a large part of the system's cost.

Supervision and control cannot be done by humans without some type of system that visualizes the data and enables interaction. Components aiding in visualization and interaction are part of the Vis/Int plane for short. The Web panel is a modern example of a Vis/Int component. It is a touch screen combined with embedded hardware that is able to display websites.

Web-panels are one example of where the web-based SCADA software that will be developed as part of this thesis can come into play. The software itself is considered a Vis/Int-component as well. In a web-based SCADA system, the software enabling monitoring and control by humans runs in a web browser and not natively as a computer program. This is further detailed in Section 3.1.

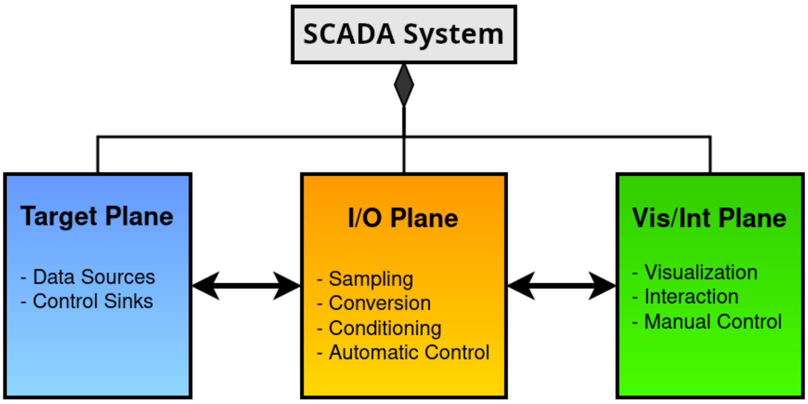


Fig. 4: Summary of SCADA systems' different domains as defined in this section. The colors used here are maintained throughout all chapters.

In rocketry, it is possible to separate the application of SCADA systems into two use cases:

- **Experiments:** High sampling rates, high resolution, complete data storage, repeatable; E.g., engine characterization on a test stand
- **Operations:** Lower sampling rates, lower resolution, partial data storage, unique and high stakes; E.g., rocket launches

This distinction will not be made here, because the SCADA system can be set up in a way that enables the transition between the two use cases to be fluent and configurable. At WARR especially, the distinction is not made at all, because data from actual launches is deemed equally or more valuable then data from experiments. The SCADA system will be designed in such a way that it is able to combine the high fidelity data acquisition needed for experiments with the reliability required in operational scenarios. Notably, this ensures that development efforts can be focused on a single SCADA software.

1.3. Thesis Structure, Scope and Objectives

Fundamentally, this thesis aims to design a web-based SCADA system that fits typical rocketry applications. As such, it was written to:

1. Objective: Answer the question of how to integrate typical rocketry components of the target plane into a web-based SCADA system.

In order to achieve objective one, an appropriate I/O plane has to be designed. This will be handled in Chapter 2. It lays out an I/O plane that fits typical rocketry applications and defines signal chains up to the Vis/Int plane. From there, Chapter 3 will design the rest of the SCADA system, with the goal to:

2. Objective: Develop a web-based SCADA software prototype.

The thesis finishes with Chapter 4, a testing chapter serving to achieve the following aim:

3. Objective: Test selected signal chains from the system design chapter together with the software developed in the software design chapter.

This thesis will not cover the target plane in detail. The target represents the "system of interest" that is to be monitored and controlled with the SCADA system. While the SCADA system *will* be developed as part of this thesis, the target plane is considered predefined and assumptions will be made about its components where necessary. Assumptions and explanations of typical rocketry components in the target plane are laid out in Section 2.1.

As an additional, general objective, this thesis is to serve as an introductory reference work for aerospace students, engineers and scientists. As such, some rather unaccustomed concepts in the field, like web-based software, are explained in more detail. This is to lay a foundation for further developments and improvements.

2. System Design

This chapter will first deal with finding a fitting general system architecture for the SCADA system. Then it will detail the hardware and signal chains necessary to integrate components of the target plane into the system to achieve objective one of this thesis (see Section 1.3). Signal chains will be developed up to the Vis/Int plane, where Chapter 3 will continue. The term stakeholders will be used to refer to WARR as well as to the SPM from now on.

Requirements

Requirements from the stakeholders for the SCADA system are listed in Appendix A. Due to the large amount of requirements they will predominantly be referenced in this section, not explicitly stated.

Chapter Structure

This chapter is structured into three parts. As a starting point, the different components typically involved in rocket launches and tests are identified and categorized from the SCADA perspective (Section 2.1).
Following this, multiple high-level aspects are considered in Section 2.2:

- Section 2.2.1: Interconnection and organization of components in a network topology
- Section 2.2.2: Location, amount and redundancy of data storage
- Section 2.2.3: Types of control and their coordination
- Section 2.2.4: Relevant aspects of safety and reliability as well as their interplay

Finally, different approaches for integration of each component into the SCADA system are compared in Section 2.3. Details like signal chains involved in the optimal approach are explained further.

2.1. Rocket Launch and Test Components Relevant for SCADA

This section will list typical components used in rocket launches and tests from the SCADA perspective and motivate their integration detailed in Section 2.3. Said components are located in the target plane and the I/O plane of the SCADA system. Each plane can be split into components that are exclusively used on the ground, like precise measurement equipment and components that are located on board of a rocket. The difference from the perspective of the SCADA system reduces to a component possibly not being reachable after the launch of the rocket. At WARR for example, the secondary flight computer (See Fig. 28) is simple and does not offer data transmission over radio during flight. Still, it is part of the SCADA system as long as the rocket is on the launch pad, because it offers a wired communication interface. An exception from the above categorization are the test computers in the Vis/Int plane of the

SCADA system. In fact, monitoring them has the potential to aid in troubleshooting scenarios. Details and one possible way of achieving this will be presented in Chapter 3.

2.1.1. Components in the Target Plane

Data Sources and Control Sinks

Sensors are data sources. Typical sensors used in rocket launches and tests are listed in Table I; Typical control sinks are listed in Table II.

Table I: *Typical* sensors used in rocket launches and tests.

Sensor Type	Output Signal Type
Pressure transmitter	Analog; 4..20 mA electrical current
Pressure transducer	Analog; 0..10 V or 0..5 V electrical voltage
Thermocouple	Analog; -100..+100 µV electrical voltage
Flow meter	Analog; Electrical frequency
Load cell	Analog; 0..45 mV electrical voltage
IMU	Digital; I^2C, SPI or CAN
Ambient Pressure Sensor	Digital; I^2C, SPI or CAN
GPS receiver	Digital; UART, SPI or I^2C

Digital sensors are typically deeply integrated into systems of the I/O plane, for example on flight computers. All of the different data produced by sensors needs to be considered in the development of the SCADA software. For example, the SCADA software has to provide a way to monitor positional data created using the Global Positioning System (GPS) or orientation data from inertial measurement units (IMUs).

Table II: *Typical* control sinks used in rocket launches and tests.

Control Sink Type	Control Signal Type
Shut-off valve	Analog and binary; On: 24 V, Off: 0 V
Regulated valve	Analog; 0..10 V electrical voltage
Ignition source	Analog and binary; 0 A or high single digit currents
Hydraulics	Analog; Many possibilities
Warning horn	Analog and binary; On: 24 V, Off: 0 V

Surveillance Cameras

The stakeholders use live cameras to monitor rocket launches and tests remotely due to the hazardous nature of the tests. Latency is a predominant concern here, because situations requiring low reaction times from test engineers can arise. It is possible to handle test surveillance with a separate system, but this approach has similar drawbacks to the use of more than one SCADA software. Integration of live video will thus be considered here as well.

Live Video Receivers
At WARR it is of high importance to monitor and save video of live cameras on board of rockets on the ground. This stems from the value that video material can bring to possible failure investigations. Video is transmitted over radio and received by a dedicated receiver in the target plane of the SCADA system on the ground.

Backup Coordinate Receivers
At most international student rocketry competitions it is customary to use backup location transmitters on the rockets to increase probability of finding rockets after a parachute landing. The transmission over radio needs to be received in the target plane of the SCADA system and integrated with the software as well. This will be covered through the integration of flight computers going forward, as necessary steps are comparable.

There are more components of the target plane that could be mentioned here, like sound suppression systems, payloads, power supplies with monitoring functionality and others. Nevertheless, integrating the devices listed so far into the SCADA system covers a wide range of scenarios in the field of rocketry.

2.1.2. Components in the I/O Plane

Flight Computers
Flight computers used on rockets are assigned to the I/O plane here, because they typically assert control based on signals they previously sampled, conditioned and converted. Valves are often controlled using pressure values that have been acquired from pressure sensors, for example. Flight computers can forward data such as status and diagnostic information, measurements and more. There are typically multiple redundant flight computers, and they usually accept control commands for coordination with test and launch procedures.
While they are launched on board the rocket and leave the SCADA system on the ground, flight computers typically stay part of the I/O plane during the flight through telemetry. Hence, integration of flight computers is split up into wired and telemetry integration. Telemetry is typically received by a dedicated receiver on the ground, which is assigned to the I/O plane of the SCADA system here.

DAQs
In this thesis, DAQs are defined as devices that not only sample signals, but are also able to act upon them and control. They are typically used near rocket engine test stands and are not located on board the rocket. DAQs can be compared to Programmable Logic Controllers (PLCs), but they usually offer higher performance in metrics like sampling rate, processing power and data storage rate. This is why they are assumed to be able to run control models here.
DAQs usually offer a large number of inputs for all of the commonly used sensors and actuators listed in Table I and Table II respectively. Typical sampling rates can range from ten

samples per second (S/s) to 150000 S/s. In this thesis, DAQs are assumed to offer the capability to store data locally and to forward it to the Vis/Int plane.
Because DAQs are able to offer a large number of inputs, they play a central role in aggregating sensor data into a single coherent communication interface for the Vis/Int plane. Some data transmission standards that can be used for DAQs have the potential to reduce development work necessary and increase ease of use across all domains of the SCADA system. This is why data transmission for DAQs is investigated extensively in Section 2.3.2.

2.2. High-Level Considerations

Before considering every component by itself and how it can be integrated into the SCADA system, a few fundamental aspects have to be investigated.

2.2.1. Connecting Components — Network Topology

If a device in the I/O plane is to be monitored and controlled, it needs to be able to communicate with the Vis/Int plane. Instead of connecting each device separately, a communication network can be designed. A model that is widely used for this task is the Open Systems Interconnections (OSI) model. It is depicted in Fig. 5. In this section, layers one to three of the OSI model will be defined and laid out considering physical constraints and requirements. The resulting network topology is summarized in Fig. 6. The physical layer (layer one) concerns the physical transmission medium used. The data link layer (layer two) enables the transition of data to a physical signal. Lastly, where data is routed to, is decided in the network layer (layer three).

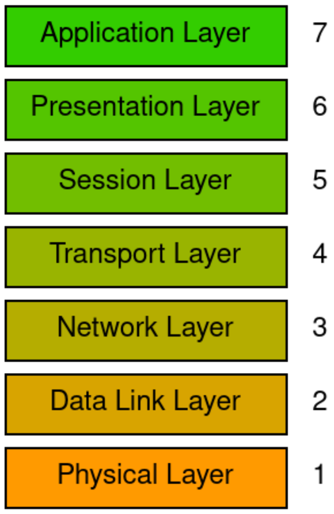


Fig. 5: The OSI model. Presentation and session layer are of low significance here.

Layer four is defined by the integration approach chosen for a specific device. It will thus be looked at in this chapter's Integration Section (Section 2.3). With the selection of a transmission protocol for layer four, development of the software integration approach can be started, because software begins to handle communication with layer five. Layers five to seven will therefore be further investigated in the Software Design Chapter (Chapter 3).

OSI Layers One, Two and Three

Ethernet (IEEE 802.3) on layers one and two as well as the Internet Protocol (IP) are already being used by the stakeholders for their current test stands. As one of the pillars of the Internet, the IP can be directly used by web-based software. As this layer combination additionally fits all stakeholder requirements, protocols of layers one to three will not be investigated further here.

Physical Constraints on the Network

The stakeholders have use cases wherein parts of the signal chain are required to be physically placed in certain locations. While in a testing laboratory with sheltered monitoring rooms, the SCADA system might only stretch distances of less than 25 m, during any dynamic tests or launches, personnel monitoring the test has to keep large safety distances. In the case of WARR, to be able to compete in some student rocketry competitions, the SCADA system has to bridge distances of over 600 m (see requirement R-CHRO-12). This raises the question of where each plane of the SCADA system should be physically placed.

Many sensors the stakeholders use, like pressure sensors or thermocouples (See requirements R-CHRO-1.1.2, R-CHRO-1.1.1, R-CHRO-1.1.6; DACS4 and Table I), use analog signals. It is unfeasible in terms of signal integrity and expensive to route large amounts of analog signal cables from the sensors to their I/O system over long distances. Thus, most of the I/O plane has to be placed close to the test setup.

As suggested by requirement R-CHRO-7, the connection between I/O systems and monitoring station on the ground should be combined into one cable. This can reduce setup effort and cable cost. Many data transmission standards do not support distances of over 600 m without additional hardware, like repeaters or extenders. Depending on transmission speed, repeaters can only be used over a range as low as 100 m in Ethernet networks [10, Table 8-1]. This is why for this SCADA system, Ethernet extenders are better suited. They modulate Ethernet signals into more robust, long distance protocols of the physical layer. WARR has acquired Single-Pair High-Speed Digital Subscriber Line (SHDSL) extenders for this purpose. They can bridge distances of up to 20 km, if the data rate is low enough. In this example, the data rate that can be used over the long distance link is limited to a maximum of around 30 Mb/s [11]. This is relevant for the approach to data storage, detailed in the next section.

Organizing Components

Large parts of the I/O systems on the ground are combined into one system, called **measurement and control station** going forward. This is an approach which can be seen in typical test setups, in the form of a 19-inch mounting rack containing most of the I/O equipment.

Because radio receivers in the I/O plane do not need to be physically close to the test setup, they are aggregated in a range of 200 m around the monitoring location to avoid Ethernet extenders or repeaters. This aggregated system will subsequently be called **radio ground station**. This approach additionally leaves open the option for hand-tracking with antennas. The added distance also offers advantages like a better position for tracking, easier maintenance access and better protection from debris.

To enable both the radio ground station as well as the measurement and control station to connect many different devices into the network, network switches are required. The switches fan out the network in the stations and the Vis/Int plane fans out the network to multiple SCADA software users. As a result, the network topology proposed here is an Ethernet based, tree or star bus topology (see Fig. 6).

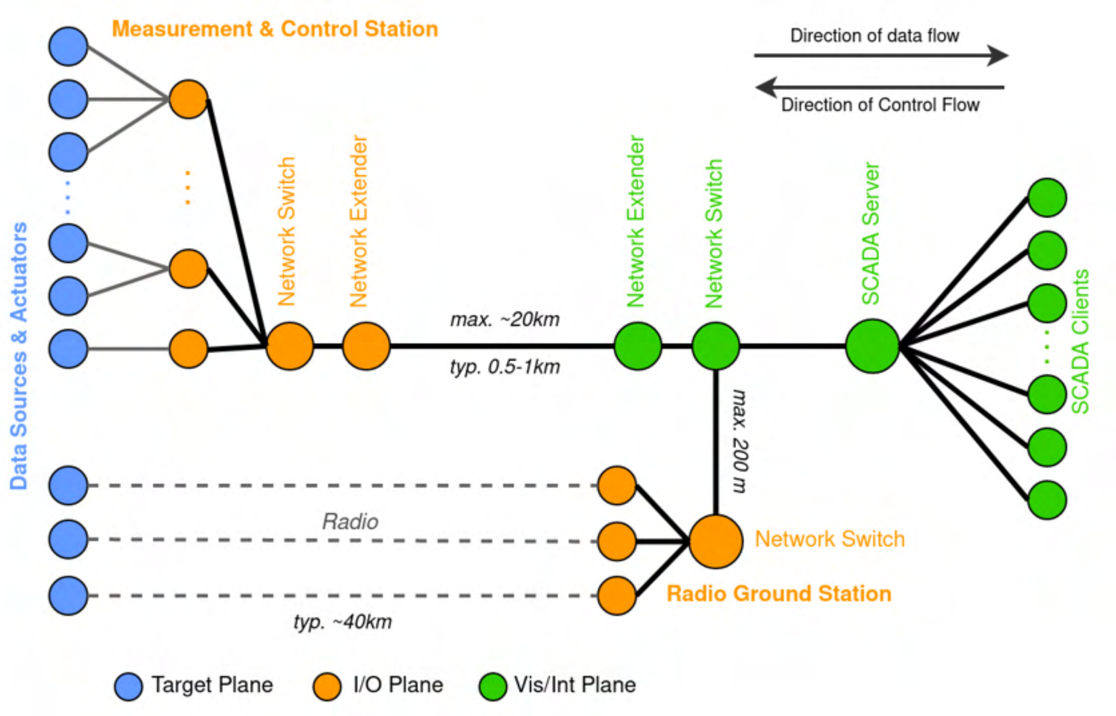


Fig. 6: Proposed network topology, fitting requirements from stakeholders and technical constraints.

2.2.2. Data Handling and Storage

This section will find a fitting approach to the way data is handled across the SCADA system and present an answer to the question of where data should be stored and how.

Primary Data Storage

In the following, the term primary data storage will be used to refer to the storage of complete, full resolution data. In the case of a DAQ, it thus refers to the storage of all acquired measurements at their original precision and sampling rate. There are two main approaches to primary data storage:

- Local, primary data storage in the I/O plane or in the target plane
- Remote, primary data storage in computer systems of the Vis/Int plane

In the case of DAQs, the stakeholders are currently following the second approach and stream all acquired data to a monitoring computer in the Vis/Int plane. There, it is handled and stored by the programming software for the DAQ, in this case LabView.
This approach is compatible with the SCADA system being developed here, but there are multiple further considerations to be made:

- Large implementation efforts are necessary to maintain high data rates across all SCADA domains, especially in the SCADA software.

- This approach is suboptimal in scenarios where the available data rate in the network is limited (e.g., Ethernet extenders)
- To retain data in the event of connection loss, data has to be stored locally as well.

The first aspect especially applies to flight computers, where the complete data stream has to be maintained over a radio connection in a dynamic environment. If the equipment used supports it, the safest approach to data storage would be to store all acquired data both locally as well as remotely. Because the main purpose of the SCADA software is to enable monitoring and control of the system, it will not be developed to support primary data storage. This is further motivated by the fact that the DAQ manufacturer's software can be used for this purpose, if required. Redundant primary data storage can also be achieved by adding dedicated data loggers to the SCADA system.

Going forward, the SCADA system will be developed using local, primary data storage, where possible. In addition to reducing network load, development and correct functioning of primary storage solutions can be deferred to manufacturers in this way. By acquiring network cameras that are able to record video locally on secure digital (SD) cards, WARR has already reduced the amount of work involved compared to the implementation of a Network Video Recording (NVR) setup using the remote primary storage approach.
To access primary data from DAQs for detailed test evaluations, SD cards can be used as well. Another approach is to access the data remotely using a File Transfer Protocol (FTP) server running on the DAQ. Using this approach, the test data can be downloaded over Ethernet to a monitoring computer, no physical access to the I/O plane is necessary.

Data Visualization Rates
The amount of samples used for visualization purposes does not have to be equal to the full sampling rate used in the I/O plane. In fact, increasing the amount of measurements shown in visualizations starts yielding diminishing returns early on due to human limitations. The motivation behind high data visualization rates is often to decrease the time needed to react to a certain event reflected in the data. Using a human reaction time to visual stimuli of 250 ms [12, p. 2], an exemplary transmission latency of 20 ms and a monitor refresh rate of 60 Hz, a relative comparison of reaction times at increasing visualization rates can be employed. Results are summarized in Table III.

Table III: Reduction of human reaction time from increased data visualization rates.

Data Visualization Rate:	10 S/s	15 S/s	30 S/s	60 S/s
Worst Case Reaction Time:	387 ms	354 ms	320 ms	304 ms
Relative Decrease:	-	9%	10%	5%

Table III does not yet consider that to react to a critical event, physical movement is necessary to press a button, for instance. It shows that if primary data is not stored remotely, the amount of samples per second forwarded over the network can be limited to avoid sending expendable data over the network. Data is used especially inefficiently when its arrival rate exceeds the refresh rate of the computer monitor used.

Secondary Data Storage

Visualizations in the SCADA software might be reconfigured by the user to show more past measurements. This functionality requires the measurements to be continuously stored while allowing simultaneous access by the software. This type of data storage will subsequently be referred to as secondary data storage.

Implementation details are out of the scope of this thesis, but regular time series databases like InfluxDB or TimescaleDB are possible starting points. The rate of forwarded measurements can be individually limited in the I/O plane such that the secondary data storage can act as a fallback solution. It would then be comprised of the minimum amount of data that can still produce satisfactory results in an analysis.

2.2.3. Automatic and Manual Control

Control during rocket launches and tests can be separated into two categories:

- Manual or supervisory control (e.g., for valve dry-checks)
- Automatic or real-time control (e.g., for valve sequences)

For manual control, commands are sent from the Vis/Int plane of the SCADA system using the SCADA software. This means the plane and especially the software used need to support manual issuing of commands. Additionally, stakeholder requirements state the need to issue emergency commands using some form of hardware initiator, like buttons. These buttons and particularly the emergency stop button are covered in Section 2.2.4.

While the origin of control commands is clear for manual control, automatic control commands can be issued in four different ways:

- Hosted control: Automatic control stems from the Vis/Int plane, i.e., a faulty connection between the Vis/Int plane and the I/O plane causes loss of control authority.
- Embedded control: Automatic control stems from the I/O plane, control authority is maintained even on loss of connection.
- Centralized control: All automatic control stems from one single device; Complete control authority is lost if it fails.
- Distributed control: There exist many automatic control sources with separate control domains. Partial loss of control authority occurs on failure of one controller.

Only embedded control is suitable for the stakeholders, due to requirements R-CHRO-4.1 and DACS12. With multiple flight computers and DAQs, SCADA systems for rocket launches and tests are typically controlled in a distributed way. Therefore, the SCADA system developed in this thesis will use distributed, embedded control.

Control and coordination have to be distinguished. Coordination of different control sources expresses itself in the coordination with other control sources and the coordination with supervising engineers. There are essentially three approaches to achieve the former:

- Coordination of control sources through a centralized main controller in the I/O plane
- Coordination of control sources through the use of the SCADA software in the Vis/Int plane
- Coordination of control sources among themselves

The SCADA system developed in this thesis will exclusively use the second approach. However, combinations of the three approaches are possible and a good solution in some scenarios. Two high-speed control loops on two sepatate controllers, for example, cannot be coordinated from the Vis/Int plane. To achieve this, coordination between the controllers would be necessary while still allowing occasional intervention from supervising engineers.
With approach two, the SCADA software is used for both coordination with supervising engineers and coordination of the control sources. Some advantages this offers are listed below:

- Different control sources do not need to communicate with each other.
- There exists a single source of coordination right where the system is monitored.
- All data that arrives at the Vis/Int plane for monitoring purposes can be used for (automated) system-wide coordination.
- Less (embedded) programming of controllers in the I/O plane is necessary, because communication between controllers does not have to be implemented.
- Coordination errors are restricted to one domain.
- Coordination with data and information that might not be available in the I/O plane like weather, go/no-go polls or launch site restrictions is possible.

As mentioned, this approach is best suited for (low frequency) coordination. It is not optimal for time sensitive automatic reactions to certain events, as data and subsequent reaction commands take the longest possible path through the SCADA network and as such are susceptible to latency.

To summarize the finalized control approach with an example, a rocket launch scenario could look like this: The fueling system on the ground is set up and the supervising engineers want to perform a dry-check of all valves. To switch the valves, they click on corresponding buttons in the SCADA software. Each button is bound to a certain command that is sent to the device connected with the valve. When the engineers click on the purge valve button, a command is sent to their DAQ to switch the valve. Then, they click on the button for the oxidizer main valve in the rocket. Subsequently, a command is sent to the flight computer telling it to switch the valve.
Dry-checks and fueling are now completed, engineers decide that they are ready for launch.

They click a state change button in the SCADA software. A number of commands is now sent out throughout the system. The DAQ is told to close all valves and stop logging temperatures in the fluid system on the ground; The flight computer receives a launch readiness command. It now has exclusive control over the rocket and prepares the engine ignition control loop.

2.2.4. Safety and Reliability

There is a variety of safety measures necessary during rocket launches and tests. This section mainly serves to highlight the interaction of safety measures commonly used by the stakeholders with the web-based approach to SCADA. It will not follow any safety classification procedures as it is merely intended to provide an overview.

Uninterrupted Power Supplies (UPS)
To increase reliability, requirements R-CHRO-6 and DACS21 specify that the SCADA system should use Uninterrupted Power Supplies (UPSs) to keep the system running in the event of electrical power loss. These devices automatically switch to a bank of charged batteries in such an event.
They can be integrated into 19-inch racks, e.g., into the measurement and control station, if it uses a 19-inch rack for mounting. If longer power outages are expected, the power draw of computers used to run the SCADA software needs to be taken into consideration, as the UPS powering them can run out of battery. UPSs are also mentioned here because of their interplay with safety measures described in the next paragraphs.

Residual Current Devices (RCD)
Residual Current Device (RCD)s are an essential safety measure for all electrical systems in use. They detect missing return currents on the mains voltage sources (230V AC in Germany) and cut power to prevent possibly deadly electrical shocks.
It is critical that RCDs are positioned after any UPSs, otherwise only mains power is cut. This would cause the UPS to engage and the dangerous mains voltage to persist (unless the UPS employs an RCD itself).

Emergency Stop Button
WARR previously used a single emergency stop (e-stop) button in the control room. The button physically cut power to the measurement and control station. Inter alia, this causes all valves to return to their unpowered state. To use e-stop buttons, a safety relay, which includes the actual switch cutting the power, is needed. A circuit with the button is connected to the relay. It detects when the circuit is opened, that is, the button is pressed.
Using just one e-stop button is not optimal in the network topology discussed in Section 2.2.1, because it can span multiple hundred meters. It has to be investigated whether e-stop circuits work over such distances. For increased safety and flexibility, a setup with two e-stop buttons, one in the control room and one at the measurement and control station is proposed. In this setup, one safety relay is assumed to be located in the measurement and control station. This means that power will only be cut there and not for any hardware in the control room. It

is questionable whether turning off power in the Vis/Int plane together with the target plane is a good practice. For this, two safety relays would have to be used on the same e-stop circuit. Going forward, it will be assumed that any e-stop circuit uses a single safety relay.

The two e-stop buttons can be installed to interrupt power before any RCD in the measurement and control station or after. Both have to be connected in series on the same circuit. This way, any of the two buttons can be used to cut power. The limiting factor for the distance of the e-stop button in the control room from the test site is the total resistance along its circuit. Safety relays only work up to specific resistances in the e-stop circuit. The value can usually be found in the relays' data sheets.

As discussed in Section 2.2.1, only a single Ethernet cable will be used between the control room and the measurement and control station. The SHDSL Ethernet extenders used at WARR use only four of the eight conductors in the cable. Therefore, two of the four spare conductors can be used for the e-stop circuit to the control room (see Fig. 7). The resistance in the circuit can then be estimated with the wire resistance of the cable used and the distance. With a five percent margin on a total distance of 1.2 km and a wire resistance of 0.06 Ω/m [13], the resulting resistance is 75.6 Ω. This exceeds the limits of some commercial safety relays and therefore special care has to be taken when selecting one for this application [14, p. 17].

Emergency Sequence Buttons

At WARR, any emergency safe state or purge sequences were previously initiated with physical buttons connected to the monitoring computer via the Universal Serial Bus (USB). These sequences are for example used when fire in a combustion chamber needs to be extinguished. This is a suboptimal approach in terms of safety, because the USB connection is handled through the monitoring software. Adding to that, the software runs on a general purpose, not a real-time operating system.

To improve this, the remaining two conductors in the Ethernet cable connecting control room and test site can be used to establish a purely electrical connection. Each button can be wired up to be normally closed to ground. A power supply in the measurement control station can be connected to each button, using 24 V for example. When a button is pressed, the voltage on its conductor increases to roughly 24 V in the measurement control station. This can be detected on a voltage input of an embedded DAQ, which can then execute programmed sequences. Instead of using relays similar to the ones previously discussed, this offers the advantage that just one conductor is necessary per button. Hence, two buttons can be used over the Ethernet cable instead of just one.

The described configuration makes sure that sequences are only executed on the press of a button and that the used voltage does not suffer from a voltage drop over the cable. Variations to this approach are possible and it remains to be tested. There is also the option to use separate voltage monitoring devices for this purpose.

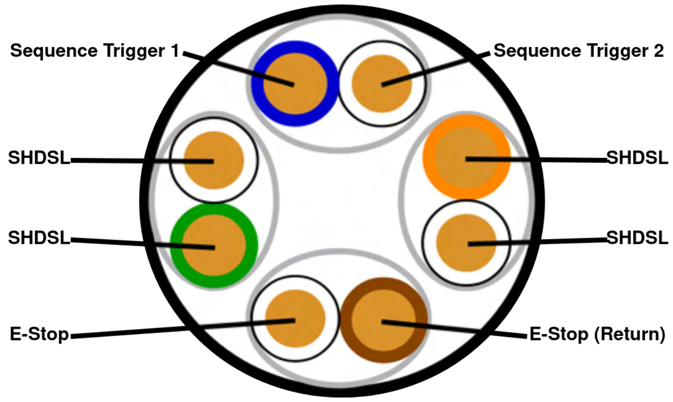


Fig. 7: Cross section of the single cable designated for connection of the measurement and control station with the Vis/Int plane, e.g., a control room. Single-Pair High-Speed Digital Subscriber Line (SHDSL) is a long range protocol that is used to extend Ethernet here.

Software Emergency Buttons

The emergency functions covered by the buttons described in the previous paragraph can be doubled in the SCADA software, if so desired. However, operators will have to be educated about the fact that the hardware buttons are inherently safer than their software counterparts. Additional emergency functions that do not underlie as strict safety limitations can be implemented in software only, if required.

2.3. Hardware Integration

In this section, hardware integration of components (for a list, see Section 2.1) into the SCADA system will be detailed. It is mostly concerned with signal chains and the steps involved like protocol conversion to ultimately arrive at data, that can be sent over the Ethernet network laid out in Section 2.2.1.

2.3.1. Flight Computers

Rocket avionics usually deliver data over physical wires while the rocket is on the launchpad and over radio during and after the flight. The radio connection is assumed to be established while the rocket is on the launchpad.

Wired Connection

Flight computers can have any number of wired communication interfaces. Options range from Ethernet, RS232 to I²C, Universal Asynchronous Receiver Transmitter (UART) and more. Almost all microcontrollers today support UART interfaces. The microcontroller used in the flight computer at WARR as well; Thus, it will be investigated how to adapt UART interfaces to the Ethernet based communication used in the SCADA system.

First of all, UART interfaces of flight computers cannot be routed through the rocket's umbilical, because they are intended for short range communication. Consequently, a first conversion step is required on the rocket. Conversion to RS485/422 was chosen by WARR; It exhibits the following properties:

- UART and RS485/422 are both serial protocols. Conversion is simple and requires only one integrated circuit. This simplicity enabled WARR to successfully use this approach for the first time in less than an hour and without prior preparation.
- RS485/422 can be used over distances of up to 1000 m. This leaves open the option to skip any further conversions and directly wire up the flight computers to the control room, if necessary.
- RS485/422 is a robust protocol, using differential signaling which is barely affected by noise.
- RS485/422 connections only use four wires, compared to eight for Ethernet, reducing weight and wiring complexity on board of the rocket.
- Debugging RS485/422 links with USB only requires a USB adapter and a terminal window on a computer. This can help in development and troubleshooting scenarios.

To integrate an RS485/422 interface into an Ethernet network, a serial device server is necessary (other terms possible). These low power devices offer Transmission Control Protocol (TCP) and or User Datagram Protocol (UDP) ports and forward all characters sent to them over Ethernet to the serial RS485/422 connection. Full-duplex versions support simultaneous communication in both directions, while half-duplex versions only support one direction at a time. For flight computers, full-duplex versions are preferable, as commands can be sent to them while they are streaming data at the same time. The resulting signal chain is summarized in Fig. 8 together with data storage and control concepts discussed in the previous section. How to interface with TCP ports with the SCADA software, will be discussed in Section 3.5.1. Testing of wired flight computer integration is documented in Section 4.1.

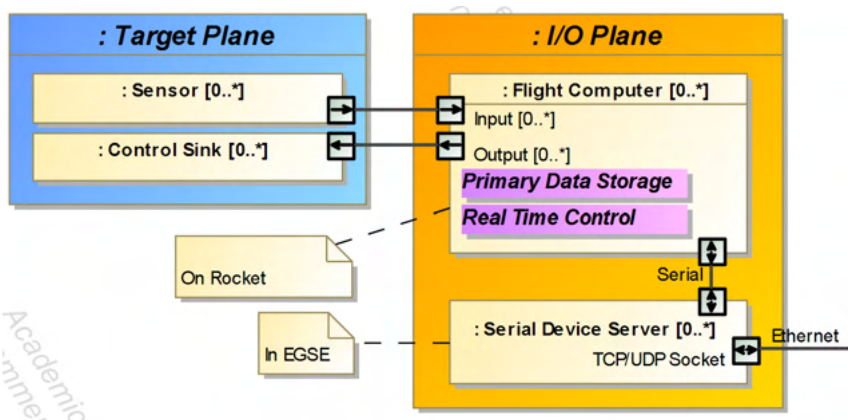


Fig. 8: Systems Modeling Language (SysML) internal block diagram summarizing the signal chain for flight computers. Integration is continued in Section 3.5.1 starting with the Ethernet interface. EGSE stands for Electrical Ground Support Equipment.

Telemetry

Flight computers typically send data over radio during the flight of the rocket. Many different frequencies and modulations are used across the industry. To receive and integrate the data that comes in over radio into the SCADA system on the ground, a number of components are necessary.

First and foremost, an appropriate antenna must be used. It needs to support the polarization and frequency of the signal. Next, an optional component, namely a low noise amplifier, can be used to increase the power of the received signal. This is especially important for trajectories spanning long distances over which atmospheric attenuation can reduce the power of the signal significantly. After the amplifier, a receiver is needed. Receivers demodulate and digitize the (amplified) radio signal into a communication protocol.

This would optimally be Ethernet in this case, but few receivers support it directly. Some intermittent conversion steps might be necessary, similar to the concepts discussed in the wired integration paragraph above. For the signal chain summarized in Fig. 9, it was assumed that the receiver only has an UART interface and necessary conversion steps were included. Interfacing from the SCADA software is then identical to the wired approach, making use of TCP or UDP sockets. This approach applies analogously to other radio systems like locating beacons.

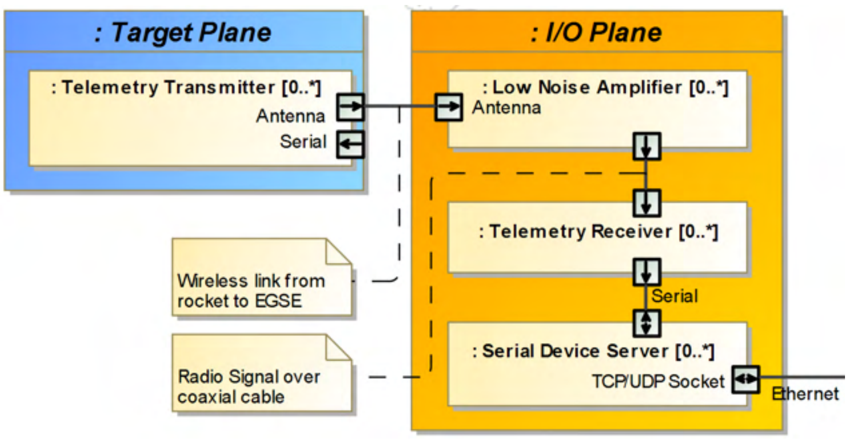


Fig. 9: SysML internal block diagram visualizing the signal chain for telemetry. Integration is continued in Section 3.5.1 starting with the Ethernet interface. EGSE is an abbreviation for Electrical Ground Support Equipment.

2.3.2. Data Acquisition Systems

In this section, an appropriate and unified approach for data transmission to and from DAQs shall be found by defining layer four of the OSI-model (see Fig. 5). It is assumed that DAQs directly support Ethernet on the data link and physical layers. An approach, wherein as much DAQ hardware as possible uses the same communication standard, offers the following advantages:

- Stakeholders can use any software supporting the standard or develop their own (like in this thesis).
- System expansion reduces to "plug and play" in the best case.
- The data transmission is automatically documented through the standard.
- The amount of supported commercial hardware is larger.

This is especially relevant, because DAQs are typically responsible for a large portion of the data produced in test setups.

Most communication standards cover not only layer four of the OSI layer, but a number of layers above. To compare different standards, selected standards are compared in Table IV. Its performance evaluation is coarse; Performance highly depends on network equipment and conditions. Publisher-Subscriber (pubsub) approaches offer advantages in terms of network load, because continuous requests for measurements are not required. In addition, the terms Peer-To-Peer (P2P) and client-server distinguish whether DAQs are able to exchange data among themselves or not, respectively. The last three rows of Table IV look at the libraries available for “Node.js”, the framework that will be used to program the software prototype later on. This can quantize implementation effort and how common a standard is in combination with “Node.js”.

Table IV: Comparison of applicable data transmission standards for DAQs.

	MQTT	OPC UA	DDS	Modbus TCP	ROS	LXI
Year Started	1999 [15]	2003 [16]	2004 [17]	1999 [18]	2007 [19, p. 2]	2005 [20]
Description	Lightweight messaging protocol designed for the Internet of things, unreliable networks and with assurance of delivery [15]	Semantic, platform independent, object-oriented architecture for data exchange [21]	Semantic, "Data-Centric Publish-Subscribe (DCPS) model for distributed application communication and integration" [22, p. 1]	Messaging protocol using "vendor neutral data representation" and action definitions via function codes. [18]	"[L]oosely-defined framework for writing robot software" [19, p. 1]; Includes communication infrastructure with data topics [19, p. 17]	Standard for Ethernet equipped instrumentation with focus on ease of use; Supports multiple ways of time synchronization [23]
OSI Layers	5-7 [24, p. 11]	5-7 [25, p.2] [26, Fig. 2]	5-6 (Middleware) [27]	7 [28, p. 2]	Unclear, likely 7	1-3 [29, p. 3]
Transport Protocol	TCP [24, p. 1]	TCP [25, p.2]	TCP, UDP, shared memory [30],[31]	TCP	TCP or UDP [32]	TCP or UDP [33, p. 41]
Publicly Available	yes	yes	yes	yes	yes	yes
Performance	Low [34, p. 8]	High [34, p. 8]	High [34, p. 8]	Medium[1]	Low [34, p. 8]	Unclear[2]
Client-Server or P2P	Both [24, p. 1]	Both [21]	Both [39, p. 3]	Client-Server [28, p. 2]	P2P [19, p. 2]	Client-Server
Polling or pubsub	pubsub [24, p. 1]	Both supported [40, Part 14]	pubsub [22, p. 1]	Polling [28, p. 2]	pubsub	Polling
Product Availability	Few data acquisition products use protocol directly; Protocol converters available	Data acquisition products, digital output modules and relays available	Few data acquisition products use protocol directly	Large amount of data acquisition and control products [18]	Commercial products are scarce and oriented towards robot development	Large amount of products; Mostly for electrical testing; Few DAQ products
“Node.js” Library	yes [41]	yes [42]	yes [43]	yes [44]	yes [45]	No; C library [46]
Library Maintained	yes [41]	yes [42]	yes [43]	yes [44]	yes [45]	yes [46]
Library Quality	Complete documentation including examples; 385715 weekly downloads [47]	Good Documentation including examples; 7,178 weekly downloads [48]	New library, documentation only on installation, no examples; One weekly download [49]	Good Documentation including examples; 6370 weekly downloads [50]	Extensive Documentation with examples; Only supports TCP; 1801 weekly downloads [51]	Basic Documentation, no examples; 18 GitHub forks [46]

[1] comparing protocol overheads in [34] and [35]
[2] Data on LXI performance is conflicting and sparse. Compare [36, p. 20], [37, p. 128] and [38, p. 11].

Looking at Table IV, the following observations can be made. MQTT is the most popular protocol used in conjunction with “Node.js”. This offers advantages in terms of support and documentation, as the community is significantly larger and more projects are being created. But MQTT lacks professional, commercial data acquisition hardware supporting it. It is less performant than other protocols in Table IV as well.
LAN eXtensions for Instrumentation (LXI) has similar drawbacks. As it originates from electrical testing hardware, like oscilloscopes, multimeters, etc., it lacks in availability of DAQ hardware [20]. It offers significant advantages in terms of timestamp synchronization (e.g., using the Precision Time Protocol) , but it only offers a library for the programming language C [33, p. 21].
The Robot Operating System (ROS) has similar issues, although its library offers extensive documentation with a dedicated "wiki" system [32].
The Data Distribution Service (DDS) can be ruled out due to the absence of a good selection of commercial DAQ products.

Both Open Platform Communications Unified Architecture (OPC UA) and Modbus TCP are equally promising when considering Table IV. In industrial automation, Modbus TCP is currently regularly referenced as an "industry standard" [52, p. 32], while OPC UA is dealt as one of the most promising standards for industry 4.0 [53]. Both standards are suitable for the application at hand. A more detailed look, however, revealed that OPC UA has some advantages over Modbus TCP.
The OPC UA specification describes a mechanism called discovery. With this, a monitoring software (client) can find all compatible OPC UA servers in the network. Additionally, the application can find out what data and control the server has to offer and how this data is organized. Available information about data points is rudimentary by default. Among some more information it includes read and write rights for variables as well as their names and descriptions. However, certain information fields used for variables can be repurposed to describe units, ranges and more. If the equipment is set up for it, this could be used to automatically configure the SCADA software for the measurement equipment that is connected. With this, in contrast to Modbus, OPC UA offers functionality that could be used to satisfy requirements R-CHRO-1.2.2 and R-CHRO-2.3.4 (detect whether sensors, actuators, etc. are plugged in).
Together with the facts that OPC UA can support future developments towards a pubsub setup, that it offers slightly higher performance and that security is taken into account from the get-go, it represents the optimal protocol for DAQ integration among the ones examined in Table IV. Additionally, Amirabbas *et al.* [54, Fig. 5.10] have shown the use of OPC UA in conjunction with DAQs in comparable applications. How OPC UA is used up to the Vis/Int plane is summarized in Fig. 10. Integration is continued in Section 3.5.2. Test results of DAQ integration using OPC UA can be found in Section 4.3.

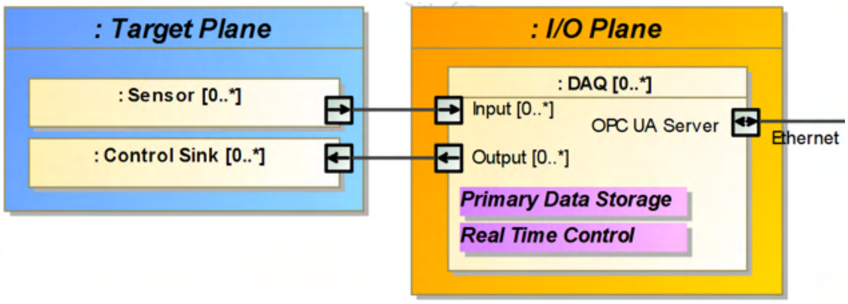


Fig. 10: SysML internal block diagram visualizing data and control flows for DAQs. Integration is continued in Section 3.5.2 starting with the Ethernet interface.

2.3.3. Live Video

Wired Video

Because the network for the SCADA system is Ethernet based, an option for monitoring tests and launches with video in real-time are so-called network or IP cameras. These cameras are ubiquitously used for video surveillance and thus the vast majority support the same transmission protocol, Real Time Streaming Protocol (RTSP). With that, integration is already covered from the hardware side and it is up to the SCADA software to connect to an RTSP port on a camera and display its video stream. This will be detailed in Section 3.5.3.
Network bandwidth limitations over the single long range interconnect between control room and test site, mentioned in Section 2.2.2 still apply. This means that compressed video streams should be used to avoid saturating the connection, if a configuration with said 30 Mb/s limit is used. As the software developed in this thesis is web-based, only h.264, not h.265 compression can be used, because of browser incompatibility [55]. Any other codec will require transcoding and add latency. There are very few browser-supported protocols that keep video latency below one second. Avoiding transcoding is one of the best measures to ensure low latency. Further details on how latency is kept low can be found in the Testing Chapter (see Section 4.2).
Some IP cameras support audio as well. Due to the web-based approach used here, audio codecs are limited. Only Pulse Code Modulation (PCM) (alaw or ulaw) is supported by the low latency software integration used in Section 3.5.3.

Wireless Video

Live video from rockets is achieved similarly to the live data transmission from flight computers. The video is sent over a specific frequency (e.g., 5.8 GHz) with a specific modulation. The same combination of antenna, low noise amplifier (LNA) and receiver is necessary. At WARR the video signal transmitted is analog and as such the output of the receiver on the ground as well.
To capitalize on the SCADA software already having to support low latency playback of RTSP video from IP cameras, live analog video received over radio can be converted accordingly. To achieve this, an analog video server (other terms possible) is necessary. Devices sup-

porting local recording of the video to an SD-card are available. This adheres well to the data storage approach developed in Section 2.2.2. The server takes in the analog video and makes it available to the SCADA system's Ethernet network as an RTSP stream. Software integration is identical to the one necessary for IP cameras from that point on, it is laid out in Section 3.5.3.

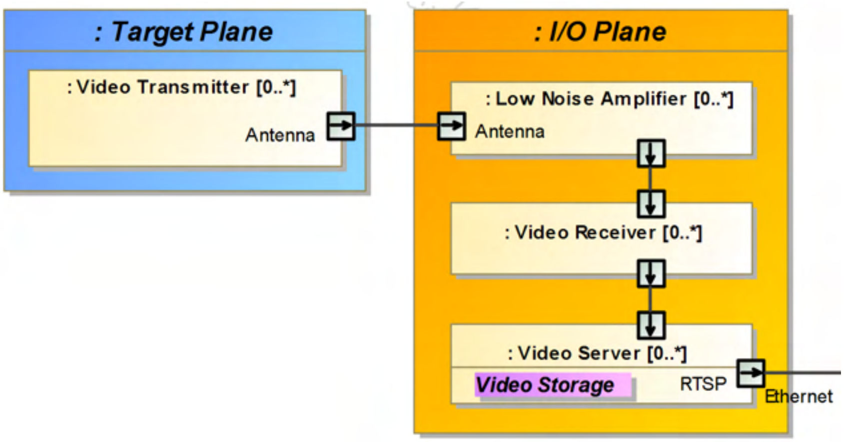


Fig. 11: SysML internal block diagram visualizing video stream conversion. Integration is continued in Section 3.5.3 starting with the RTSP interface.

2.3.4. Serial Devices over USB

As mentioned in Section 2.3.1, serial devices like flight computers can be directly attached to computers to enable debugging. To use the visualization capabilities of the SCADA system in such a scenario, integration of USB connections has to be investigated.
Usually the only device required is an appropriate serial to USB adapter. Data coming from this connection is not part of the SCADA system's Ethernet network, but it can still be made available to it through the SCADA software, if necessary. How the software is able to access the data in the first place is detailed in Section 3.5.4. This scenario has also been tested and results can be found in Section 4.1.1.

2.3.5. General Purpose Computers

Computers running SCADA software or other software represent data sources themselves. As they are part of the Ethernet network used for the SCADA system already, no further hardware is necessary for integration. Short testing documentation can be found in Section 4.4 and the software approach used is described in Section 3.5.5.

3. Software Design

This chapter deals with all aspects of designing a web-based SCADA software prototype to achieve objective two of this thesis (see Section 1.3).

Requirements
Table A.1 and A.2 in the appendix contain requirements from WARR and the SPM for their respective SCADA systems. Due to the large amount of requirements they will only be referenced in this section, not explicitly stated.

Chapter Structure
This chapter is structured into six parts. To begin, the concept of web-based software is introduced in Section 3.1. Sequently, the conceptual software project EX-UI, which was started at WARR, is presented as the basis for the SCADA software developed in this chapter.
Explanation of software concepts and design is then split up over a frontend section (3.3) and a backend section (3.4). Then, integration of target plane components started in the previous chapter is completed in Section 3.5. Finally, this chapter ends with a brief overview over the software repository created (Section 3.6).

3.1. Introduction to Web-Based Software

Definition
Web-Based software runs in the web browser. As such, it can be considered a website or web application (terms can and will be used interchangeably). It cannot run outside the environment of the browser. Hence, its code does not run directly on a computer like native computer programs, but on the additional layer of the browser.

The Browser Environment
The browser can be considered an "entire operating system" [56, chapter 14] specifically targeted towards delivery of web applications. It loads website code and runs it in a constrained and standardized environment. This is what enables the Internet as it is known today, where websites can be used on essentially any device with a browser. As such, browsers can be considered a tool for simple, widespread distribution of software.
The vast majority of modern browsers are able to run JavaScript code, which is almost exclusively used to program websites today [57].
The browser environment specifies a suite of Application Programming Interface (API)s, called web APIs. This suite can vary slightly between browsers. Chrome, for example supports direct access to serial devices connected to a device's USB port, while Firefox does not [58]. These APIs can be used by the website loaded into the browser and as such, a large part of them are JavaScript-based APIs.

Browser Networking

Because browsers are such a constrained environment, there are only a handful of ways for websites to communicate with endpoints outside it. To begin with, browsers only support a single communication protocol. It is the Hypertext Transfer Protocol (HTTP) protocol, which is used to load a website when a user requests it through the browser's address bar. Any additional modes of communication are specified by the browser's web APIs.

When a user requests a website with an address, the browser requests an "index.html" file from the server corresponding to that address. Servers like this are called **web servers**, because they supply the website content to the browser. The website's index describes all the content on the website, including JavaScript code, images, text and more. After loading the index, the browser loads those contents from the web server as well. Web servers can be considered the **backend** of a web application, the website code as the **frontend**.

As soon as the website code is loaded into the browser, it can use the browser's web APIs to establish additional connections itself. There are only a handful of web APIs that enable additional communication with servers. The most relevant ones are depicted in Fig. 12. The web application developed in this thesis will widely rely on WebSockets, for instance.

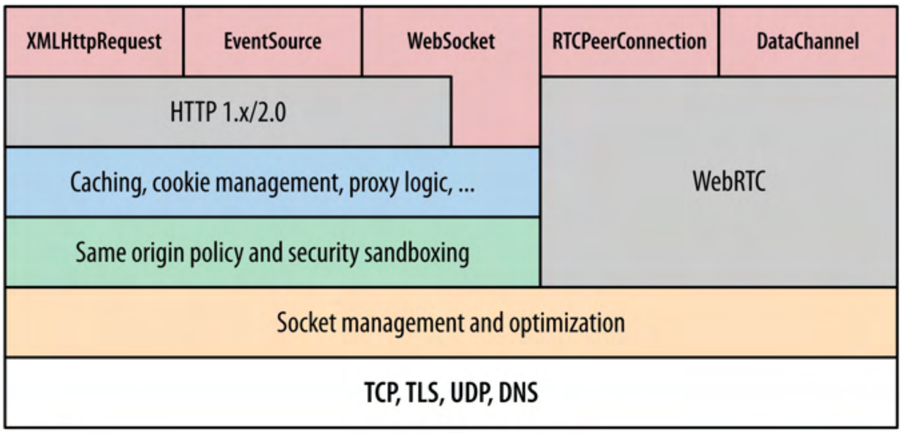


Fig. 12: An overview of most relevant networking APIs available in modern browsers. The white colored layer containing TCP and others corresponds to the fourth OSI layer. [56, Fig. 14-1]

Web Stacks

The collection of all code components used to realize a web application across the front- and the backend is called the **web stack**. In general, web stacks can be categorized in the following, going from the frontend to the backend:

Frontend:

- **Styling:** Cascading Stylesheets (CSS), Syntactically Awesome Stylesheets (SASS), . . .
- **UI Framework:** “Vue.js”, React, . . .
- **Website Code (or Runtime):** JavaScript, TypeScript, . . .

Backend:

- **Web Server:** Express, Django, PHP, . . .
- **Database:** InfluxDB, CouchDB, . . .
- **Module Bundler:** WebPack, rollup, . . .
- **Server Runtime:** “Node.js”, Python, . . .
- **Deployment:** Docker, Kubernetes, . . .

An essential aspect of any JavaScript based website is that it runs in the JavaScript runtime environment of the browser, which is fundamentally single-threaded and event-based. To illustrate: A while-loop in the website’s JavaScript code without an exit condition will block the single thread it is running in. This effectively stops any further code from being run and can cause the browser to behave in an unstable way or even crash. This is why in the JavaScript runtime, any code portion that might exhibit such behaviour, like a WebSocket connection waiting for data, is deferred to the so-called event loop of the browser’s JavaScript runtime [59, chapter 3]. The runtime will not be further explained here, but documentation by the developers of Firefox [60] offers further details.

3.2. EX-UI — A Starting Point with Guidelines

a: The EX-UI logo.

b: The WARR logo.

Fig. 13: EX-UI and WARR logos. The letters "U" and "I" were designed to bear resemblance to the WARR logo.

EX-UI is a software project that was started at WARR to solve the problems mentioned in Section 1.1. The name EX-UI stems from the currently common naming scheme for rockets at WARR. It is in a conceptual, path finding state and will serve as a basis for the SCADA software development here. The next sections will design its structure and add first capabilities.

As a result of the problems it tries to solve, EX-UI comes with fundamental development principles that need to be taken into consideration:

- As much of the software start as possible is to be handled by just one "click" or action.
- Green colors are positively connoted, yellow colors negatively and red colors more so.
- As much of the UI as possible is to be designed so that it can be understood intuitively. Visual elements resembling their real counterparts are preferred.

- The software must be written to be as reliable and dependable as possible.

Additionally, WARR requires EX-UI to support use with flight computers; This is a new capability that was previously not available. As can be observed, EX-UI's principles are influenced by the fact that it is mainly intended for use in a student initiative environment. Another consequence of this is the use of a preexisting code base, to reduce required development workload and build upon external documentation. The main code library EX-UI uses is Open Mission Control Technologies (OpenMCT). It is detailed in Section 3.3.1.

3.3. Frontend Architecture

3.3.1. Overview of OpenMCT

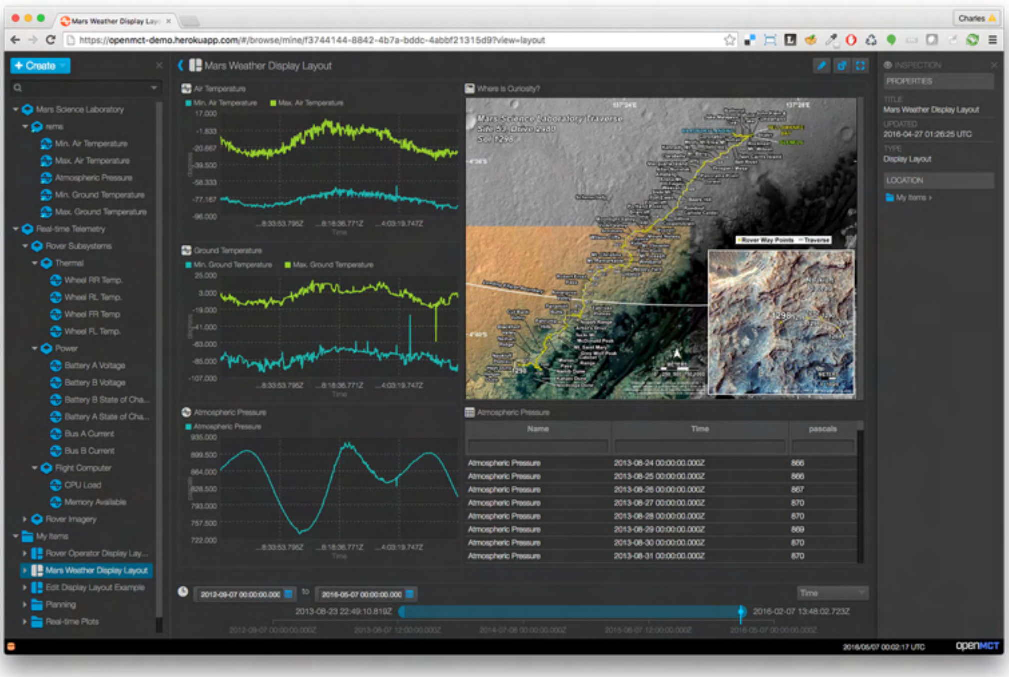


Fig. 14: This is one of the main visualization screens in the OpenMCT live demonstration available online. The ready-made graphical elements, like the real-time plots and the map, make it particularly suitable for this application. [61]

OpenMCT, as described on its website, is an open-source, "next-generation mission control framework for visualization of data" [62] and can be used "as the basis for building applications for planning, operation, and analysis of any systems producing telemetry data" [62]. It is being jointly developed by the National Aeronautical and Space Administration's (NASA) Advanced Multi-Mission Operations System (AMMOS) team, the Ames Research Center and the Jet Propulsion Laboratory (JPL) [62]. It was created in 2014[3] to satisfy the "rapidly evolving needs of mission control systems", including "distributed operations, access to data anywhere [...] and flexible reconfiguration to support multiple missions and operator use cases" [62].

[3] Based on year of first commit to public GitHub repository

The OpenMCT Git repository[4] is being released under version 2.0 of the Apache License. It can be used directly, because it includes a complete web stack. Accompanying tutorials are located in a separate repository[5]. OpenMCT is mainly documented through a developer and a user guide as well as an API reference [63], [64], [65].

To use OpenMCT for a project it is mainly modularly expanded via plugins (in the frontend) [65]. As its code is publically available, it can be modified directly as well. The use of OpenMCT impacts development of EX-UI across the front- and the backend. OpenMCT release 2.0.5 and its web stack are adopted one-to-one for EX-UI. An overview of the web stack is given in Fig. 15.

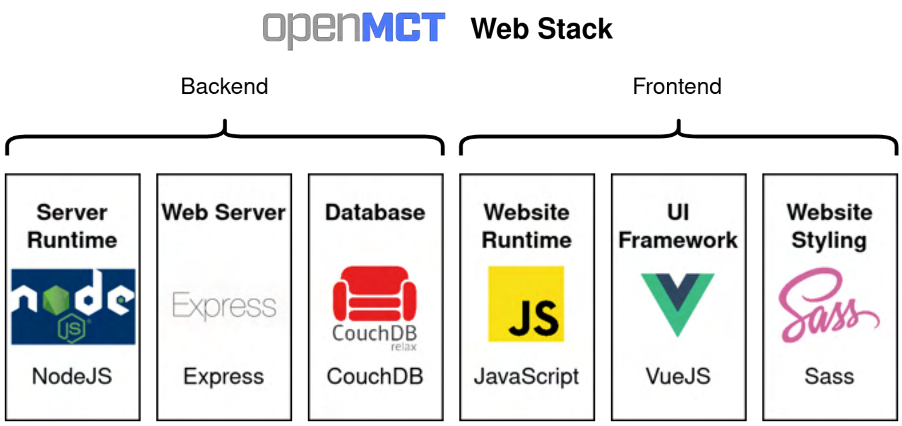


Fig. 15: The web stack OpenMCT uses by default. It was found by analyzing its repository [66]. Logos adapted from [67], [68], [69], [70], [71], [72], [73]

OpenMCT offers a variety of configurable data visualizations out of the box [64, p. 40]. The most relevant ones for applications in rocketry are listed below:

- Overlay Plots (multiple data series in one plot; supports single data series)
- Stacked Plots (multiple data series constrained to horizontal sections of one plot)
- Bar Plots
- Tables
- Display Layouts (configurable positioning of multiple visualizations)
- Flexible Layout (Automatic sizing and positioning of multiple visualizations)
- Tab Views (Contained views accessible via clickable tabs)
- Web Pages

[4] OpenMCT's repository can be found under `https://github.com/nasa/openmct`
[5] OpenMCT tutorials can be found under `https://github.com/nasa/openmct-tutorial/`

All of the above mentioned visualizations are created by the user: Creation is started with the plus button in the upper left of the UI. In the creation screen, data points can be dragged onto visualizations. Colors are customizable in most visualizations and some of them, like plots, additionally offer individualization of interpolation, data point geometry, y-axis range and more [64, pp. 52-57]. Full- and multi screen setups are supported natively; Visualizations can be distributed across multiple browser tabs, which can in turn be distributed over multiple screens.[6]
All visualization types, buttons and more are stored in a hierarchical directory-like structure, which is displayed on the left of the website. Objects stored there are called domain objects by the OpenMCT developers [63, p. 22]. The domain object tree will play an essential role in displaying all relevant data sources and control sinks of the SCADA system's target plane.

The ability to embed other websites into OpenMCT is essential for its expansion with new visualization types. If a specific visualization is not supported by OpenMCT, it can be added by developing an appropriate website and embedding it. Multiple integration approaches will make use of this capability. OpenMCT's API for adding custom visualizations directly is currently still under development [65].

3.3.2. UI Modifications in OpenMCT

The OpenMCT library was customized in its visual appearance to better fit the stakeholders and their typical applications:

- Create Button: Moved out of website header into domain object browser; Made significantly smaller due to unlikely use in operational environment.
- Colors: The standard color theme of OpenMCT was replaced by a theme adhering to the WARR corporate design guide.
- Status Indicators: Website header was updated to include all so-called indicators available in OpenMCT, most notably a performance indicator displaying frames per second.
- Real-Time Mode: Enabled by default; Standard plot time range set to show the last minute and the next 5 s.

UI modification in OpenMCT is mainly performed through SASS styling files and the main "layout.vue" file, describing the website's layout on a high level.

3.3.3. Equipment Configuration File and EQ Plugin

Because the contents of OpenMCT are highly dependant on the hardware that is to be monitored and controlled, a single **equipment configuration file** is defined to serve automatic configuration across all code domains. It represents one of two user-centric files in EX-UI.

[6] An online demonstration of OpenMCT can be accessed publicly under `openmct-demo.herokuapp.com/`(Last accessed Aug. 2022) .

The JavaScript Object Notation (JSON) file format is used here, because it is natively supported by JavaScript as well as human readable and editable. JSON editors with syntax highlighting and tree-style organization options are publicly available online. OpenMCT stores created UI layouts as JSON files too (see 3.3.4). JSON files do not support comments, however. An example of how to include a similar functionality anyway can be seen in Listing 1, which contains a minimal version of an equipment configuration file.

Listing 1: Minimal example of an equipment configuration file

```
{ "title": "Minimal EQ configuration",
          "description": "Minimal Example, one data source with two data points",
          "computers": [{ "name": "Main Computer",
                          "ip": "192.168.178.22",
                          "type": "manager"
                        }],
          "datasources": [{ "name": "Secondary Flight Computer",
                            "key": "sfc",
                            "type": "TCP",
                            "ip": "192.168.1.3",
                            "sourceport": 10001,
                            "destport": 9001,
                            "pollcommand": "Zs",
                            "pollrate": 2,
                            "delimiter": "=",
                            "datapoints": [{ "name": "Ambient Pressure",
                                             "key": "sfc.apress",
                                             "label": "apress",
                                             "values": [{ "key": "value",
                                                          "name": "Value",
                                                          "units": "Pascal",
                                                          "format": "float",
                                                          "min": 0,
                                                          "max": 120000,
                                                          "hints": { "range": 1 }
                                                        },
                                                        { "key": "utc",
                                                          "name": "Timestamp",
                                                          "format": "utc",
                                                          "hints": { "domain": 1 }
                                                        }]
                                           },
                                           { "name": "log",
                                             "key": "log",
                                             "label": "log",
                                             "values": [{ "key": "value",
                                                          "name": "Value",
                                                          "unit": "raw",
                                                          "format": "string"
                                                        },
                                                        { "key": "utc",
                                                          "name": "Timestamp",
                                                          "format": "utc",
                                                          "hints": { "domain": 1 }
                                                        }]
                                           }]
                           }]
        }
```

Listing 1 describes a single data source supplying two **data points**. Both data points consist of their actual (measurement) value and their timestamp. The description of data points follows OpenMCT's definition closely [65]. Notably, data points do not represent a single measurement, but a series of measurements sequentially arriving in real time. As such one data point is associated with one measurement channel on a DAQ, for example. In this case, the data source is of the type "TCP". Integration of such sources is detailed in Section 3.5.1. For now, it suffices to note that the additional parameters "pollcommand", "pollrate", "delimiter" and "label" describe how data can be requested from the data source and how its response can be deconstructed.

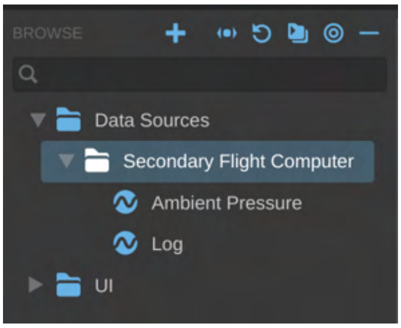


Fig. 16: Domain object tree created from Listing 1 using the EQ plugin.

To automatically parse the equipment configuration file into domain objects, a plugin named EQ was written. Its code is attached in Appendix C.6. The resulting object tree can be seen in Fig. 16.
OpenMCT plugins are written in JavaScript and run in the browser. The process of installing a new plugin is detailed in OpenMCT's tutorials. However, explanations there neglect that new plugins have to be registered in the file "plugins.js" in the "src/plugins" folder of OpenMCT.

3.3.4. UI Configuration File

OpenMCT supports storing the interface created by the user in a JSON file. This file can later be loaded back into OpenMCT, by right clicking on the UI folder in the object tree and choosing the corresponding option in the context menu. The term **UI configuration file** will be used from here. Together with the equipment configuration file it represents the two pillars of EX-UI's configuration by the user for a specific application (e.g., a test stand).
Most notably, any given UI configuration file just maps data points of the object tree onto visualizations. This means that if the equipment configuration file is changed, visualizations still referencing old data sources will not display any data. Moreover, the UI configuration file is not intended for manual edits.

The UI configuration file enables managing different "channel configurations" of devices in the I/O plane. This applies to flight computers and DAQs, for example. **While the equipment configuration only generally describes measurements, the UI configuration ascribes meaning to them.** For example: If a pressure sensor port on a DAQ is used to measure the pressure of a liquid oxygen tank, the equipment configuration only describes its data point as "Pressure Sensor Channel 1". The UI configuration on the other hand, would create a plot from the data point and name it "LOx Tank Pressure". If the port on the DAQ was now used to measure helium tank pressure instead, the equipment configuration would remain unchanged, while the plot in the UI would have to be renamed.

Because all code used for EX-UI is in the public domain, it could technically be used to enable calibrations for measurements. Namely, when using a current based pressure sensor, EX-UI would calculate the corresponding pressure values from it and display them in plots. However, calibrated measurements are typically necessary in the I/O plane itself to be used in control loops or state control. Because of this, calibrations are retained as an embedded programming task (for DAQs, flight computers, etc.) going forward.

3.4. Backend Architecture

Because EX-UI mainly uses the default OpenMCT web stack (see Fig. 15), backend code is written in JavaScript. JavaScript was originally developed for use in the frontend, but with the server-side runtime “Node.js”, it can be used in the backend too. As it is event-driven by default, developing efficient communication code is simpler than in some other languages which might have to be extended to support event-driven developments. Event-driven programming is especially suited for communication, because code is only executed when data needs to be sent or arrives.

Web Server
The web server used in OpenMCT is an Express web server. It is programmed in JavaScript and was only slightly extended to host the equipment configuration under `http://<serverIP>:8080/eq.json`. In this way, any EX-UI user can view the current equipment configuration, even if they are not in possession of the file.

Databases
OpenMCT comes with CouchDB as a default database to store user-created objects. For secondary data storage, an additional time-series database like InfluxDB has to be implemented. This is however not in the scope of this thesis.

3.4.1. Adapter Concept

To integrate devices into EX-UI, intermediary code on the server side is needed to map their interfaces to an interface browsers support. Code that only translates commands and data between interfaces like this will be called **adapter** from now on (akin to physical adapters used for cable connectors). Tests (see Section 4.1.1) revealed that WebSockets are suited as a browser-oriented interface for data that can be directly visualized by OpenMCT. For example, to integrate the TCP based secondary flight computer described in Listing 1, an adapter would be required that acts as a TCP client on one side and a WebSocket server on the other. However, The flight computer likely supplies data that cannot be directly visualized by OpenMCT. As previously mentioned, this data can instead be visualized by hosting a separate website. To keep adapters as modular as possible, this task will be handled in an additional, separate adapter. Otherwise, adapters would have to additionally include a web stack for each incoming data point that is not supported by OpenMCT. This concept will be used to integrate flight computers in Section 3.5.1, which contains a matching visualization in Fig. 23.

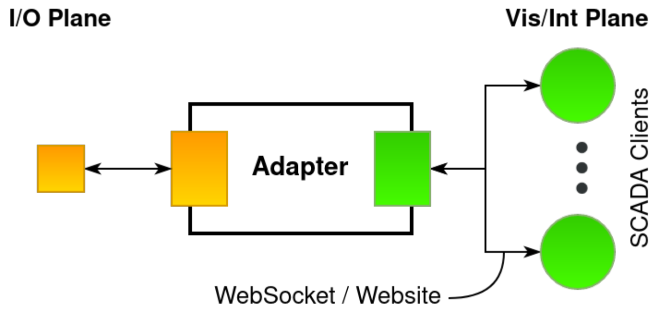


Fig. 17: Adapters in EX-UI visualized. The protocol between the I/O plane and the adapter is usually defined by the device that needs to be integrated.

Development of WebSocket based adapters is essentially identical for adaption of all communication interfaces and can be structured into three stages:

1. Import the WebSocket library and the library suited for the communication protocol to be adapted.
2. Implement access to the equipment configuration for determination of which device needs to be connected to.
3. Define all event handlers associated with both libraries. This includes programming what happens if data arrives from the device, for instance. Transcoding can be implemented at this stage.

3.4.2. Redundant, Distributed Computing Using Docker

Some adapters, for instance adapters for live video, can be quite computationally intensive. With the number of adapters needed, it becomes increasingly unfeasible to run all code for EX-UI on just one computer. Aside from the single point of failure this approach would create, all of the adapters have to be started in some way that does not clash with the "one-click" philosophy of EX-UI. In summary, to solve this, a multitude of adapters and a web server have to be started on multiple computers by just one click. The solution found to accomplish this is an open-source tool called Docker. It is introduced in the following.

Introduction to Docker

Docker can be considered a deployment tool for software. At its core, it supplies a consistent environment for software to be installed and run in across multiple platforms. The environments that can be created with Docker are called **containers** [74, p. 30]. To define a container for a specific software, a Dockerfile has to be created, describing its installation and start process. In this way, the software can be installed on any computer that has Docker and access to the Internet by using *only* its Dockerfile [74, pp. 29-32]. Docker also ensures that containers are automatically started on computer startup.

Docker Compose

Docker can be expanded with Docker Compose, an additional tool facilitating installation and connection of multiple Docker containers [74, p. 35]. To use it, a "YAML Ain't Markup Language" (YAML) file called "docker-compose.yaml" is required, listing container amount and names, their Dockerfile locations, networks, ports, restart policies and more. By using the command `docker compose up`, multiple software components can be installed and run. Notably, how and if Docker is supposed to restart containers on an (unexpected) exit can be defined [75]. This offers potential to increase EX-UI's dependability.

Docker Swarm

As with compose, Docker can be expanded with Docker Swarm. Swarm enables control over software deployment on any number of computers called nodes. As long as a computer has Docker installed, it can be added to a swarm either as a manager node or a worker node [76]. Manager nodes are in charge of maintaining the software's optimal state. Crucially, they can redistribute containers to other nodes, in case one fails. Which containers are preferred to be running where can be specified in the docker-compose file as well [75]. For instance, video transcoding containers could be preferably distributed to nodes that have capable graphics cards.
It is important to note that no setup is necessary on the nodes except installing Docker and adding them to the swarm. Docker automatically distributes the Dockerfiles to the nodes, which can then subsequently use them to install their software component. Again, only a single command is necessary to install and run the software on all computers: `docker stack deploy`. Here, the term stack refers to all combined software components described by the docker-compose file.

Swarm Networking Concepts

At the point where containers are programmatically distributed over multiple computers with different IP addresses, users cannot know for certain which IP address to use to access a desired container. To solve this, Docker swarm uses a mechanism called the **ingress network**, which automatically forwards requests for a specific container to the computer it is running on.
For communication between containers, a so-called **overlay network** can be defined, stretching over all containers that need to communicate with each other. Docker does this by default when deploying a stack [77].

Applying Docker's concepts to EX-UI yields the following conclusions:

- Each adapter and the web server become a Docker container.
- Adapters and the web server are their own separate software components, each containing their own Dockerfile.

- EX-UI with all its components is defined through a "docker-compose.yaml" file.
- Every computer that is intended to run part of EX-UI requires Docker and needs to be added to the Docker swarm.
- All components needed to make EX-UI work are distributed, installed and run on multiple computers by the `docker stack deploy` command.

A summary of relevant Docker concepts is depicted in Fig. 18.

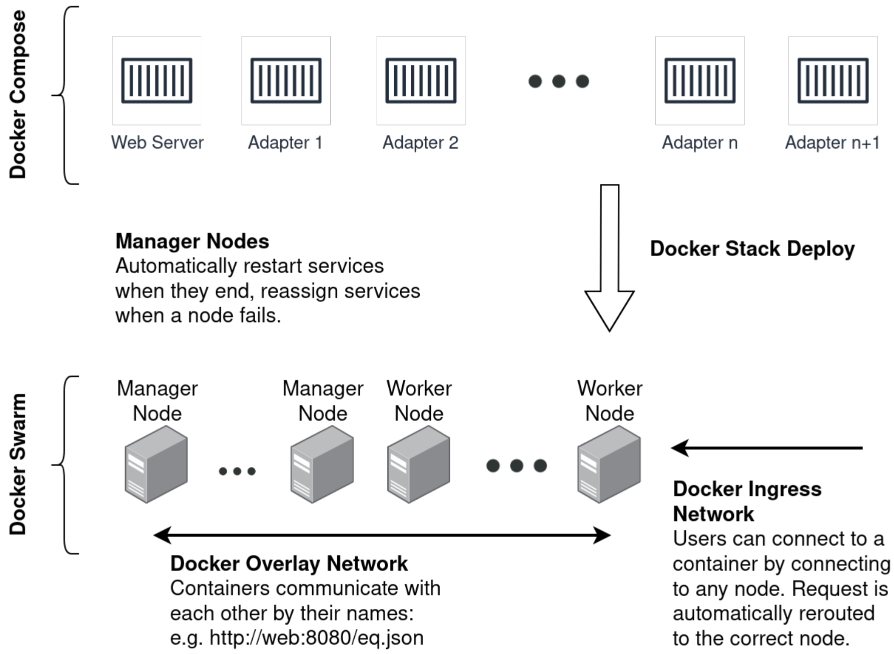


Fig. 18: Visualization of Docker's core concepts. Each container requires its own Dockerfile describing installation and execution.

The Docker Registry

To enable the distribution of Docker containers over a swarm, a Docker registry is necessary. It distributes so-called Docker images, which are compiled versions of the Docker containers in the stack [74, p. 35], [75]. There exists a public registry, but for this application a registry must be deployed locally to enable stakeholders to start EX-UI without Internet access. As a consequence of the registry concept, the complete EX-UI repository is only necessary on one computer in the Docker swarm. Other computers only need the installation script described in Section 3.4.3. Another effect of this that was observed in testing is that using the same base layer image across all adapters and the web server requires less memory and build time.

Scaling Adapters with Data Sources

If each adapter handled all data sources of its type, ensuring that it works with more than a few would be necessary. The same code would have to handle multiple bidirectional connections concurrently. To enable this in a way that does not cause significant loads on the Central Processing Unit (CPU), handling of each data source has to be performed in a separate, event-based process. The event-based approach ensures that the CPU is only used when data or commands arrive. Attempts to implement this approach in Python were unsuccessful and revealed that it significantly increases programming complexity. Docker already includes features that offer a comparatively simpler solution to this issue.

There are four mechanisms built into Docker compose that allow simple scaling of adapters with the amount of their respective data sources [75]:

- **Replicas:** Enables scaling number of instances of a container.
- **Configuration sharing:** Docker allows to share a configuration file with each container in the stack by mounting the file into each container's file system.
- **Environment Variables:** Each container can be given environment variables prior to startup.
- **Port mapping:** Each container replica can supply its services on the same port, Docker maps it to a unique port in the swarm that can be accessed through the ingress network.

If every adapter is developed in an event-based fashion, the remaining step is to create a new process for each adapter. This can be handled with Docker's replica mechanism. No additional coding in the adapter is necessary. "Node.js" was chosen as the programming environment for the adapters exactly for this reason; It is event-based by default.
Continuing, if every adapter handles only one data source of its type, Docker can tell it which data source it is assigned to by passing the number as an environment variable. Docker's "{{.Task.Slot}}" property contains the current replica number and will be used for this purpose [78]. How this approach is reflected in the docker-compose file can be seen in Appendix C.7.
Thirdly, to give every adapter access to the equipment configuration, it can be mounted into each container's file system using Docker's configuration file functionality. By using this approach, letting a changed equipment configuration take effect only requires a restart of the Docker stack.
Finally, to make sure each adapter is accessible by the browser on a unique port, Docker can assign a unique port in a range to each adapter. For adapters using WebSocket connections, the default port 9000 was selected.

3.4.3. Automation Scripts and Resulting User Experience

To adhere to EX-UI's "one-click" philosophy, its installation and start have to be automated as much as possible. For this, the programming language used has to satisfy the following integration requirements:

- JSON integration to parse equipment configuration file.
- YAML integration for automatic creation of docker-compose file from equipment configuration.
- Graphics integration to show prompts for password, equipment file selection and more.
- Docker integration to deploy the EX-UI stack.
- Secure Shell (SSH) integration to automatically add computers to the Docker swarm.

Python was chosen here, due to its use for similar tasks in professional settings, in the GÉANT project for example [79]. It is also well known in the student environment and a beginner-friendly programming language lowering barrier of entry for future modifications. Moreover, most modern Linux distributions come with Python 3 installed by default. The standard libraries subprocess and os enable visually requesting a user's password to authorize installation tasks. Docker integration can be handled with subprocess as well. Furthermore, PySide6 enables creation of modern graphical user interfaces. Lastly, Python supports JSON, YAML and SSH through JSON, PyYAML and paramiko libraries respectively [80].

Installation Script

An installation script was written to automate installation of platform and Python dependencies for EX-UI. Its code is attached in Appendix C.1. Some of its functionality was implemented with code from a publicly available installation script, which was developed by the Leibniz Supercomputing Centre for the GÉANT project [79]. EX-UI's installation script needs to be run on every computer that is to be used as part of EX-UI's Docker swarm. It manages the following tasks sequentially:

1. Determination of operating system, available package manager and graphics environment
2. Check for Internet availability
3. Visual prompt for password (Password is only stored in volatile memory and deleted after installation tasks requiring it are finished)
4. Determination of necessary installation steps by checking already satisfied requirements
5. Installation of system dependencies like Docker and SSH
6. Addition of a dedicated "exui" user with standard password for later SSH access

7. Installation of Python packages used in the start script
8. Installation and start of a Docker registry
9. Copying of OpenMCT plugins into their dedicated directory in the web server

This process was only implemented for systems running Linux with the advanced packaging tool (APT) as the package manager and zenity as the tool for visual dialogues; It was tested on computers with the operating system Ubuntu 22.04. The script only requires Python 3 to be installed on the computer prior to execution and it uses default libraries exclusively. In theory, it can be expanded for usage on Windows and macOS as well; Necessary expandability was respected in development.

Start Script
The start script automates all tasks necessary to build and deploy EX-UI's Docker stack in a way that suits the equipment configuration. Its code is appended in Appendix C.2, for which a public Git repository by W. Magalhaes [81] served as a basis. The start script runs through the following process:

1. Show a start screen with a button to open a file selection dialog for the equipment configuration. (see Fig. 19a).
2. Parse the equipment configuration and reorder data sources alphabetically by type.
3. Show a progress bar for all subsequent processes (currently not functional).
4. Check whether local IP address matches a computer of type "manager" in the equipment configuration file (Docker stack can only be deployed from a manager node).
5. Initialize the Docker swarm.
6. Add computers to the swarm using the SSH endpoints created by the installation script.
7. Create and store a docker-compose file that fits the equipment configuration.
8. Build the Docker stack.
9. Push images used in the stack to the Docker registry for distribution over the network.
10. Deploy the stack and show completion screen (see Fig. 19b).

At the center of this process lies the creation of the docker-compose file which describes all Docker containers that have to be run in the swarm. The port mapping for replicas in the docker-compose file is specified in the format "9001-9002:9000". The port number right of the colon specifies the WebSocket port used by the container, the range to the left specifies the port actually published in the ingress network. To create this format automatically, the data sources in the equipment configuration are reordered by their type and the configuration is saved. An exemplary Docker compose file generated by the start script is attached in

Appendix C.7. Containers that need to be run on every computer in the swarm can be deployed in global, instead of replicated mode. This was tested successfully with the computer monitoring container Netdata, further described in Section 3.5.5 and Section 4.4.

a: Start screen with button for selection of the equipment configuration file from the user's file system.

b: Start progress screen with progress bar after successful start of EX-UI.

Fig. 19: The two different UIs shown by the start script using PySide6.

Resulting User Experience

The user experience resulting from the developed backend architecture can be summarized in the instructions for initial installation and start of EX-UI:

1. Clone EX-UI's repository to obtain installation files (only required on computer used for running start script).
2. Run installation script on every computer that is to be running EX-UI as part of the Docker swarm.
3. Create equipment configuration: Define data and control sources; List all participating computers with their IP addresses.
4. Run start script on a computer that is to be used as a manager node.
5. Access OpenMCT via `http://<AnySwarmNodeIPaddress>:8080`.
6. Create visualizations from the data points automatically listed in the domain object tree.
7. Aggregate visualizations in folders or layouts.
8. Store the UI configuration for later reuse by right clicking the UI folder and choosing "export to json" in the context menu.
9. Monitor and control test or launch.

Most notably, the Docker ingress network enables using any node IP address to connect to a service. E.g., to connect to the web server from any swarm node, one can use any swarm node IP address or the loopback IP address "localhost". It is the web server's port, not its IP address that uniquely identifies it in the swarm. To access a service from outside the swarm, from a smartphone or tablet in the same local network for example, one can use any node IP address. This was tested successfully.

After a successful execution of the start script, the EX-UI stack is deployed with the standard name "exui". With the command `docker stack ps exui` it is possible to investigate which containers are running how often and where in the swarm. For a basic equipment configuration with two computers, one TCP data source and one benchmark data source, the command yields the results shown in Listing 2.

Listing 2: Result of the `docker stack ps exui` command (shortened). It shows which containers are running where in the Docker swarm and status information. Note that execution of EX-UI is distributed over two different nodes.

```
NAME               IMAGE                                NODE
exui_benchmark.1   127.0.0.1:5000/exui_benchmark:latest   Arcticgnu
exui_tcp.1         127.0.0.1:5000/exui_tcp:latest         Arcticgnu
exui_web.1         127.0.0.1:5000/exui_web:latest         Ubuntubook
```

All containers are configured by the start script to log any console output. To access the log of a specific container, the command `docker service logs -f <container id> --raw` can be used. It is useful to investigate why a container is exhibiting abnormal behavior, for example.

The stack can be shut down by issuing the command `docker stack rm exui`. If it is left running at shutdown of the computer that the start script was run on, it will automatically restart when the computer is booted up. Due to this, the start script does not have to be run again until the equipment configuration has been changed. In a static test setup where target and I/O plane are not modified after setup, the start script would rarely have to be run, for instance.

3.5. Software Integration

This section's goal is to find suitable and performant code libraries for adapters. Each device type that needs to be integrated receives a separate adapter. Devices providing data that requires special visualizations, like a geographical map, are handled with two adapters. In this case, one adapter handles communication transcoding, while the other handles the website necessary to expand OpenMCT with the new visualization.

3.5.1. Flight Computers

As explained in Section 2.3.1, communication interfaces to flight computers, both wired and radio, end up being TCP sockets. As such, there are few differences between telemetry and wired integration of flight computers from a software perspective. The adapter to integrate TCP connections will be called "tcp" adapter.
Telemetry and wired communication can carry the same information in part or completely. This raises the question of how to transition from wired to telemetry data as soon as wired connection is lost at lift off, for example. OpenMCT offers one option to handle this: Corre-

sponding telemetry and wired data points can be joined in one overlay plot visualization. As soon as wired data is unavailable, only telemetry data remains in the plot.

Functionality

The TCP adapter connects to an IP address and port specified in the equipment configuration. At that address it expects a TCP socket supplied by a server. It can therefore be considered a TCP client. The adapter maps the TCP socket to a WebSocket. As soon as a data point is accessed in OpenMCT, it connects to its corresponding WebSocket to receive the data.

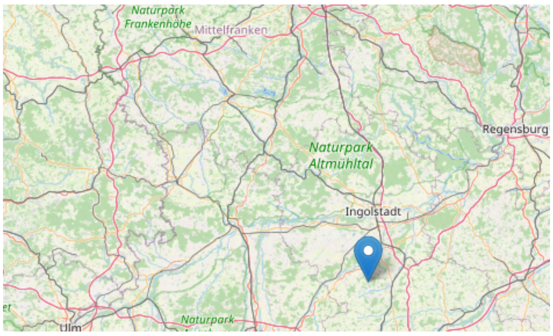
Apart from its core functionality, there are the following requirements the TCP adapter was developed with:

- Recover from physical loss of connection to TCP socket.
- Recover from physical loss of connection between TCP server and flight computer.
- Detect and handle untimely reactions (Timeout).
- Support multiple WebSocket clients.

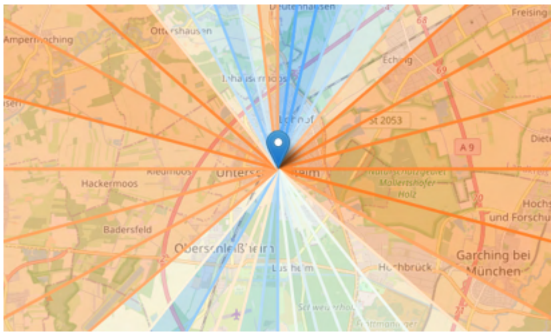
All of these requirements where met during development. The libraries used were the standard “Node.js” libraries Net and ws. The code for the TCP adapter can be found in Appendix C.3; It was tested together with testing of the complete integration approach for flight computers, results are documented in Section 4.1.2.

Positional Data

There are some data types supplied by flight computers that cannot be visualized by using the options included in OpenMCT. GPS coordinates containing positional information are one example of this. Here, positional data is regarded to as any combination of values that hints to where a vehicle might be located. Notably, this can include height information.
One way to visualize positional data is to use two-dimensional (2D) geographical maps. Using OpenMCT’s capability to embed websites, it can be extended to show a map with a marker indicating flight computer position. An additional web stack is necessary to serve such a website. The adapter containing it will be called "locat2d" going forward.
One possible implementation of the locat2d adapter using the open-source projects Klokantech Tileserver, Leaflet and Openstreet Maps can be seen in Fig. 20. It is currently in the prototype stage and was developed at WARR, not as part of this thesis. It is not only able to visualize vehicle position directly using a marker, but by overlaying received signal strength and direction onto the map (see Fig. 20b).



a: A geographical map visualization showing positional data.



b: A waterfall diagram visualizing signal strength and origin.

Fig. 20: Two visualization prototypes for positional data developed at WARR.

This approach to positional visualization can also be used for backup locating mechanisms like beacons or additional coordinate transmitters. There is the possibility to create a mobile computer setup with corresponding receivers to run the "locat2d" container for recovery operations.

Next to 2D visualization of positional data, three-dimensional (3D) visualization was found to be achievable in a similar way. A benefit of 3D visualizations is their ability to show vehicle height by providing terrain for context. One public library that can be used for this is three-geo [82]. To test whether 3D visualization of positional data is feasible, one of the three-geo's demonstration websites[7] was embedded in OpenMCT. The result is shown in Figure 21.

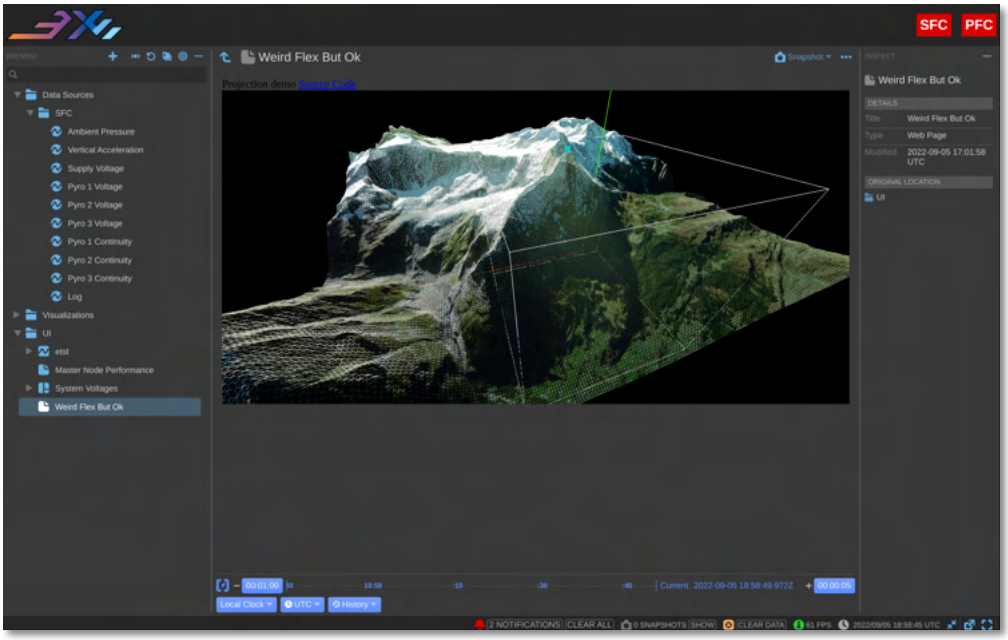


Fig. 21: Website visualizing 3D terrain around a center coordinate embedded in OpenMCT.

[7] The website is publicly available under `https://w3reality.github.io/three-geo/examples/projection/index.html` (last accessed 29.09.2022).

Orientation Data

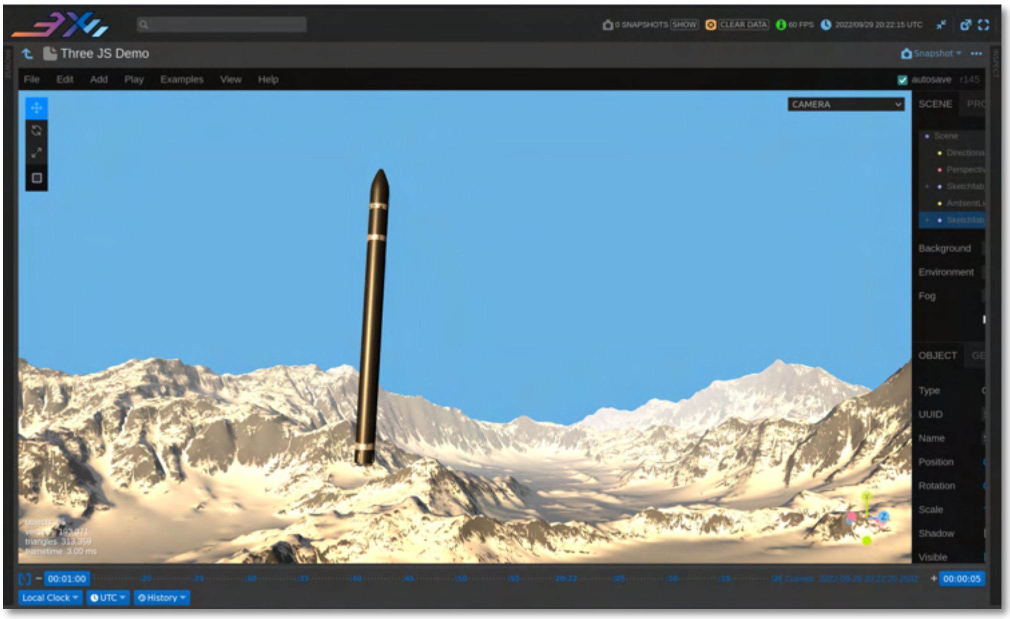


Fig. 22: A static 3D scene created with the Three.js editor embedded in OpenMCT.

Like positional data, OpenMCT is not able to visualize orientation data of a vehicle out of the box. In order to allow adequate monitoring of roll, pitch and yaw of a vehicle, a 3D visualization is necessary. The solution proposed here is fundamentally identical to the one described in the previous paragraph: An additional website visualizing orientation data, served by a dedicated adapter. Going forward, this adapter will be called "orien3d". One library that can be used for this, is "Three.js". It supports visualization of 3D models in the browser and most notably, it supports rotating them in real time using quaternions[8] [83]. In this way, the 3D model of a rocket can be visualized in the orientation of its real counterpart.
The orien3d adapter will not be developed here, instead it will be shown that the proposed approach is feasible. Similar to the proof of concept seen above, a demonstration website provided with the "Three.js" library was embedded in OpenMCT. In this case, the website includes a capable 3D model editor, which was used to import a model of a rocket and terrain. The result is depicted in Fig. 22 and a final SysML summary for flight computer integration is located in Fig. 23.

[8] A demonstration for this is accessible in the following GitHub repository: `https://github.com/ZaneL/quaternion_sensor_3d_nodejs`

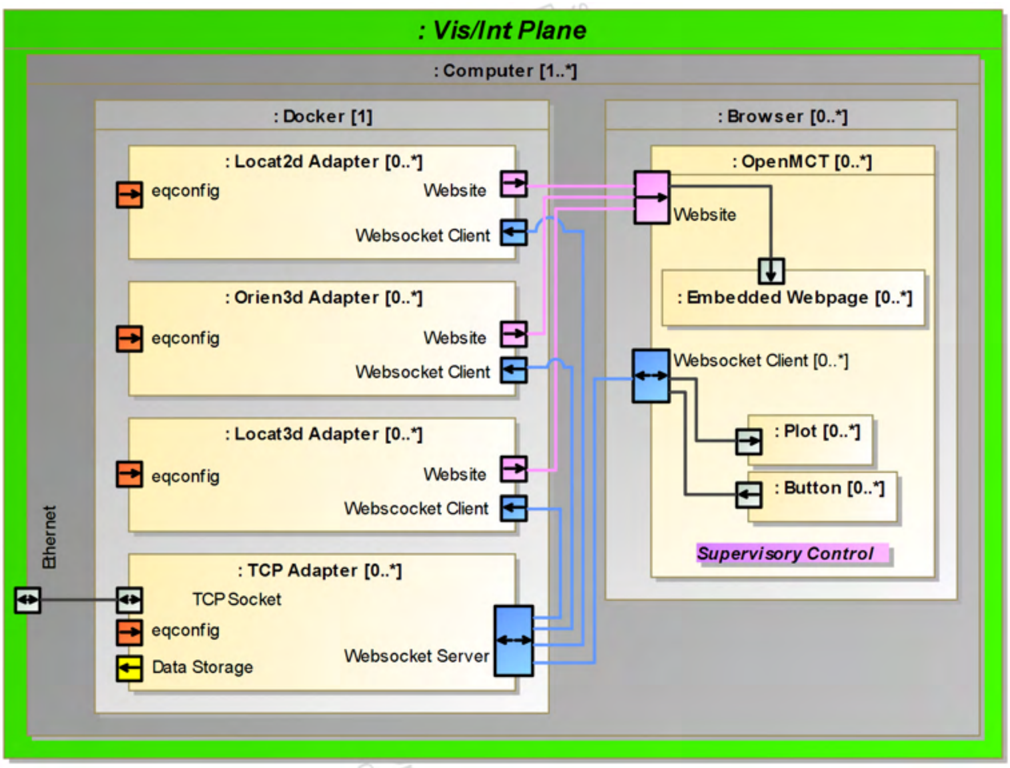


Fig. 23: SysML internal block diagram summarizing software integration of control and measurement capabilities of flight computers.

3.5.2. Data Acquisition Systems

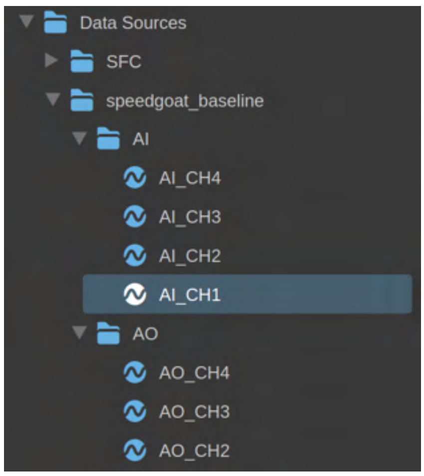


Fig. 24: An OPC UA DAQ with four analog inputs and three outputs mapped into OpenMCT.

In the System Design Chapter, OPC UA was found to be a suitable DAQ communication interface with added benefits. To integrate OPC UA DAQs into EX-UI, an adapter is needed that maps OPC UA communication onto a WebSocket. It is essential to note that the DAQ is considered to be an OPC UA server in this case, because it primarily supplies data and does not request it itself in any way.

OPC UA was considered over Modbus because of its additional features, particularly semantic information modeling. Using it enables querying DAQs about their measurement and control capabilities. If a DAQ's OPC UA server is programmed correctly, EX-UI can autonomously determine, which measurement channels the DAQ offers, which control capabilities it has as well as channel names, measurement types and more. This functionality was implemented in the "opcua" adapter and tested. The standard “Node.js” library ws was used for the WebSocket side of the adapter, node-opcua was used for OPC UA [42].

To further detail how the adapter works: It first connects to its OPC UA endpoint (e.g., `opc.tcp://192.168.1.7:4840`) specified in the equipment configuration. Subsequently, it queries the OPC UA device for all variables in its object tree. These variables could be measurements (readable) or control variables (writable and optionally readable). The adapter subsequently sets up subscriptions to all measurements. Then, the OPC UA object tree is translated into OpenMCT's domain object tree and sent over a dedicated WebSocket to all EX-UI clients. These can then restructure the object tree on OpenMCT's website to correspond to the one on the OPC UA devices. An exemplary result of this can be seen in Fig. 24. With this approach, the user is not required to list any data points of the DAQ in the equipment configuration. Further details on this sequence of events can be found in Testing Section 4.3.

Control over writable variables on OPC UA DAQs from within OpenMCT was not investigated. A starting point could be adding UI elements to OpenMCT that send out target values for variables when clicked. The OPC UA adapter can then forward the variable change request to the OPC UA device. Fig. 25 includes buttons in OpenMCT for this reason and finalizes this section with a summary.

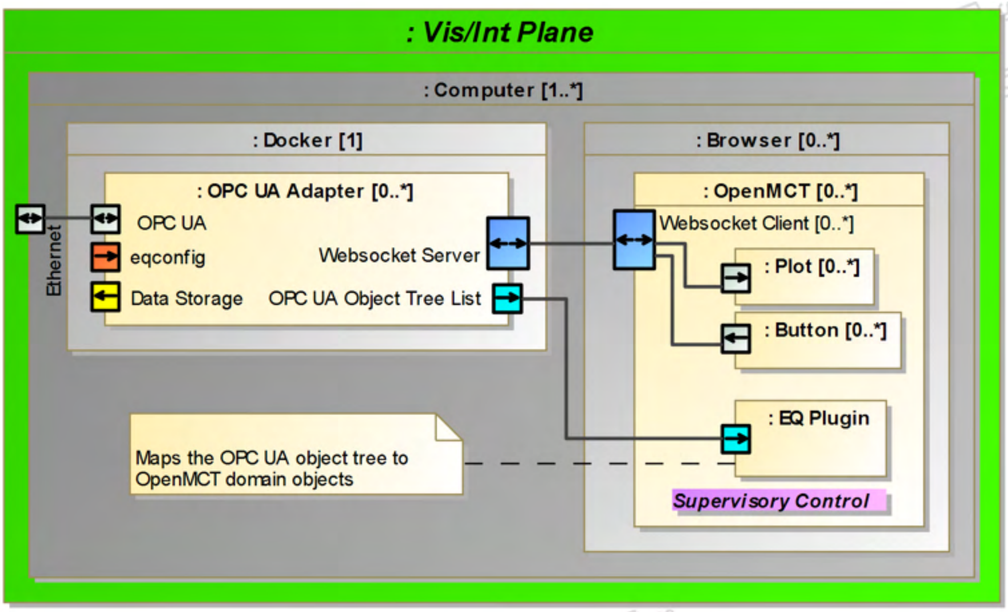


Fig. 25: SysML internal block diagram visualizing how measurement and control capabilities of DAQs can be integrated into EX-UI.

3.5.3. Live Video

In the system design chapter, integration of live video was described to be handled by conversion to RTSP streams both for wireless and IP video. RTSP is not supported in browsers, neither is h.265 encoding (with the exception of Safari). Moreover, OpenMCT does not include visualizations for video. Therefore the only way to integrate video into OpenMCT is to make use of its ability to embed websites again. The video adapter thus has to accept RTSP video streams on one end and display them on the other using a website.
Some IP cameras include microphones and stream recorded audio as well. Because live audio could potentially benefit remote operation of fueling systems and more, it will be taken into consideration here as well for completeness. A summary is depicted in Fig. 26.

There are only a handful of options for streaming live video into a browser. The most common approaches and corresponding libraries found are listed in Table V, untypical audio codecs for IP cameras were left unnoted [84]:

Table V: Approaches to getting live video into a web browser and libraries that can be used for them.

Approach	Description	Libraries
WebRTC	Dedicated Web API for real-time communication; Supports PCM mulaw or alaw audio	RTSPtoWeb [85], RTSP-toWebRTC [86], Janus Gateway [87], Kurento Mediaserver [88]
Image Stream	Uncompressed; No audio support	RTSPtoImage [89], Zone-minder [90], Shinobi [91]
Media Source Extensions (MSE)	Bytestream into html video and audio elements; AAC audio support	RTSPtoWeb
HTTP-Livestreaming (HLS)	Partial video file transfer; AAC audio support	RTSPtoWeb, Shinobi
Dynamic Adaptive Streaming over HTTP (DASH)	Bit rate adaptive HLS; AAC audio support	Not investigated
WebSockets and jsmpeg	Transfer of MPEG1 video segments, decoding in the browser; No applicable audio support	Steaming-IP-Camera-Nodejs (sic!) [92]

Some libraries listed are ruled out from the start: HLS is only supported in Safari browsers, for instance [84]. In addition, Shinobi and Zoneminder - while well documented and capable - are not well programmatically configurable. Zoneminder was found to require undocumented database operations for automated configuration and Shinobi only manually allows adding

video streams through its web interface. Janus was found to be unsuitable too, due to an error which prevented it from running at all. The error appears to be stemming from missing or unexpected metadata in the IP cameras' RTSP streams. Further research into this was not conducted, as other options worked without any errors.
To decide whether some of the remaining approaches offer advantages compared to others, the latency was measured and compared during testing. Results and methodology are documented in Section 4.2.

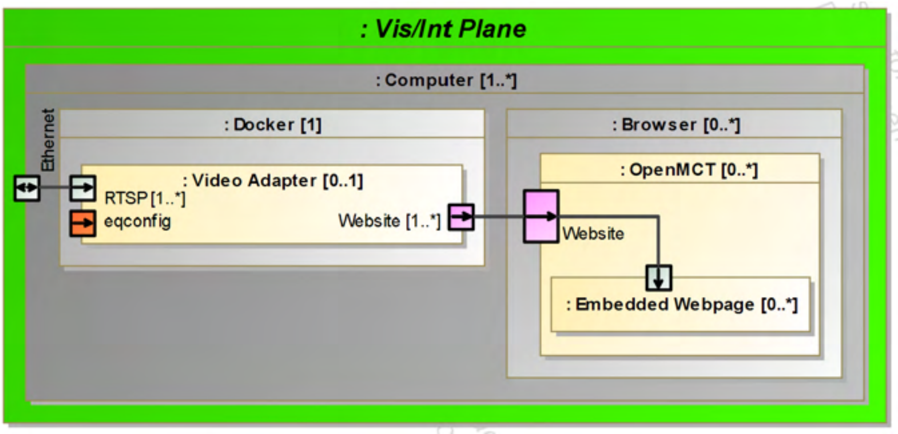


Fig. 26: SysML internal block diagram visualizing how video streams can be integrated into EX-UI.

3.5.4. Serial Devices over USB

Serial devices can be integrated into OpenMCT by designing an adapter that is able to interface with serial ports on one side and offer a WebSocket on the other. As adapters are programmed in "Node.js", the library node-serial was used for this task. WebSockets can be handled by the standard ws library again. Data arriving over the WebSocket can be visualized as usual in OpenMCT.
As noted in the System Design Chapter, this integration directly uses USB. USB ports on computers are a hardware resource and typically require root rights to be accessed. The adapter was not updated for Docker for this reason and is omitted in the appendix. Nevertheless, documentation of a successful test was kept in Section 4.1.1 for illustration purposes.

3.5.5. General Purpose Computers

The ability to monitor computers, the same ones running EX-UI for example, was found to be achievable as well. Instead of programming an adapter to access hardware information and send it over WebSockets, integration can be achieved by using open-source Docker containers developed for this application. One such containers is Netdata for example [93]. Its website can be embedded in OpenMCT and shows detailed computer performance statistics. This was tested and documented in Section 4.4.

3.6. Repository Structure

Setting up a Git repository was found to be a facilitating foundation for active development of EX-UI. To create the repository, a suited strategy for incorporating other open-source repositories into the project is necessary. The following strategies were investigated:

- Git submodules referencing original libraries
- Git submodules referencing forks of libraries
- Subtree merging strategy [94]
- Cloning or copying libraries into EX-UI's repository

Comparing and testing the different approaches showed that the subtree merging strategy is best in line with EX-UI's principles. It combines the version control ability of second approach with the centralization of the last approach. In this way, all files for EX-UI are collected in one repository and all files are downloaded when the repository is cloned. This represents a lower barrier of entry for new developers and users. Moreover, the subtree merging strategy allows updating incorporated repositories to newer versions [94]. This is essential to build upon third party development work and improvements.

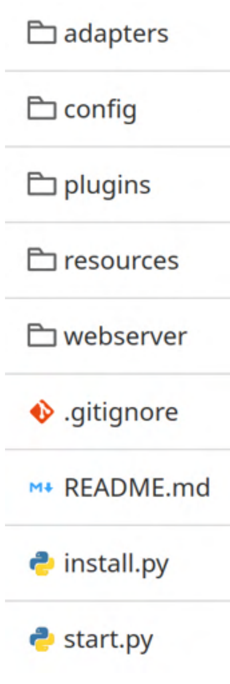


Fig. 27: EX-UI's repository structure.

After finding a suited strategy, the repository was structured as shown in Fig. 27 and third party libraries were integrated[9]. In particular, OpenMCT version 2.0.5 was merged into the "webserver" folder. The "adapters" folder contains one directory for each adapter and finally, the configuration folder was created to offer users an accessible place to store configuration files.

[9] A public, read-only instance of the repository is provided under `https://gitfront.io/r/antoniosteiger/RqVNNoGSL8wM/exui/`.

4. Testing

This chapter deals with testing hardware and software integration of selected examples for target plane components. All tests have been performed on a laptop computer; Its specifications are summarized in Table VI.

Table VI: Specifications of personal computer used for testing.

Manufacturer:	Microsoft
Model:	Surface Book 1
CPU:	Intel i7 6600U
RAM:	16 GB LPDDR3 at 1867 MT/s
OS:	Ubuntu 22.04 LTS

Tests involving Ethernet were performed on a dedicated local network with one router and one network switch. The laptop was connected to the network using a physical Ethernet to USB adapter. Detailed technical specifications of router, switch and adapter can be found in the appendix in Table XIII, XII and XIV respectively. The wireless functionality of the router was not used.

4.1. Flight Computer

To test integration with flight computers, a commercial model rocketry flight computer with a serial port was used. At WARR, this flight computer is scheduled to be used as a secondary flight computer in multiple upcoming sounding rocket launches. Its specifications are summarized in Table VII. In all of the following tests it was powered with a laboratory power supply (specifications in Table XV).

Table VII: Specifications of flight computer used for testing.

Manufacturer:	Rocketronics GmbH
Model:	AltiMax G4 Hardware Rev. G4STD, Firmware ver. 1
Operating Voltage:	7 - 12 V DC
Communication Interface:	UART; Serial Port Details: 38400 baud, 8 data bits, 2 stop bits, no flow control, no parity.
Functions:	Timers, Apogee Detection, Dual Pyrotechnic Events; Measurements: Ambient pressure and temperature; Vertical Acceleration, System Voltages

4.1.1. Wired Testing as a Serial Device Using USB

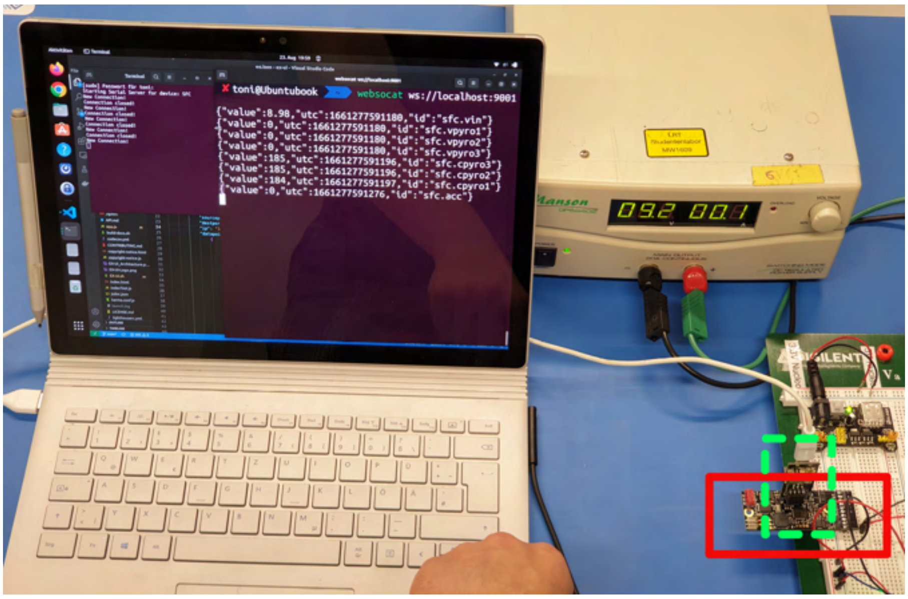


Fig. 28: Test setup used for directly integrating the AltiMax over a serial link. The large terminal on the laptop acts as a WebSocket client, simulating EX-UI. The flight computer is highlighted in solid red and the serial to USB adapter in dashed green.

This test aimed to directly receive data from the AltiMax over USB and display it in OpenMCT. The serial adapter developed for this purpose remains to be fully integrated into OpenMCT and EX-UI's Docker environment. It was used in conjunction with this test to determine whether WebSocket connections are a viable path forward for further device integrations.
In a development and pathfinding scenario, it is desirable to directly connect the flight computer to a personal computer. In this case, a common serial to USB converter is needed. In this test, a converter from Rocketronics was used. It directly attaches to the flight computer and is based on an FTDI FT232RL chip. The complete test setup is depicted in Fig. 28.

Results proved that a fast translation of data onto the WebSocket connections used by OpenMCT is possible. Along with that, real-time visualization using OpenMCT's plots was confirmed to be performant and appropriate for the application at hand.

4.1.2. Wired Testing Using a TCP Connection

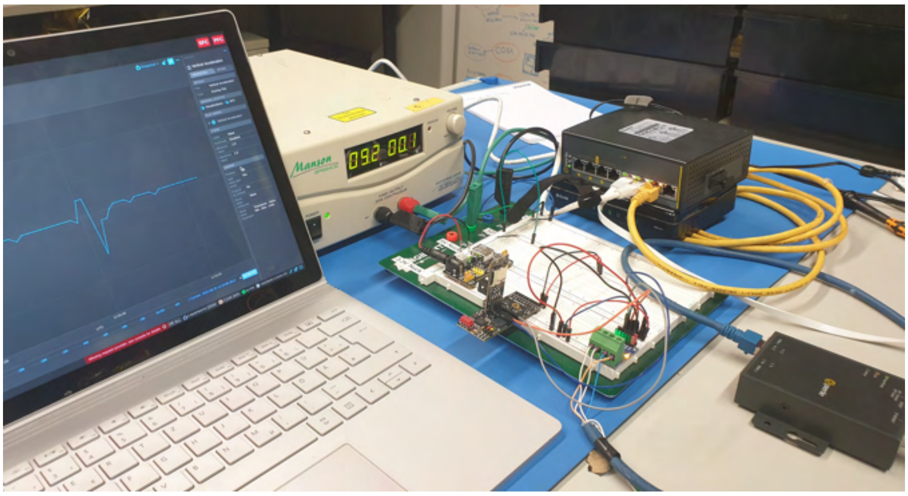


Fig. 29: Test setup used for integrating the AltiMax flight computer into the SCADA system via TCP. EX-UI is running on the laptop and displaying vertical acceleration data polled from the AltiMax. The data arrives at the laptop over Ethernet and TCP, no USB connection to the AltiMax is present. The small black device in the lower right corner is the serial device server.

In this test, the AltiMax was connected using the TCP signal chain summarized in Fig. 8. The test setup is depicted in Fig. 29 showing AltiMax, serial device server and network equipment. This test was performed to achieve three goals:

1. Determine whether the signal chain works.
2. Evaluate Docker's usability by running the TCP adapter in a container.
3. Observe whether the test setup can recover from physical connection loss at any point in the signal chain.

Results showed that the signal chain, Docker and recovery from connection losses works. Additionally, it was further established that Docker is a viable deployment option for EX-UI.

4.2. Live Video

Fig. 30: A picture of the test setup used to investigate live video and audio integration.

Wired IP camera integration was first tested with a single Reolink RLC-510a camera. Its software did not support other audio codecs than AAC. Tests were repeated in an extended form with two Dahua cameras where audio codecs could be configured accordingly. Their specifications are listed in Table VIII.

Table VIII: Specifications of the IP cameras used for testing.

Manufacturer:	Dahua
Model:	IPC-HFW4239TP-ASE-0360B
Power Input:	Power over Ethernet (PoE) or 12 V DC
Supported Ethernet Speeds:	10/100Base-TX
Video/Audio Protocol and Codecs:	RTSP h.264; AAC, G.711a and G.711mu @ 8kHz, G.726
Resolution:	1080P (NTSC @ 1024 kb/s)
Framerate:	30 FPS
Functions:	Local video recording to micro SD card, alarm output, audio input, audio output, web interface

The goal of this test was to compare different software integration approaches (see Section 3.5.3) for live RTSP video to arrive at a conclusion as to which one is optimal. Kurento and RTSPtoWSMP4f were not investigated in this test because they exhibited multi-second latencies. Being an essential metric, latency was measured by filming a smartphone timer with the IP cameras. With a different camera, pictures were then taken showing both the IP video streams in OpenMCT as well as the smartphone timer. Fig. 30 shows the test setup used. By looking at the picture and subtracting the smartphone's timer value from the one shown in OpenMCT, the latency can be determined. This was done for each adapter approach by taking five pictures roughly every five seconds. Results are shown in Fig. 31.

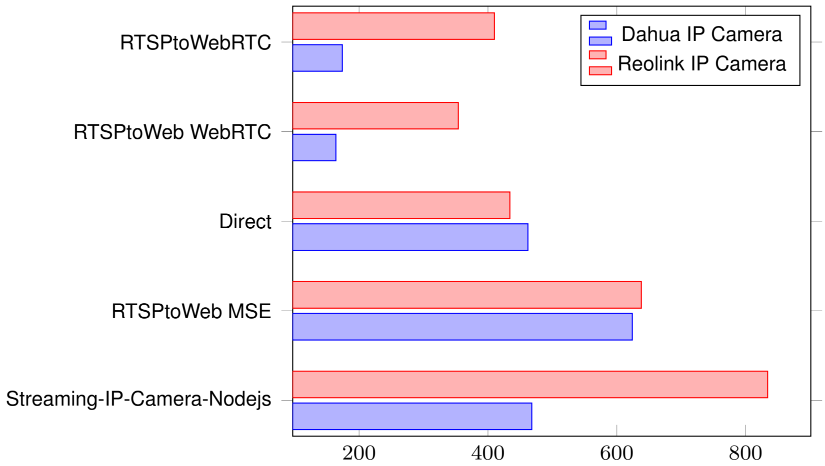


Fig. 31: Latency in ms resulting from different libraries that make RTSP streams from IP cameras available in the browser. "Direct" refers to the website implementations supplied by the manufacturers.

In some areas, Fig. 31 reveals large differences in latency between cameras from different manufacturers. Both cameras were configured to stream with a constant bit rate around 1 Mb/s. The large differences could stem from the use of differently performing hardware for the streaming pipeline in the cameras. The raw image data from the camera's sensors has to be packaged and compressed into an h.264 RTSP stream. Added latency could come from suboptimal hardware for the compression step, however this hypothesis needs to be confirmed in further testing.

Most notably, latency results showed that integration approaches using WebRTC offer the lowest latencies. Because both WebRTC libraries used are programmatically configurable with JSON files, both of them are appropriate options for EX-UI. Audio was confirmed to work with both of them as well.

4.3. DAQ

OPC UA based integration of DAQs was tested with one that could be programmed using MATLAB Simulink and offered OPC UA server blocks in its Simulink library. Specifications are listed in Table IX. The Simulink model running on the DAQ was laid out in a way where all analog channels were linked to a variable-type node in its OPC UA object tree. Analog input nodes were aggregated under an "AI" object-type node, analog outputs under "AO". One output channel was not routed into an OPC UA node, but instead driven by a slow sine wave generated within the model. This output was physically connected to an input, to create reference data.

Table IX: Specifications of the DAQ used for testing.

Manufacturer:	Speedgoat
Model:	Baseline Real-Time Target Machine
Operating System:	Free DOS Version 1.0
Supported Ethernet Speeds:	10/100/1000Base-T
Capabilities:	4x analog input, 8x analog output, I2C, SPI, UART, (Modbus) TCP, UDP
Programming Tool:	MATLAB Simulink up to R2022a

With this Simulink configuration running on the DAQ, a test was performed to achieve the following objectives:

- Prove successful connection of EX-UI's OPC UA adapter with the DAQ
- Evaluate benefits of OPC UA's object tree browsing features and whether the adapter is able to use them correctly
- Develop code that correctly maps the OPC UA object tree in the DAQ to a domain object tree in OpenMCT

The test setup used for this is depicted in Fig. 32.

Browsing the server for the structure of its object tree and mapping it to domain objects in OpenMCT worked successfully. However, some occasional timing and execution order issues were caused on the OpenMCT website by the OPC UA adapter. A possible reason for this could be OpenMCT's plugin installation process, which is asynchronous. This can lead to race conditions, wherein OpenMCT itself expects the domain object tree to be ready when it is started, but it is not due to the time the EQ plugin needs to fetch the OPC UA tree from the adapter. The concept of automatically listing available measurements in OpenMCT by using functionality provided by OPC UA was proven to work with this test. It is well suited for the SCADA system and was found to offer potential in increasing ease of use.

The sine wave output was connected to an analog input channel and could successfully be visualized in an overlay plot. The sampling rate appears to be limited to 10 S/s by the node-opcua library, but this has to be investigated in detail in the future. Sampling rate on the server side was set to 1000 S/S across all Simulink blocks and it has been proven that OpenMCT supports higher sampling rates (see Section 4.5). The resulting visualization is shown in Fig. 32.

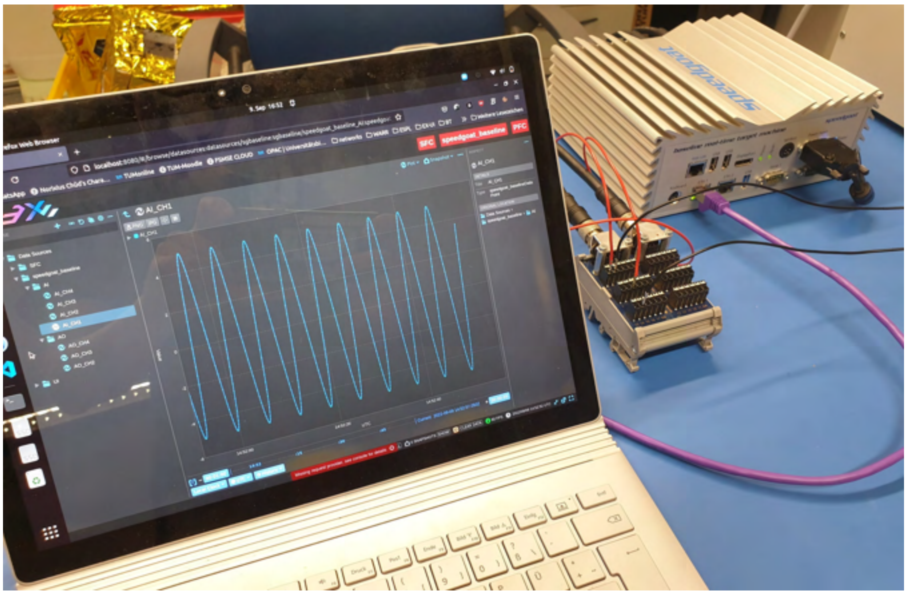


Fig. 32: Test setup used to investigate OPC UA based DAQ integration. OpenMCT is plotting the voltage of an analog input on the DAQ. The sine wave shape is coming from a programmed output of the DAQ.

During development of the OPC UA adapter, ready-made graphical OPC UA clients were used to verify that the adapter is working correctly. While the ability to change output values on the OPC UA server from within OpenMCT was not developed as part of this thesis, changing them from within the reference clients was achieved.

4.4. General Purpose Computer

Displaying computer performance statistics in OpenMCT was tested as well. For this, a Netdata container was run as usual and its web page was embedded in OpenMCT. The result can be seen in Fig. 33.

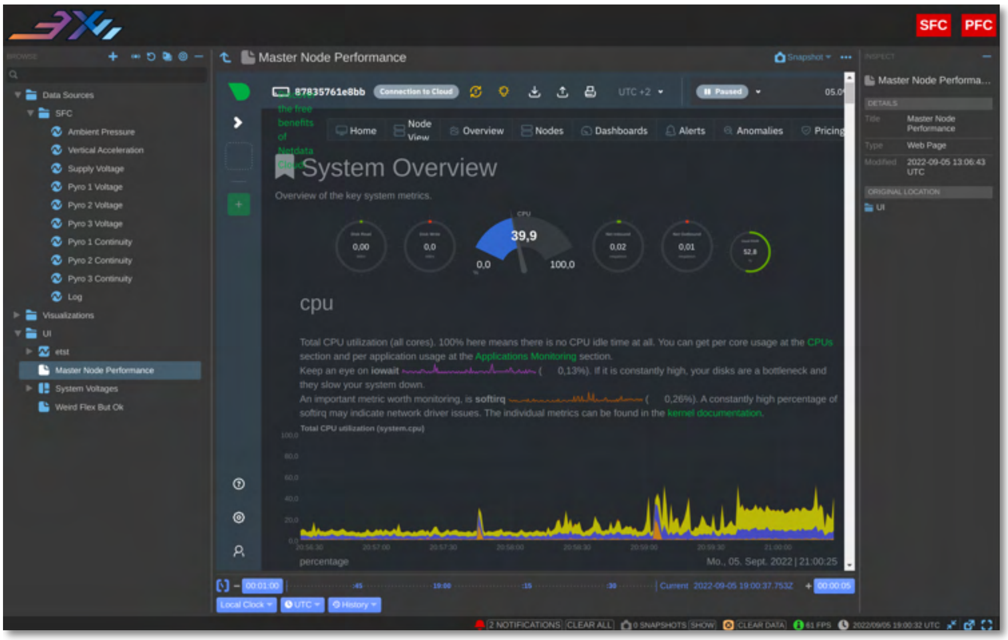


Fig. 33: Netdata shows a wide range of computer performance statistics including CPU, network and disk loads. It is embedded in OpenMCT here using its ability to embed websites.

4.5. Benchmarking

To evaluate performance of OpenMCT across a range of sampling rates on multiple plots, the benchmark adapter was created (see Appendix C.5). It is not an adapter in the typical sense, because it supplies dummy data itself. The dummy data is a sine wave. No extensive performance testing was done as part of this thesis, but the following observations were made:

- The adapter as it is currently implemented can be used up until 20 samples per second before plots in OpenMCT start to behave abnormally, showing multiple waves at the same time.
- It is possible to use four plots at the same time with 20 samples per second. The browser reported 50 frames per second using this setup. A screenshot of OpenMCT during this test can be seen in Fig. 34.

Investigations into how the visualization rates can be increased with a lower performance impact are left to future work in this area.

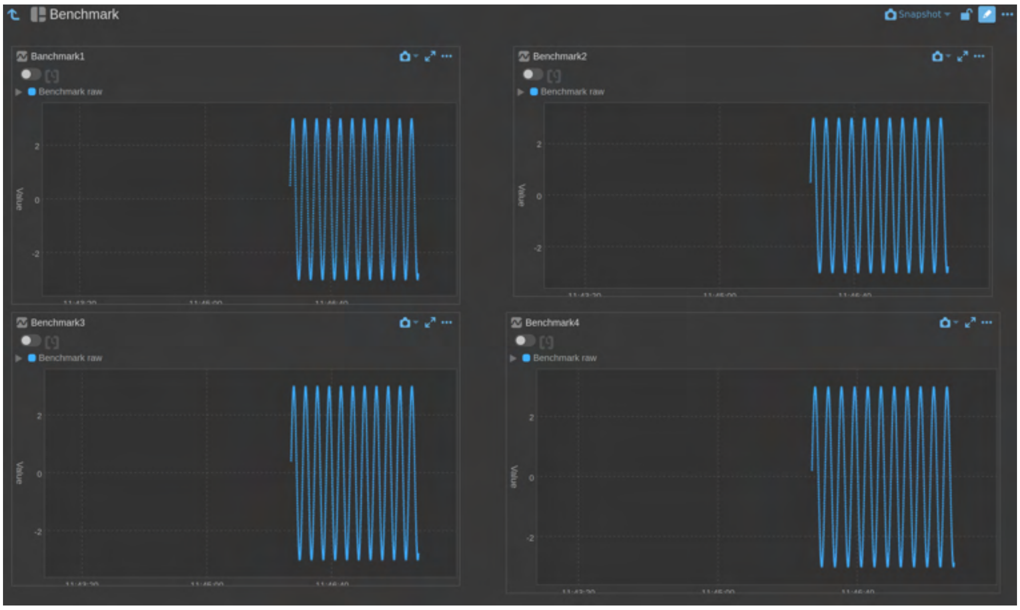


Fig. 34: A UI configuration of OpenMCT visualizing four benchmark data sources in plots. Each plot visualizes 20 measurements per second.

5. Summary and Outlook

All essential concepts that will be reemphasized in the following are reflected in Fig. 35, which is a SysML diagram created to serve as a visual summary. It is framed by the target plane, the I/O plane and the Vis/Int plane; The three SCADA domains introduced at the beginning of this thesis using the automation pyramid.

The first objective of this thesis was to **answer the question of how to integrate typical rocketry components of the target plane into a web-based SCADA system**. It was achieved by first laying out an Ethernet based IP network with a tree topology; A network particularly suited for the *web-based* approach to SCADA. Target plane components were then integrated by building up a signal chain that ultimately ends in this network. In this process, an aggregation strategy was used when assigning analog devices in the target plane to DAQs. Having a reduced number of distinct DAQ interfaces, integration was further streamlined by using OPC UA as a data transmission standard for them. With its semantic information modeling, extensive descriptions of DAQ capabilities, like in "channel configuration" files, were rendered obsolete.
The concept of local, primary data storage was selected as a compromise between required implementation efforts and network load. Additionally, secondary data storage in the Vis/Int plane was identified to be suitable as a fallback solution.
Only an embedded and distributed approach to control, wherein control commands are issued by multiple devices in the I/O plane, was found to fit stakeholder requirements. Coordination of said control sources was constrained to stem from the Vis/Int plane, because it benefited stakeholder operations.
Finally, all design decisions above were found to be compatible with the typical safety measures employed by the stakeholders.

The second goal of this thesis was to **Develop a web-based SCADA software prototype**. NASA's OpenMCT project served as a starting point, enabling basic visualization of data in plots and tables as well as their arrangement in user-creatable layouts. EX-UI, a conceptual software project initialized by WARR, was selected to contain development of the prototype. Its principles were taken into consideration throughout the design process.
One of EX-UI's principles is the "one-click" philosophy. To adhere to it, an installation script and a start script, automating as much of the initial user experience as possible, were written in Python. In addition, the equipment configuration file and the UI configuration file were introduced as the two main tools for tailoring the SCADA software to a specific application.
EX-UI's principles call for high reliability and dependability. To accomplish this, the open-source deployment tool Docker was researched. It enables redundant, distributed execution of EX-UI and streamlines its deployment process. This required a well-defined modularization, which was handled through the introduction of the adapter concept. Adapters use WebSockets to pass on data if it can be visualized by OpenMCT *or* websites if not. Some

of the website-based visualizations tested included 3D visualizations of rocket and terrain, geographical maps and live video with latencies under 180 ms. They can be integrated into the UI by using OpenMCT's ability to embed websites.

The last goal of this thesis was to **test selected signal chains from the system design chapter together with the software developed in the software design chapter**. In doing so, all proposed integration approaches were found to be feasible. However, dynamic expansion of OpenMCT's domain object tree with OPC UA objects exhibited execution order issues. Although issuing OPC UA control commands for DAQs from within OpenMCT was not tested, the concept was found to work using a reference OPC UA client. Finally, tests additionally revealed limitations to data visualization rates in OpenMCT and performance degradations associated with increasing them.

Performance degradation not only from increased visualization rates, but large numbers of simultaneous visualizations has to be further investigated in the future. Especially research into performance improvements could be beneficial for the stakeholders. It remains to be seen whether work on this will be picked up by WARR and the SPM as part of persistent and active development efforts on EX-UI. First starting points for this could also be finishing the implementation of OPC UA, serial and video adapters as Docker containers handling single devices.

A large and essential topic that still requires significant investigation and development is manual control and coordination from within EX-UI. Combining this with tests of Ethernet extensions has the potential to make EX-UI a *fully* suited SCADA software for rocket launches and tests.

First tests of EX-UI in operational scenarios are scheduled for November 2022, when WARR will perform integrated campaigns of an experimental hybrid sounding rocket. If active software development is carried out during these tests, EX-UI might solve one of the problems that originally motivated its creation.

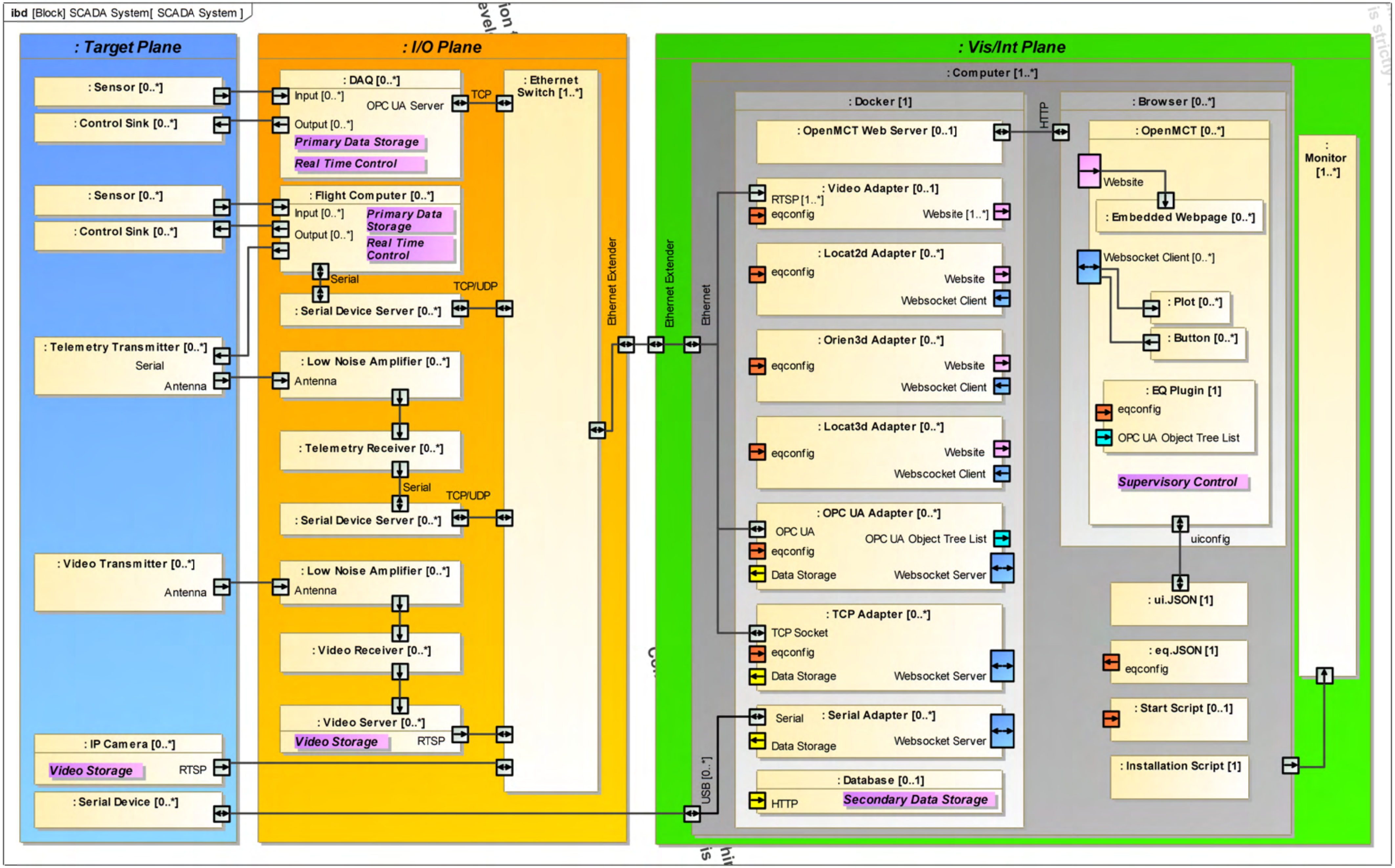


Fig. 35: SysML internal block diagram visualizing the designed SCADA system from a data and control perspective. Most adapter connections were left color-coded, but unconnected for clarity.

Bibliography

5.1. Main Sources

[9] M. Åkerman, **Implementing Shop Floor IT for Industry 4.0**, Ph.D. dissertation, Jun. 2018.

[10] **Internetworking Technologies Handbook**, Cisco Systems, Inc. (1999), [Online]. Available: https://learning.oreilly.com/library/view/internetworking-technologies-handbook/1587051192/.

[12] R. J. Kosinski, **A Literature Review on Reaction Time**, 2013. [Online]. Available: https://www.sciencebuddies.org/Files/16478/4/clemson.rt.pdf (visited on 10/02/2022).

[19] M. Quigley, B. Gerkey, and W. D. Smart, "'Introduction,'" in **Programming Robots with ROS**. O'Reilly Media, 2015. [Online]. Available: https://learning.oreilly.com/library/view/programming-robots-with/9781449325480/ (visited on 10/02/2022).

[22] **OMG Data Distribution Service (DDS)**, version 1.4, Object Management Group, Apr. 2015. [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF (visited on 06/19/2022).

[24] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, Eds., **MQTT Specification**, version 5.0, Organization for the Advancement of Structured Information Standards (OASIS), Mar. 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html (visited on 06/19/2022).

[25] A. Neumann, L. Wisniewski, O. Givehchi, and J. Jasperneite, **Utilizing OPC UA as Comprehensive Communication Technology for Cyber Physical Production Systems**, presented at the 9th International Workshop on Service-Oriented Cyber-Physical Systems in Converging Networked Environments (SOCNE) in conjunction with 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) (Luxembourg, Sep. 8, 2015).

[26] C. Schranz, M. Schmoigl-Tonis, and F. Strohmeier, **From Modelling to Designing: A Streaming Network for Multi-Tenant Digital Twin Platforms Based on the Reference Architecture for Industry 4.0. Designed for Cross-Domain Applications**, presented at the Fourteenth International Conference on Systems and Networks Communications(ICSNC) (Valencia, Nov. 2019). DOI: 10.13140/RG.2.2.29690.52166.

[28] **Modbus Application Protocol Specification**, version 1.1b3, Modbus Organization, Apr. 2012. [Online]. Available: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf (visited on 06/19/2022).

[29] **LXI Networking Basics**, LXI Consortium. (Aug. 2013), [Online]. Available: https://www.lxistandard.org/members/Adopted%20Specifications/Latest%20Version%20of%20Standards_/LXI%20Standard%201.5%20Specifications/LXI%20Device%20Specification%20v1_5_01.pdf (visited on 06/19/2022).

[33] **LXI Device Specification 2016**, version 1.5.01, LXI Consortium, Mar. 2017. [Online]. Available: https://www.lxistandard.org/members/Adopted%20Specifications/Latest%20Version%20of%20Standards_/LXI%20Standard%201.5%20Specifications/LXI%20Device%20Specification%20v1_5_01.pdf (visited on 06/19/2022).

[34] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, **OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols**, in *2019 IEEE International Conference on Industrial Technology (ICIT)*, 2019, pp. 955–962. DOI: 10.1109/ICIT.2019.8755050.

[35] T. Hu and I. C. Bertolotti, **Overhead and ACK-induced jitter in Modbus TCP communication**, in *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, 2015, pp. 392–397. DOI: 10.1109/RTSI.2015.7325130.

[37] A. B. McCarthy and F. Peng, **Comparing GPIB, LAN/LXI, PCI/PXI Measurement Performance in Hybrid Systems**, in *2006 IEEE Autotestcon*, 2006, pp. 122–128. DOI: 10.1109/AUTEST.2006.283608.

[40] **OPC UA Online Reference**, Open Platform Communications Foundation. (Oct. 2022), [Online]. Available: https://reference.opcfoundation.org/ (visited on 09/11/2022).

[52] S. Banerji, **Study and Development of a Data Acquisition and Control (DAQ) System using TCP/Modbus Protocol**, Tech. Rep., Jul. 2013.

[54] A. Bahador *et al.*, **Condition Monitoring for Predictive Maintenance of Machines and Processes in ARTC Model Factory**, in *Implementing Industry 4.0* (Intelligent Systems Reference Library), Intelligent Systems Reference Library. Springer, 2021, ISBN: 978-3-030-67270-6.

[56] I. Grigorik. **High Performance Browser Networking**. (2013), [Online]. Available: https://learning.oreilly.com/library/view/high-performance-browser/9781449344757/ (visited on 10/02/2022).

[59] T. Hughes-Croucher and M. Wilson, "'Building Robust Node Applications,'" in **Node: Up and Running**. O'Reilly Media, 2012. [Online]. Available: https://learning.oreilly.com/library/view/node-up-and/9781449332235/ (visited on 10/02/2022).

[63] V. Woeltjen. **Open MCT Web Developer Guide**. version 1.0. (Jun. 2015), [Online]. Available: https://ntrs.nasa.gov/api/citations/20150021312/downloads/20150021312.pdf (visited on 06/09/2022).

[64] C. Hacskaylo and T. Truong. **Open MCT User's Guide**. version 1.0. (Mar. 2021), [Online]. Available: https://nasa.github.io/openmct/static/files/Open_MCT_Users_Guide.pdf (visited on 06/09/2022).

[65] OpenMCT, **API.md**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/nasa/openmct/blob/master/API.md (visited on 10/01/2022).

[66] NASA, **OpenMCT**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/nasa/openmct/ (visited on 10/01/2022).

[74] J. Cook, **Docker for Data Science : Building Scalable and Extensible Data Infrastructure Around the Jupyter Notebook Server**. Apress, 2017. DOI: 10.1007/978-1-4842-3012-1. (visited on 10/02/2022).

[75] **Compose Specification**, *Docker Documentation*, Sep. 2022. [Online]. Available: https://docs.docker.com/compose/compose-file (visited on 10/02/2022).

[76] **Swarm mode overview**, *Docker Documentation*, Sep. 2022. [Online]. Available: https://docs.docker.com/engine/swarm (visited on 10/02/2022).

[94] **About Git subtree merges**, *GitHub Docs*, Oct. 2022. [Online]. Available: https://docs.github.com/en/get-started/using-git/about-git-subtree-merges (visited on 10/01/2022).

5.2. Supplemental Sources

[1] M. Wall, **SpaceX launches 46 Starlink satellites, lands Falcon 9 rocket for 100th time**, *Space.com*, Feb. 21, 2022. [Online]. Available: https://www.space.com/spacex-starlink-satellite-launch-4-8-rocket-landing (visited on 10/01/2022).

[2] E. Berger, **Rocket Report: Used Electron engine gets retested; Canadian spaceport is a go**, *Ars Technica*, Feb. 9, 2022. [Online]. Available: https://arstechnica.com/science/2022/09/rocket-report-used-electron-engine-gets-re-tested-canadian-spaceport-is-a-go/ (visited on 10/01/2022).

[3] **A modular Ground Station Software | Telestion (a WüSpace e. V. project)**. (May 2022), [Online]. Available: https://telestion.wuespace.de (visited on 05/21/2022).

[4] D. Schiemann, **JavaScript Reaches the Final Frontier: Space**, *InfoQ*, Jun. 30, 2020. [Online]. Available: https://www.infoq.com/news/2020/06/javascript-spacex-dragon/ (visited on 10/01/2022).

[5] SpaceX, NASA, and ESA, **SpaceX_Crew_Dragon_training_pillars.jpg**, JPEG Image, 1920 × 1280 pixels, scaled, Jul. 2020. [Online]. Available: https://www.esa.int/var/esa/storage/images/esa_multimedia/images/2020/07/spacex_crew_dragon_training/22153084-1-eng-GB/SpaceX_Crew_Dragon_training_pillars.jpg (visited on 05/21/2022).

[6] **Rocket Lab Production Complex**, JPEG Image, 4500 × 3000 pixels, scaled, 2018. [Online]. Available: https://www.dropbox.com/sh/kumqcl235k3sghq/AAB7ecs006yywvpxtHhXv5YQa/Rocket%20Lab%20Production%20Complex?dl=0&preview=Rocket+Lab+Mission+Control.JPG&subfolder_nav_tracking=1 (visited on 05/21/2022).

[7] **STAR Dresden on Twitter**, JPEG Image, 854 × 633 pixels, scaled, 2021. [Online]. Available: https://twitter.com/STAR_Dresden/status/1409486610677305345/photo/1 (visited on 05/21/2022).

[8] **TU Wien Space Team on Twitter**, JPEG Image, 1080 × 864 pixels, scaled, May 2022. [Online]. Available: https://twitter.com/tuspaceteam/status/1522644328400801794/photo/1 (visited on 10/02/2022).

[11] **TC EXTENDER 2001 ETH-2S**, 106914_en_05, Phoenix Contact. [Online]. Available: https://www.phoenixcontact.com/de-de/produkte/modem-tc-extender-2001-eth-2s-2702409#downloads-link-target (visited on 10/01/2022).

[13] **Ethernet Cat.7, Cat.7A, Cat.7e**, 03.15.02.03, TKD Kabel. [Online]. Available: https://www.tkd-kabel.de/datenblaetter/03/03.15.02.03.pdf (visited on 10/01/2022).

[14] **PNOZ s1 Safety Relay**, 21393-EN-12, PILZ. [Online]. Available: https://www.pilz.com/download/open/PNOZ_s1_Operat_Man_21393-EN-12.pdf (visited on 10/01/2022).

[15] **FAQ**, Organization for the Advancement of Structured Information Standards. (May 2022), [Online]. Available: https://mqtt.org/faq (visited on 06/19/2022).

[16] **History**, OPC Foundation. (Aug. 2021), [Online]. Available: https://opcfoundation.org/about/opc-foundation/history (visited on 06/19/2022).

[17] **About the Data Distribution Service Specification Version 1.4**, Organization for the Advancement of Structured Information Standards. (Jun. 2022), [Online]. Available: https://www.omg.org/spec/DDS (visited on 06/19/2022).

[18] **Modbus FAQ**, Modbus Organization. (Jun. 2022), [Online]. Available: https://modbus.org/faq.php (visited on 06/19/2022).

[20] **History of the LXI Consortium**, LXI Consortium. (Jun. 2022), [Online]. Available: https://www.lxistandard.org/About/History.aspx (visited on 06/19/2022).

[21] **Unified Architecture**, OPC Foundation. (Sep. 2019), [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua (visited on 06/19/2022).

[23] **Benefits of LXI Instruments for Test & Measurement Applications**, LXI Consortium. (Jun. 2022), [Online]. Available: https://www.lxistandard.org/About/Benefits.aspx (visited on 06/19/2022).

[27] **What is DDS?** DDS Foundation. (Jun. 2022), [Online]. Available: https://www.dds-foundation.org/what-is-dds-3 (visited on 06/19/2022).

[30] **Transport Layer — Fast DDS 2.6.1 documentation**, eProsima. (Jun. 2022), [Online]. Available: https://fast-dds.docs.eprosima.com/en/latest/fastdds/transport/transport.html (visited on 06/19/2022).

[31] **Introduction to OpenDDS**, OpenDDS Foundation. (Jun. 2022), [Online]. Available: https://opendds.org/about/articles/Article-Intro.html (visited on 06/19/2022).

[32] **Topics**, *ROS Wiki*, Jun. 2022. [Online]. Available: http://wiki.ros.org/Topics (visited on 06/19/2022).

[36] **DAQ970A/DAQ973A Datasheet**, 5992-3168, Keysight Technologies, Jun. 2022. [Online]. Available: https://www.keysight.com/de/de/assets/7018-06259/technical-overviews/5992-3168.pdf (visited on 06/19/2022).

[38] **InfiniiVision 6000L Series Data Sheet**, 5989-5470, Keysight Technologies, Oct. 2019. [Online]. Available: https://www.keysight.com/us/en/assets/7018-01421/data-sheets/5989-5470.pdf (visited on 06/19/2022).

[39] **What can DDS do for You?** Object Modeling Group. (Jun. 2022), [Online]. Available: https://www.omg.org/hot-topics/documents/dds/CoreDX_DDS_Why_Use_DDS.pdf (visited on 06/19/2022).

[41] **MQTT.js**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/mqttjs/MQTT.js (visited on 06/19/2022).

[42] **node-opcua**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/node-opcua/node-opcua (visited on 06/19/2022).

[43] **node-opendds**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/oci-labs/node-opendds (visited on 06/19/2022).

[44] **node-modbus**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/Cloud-Automation/node-modbus (visited on 06/19/2022).

[45] **rosnodejs**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/RethinkRobotics-opensource/rosnodejs (visited on 06/19/2022).

[46] **liblxi**, *GitHub*, Jun. 2022. [Online]. Available: https://github.com/lxi-tools/liblxi (visited on 06/19/2022).

[47] **mqtt**, *npm*, Jun. 2022. [Online]. Available: https://www.npmjs.com/package/mqtt (visited on 06/19/2022).

[48] **node-opcua**, *npm*, Jun. 2022. [Online]. Available: https://www.npmjs.com/package/node-opcua (visited on 06/19/2022).

[49] **opendds**, *npm*, Jun. 2022. [Online]. Available: https://www.npmjs.com/package/opendds (visited on 06/19/2022).

[50] **jsmodbus**, *npm*, Jun. 2022. [Online]. Available: https://www.npmjs.com/package/jsmodbus (visited on 06/19/2022).

[51] **rosnodejs**, *npm*, Jun. 2022. [Online]. Available: https://www.npmjs.com/package/rosnodejs (visited on 06/19/2022).

[53] D. Greenfield, **Industry 4.0 and OPC UA**, *AutomationWorld*, [Online]. Available: https://www.automationworld.com/products/networks/blog/13311977/industry-40-and-opc-ua (visited on 10/01/2022).

[55] **Web video codec guide**, *Mozilla Developer Network (MDN) Web Docs*, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Video_codecs#hevc_h.265 (visited on 10/01/2022).

[57] **Usage statistics of JavaScript as client-side programming language on websites**, W3Techs. (), [Online]. Available: https://w3techs.com/technologies/details/cp-javascript (visited on 10/01/2022).

[58] **SerialPort**, *MDN Web Docs*, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/SerialPort (visited on 10/01/2022).

[60] **Event Loop**, *MDN Web Docs*, Oct. 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop (visited on 10/01/2022).

[61] **Open-MCT.Browse.Layout.Mars-Weather-1.jpg**, JPEG, 1280 × 860 pixels, scaled, NASA, Oct. 2021. [Online]. Available: https://nasa.github.io/openmct/static/res/images/Open-MCT.Browse.Layout.Mars-Weather-1.jpg (visited on 06/09/2022).

[62] **About Open MCT**. (Oct. 2021), [Online]. Available: https://nasa.github.io/openmct/about-open-mct (visited on 06/07/2022).

[67] **logo-openmct.svg**, SVG Image, Sep. 2022. [Online]. Available: https://github.com/nasa/openmct/blob/master/src/ui/layout/assets/images/logo-openmct.svg (visited on 09/11/2022).

[68] **logo.svg**, SVG Image, Node.js homepage, Sep. 2022. [Online]. Available: `https://nodejs.org/static/images/logo.svg` (visited on 09/11/2022).

[69] **express-facebook-share.png**, PNG Image, 435 × 157 pixels, scaled, Express homepage, Aug. 2022. [Online]. Available: `https://expressjs.com/images/express-facebook-share.png` (visited on 09/11/2022).

[70] **Apache_CouchDB_logo.svg.png**, PNG Image, 1920 × 1940 pixels, scaled, Wikimedia, Apr. 2021. [Online]. Available: `https://upload.wikimedia.org/wikipedia/commons/thumb/7/72/Apache_CouchDB_logo.svg/1920px-Apache_CouchDB_logo.svg.png` (visited on 09/11/2022).

[71] **Unofficial_JavaScript_logo_2.svg.png**, PNG Image, 1920 × 1920 pixels, scaled, Wikimedia, Mar. 2022. [Online]. Available: `https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Unofficial_JavaScript_logo_2.svg/1920px-Unofficial_JavaScript_logo_2.svg.png` (visited on 09/11/2022).

[72] **logo.png**, PNG Image, 400 × 400 pixels, scaled, Vue.js homepage, Sep. 2022. [Online]. Available: `https://vuejs.org/images/logo.png` (visited on 09/11/2022).

[73] **logo-b6e1ef6e.svg**, SVG Image, Sass homepage, Sep. 2022. [Online]. Available: `https://sass-lang.com/assets/img/logos/logo-b6e1ef6e.svg` (visited on 09/11/2022).

[77] **Use overlay networks**, *Docker Documentation*, Sep. 2022. [Online]. Available: `https://docs.docker.com/network/overlay` (visited on 10/01/2022).

[78] **Docker Service Create**, *Docker Documentation*, Sep. 2022. [Online]. Available: `https://docs.docker.com/engine/reference/commandline/service_create` (visited on 10/02/2022).

[79] **Eduroam Installation Script**, Leibniz Supercomputing Center (LRZ), Oct. 2022. [Online]. Available: `https://wlan.lrz.de/static/cat-test/eduroam-linux-de.py` (visited on 10/01/2022).

[80] **Paramiko**, *GitHub*, Oct. 2022. [Online]. Available: `https://github.com/paramiko/paramiko` (visited on 10/02/2022).

[81] W. Magalhaes, **Splash_Screen_Python_PySide2**, *GitHub*, Oct. 2022. [Online]. Available: `https://github.com/Wanderson-Magalhaes/Splash_Screen_Python_PySide2` (visited on 10/02/2022).

[82] **three-geo**, *GitHub*, Oct. 2022. [Online]. Available: `https://github.com/w3reality/three-geo` (visited on 10/02/2022).

[83] **quaternion_sensor_3d_nodejs**, *GitHub*, Oct. 2022. [Online]. Available: `https://github.com/ZaneL/quaternion_sensor_3d_nodejs` (visited on 10/02/2022).

[84] **Live streaming web audio and video**, *MDN Web Docs*, Oct. 2022. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/Guide/Audio_and_video_delivery/Live_streaming_web_audio_and_video` (visited on 10/01/2022).

[85] **RTSPtoWeb**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/deepch/RTSPtoWeb (visited on 10/02/2022).

[86] **RTSPtoWebRTC**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/deepch/RTSPtoWebRTC (visited on 10/02/2022).

[87] **janus-gateway**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/meetecho/janus-gateway (visited on 10/02/2022).

[88] **kurento-rtsp2webrtc**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/lulop-k/kurento-rtsp2webrtc (visited on 10/02/2022).

[89] **RTSPtoImage**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/deepch/RTSPtoImage (visited on 10/02/2022).

[90] **zoneminder**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/ZoneMinder/zoneminder (visited on 10/02/2022).

[91] **Shinobi**, *GitHub*, Oct. 2022. [Online]. Available: https://github.com/ShinobiCCTV/Shinobi (visited on 10/02/2022).

[92] **Steaming-IP-Camera-Nodejs (sic!)** *GitHub*, Oct. 2022. [Online]. Available: https://github.com/xpcrts/Steaming-IP-Camera-Nodejs (visited on 10/02/2022).

[93] **netdata**, *GitHub*, Oct. 2022, [Online; accessed 3. Oct. 2022]. [Online]. Available: https://github.com/netdata/netdata (visited on 10/02/2022).

A. Requirements

A.1. Requirements from WARR

Table X: Current requirements from WARR Rocketry for SCADA systems.

ID	Name	Text
R-CHRO-1	Sensor Support	Chronos must support the sensor handling for rocket propulsion tests and flight operations.
R-CHRO-1.1	Sensor Types	Chronos shall support all necessary sensor types specified as sub-requirements.
R-CHRO-1.1.1	Thermocouples	Chronos shall support a total of 75 Thermocouples.
R-CHRO-1.1.1.1	Type-K - Thermocouples	Chronos shall support a number of 50 Type K Thermocouples
R-CHRO-1.1.1.2	Type-T-Thermocouples	Chronos shall support a number of 25 Type T Thermocouples
R-CHRO-1.1.2	Static Pressure Sensors	Chronos shall support a minimum of 24 static pressure sensors.
R-CHRO-1.1.3	Dynamic Pressure Sensors	Chronos shall support a minimum of 3 dynamic pressure sensors.
R-CHRO-1.1.4	Regulated Valve Position Sensors	Chronos shall support a minimum of 2 sensors indicating the position of regulated valves.
R-CHRO-1.1.5	Contact Sensors	Chronos shall support a minimum of 5 contact sensors
R-CHRO-1.1.6	Load Cells	Chronos shall support a minimum of 4 Load Cells.
R-CHRO-1.1.7	Measurement Turbines	Chronos shall support a minimum of 2 measurement turbines.
R-CHRO-1.1.8	Coriolis Sensor	Chronos shall support a minimum of 2 coriolis sensors.
R-CHRO-1.1.9	Unspecified Voltage Output Sensors	Chronos shall support a minimum of 5 unspecified sensors with voltage output.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-1.2	Sensor Functionality	Cronos shall support all necessary sensor operations specified as sub-requirements.
R-CHRO-1.2.1	Sensor Accuracy and Precision	The sensors' own accuracy and precision limitations should be the dominant error.
R-CHRO-1.2.2	Detect Sensor Port Connection	Chronos shall be able to determine whether a sensor is plugged in or not.
R-CHRO-1.2.3	Sensor Fault Detection	Chronos shall be able to detect sensor faults. At a minimum, this means detecting shorts.
R-CHRO-1.2.4	Sensor Value Processing Time	The sensor processing time shall be insignificant compared to human reaction time of 300ms and below 60ms.
R-CHRO-1.2.5	Redline Latency	IF a value limit is exceeded on a sensor channel, Chronos shall initiate emergency procedures independently within 30ms.
R-CHRO-1.3	Gas Protection	Chronos shall be able to protect against dangerous gas concentrations
R-CHRO-1.3.1	Gas Detection	Chronos shall be able to detect dangerous gas levels.
R-CHRO-1.3.1.1	O2 Concentration Sensor	Chronos shall contain one O2 Concentration Sensor.
R-CHRO-1.3.1.2	Ethanol Concentration Sensor	Chronos shall contain one Ethanol vapor Concentration Sensor.
R-CHRO-1.3.1.3	Hydrogen Concentration Sensor	Chronos shall contain one Hydrogen Concentration Sensor.
R-CHRO-1.3.1.4	Methane Concentration Sensor	Chronos shall contain one Methane Concentration Sensor.
R-CHRO-1.3.2	Gas Information Transfer	Chronos shall be capable of transferring gas concentration data to Cor in all operational phases.
R-CHRO-1.3.3	Gas Automatic Protection	Chronos shall be able to take automatic protective measures if high gas concentrations occur

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-1.3.3.1	Gas Warning Horn	If the gas concentration exceeds a limit configurable by Cor, Chronos shall be able to activate a programmable horn sequence automatically.
R-CHRO-1.3.3.2	Gas Ignition Disable	If a Cor configurable gas concentration limit is exceeded, Chronos should be able to disable Ignition commands until safe levels are reached.
R-CHRO-1.3.3.3	Gas Safe State	If a Cor configurable gas concentration limit is exceeded, Chronos should optionally be able to engage the Safe State.
R-CHRO-2	Valve Support	Chronos shall support valve handling for rocket propulsion tests and flight operations.
R-CHRO-2.1	24V Valves	Chronos shall support at least 32 24V shut-off valves.
R-CHRO-2.1.1	24V Regulated Valves	Chronos shall support at least 2 regulated valves with analog control input.
R-CHRO-2.2	230V Valves	Chronos shall support at least 12 230V shut-off valves.
R-CHRO-2.3	Valve Functionality	Cronos shall support all necessary valve operations specified as sub-requirements.
R-CHRO-2.3.1	Valve Activation/Deactivation	Chronos shall be able to activate and deactivate each connected alve.
R-CHRO-2.3.2	Send Analog Signal	Chronos shall be able to send an analog control signal to the regulated valves
R-CHRO-2.3.3	Valve Fault Detection	Chronos shall be able to detect electrical valve faults.
R-CHRO-2.3.4	Detect Valve Port Connection	Chronos shall be able to determine whether a valve is plugged in or not.
R-CHRO-2.3.5	Valve Signal Latency	The Chronos internal valve actuation frequency shall be insignificant compared to the actuation time of a solenoid valve and no larger than 10 ms.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-2.3.6	Discrete Boundary Valve Automation	Chronos shall be able to support discrete valve control based on a lower and a upper value of a connected sensor.
R-CHRO-3	Igniter Support	Chronos shall support ignition of rocket engines.
R-CHRO-3.1	Spark Torch Igniter Support	Chronos shall be able to drive a spark torch igniter.
R-CHRO-3.2	Pyrotechnic Igniter	Chronos shall support two redundant pyrotechnic igniters.
R-CHRO-3.2.1	Pyrotechnic Circuit Redundancy	Chronos shall employ two completely separate identical pyrotechnic igniter circuits
R-CHRO-3.2.2	Continuity Detection	Chronos shall be able to detect that a pyro circuit is closed.
R-CHRO-3.2.3	Pyrotechnic Igniter Circuit Arming	Chronos shall support a key operated circuit interruptor to arm both pyro circuits.
R-CHRO-3.2.4	Independence of valve and sensor circuits from pyro circuits.	Failure within any pyro circuits must not influence valve and sensor circuits.
R-CHRO-4	Sequence Support	Chronos shall support the execution of test and flight valve actuation sequences.
R-CHRO-4.1	Autonomous Sequence Execution	Chronos shall be able to execute safe state, emergency purge and main sequences autonomously.
R-CHRO-4.1.1	Sequence Saving	Chronos shall be capable of saving a sequence of each type, safe state, emergency purge, and main, internally.
R-CHRO-4.1.1.1	Sequence Saving Confirmation	Chronos shall transmit confirmation of successful sequence saving to Cor upon completion.
R-CHRO-4.1.2	Sequence Receiving	Chronos shall be capable of receiving sequences from Cor.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-4.1.3	Sequence Sending	Chronos shall be capable of sending the currently saved sequences to Cor for verification.
R-CHRO-4.1.4	Receive and Send Initiation	The action of receiving and sending a sequence shall be initiated from Cor.
R-CHRO-4.1.5	Sequence Hierarchy	Initiation of the Safe State or Emergency Purge shall supersede any other active processes or sequences.
R-CHRO-4.1.6	Sequence Remote Initiation	Chronos shall be capable of initiating each sequence type upon command by Cor.
R-CHRO-4.1.7	Sequence Direct Initiation	Chronos shall be capable of directly initiating each sequence type using an control box independent of Cor.
R-CHRO-4.1.7.1	Control Box Wiring Distance	The Chronos control box shall have a wired connection of at least 25 m length.
R-CHRO-4.1.7.2	Control Box Emergency Stop	The control box shall have a standardized emergency stop that immediately cuts-off the power to the entire control box.
R-CHRO-4.1.7.2.1	Control Box Emergency Stop Restart	The control box emergency stop shall have a button to remotely restart Chronos.
R-CHRO-4.1.7.2.2	Emergency Shut Off	The emergency shut off shall shut off power after the UPS.
R-CHRO-4.1.7.3	Control Box Safe State Button	The safe state shall be initiated using an always accessible yellow button of equal diameter to the emergency purge.
R-CHRO-4.1.7.4	Control Box Purge Button	The Emergency Purge shall be initiated on the control box using an always accessible red button of no more than 2/3 the diameter of the emergency stop.
R-CHRO-4.1.7.5	Control Box Main Sequence Button	The main sequence should be initiated using an accidental contact protected green button of equal diameter to the purge.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-4.1.7.5.1	Control Box Main Sequence Key Interrupt	The control box main sequence initiation button should have optional 0 to 3 key interrupts.
R-CHRO-4.1.8	Sequence Initiation Latency	Upon arrival of command Chronos shall initiate the sequence with a latency insignificant compared to the actuation time of a solenoid valve and no larger than 10 ms.
R-CHRO-4.2	Safe State Sequence Definition	The safe-state sequence shall be a single step sequence with a predefined value on all valve and ignition channels.
R-CHRO-4.3	Emergency Purge Sequence Definition	The emergency purge sequence shall be a multi-step sequence of a maximum of 10 steps on all valve and ignition channels.
R-CHRO-4.4	Main Sequence Specifications	The main sequence is a complex sequence and decision configuration as defined by the sub requirements.
R-CHRO-4.4.2	Main Sequence Data Acquisition	The main sequence shall be capable of optionally initiating and ending data acquisition at a given time step.
R-CHRO-4.4.3	Main Sequence Redline Sequences	The main sequence shall support a minimum of 12 red-line sequences.
R-CHRO-4.4.3.1	Red Line Sequence Definition	Red line sequences are sequences across all valve and ignition channels with support for a minimum of 64 time steps.
R-CHRO-4.4.3.2	Red Line Sequence Red Lines	Once a red line sequence is initiated, the sequence shall be run to completion. No further red-lines shall be considered. The main sequence shall be terminated upon completion of the red line sequence.
R-CHRO-4.4.3.3	Red Line Sequence Selection	The appropriate red-line sequence shall be saved within the sequence in each time step.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-4.4.3.4	Red Line Initiation	A red line sequence shall be initiated on a time step if a red line is violated. A red line specifies an optional upper and optional lower limit on a sensor value on a given time step.
R-CHRO-4.4.3.5	Red Line Threshold	Chronos should be able to set a threshold for consecutive time steps in violation of the red line or number of time steps in violation over a set time period, necessary to initiate a red-line sequence.
R-CHRO-5	Data Acquisition	Chronos shall be able to independently save time, sensor and valve position data.
R-CHRO-5.1	Acquisition Duration	Chronos shall be able to save all data aquired over 10 minutes.This requirement considers the three time codes 100, 1000, 20000 Hz for thermocouples, standard sensors, and high-frequency sensors.
R-CHRO-5.2	Three saving frequencies	Chronos shall support at least three different data saving frequencies with the highest being at least 20kHz.
R-CHRO-5.3	Acquisition Start and Stop	Data Acquisition shall be able to be initiated and stopped by command by Cor.
R-CHRO-5.4	Data Transfer	Chronos shall automatically initiate data transfer to Cor upon completion of acquisition and internal saving.
R-CHRO-5.5	Data Persistence	Following data transfer, Chronos shall maintain the last data acquisition until a new acquisition is started.
R-CHRO-5.6	Data Persistence - Shut-off	In case of power-loss, at most 500 ms of data should be lost.
R-CHRO-5.7	Data Transfer - Reconnect	In case of connection loss, Chronos should automatically attempt to uplink stored data to Cor upon reconnection.
R-CHRO-5.8	Data Hashing	It would be nice if Chronos could optionally hash the stored data to confirm exact data transfer to Cor.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-6	Uninterrupted Power Supply	Chronos shall have an uninterupted power supply with sufficient energy to power Chronos for 30 minutes in case of power failure.
R-CHRO-7	Aggregated Data Cables	All cables exiting chronos and goint to the control room should be aggregated into one cable shroud.
R-CHRO-8	Transmission Frequency	Chronos shall send all sensor values at least 240 times a second to the control room.
R-CHRO-9	Internal PoE Ethernet Switch	Chronos shall have an internal Ethernet switch with at minimum 4 free PoE ports.
R-CHRO-9.1	Switch Connection	Chronos shall itself be connected to the Ethernet switch
R-CHRO-10	Master Node	It would be nice if Chronos could act as a Master Node with flight computers.
R-CHRO-10.1	Timecode Generation	Chronos shall output a time code signal for synchronization with other computers.
R-CHRO-10.1.1	Genlock Timecode	It would be nice if the time code was Genlock compatible for synchronisation with COTS time code devices.
R-CHRO-10.1.2	Timecode Port	The time code should be communicated over a standardized port.
R-CHRO-10.1.3	Timecode Tolerance	The time code synchronization should be reliable within on standard sensor time step (1 ms).
R-CHRO-10.2	Master Control	Chronos shall be able to control 1 slave node.
R-CHRO-10.2.1	Master Sensor Access	Chronos shall be able to receive sensor and valve position data and treat these as its own.
R-CHRO-10.2.2	Master Valve Access	Chronos shall be able to actuate slave valves as its own.
R-CHRO-10.2.2.1	Node Valve Synchronization	The timing difference between nodes' valve actuation shall be no larger than the time code synchonization tolerance achievable.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-10.2.3	Node Connection Loss	Chronos shall optionally continue the sequence with its own sensors and valves or trigger a red line upon connection loss with the slave node.
R-CHRO-10.2.3.1	Node Connection Loss Data Handling	If connection to the slave node is lost but the sequence continued the slave node sensor and valve shall be 'NaN' or equivalent Not-a-Number flag for the period of connection loss.
R-CHRO-10.2.4	Master Sequence Sending	Chronos shall be able to send the relevant portion of a sequence to a slave node for independent execution in connection loss.
R-CHRO-10.2.5	Master Sequence Receiving	Chronos shall be able to receive sequences from the slave node and verify that these are identical to the previously transmitted sequence.
R-CHRO-10.2.6	Node Red Line Exchange	Master and Slave Nodes shall be able to communicate triggered red lines with one another.
R-CHRO-10.2.6.1	Node Red Line Independence	If a Node triggers a red line it shall immediately initiate the red line sequence and not wait for communication with the other node.
R-CHRO-10.2.6.2	Node Red Line Dead Man's Switch	The nodes shall communicate their red lines in the form of a dead man's switch. A signal is send and verified if everything is ok and if a red line is triggered the signal is modified or disabled.
R-CHRO-10.2.6.3	Node Red Line During Connection Loss	If a node triggers a red line during a period of connection loss, it shall communicate the redline to the other node on reconnection.
R-CHRO-10.2.6.3.1	Node Red Line Desynchronization	If the desynchronization exceeds a Cor configured level, the trailing node shall immediately go into the Emergency Purge.
R-CHRO-11	19-Inch Rack-Mount	Chronos must be rack-mountable.

Continued on next page

Table X – continued from previous page

ID	Name	Text
R-CHRO-12	Connection Distance	The connection between Chronos and the control room must be able to span at least 600 m.

A.2. Requirements from the SPM Chair

Table XI: Requirements from the SPM for new Data Acquisition and Control Systems (DACS).

ID	Text	Value / Content	Justification
DACS1	The DACS shall be expandable in terms of number of sensors and actuators	Channel count first iteration x2 of initially required	Long term strategy of test bench
DACS2	The DACS shall be flexible in terms of hardware and software components		Mixing of measurement equipment from multiple suppliers
DACS3	The DACS shall contain uniform components, where possible		Reduction of error sources (compatibility), cost, bug fixing
DACS4	The DACS shall allow to measure all physical quantities interesting for the principal investigator	Thrust, pressure, mass flow, temperature, vibrations, optical measurements ...	Research purposes
DACS5	The DACS shall acquire and store data with time flag		Data synchronisation

Continued on next page

Table XI – continued from previous page

ID	Text	Value / Content	Justification
DACS6	The DACS shall allow the integration and flexible exchange / adaption of the control algorithm and architecture	Open vs. closed, feed forward vs. feed backward architecture, model based vs. AI based plant, . . .	Research purposes
DACS7	The DACS shall allow the control of the test specimen over the entire operational envelope		Research purposes
DACS8	The DACS shall allow to change the test procedure (test cell X to Y or different test sequence) rapidly and without sophisticated knowledge of the DACS software		Quick turn over times
DACS9	The DACS software shall be modular in terms of the UI		Only relevant information on UI for certain test cell
DACS10	The DACS hardware shall comply with regulations regarding explosion safety		Safety linked to ATEX regulations that may apply
DACS11	The DACS hardware shall be robust		Mechanically loaded environment (vibrations)
DACS12	The DACS shall contain an autonomous fail-safe emergency sequence and state		Operational Safety
DACS13	The DACS shall have an anomaly and fault detection		
DACS14	The DACS shall allow to define redlines		

Continued on next page

Table XI – continued from previous page

ID	Text	Value / Content	Justification
DACS15	The DACS shall be connected to the gas detectors		Warning message by DACS
DACS16	The DACS shall run on Linux		Delay times (10ms) naturally embedded into Windows environment
DACS17	The DACS software shall be operated on an offline machine network		Data safety
DACS18	The DACS software shall run on a central control entity / network		
DACS19	The DACS setup shall allow to work on multiple test cells / sections simultaneously		Quick preparation times
DACS20	The DACS UI shall display critical quantities for the operation		
DACS21	The DACS shall contain a backup power supply		Controllability of Test Bench
DACS22	In case of power-loss, at most 500 ms of data should be lost		
DACS23	The Data Acquisition process shall be initiated and stopped by command		
DACS24	The DACS shall have a certain temporal accuracy	< 1ms	
DACS25	The DACS software shall be comprehensible and well documented		Reasonable amount of work to understand software and make future adaptions

Continued on next page

Table XI – continued from previous page

ID	Text	Value / Content	Justification
DACS26	All cables shall be aggregated into as few as possible cable shrouds and boxes		Cleanliness of Workspace
DACS27	The DACS shall be able to display the telemetry data of the Hopper		
DACS28	The DACS shall be able to upload the trajectory / mission data on the Hopper system		
DACS29	The DACS shall support different actuators with continuous signals		Continuous control of control valves and pumps

B. Testing Equipment Specifications

Table XII: Specifications of Ethernet switch used for testing.

Manufacturer:	Perle Systems
Model:	IDS-108FPP-M2SC2-XT
Supported Ethernet Speeds:	10/100Base-TX

Table XIII: Specifications of router used for testing.

Manufacturer:	D-Link
Model:	DIR-615
Supported Ethernet Speeds:	10/100Base-TX

Table XIV: Specifications of Ethernet adapter used for testing.

Manufacturer:	TRENDnet
Model:	TU2-ET100
Supported Ethernet Speeds:	10/100Base-TX

Table XV: Specifications of lab power supply used for testing.

Manufacturer:	Manson
Model:	SPS9402

C. Developed Code

C.1. Installation Script Code

Listing 1: Python code for EX-UI's installation script.

```
###### IMPORTS ######
import subprocess as sh # Shell Integration
import platform # Computer/Python Information
import os # Graphics Environent
from importlib.util import find_spec # Check for python packages
from urllib import request
from urllib.error import HTTPError, URLError


###### FUNCTIONS ######
def _print(input):
    print("[Install:] " + str(input))

def _printerr(input):
    print("[ERROR:] " + str(input))

def checkPlatform():
    # Determines platform, checks it for compatibility and passes on the OS
    _print("Determining platform...")
    OS = platform.system()
    PYVER = platform.python_version()
    MACHINE = platform.machine()

    if PYVER.startswith('3'): #TODO: Check machine
        return (MACHINE, OS)
    else:
        _printerr("Python version " + PYVER + " is not supported.")
        exit()

def linux_isCMDAvailable(command):
    # GEANT PROJECT
    shell_command = sh.Popen(['which', command],
                                    stdout=sh.PIPE,
                                    stderr=sh.PIPE)
    shell_command.wait()
    if shell_command.returncode == 0:
        return True
    else:
        return False

def linux_checkPKGMGR():
    for cmd in ['apt', 'pacman', 'yum']: # Add more package managers here
            if linux_isCMDAvailable(cmd) == True:
                return cmd

def linux_checkGFX():
    # GEANT PROJECT
    _print("Determining graphics environment...")
    if os.environ.get('DISPLAY') is not None:
        for cmd in ['zenity', 'kdialog', 'yad']:
            if linux_isCMDAvailable(cmd) == True:
                return cmd
```

```
        return -1

def linux_promptInput(GFX, text):
    # GEANT PROJECT

    if GFX == 'zenity':
        command = ['zenity', '--entry', '--hide-text',
                   '--width=500', '--text=' + text]
    # Add more GFX support here
    else:
        _printerr("Graphics environment not supported!")
        exit()

    output = ''
    while not output:
        shell_command = sh.Popen(command,   stdout=sh.PIPE,
                                            stderr=sh.PIPE)
        out, _ = shell_command.communicate()
        output = out.decode('utf-8')
        if GFX == 'yad':
            output = output[:-2]
        output = output.strip()
        if shell_command.returncode == 1:
            _printerr("User quit or prompt could not be opened.")
            exit()
    return output

def linux_handleRootTasks(GFX, PKGMGR):
    # Password handling in this function should be most secure as it is implemented.
    # Encryption is not necessary here, because the password does not have to be stored.
    # A root process is created quickly and the password can be discarded.
    EXUIUSERNAME = "exui"
    EXUIPW = "exui"

    _print("Asking for password...")
    pw1 = "a"
    pw2 = "b"
    # Get Password
    while pw1 != pw2:
        # GEANT Project Code
        pw1 = linux_promptInput(GFX, "Please enter your password to " +
                                "authorize package installations:")
        pw2 = linux_promptInput(GFX, "Repeat password to confirm")
        if pw1 != pw2:
            linux_alert(GFX, "Passwords differ! Try again")

    # Check if bash is installed
    if linux_isCMDAvailable("bash") == False:
        _printerr("Couldn't find bash! Bash shell is currently required for Linux installation process.")
        exit() # TODO: implement different installation processes depending on shell, similar to GFX commands

    # Dynamically create installation script
    installcmds = ""

    # Use id as the first, dummy sudo command to acquire root permissions
    installcmds = "echo \"" + pw1 + "\"| sudo -S id\n" + installcmds
```

```
    # Clear password from memory
    pw1 = None
    pw2 = None
    # Add exui user (only if non existant)
    _print("Adding EX-UI user if necessary...")
    installcmds += "EXUIPWCRYPT=$(perl -e 'print crypt($ARGV[0], \"salt\")' " + EXUIPW + ")\n"
    installcmds += "id -u " + EXUIUSERNAME + " || sudo useradd -s /bin/bash -d /home/exui/ -m -p ${EXUIPWCRYPT} -G sudo exui\n" #TODO: Remove sudo rights if they turn out to be unnecessary
    # Add system installation commands
    if(PKGMGR == "apt"):
        if linux_isCMDAvailable("ssh") == False:
            _print("Could not find ssh client, will install...")
            installcmds += "sudo apt-get update\n"
            installcmds += "sudo apt-get install -y openssh-client\n"

        if linux_isCMDAvailable("sshd") == False:
            _print("Could not find ssh server, will install...")
            installcmds += "sudo apt-get update\n"
            installcmds += "sudo apt-get install -y openssh-server\n"
        else:
            installcmds += "sudo systemctl enable --now ssh\n"

        # TODO?: Use docker-machine for more simplicity and compatibility (But uses VMs)
        if linux_isCMDAvailable("docker") == False:
            _print("Could not find docker, will install...")
            # Install Docker
            installcmds += "sudo apt-get update\n"
            installcmds += "sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common\n"
            installcmds += "curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg\n"
            installcmds += "echo \"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable\" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null\n"
            installcmds += "sudo apt-get update\n"
            installcmds += "apt-cache policy docker-ce\n"
            installcmds += "sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin\n"
            installcmds += "sudo usermod -aG docker ${USER}\n"
            installcmds += "sudo usermod -aG docker " + EXUIUSERNAME + "\n"
            installcmds += "sudo systemctl enable docker\n"
            # Install Docker Compose
            installcmds += "mkdir -p ~/.docker/cli-plugins/\n"
            installcmds += "curl -SL https://github.com/docker/compose/releases/download/v2.3.3/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-compose\n"
            installcmds += "chmod +x ~/.docker/cli-plugins/docker-compose\n"
        else:
            installcmds += "sudo usermod -aG docker ${USER}\n"
            installcmds += "sudo usermod -aG docker " + EXUIUSERNAME + "\n"

        if linux_isCMDAvailable("pip") == False:
            _print("Could not find pip, will install...")
            installcmds += "sudo apt-get update\n"
            installcmds += "sudo apt-get install -y python3-pip\n"
```

```
    else:
        _printerr("Package manager not supported. Exiting...")
        exit()


    # open bash and execute all system install commands
    _print("Starting main installation process. This can take a few minutes...")
    process = sh.Popen('/bin/bash', stdin=sh.PIPE, stdout=sh.PIPE)
    out, err = process.communicate(installcmds.encode("utf-8"))
    _print(out.decode("utf-8"))
    process.wait() # end installation process with root permissions
    installcmds = None # contains password, so remove from memory

    return

def linux_installPyDeps():
    # Dynamically install libraries used in start.py
    # No root permissions necessary here
    pipcmds = ""

    if find_spec("PySide2") == None:
        pipcmds += "pip install PySide2\n"
    if find_spec("PyYAML") == None:
        pipcmds += "pip install PyYAML\n"
    if find_spec("docker") == None:
        pipcmds += "pip install docker\n"
    if find_spec("redexpect") == None:
        pipcmds += "pip install paramiko\n"

    # open bash and execute all pip commands
    _print("Now installing python packages...")
    process = sh.Popen('/bin/bash', stdin=sh.PIPE, stdout=sh.PIPE)
    out, err = process.communicate(pipcmds.encode("utf-8"))
    _print(out.decode("utf-8"))
    process.wait()

    return

def linux_alert(GFX, text):
    # GEANT PROJECT
    """Generate alert message"""
    if GFX == 'zenity':
        command = ['zenity', '--warning', '--text=' + text]
    elif GFX == "kdialog":
        command = ['kdialog', '--sorry', text]
    elif GFX == "yad":
        command = ['yad', '--text=' + text]
    else:
        exit(1)
    sh.call(command, stderr=sh.DEVNULL)

def linux_copyPlugins():
    process = sh.Popen(["cp", "-r", "./plugins/OpenMCT/.", "./webserver/src/plugins
        /"])
    process.wait()

def linux_createDockerRegistry():
    process =   sh.Popen(["docker", "service", "create", "--name", "registry",
                "--publish", "5000:5000", "registry:2"])
    process.wait()
```

```python
    # TODO: suppress error msg if registry already exists

def verifyInternetConnection():
    try:
        request.urlopen("https://archlinux.org", timeout=5)
        return True
    except HTTPError:
        return False
    except URLError:
        return False



###### MAIN ######
if __name__ == "__main__":
    # Entry Point
    _print("###### EX-UI Installation Script ######")

    MACHINE = ""
    OS = ""
    PKGMGR = ""
    GFX = ""

    REQS = ""

    MACHINE, OS = checkPlatform()
    if OS == "Linux":
        PKGMGR = linux_checkPKGMGR()
        GFX = linux_checkGFX()

        if verifyInternetConnection() == True:
            linux_handleRootTasks(GFX, PKGMGR)
            linux_installPyDeps()
            linux_createDockerRegistry()
            linux_copyPlugins()
            _print("Installation complete. You can use start.py now.")
            exit(0)
        else:
            print("WARN: You are not connected to the Internet!")
            linux_copyPlugins()
            _print("Installation complete, but WARN: Internet connection was unavailable. Packages might be missing.")
            exit(0)

    # Add support for more OSs here
    else:
        _printerr("Platform " + OS + "  " + MACHINE + " is not supported")
        exit(1)
```

C.2. Start Script Code

Listing 2: Python code for EX-UI's start script.

```python
### IMPORTS ###
import sys
import subprocess as sh
from threading import Thread, Event

from PySide6 import QtCore, QtSvgWidgets
from PySide6.QtCore import (QMetaObject, QObject, QRect, QSize, Qt, Signal)
from PySide6.QtGui import (QColor, QFont, QScreen)
from PySide6.QtWidgets import *

import json
from yaml import safe_dump_all
import socket
from paramiko.client import SSHClient as ssh
from paramiko import AutoAddPolicy, ssh_exception


### ENVIRONMENT ###
EXUIUSR = "exui"
EXUIPW = "exui"
EQFILEMOUNTPT = '/eqconfig'
REGISTRYLOCATION = '127.0.0.1:5000' # this is defined by install.py
WEBSERVERPORT = 8080
ADAPTERPORT = 9000
ADAPTERFOLDER = './adapters/'

ADAPTERLIST = [ # REGISTER NEW ADAPTERS HERE
    'tcp',
    'opcua',
    'serial',
    'video',
    'benchmark',
    'locat2d',
    'locat3d',
    'orien3d',
    'aprs',
    'netdata',
]

SPECIALDATADICT = { # REGISTER NEW SPECIAL DATA TYPES THAT NEED
    'gps': ['locat2d', 'locat3d'], # THEIR OWN ADAPTERS HERE
    'orientation': ['orien3d']
}

DOCKERFILENAME = "Dockerfile"

USENETDATA = True

### GLOBAL VARS ###
progressValue = 0
EQPATH = ""
EQCONFIG = {}


### UI CLASSES ###
class UI_ProgressScreen(object):
```

```
    def setupUi(self, SplashScreen, width, height):
        if SplashScreen.objectName():
            SplashScreen.setObjectName(u"SplashScreen")

        self.centralwidget = QWidget(SplashScreen)
        self.centralwidget.setObjectName(u"centralwidget")

        self.verticalLayout = QVBoxLayout(self.centralwidget)
        self.verticalLayout.setSpacing(0)
        self.verticalLayout.setObjectName(u"verticalLayout")
        self.verticalLayout.setContentsMargins(10, 10, 10, 10)

        self.dropShadowFrame = QFrame(self.centralwidget)
        self.dropShadowFrame.setObjectName(u"dropShadowFrame")
        self.dropShadowFrame.setStyleSheet(u"QFrame {    \n"
            "    background-color: rgb(57, 57, 57);       \n"
            "    color: rgb(220, 220, 220);\n"
            "    border-radius: 10px;\n"
            "}") # OpenMCT colors
        self.dropShadowFrame.setFrameShape(QFrame.StyledPanel)
        self.dropShadowFrame.setFrameShadow(QFrame.Raised)

        # EX-UI Logo
        self.label_title = QtSvgWidgets.QSvgWidget(self.dropShadowFrame, "./resources/logos/exui_logo.svg")
        self.label_title.load("./resources/logos/exui_logo.svg")
        self.label_title.setGeometry(QRect(0.3 * width - 10, 0.1 * height, 0.4 * width, 0.15 * height))

        self.label_description = QLabel(self.dropShadowFrame)
        self.label_description.setObjectName(u"label_description")
        self.label_description.setGeometry(QRect(0, 0.6 * height, width - 10, 0.05 * height))
        self.label_description.setText("Starting")
        font1 = QFont()
        font1.setFamily(u"Segoe UI")
        font1.setPointSize(14)
        self.label_description.setFont(font1)
        self.label_description.setStyleSheet(u"color: rgb(190, 190, 190);")
        self.label_description.setAlignment(Qt.AlignCenter)

        self.progressBar = QProgressBar(self.dropShadowFrame)
        self.progressBar.setObjectName(u"progressBar")
        self.progressBar.setGeometry(QRect(0.05 * width - 10, 0.5 * height, 0.9 * width, 0.05 * height))
        self.progressBar.setAlignment(Qt.AlignCenter)
        self.progressBar.setStyleSheet(u"QProgressBar {\n"
            "    \n"
            "    background-color: #9bdafa;\n"
            "    border-style: none;\n"
            "    border-radius: 10px;\n"
            "    text-align: center;\n"
            "    color: #ffffff;\n"
            "}\n"
            "QProgressBar::chunk{\n"
            "    border-radius: 10px;\n"
            "    background-color: qlineargradient(spread:pad, x1:0, y1:0.5, x2:1, y2:0.5, stop:0 rgba(250, 142, 62, 255), stop:0.323529 rgba(125, 73, 151, 255), stop:0.593137 rgba(7, 97, 176, 255), stop:0.828431 rgba(106, 178, 225, 255), stop:1 rgba(155, 218, 250, 255));\n"
```

```
            "}") # EX-UI logo colors
        self.progressBar.setFont(font1)
        self.progressBar.setValue(0)

        self.label_credits = QLabel(self.dropShadowFrame)
        self.label_credits.setObjectName(u"label_credits")
        self.label_credits.setGeometry(QRect(-10, 0.85 * height, width, 0.05 * height))
        font3 = QFont()
        font3.setFamily(u"Segoe UI")
        font3.setPointSize(10)
        self.label_credits.setFont(font3)
        self.label_credits.setStyleSheet(u"color: rgb(120, 120, 120);") #OpenMCT Color
        self.label_credits.setAlignment(Qt.AlignCenter)
        self.label_credits.setText("EX-UI Start Application Version 1.0.  Authors: Antonio Steiger")

        self.verticalLayout.addWidget(self.dropShadowFrame)

        SplashScreen.setCentralWidget(self.centralwidget)

        QMetaObject.connectSlotsByName(SplashScreen)

class UI_StartScreen(object):
    def setupUi(self, StartScreen, width, height):
        if StartScreen.objectName():
            StartScreen.setObjectName(u"StartScreen")

        self.centralwidget = QWidget(StartScreen)
        self.centralwidget.setObjectName(u"centralwidget")

        self.verticalLayout = QVBoxLayout(self.centralwidget)
        self.verticalLayout.setSpacing(0)
        self.verticalLayout.setObjectName(u"verticalLayout")
        self.verticalLayout.setContentsMargins(10, 10, 10, 10)

        self.dropShadowFrame = QFrame(self.centralwidget)
        self.dropShadowFrame.setObjectName(u"dropShadowFrame")
        self.dropShadowFrame.setStyleSheet(u"QFrame {    \n"
            "    background-color: rgb(57, 57, 57);      \n"
            "    color: rgb(220, 220, 220);\n"
            "    border-radius: 10px;\n"
            "}") # OpenMCT colors
        self.dropShadowFrame.setFrameShape(QFrame.StyledPanel)
        self.dropShadowFrame.setFrameShadow(QFrame.Raised)

        # EX-UI Logo
        self.label_title = QtSvgWidgets.QSvgWidget(self.dropShadowFrame, "./resources/logos/exui_logo.svg")
        self.label_title.load("./resources/logos/exui_logo.svg")
        self.label_title.setGeometry(QRect(0.3 * width - 20, 0.1 * height, 0.4 * width, 0.15 * height))

        # Select File Button
        self.filebutton = QPushButton("Select File...", self.dropShadowFrame)
        self.filebutton.setGeometry(QRect(0.2 * width - 10, 0.6 * height, 0.15 * width, 0.05 * height))
        self.filebutton.clicked.connect(StartScreen.showFileDialog)
        self.filebutton.setStyleSheet("""
```

```
            QPushButton {
                color: white;
                background-color: #0761b0;
                border-radius: 10px;
            }
            QPushButton:hover {
                background-color: #6ab2e1;
            }
            QPushButton:pressed {
                background-color: #FA8E3E;
            }
            """)

        # Description next to file select button
        self.filebuttonlabel = QLabel(self.dropShadowFrame)
        font = QFont()
        font.setFamily(u"Segoe UI")
        font.setPointSize(13)
        self.filebuttonlabel.setFont(font)
        self.filebuttonlabel.setStyleSheet(u"color: rgb(190, 190, 190);")
        self.filebuttonlabel.setGeometry(0.36 * width - 10, 0.6 * height, 0.6 * width, 0.05 * height)
        self.filebuttonlabel.setText("Choose Equipment Configuration File")

        # Credits, Version
        self.label_credits = QLabel(self.dropShadowFrame)
        self.label_credits.setObjectName(u"label_credits")
        self.label_credits.setGeometry(QRect(-10, 0.85 * height, width, 0.05 * height))
        font3 = QFont()
        font3.setFamily(u"Segoe UI")
        font3.setPointSize(10)
        self.label_credits.setFont(font3)
        self.label_credits.setStyleSheet(u"color: rgb(120, 120, 120);")
        self.label_credits.setAlignment(Qt.AlignCenter)
        self.label_credits.setText("EX-UI Start Application Version 1.0.  Authors: Antonio Steiger")

        self.verticalLayout.addWidget(self.dropShadowFrame)
        StartScreen.setCentralWidget(self.centralwidget)

        QMetaObject.connectSlotsByName(StartScreen)


### WINDOW CLASSES ###
class ProgressScreen(QMainWindow):
    def __init__(self, xsize, ysize):
        QMainWindow.__init__(self)
        self.ui = UI_ProgressScreen()
        self.ui.setupUi(self, width=xsize, height=ysize)

        self.setWindowTitle("EX-UI")

        self.setFixedSize(QSize(xsize, ysize))
        # Center Window on screen
        center = QScreen.availableGeometry(QApplication.primaryScreen()).center()
        geo = self.frameGeometry()
        geo.moveCenter(center)
        self.move(geo.topLeft())

```

```
        # Remove Title Bar
        self.setWindowFlag(QtCore.Qt.FramelessWindowHint)
        self.setAttribute(QtCore.Qt.WA_TranslucentBackground)

        # Drop Shadow Effect
        self.shadow = QGraphicsDropShadowEffect(self)
        self.shadow.setBlurRadius(20)
        self.shadow.setXOffset(0)
        self.shadow.setYOffset(0)
        self.shadow.setColor(QColor(0, 0, 0, 60))
        self.ui.dropShadowFrame.setGraphicsEffect(self.shadow)

class StartScreen(QMainWindow):
    # Signal for switching to next window
    switch_window = Signal()

    def __init__(self, xsize, ysize):
        super().__init__()
        self.ui = UI_StartScreen()
        self.ui.setupUi(self, width=xsize, height=ysize)

        self.setWindowTitle("EX-UI")

        self.setFixedSize(QSize(xsize, ysize))
        # Center Window on screen
        center = QScreen.availableGeometry(QApplication.primaryScreen()).center()
        geo = self.frameGeometry()
        geo.moveCenter(center)
        self.move(geo.topLeft())

        #Remove Title Bar
        self.setWindowFlag(QtCore.Qt.FramelessWindowHint)
        self.setAttribute(QtCore.Qt.WA_TranslucentBackground)

        # # Drop Shadow Effect
        self.shadow = QGraphicsDropShadowEffect(self)
        self.shadow.setBlurRadius(20)
        self.shadow.setXOffset(0)
        self.shadow.setYOffset(0)
        self.shadow.setColor(QColor(0, 0, 0, 60))

    def transform(self, text):
        return QObject.tr(text)

    def showFileDialog(self):
        global EQPATH
        filewindow = QMainWindow()
        #filewindow.setCentralWidget(centralwidget)
        #filewindow.show()
        EQPATH, _ = QFileDialog.getOpenFileName(filewindow, self.transform("Load Equipment Configuration"),
                                self.transform("."), self.transform("JSON Files (*.json)"))
        filewindow.close()
        # Only transition to start process with progress screen if non null string is returned
        if(EQPATH != ""):
            self.switch_window.emit()

class WindowController():
```

```python
    def __init__(self, xsize, ysize):
        self.xsize = xsize
        self.ysize = ysize

    def showStartScreen(self):
        self.startScreen = StartScreen(self.xsize, self.ysize)
        self.startScreen.show()
        self.startScreen.switch_window.connect(self.showProgressScreen)

    def showProgressScreen(self):
        global progressValue

        self.startScreen.close()
        self.progressScreen = ProgressScreen(self.xsize, self.ysize)
        self.progressScreen.show()

        progressEvent = Event()
        th = Thread(target=mainStart, args=(progressEvent,))
        th.start()
        # Thread is blocking for some reason at the moment.

        progressEvent.wait()
        self.progressScreen.ui.progressBar.setValue(progressValue)
        self.progressScreen.ui.label_description.setText("Parsed equipment configuration.")
        progressEvent.clear()

        th.join()

        self.progressScreen.ui.progressBar.setValue(100)
        self.progressScreen.ui.label_description.setText("Start Complete. Closing in 5s.")
        # Close start application after 5s
        QtCore.QTimer.singleShot(5000, self.progressScreen.close)



### CLASSES ###
class DockerService():
    def __init__(self, name, initialPort):
        global EQFILEMOUNTPT
        global REGISTRYLOCATION
        global ADAPTERFOLDER
        global ADAPTERPORT
        global DOCKERFILENAME

        self.yaml = {
            'image': '',
            'build': {
                'context': '',
                'dockerfile': ''
            },
            'deploy': {
                'replicas': 1
            },
            'ports': [],
            'tty': True,
            'configs': []
        }

```

```
        self.yaml['ports'].append(str(initialPort) + "-" + str(initialPort) +
                            ":" + str(ADAPTERPORT))
        self.yaml['build']['context'] = ADAPTERFOLDER + name

        if name.casefold() == "web" or name.casefold() == "webserver":
            self.yaml["build"]["context"] = "./webserver"
            self.yaml["ports"][0] = str(WEBSERVERPORT) + ":" + str(WEBSERVERPORT)
        else:
            self.yaml["entrypoint"] = [
                "/bin/sh",
                "-c",
                "node adapter.js $$TASKID"
            ]
            self.yaml["environment"] = {
                "TASKID": '{{.Task.Slot}}'
            }

        self.yaml['image'] = REGISTRYLOCATION + '/exui_' + name
        self.yaml['build']['dockerfile'] = DOCKERFILENAME
        self.yaml['configs'].append(EQFILEMOUNTPT.strip('/'))

        self.portrange = [initialPort, initialPort]

    def increaseReplicas(self):
        self.yaml['deploy']['replicas'] += 1
        self.portrange[1] += 1
        portmap = self.yaml['ports'][0].split(":")
        portmap[0] = str(self.portrange[0]) + '-' + str(self.portrange[1])
        self.yaml['ports'][0] = portmap[0] + ':' + portmap[1]

    def getYAML(self):
        return self.yaml


### FUNCTIONS ###
def mainStart(progressEvent):
    global progressValue

    # TODO:
    # Progress bar currently not functional. Likely due to mainStart
    # blocking main rendering loop although it is launched in a thread

    parseEqConfig()
    progressValue = 5
    progressEvent.set()

    ip = checkHostIp()
    progressValue = 10
    progressEvent.set()

    managertoken, workertoken = initDockerSwarm(ip)
    progressValue = 15
    progressEvent.set()

    addSwarmNodes(managertoken, workertoken, ip)
    # progressValue = 25
    # progressEvent.set()

    createDockerCompose()
    # progressValue = 35
```

```
    # progressEvent.set()

    buildDockerCompose()
    # progressValue = 75
    # progressEvent.set()

    pushDockerCompose()
    # progressValue = 85
    # progressEvent.set()

    dockerStackDeploy()
    # progressValue = 100
    # progressEvent.set()

def parseEqConfig():
    global EQPATH
    global EQCONFIG

    try:
        eqfile = open(EQPATH)
        EQCONFIG = json.load(eqfile)

    except OSError:
        print("Could not open eq config file " + EQPATH +" Maybe it is in a protected directory?")
        sys.exit(1) #TODO show error in UI and allow to try again
    except json.JSONDecodeError:
        print("Error parsing equipment configuration!")
        sys.exit(1) # TODO show error in UI and allow to try again

    # adapter ports start with 9001 and increase by 1 with each data source
    # docker cannot map replicas to 9001 and 9007 for example, it has to be
    # a range like 9001-9002. Therefore: Reorder eqconfig by source type
    # Reorder:
    EQCONFIG["datasources"].sort(key=lambda source: str.lower(source["type"]))
    # Overwrite the eqconfig:
    try:
        eqfile = open(EQPATH, "w")
        json.dump(EQCONFIG, eqfile, indent=4)
    except OSError:
        print("Could not open eq config file " + EQPATH +" Maybe it is in a protected directory?")
        sys.exit(1) #TODO show error in UI and allow to try again


def checkHostIp():
    global EQCONFIG

    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(0)
    try:
        s.connect(('10.254.254.254', 1))
        local_ip = s.getsockname()[0]
    except Exception:
        print("ERROR: Could not determine local IP address.")
        sys.exit()
    finally:
        s.close()

    for pc in EQCONFIG["computers"]:
```

```
        if pc["ip"] == local_ip and pc["type"] == "manager":
            return local_ip

    print("ERROR: Your local IP address is not in the equipment configuration.")
    sys.exit()

def initDockerSwarm(ip):
    managertoken = workertoken = ""

    try:
        # try to get join tokens
        workertoken = sh.check_output(['docker', 'swarm', 'join-token',
                        '-q', 'worker'])
        managertoken = sh.check_output(['docker', 'swarm', 'join-token',
                        '-q', 'manager'])
    except sh.CalledProcessError:
        # except swarm is not initialized
        try:
            # Initialize swarm
            sh.check_output(['docker', 'swarm', 'init',
                '--advertise-addr', str(ip)])
            workertoken = sh.check_output(['docker', 'swarm', 'join-token',
                            '-q', 'worker'])
            managertoken = sh.check_output(['docker', 'swarm', 'join-token',
                            '-q', 'manager'])
        except sh.CalledProcessError:
            # Some unexpected error occured
            print("ERROR: Could not initialize docker swarm.")
            sys.exit()

    managertoken = managertoken.decode().strip('\n')
    workertoken = workertoken.decode().strip('\n')

    return managertoken, workertoken

def addSwarmNodes(managertoken, workertoken, ip):
    for pc in EQCONFIG["computers"]:
        if pc["ip"] != ip:
            session = ssh()
            session.set_missing_host_key_policy(AutoAddPolicy())
            # Connect to computer's ssh
            try:
                session.connect(pc["ip"], username=EXUIUSR, password=EXUIPW)
            except ssh_exception.AuthenticationException:
                print("ERROR: Adding computer " + pc["name"] + " with ip " + pc["ip
                    "] +
                    " failed. Did you run install.py on that pc?")
                sys.exit()
            # Add it to swarm
            try:
                if pc["type"] == "worker":
                    stdin, stdout, stderr =  session.exec_command("docker swarm
                        join --token " + workertoken +
                                    " --advertise-addr " + pc["ip"] + " " + ip + "
                                        :2377" )
                    #stdin, stdout, stderr = session.exec_command("echo $USER")
                elif pc["type"] == "manager":
                    stdin, stdout, stderr =  session.exec_command("docker swarm
                        join --token " + managertoken +
```

```
                                    "--advertise-addr" + pc["ip"] + " " + ip + ":" + "2377" )
                exitcode = stdout.channel.recv_exit_status()

                if exitcode != 0:
                    print("ERROR: Swarm join command could not be run on pc \"" +
                    pc["name"] + "\".")
                    sys.exit(1)

            except ssh_exception.SSHException:
                print("ERROR: Swarm join command could not be run on pc \"" +
                    pc["name"] + "\".")
                sys.exit(1)

            session.close()

def createDockerCompose():
    global EQCONFIG
    global EQFILEMOUNTPT
    global REGISTRYLOCATION
    global WEBSERVERPORT
    global ADAPTERPORT
    global ADAPTERLIST
    global SPECIALDATADICT
    global USENETDATA

    composeyml = [{
        'version': '3.9',
        'services': { },
        'configs': { }
    }]

    composeyml[0]["configs"][EQFILEMOUNTPT.strip("/")] = {}
    composeyml[0]["configs"][EQFILEMOUNTPT.strip("/")]["file"] = EQPATH

    # webserver service
    webserver = DockerService("web", 8080)
    composeyml[0]["services"]["web"] = webserver.getYAML()

    if USENETDATA:
        # netdata service
        composeyml[0]["services"]["netdata"] = {
            'image': '127.0.0.1:5000/exui_netdata',
            'build': {
                'context': './adapters/netdata',
                'dockerfile': 'Dockerfile'
            },
            'ports': [],
            'cap_add': [ 'SYS_PTRACE' ],
            'security_opt': [ 'apparmor:unconfined'],
            'volumes': [
                'netdataconfig:/etc/netdata',
                'netdatalib:/var/lib/netdata',
                'netdatacache:/var/cache/netdata',
                '/etc/passwd:/host/etc/passwd:ro',
                '/etc/group:/host/etc/group:ro',
                '/proc:/host/proc:ro',
                '/sys:/host/sys:ro',
                '/etc/os-release:/host/etc/os-release:ro'
            ],
```

```
            'deploy': {
                'mode': 'global' # runs netdata container on every node
            }
        }
        composeyml[0]["volumes"] = {
            'netdataconfig': {},
            'netdatalib': {},
            'netdatacache': {}
        }
        portrange = [20001, 20001]
        for pc in EQCONFIG["computers"]:
            composeyml[0]["services"]["netdata"]["ports"].append(str(portrange[0]) +
                "-" + str(portrange[1]) + ":" + "19999")
            portrange[1] += 1

    neededAdapters = {}
    curfreeport = 9001

    for i, source in enumerate(EQCONFIG["datasources"]):
        if source["type"].casefold() in ADAPTERLIST:
            # Check which adapter is needed for data source
            if source["type"] in neededAdapters:
                neededAdapters[source["type"].casefold()].increaseReplicas()
            else:
                neededAdapters[source["type"].casefold()] = DockerService(
                    source["type"].casefold(), curfreeport)
            # put port in eqconfig
            EQCONFIG["datasources"][i]["destport"] = curfreeport
            curfreeport += 1
        else:
            print("WARN: Data source type " + source["type"] + " is not supported!\n" +
                "Supported types: " + ADAPTERLIST)

    # Check if any data source supplies special data
    for datatype in SPECIALDATADICT:
        for adapter in SPECIALDATADICT[datatype]:
            for i, src in enumerate(EQCONFIG["datasources"]):
                if datatype in src:
                    if adapter in neededAdapters:
                        neededAdapters[adapter.casefold()].increaseReplicas()
                    else:
                        neededAdapters[adapter.casefold()] = DockerService(
                            adapter.casefold(), curfreeport)
                    # put port in eqconfig
                    EQCONFIG["datasources"][i][adapter] = curfreeport
                    curfreeport += 1

    for service in neededAdapters:
        composeyml[0]["services"][service] = neededAdapters[service].getYAML()

    # save docker-compose.yml
    try:
        ymlfile = open("./docker-compose.yml", "w")
    except OSError:
        print("ERROR: Could not open docker-compose.yml")
        sys.exit(1)

    safe_dump_all(composeyml, ymlfile, sort_keys=False, indent=2)
```

```

    # save eqconfig
    try:
        eqfile = open(EQPATH, "w")
        json.dump(EQCONFIG, eqfile, indent=4)
    except OSError:
        print("Could not open eq config file " + EQPATH +" Maybe it is in a protected directory?")
        sys.exit(1)

def buildDockerCompose():
    process = sh.Popen(["docker", "compose", "build"])
    process.wait()
    if process.returncode != 0:
        print("ERROR: Docker compose build failed.")
        sys.exit(1)

def pushDockerCompose():
    process = sh.Popen(["docker", "compose", "push"])
    process.wait()
    if process.returncode != 0:
        print("ERROR: Docker compose push failed.")
        sys.exit(1)

def dockerStackDeploy():
    process = sh.Popen(["docker", "stack", "deploy", "--compose-file", "docker-compose.yml", "exui"])
    process.wait()
    if process.returncode != 0:
        print("ERROR: Docker stack deploy failed.")
        sys.exit(1)


### MAIN ###
if __name__ == "__main__":
    # Entry Point
    app = QApplication(sys.argv)

    # TODO: Determine appropriate window resolution to fill quarter of screen

    controller = WindowController(860, 540)
    controller.showStartScreen()

    sys.exit(app.exec())
```

C.3. TCP Adapter Code

Listing 3: TCP adapter, written in JavaScript. Acts as a TCP socket client and translates connection onto a WebSocket to interface with a web browser.

```
var argv = require('minimist')(process.argv.slice(2))
const Net = require('node:net');
const WebSocketServer = require('ws').Server
const fs = require('fs');

// ENVIRONMENT
const EQFILEMOUNTPT = '/eqconfig';
// The equipment configuration is mounted into the adapters'
// docker container by docker swarm using the "configs" property
// in docker-compose.yml
const TASKSLOT = argv._[0];
//The task slot identifies the replica number of this adapter.
//If the task slot is 1, the adapter is supposed to ONLY handle
//the first tcp source in the eqconfig. This way, adapters can be
//scaled and distributed with number of data sources
const DESTPORT = 9000
// The destination port is identical for every adapter, because it
// is automatically mapped to an unused port in range specified in
// docker-compose.yml

var EQCONFIG = {};
var SRC = {}

// GLOBAL VARS
var srcSocket = {};
var destSocket = {}
var timeout = {};
var polltimeout = {};
var timeoutstate = {};

function sleep(ms)
{
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}


function parseEqConfig()
{
    // Get equipment configuration file from file system
    // It has previously been mounted into the docker container by the "configs"
    // property in the docker-compose.yml

    let eqfile = fs.readFileSync(EQFILEMOUNTPT);
    EQCONFIG = JSON.parse(eqfile);
    // console.log(eqconfig)

    let tcpcounter = 0
    EQCONFIG.datasources.forEach( (source, i) => {
        if(source.type == "TCP" || source.type == "tcp") {
            tcpcounter += 1;
            if(tcpcounter == TASKSLOT) { //i.e. Only handle first tcp source if
                TASKSLOT=1
                SRC = source;
            }
```

```
        }
    });
}

async function connectToSources()
{
        // Try to connect to source
        srcSocket.connect(SRC.sourceport, SRC.ip)
}

async function setupSockets()
{
    destSocket= new WebSocketServer({   port: DESTPORT,
                                        host: "0.0.0.0",
                                        clientTracking: true });
    srcSocket = new Net.Socket();
    srcSocket.setEncoding("UTF-8");
    timeoutstate = false;  //xx
}


function defineEvents()
{
    delimiter = SRC.delimiter;

    formats = {}
    for (const point of SRC.datapoints) {
        formats[point.label] = point.values[0].format
    }
    ids = {}
    for (const point of SRC.datapoints) {
        ids[point.label] = point.key
    }

    //If an error on connection to data source occurs, try to reconnect periodically
    srcSocket.on('error', async(error) => {
        console.log("[" + SRC.name + "]\t" + "Connection to " + SRC.name +
            " failed. Reason: \"" + error + "\" Trying to reconnect in 1s")
        //Let ex-ui client know about error:
        destSocket.clients.forEach(client => {
            client.send(JSON.stringify({
                key: '_CONN_',
                utc: Date.now(),
                value: 'Error'
            }));
        })
        await sleep(1000)
        // Reconnection interval will not be 1s but conn. establishment time + 1s
        connectToSources();
    });

    srcSocket.on('ready', () => {
        console.log("[" + SRC.name + "]\t" + "Connected.");
        //Start polling procedure (poll once)
        if("pollcommand" in SRC) {
            srcSocket.write(SRC.pollcommand);
        }
        //Let ex-ui client know about successful connection:
        destSocket.clients.forEach(client => {
```

```
            client.send(JSON.stringify({
                key: '_CONN_',
                utc: Date.now(),
                value: 'Good'
            }));
        })

        if ("timeout" in SRC) {
            if(SRC.timeout > 1) {
                timeout = setTimeout(async() => {
                    console.log("[" + SRC.name + "]\t" + "Not responding!")
                    //Let ex-ui client know about timeout
                    destSocket.clients.forEach(client => {
                        client.send(JSON.stringify({
                            key: '_CONN_',
                            utc: Date.now(),
                            value: 'Timeout'
                        }));
                    })

                    //If we just transitioned into timeout state, put one poll in write buffer to
                    //trigger data event when device responds again
                    if("pollcommand" in SRC && timeoutstate == false) {
                        srcSocket.write(SRC.pollcommand);
                    }
                    timeoutstate = true;

                    timeout.refresh();
                }, SRC.timeout);
            }
        }
    })

    // Handler for incoming TCP data from source
    var line = "";
    srcSocket.on('data', data => {
        //console.log("[" + SRC.name + "]\t" + "Received Data.");

        // If data arrives after a timeout event, log successful reconnection
        if(timeoutstate == true) {
            timeoutstate = false;
            //Let ex-ui client know connection is good
            destSocket.clients.forEach(client => {
                    client.send(JSON.stringify({
                        key: '_CONN_',
                        utc: Date.now(),
                        value: 'Good'
                    }));
                })
            console.log("[" + SRC.name + "]\t" + "Reconnected.")
        }
        // Data arrived, so everything is ok -> Reset timeout
        timeout.refresh();
        // Data arrived, so it is allowed to schedule next poll
        if("pollcommand" in SRC) {
            // The next block ensures that the poll is only rescheduled once and not
            // for every received character.
            if(polltimeout == null) {
```

```
                polltimeout = setTimeout(() => {
                    polltimeout = null;
                    srcSocket.write(SRC.pollcommand);
                },
                1000 / SRC.pollrate)
                //polltimeout.refresh();
            }
        }

        //Parse incoming Data
        str_data = data.toString();
        line += str_data;
        if(str_data === "\n") {
            split_line = line.split(delimiter);
            let format = formats[split_line[0]]
            let value = 0
            let packet = {}

            if(split_line[0] != undefined) {
                if (format == "float") {
                    value = parseFloat(split_line[1])
                    //console.log(value)
                    packet = {
                        value: value,
                        utc: Date.now(),
                        key: ids[split_line[0]]
                    }
                    destSocket.clients.forEach(client => {
                        client.send(JSON.stringify(packet));
                    })
                }
                else if ( format == "integer") {
                    value = parseInt(split_line[1])
                    //console.log(value)
                    packet = {
                        value: value,
                        utc: Date.now(),
                        key: ids[split_line[0]]
                    }
                    destSocket.clients.forEach(client => {
                        client.send(JSON.stringify(packet));
                    })
                }
                else if (format == "string") {
                    value = split_line[1]
                    //console.log(value)
                    packet = {
                        value: value,
                        utc: Date.now(),
                        key: ids[split_line[0]]
                    }
                    destSocket.clients.forEach(client => {
                        client.send(JSON.stringify(packet));
                    })
                }
                // If message has a label that is not in eq.json, it is treated as a log msg
                else if (ids.hasOwnProperty("log")) {
                    packet = {
                        value: line,
```

```
                        utc: Date.now(),
                        key: ids["log"]
                    }
                    destSocket.clients.forEach(client => {
                        client.send(JSON.stringify(packet));
                    })
                }
            }
            // Clear line buffer for next TCP data
            line = "";
        }
    })

    //WebSocket Server Listening Event
    destSocket.on('listening', () => {
        console.log("[" + SRC.name + "]\t" + "WebSocket Server is listening.");
    });

    //WebSocket Server Connection Event
    destSocket.on('connection', (ws, req) => {

        //Event: Connection established
        console.log("[" + SRC.name + "]\t" + "Successful Client Connection from: " +
                req.socket.remoteAddress);
        //Event: Client sends message
        ws.on('message', data => {
            console.log("[" + SRC.name + "]\t" + "<- Sending Command: " + data);
            srcSocket.write(data);
        });
        //Event: Client closes connection
        ws.on('close', () => {
            console.log("[" + SRC.name + "]\t" + "Connection with client at " +
                ws._socket.remoteAddress + " closed.");
        })
        //Event: Some error with client
        ws.on('error', () => {
            console.log("[" + SRC.name + "]\t" + "Error communicating with Client at" +
                ws._socket.remoteAddress);
        })
    });

    destSocket.on('close', () => {
        console.log("[" + SRC.name + "]\t" + "Websocket Server closed.")
    })
}

async function setupPolling()
{
    if("pollcommand" in SRC) {
        polltimeout = setTimeout(() => {
            polltimeout = null;
            srcSocket.write(SRC.pollcommand);

        },
        1000)
    }
}
```

```
async function main()
{
    await parseEqConfig();
    await setupSockets();
    // src = {} means connect to all sources
    await defineEvents();
    await connectToSources();
    setupPolling();
}


// ### SOFT SHUTDOWN ### //
process.on('SIGTERM', () => {
    console.log("Soft shutdown requested, bye...")
    // Soft close
    destSocket.clients.forEach((socket) => {
        socket.close();
    })
    // If a socket somehow stayed open after 5s, close it
    setTimeout(() => {
        destSocket.clients.forEach((socket) => {
          if ([socket.OPEN, socket.CLOSING].includes(socket.readyState)) {
            socket.terminate();
          }
        });
        process.exit(0)
      }, 5000);

    // TODO: handle srcSocket closing
})

main();
```

C.4. OPC UA Adapter Code

Note that this adapter has not yet been updated to the scaling approach described in Section 3.4.3. It therefore does not handle just a single OPC UA data source, nor does it get the required environment variable to determine which one it would be assigned to. Finally, it does still fetch the equipment configuration from the web server. As described in Section 3.4.3, this approach has been replaced with the file being mounted into the adapters file system by Docker.

These aspects are correctly implemented in the benchmark adapter (see Appendix C.5) and the TCP adapter (see Appendix C.3).

Listing 4: OPC UA Adapter, written in JavaScript. Acts as an OPC UA client and maps OPC UA communication onto a WebSocket for the browser; Publishes the object tree of the OPC UA server.

```
const fetch = require('node-fetch');
const WebSocketServer = require('ws').Server;
const util = require('util');

//OPC UA imports and definitions
const nodeopcua = require('node-opcua');
const OPCUAClient = nodeopcua.OPCUAClient;
const MessageSecurityMode = nodeopcua.MessageSecurityMode;
const SecurityPolicy = nodeopcua.SecurityPolicy;
const AttributeIds = nodeopcua.AttributeIds;
const makeBrowsePath = nodeopcua.makeBrowsePath;
const ClientSubscription = nodeopcua.ClientSubscription;
const TimestampsToReturn = nodeopcua.TimestampsToReturn;
const MonitoringParametersOptions = nodeopcua.MonitoringParametersOptions;
const ReadValueIdLike = nodeopcua.ReadValueIdLike;
const ClientMonitoredItem = nodeopcua.ClientMonitoredItem;
const DataValue = nodeopcua.DataValue;


// GLOBAL VARS
var eqconfig = {};
var clients = {};
var sessions = {};
var destSockets = {};
var datapoints = {};

var browseSocket = new WebSocketServer({    port: 9999,
                                            host: "0.0.0.0",
                                            clientTracking: true });

var subscriptionparams =    {
                                requestedPublishingInterval: 100, //Currently less
                                    than 100 is blocked somewhere
                                requestedLifetimeCount:      1500,
                                requestedMaxKeepAliveCount:   20,
                                maxNotificationsPerPublish:  0,
                                publishingEnabled: true,
                                priority: 10
                            }

// FUNCTIONS

function sleep(ms)
```

```
{
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}

function setupDestServers()
{
        eqconfig.datasources.forEach(source => {
        if(source.type == 'opcua') {
            destSockets[source.key] = new WebSocketServer({     port: source.destport,
                                                                host: "0.0.0.0",
                                                                clientTracking: true });
        }
    })

    defineDestEvents();
}

function defineDestEvents()
{
    eqconfig.datasources.forEach(source => {
        if(source.type == 'opcua') {
            //WebSocket Server Listening Event
            destSockets[source.key].on('listening', () => {
                console.log("[" + source.name + "]\t" + "WebSocket Server is listening.");
            });

            //WebSocket Server Connection Event
            destSockets[source.key].on('connection', (ws, req) => {

                //Event: Connection established
                console.log("[" + source.name + "]\t" + "Successful Client Connection from: " +
                        req.socket.remoteAddress);
                //Event: Client sends message
                ws.on('message', data => {
                    // if(data == '_BROWSE_\n') {
                    //     // Possibly wait
                    //     browseSocket.send(JSON.stringify(datapoints));
                    // }
                    // else {
                        console.log("[" + source.name + "]\t" + "<- Sending Command : " + data);
                        //send actual command or set variable XX
                    //}
                });
                //Event: Client closes connection
                ws.on('close', () => {
                    console.log("[" + source.name + "]\t" + "Connection with client at " +
                        ws._socket.remoteAddress + " closed.");
                })
                //Event: Some error with client
                ws.on('error', () => {
                    console.log("[" + source.name + "]\t" + "Error communicating with Client at" +
```

```
                        ws._socket.remoteAddress);
                })
            });

            // WebSocket Server close event
            destSockets[source.key].on('close', () => {
                console.log("[" + source.name + "]\t" + "Websocket Server closed.")
            })
        }
    })

}

function setupSourceClients()
{
    eqconfig.datasources.forEach(source => {
        if(source.type == 'opcua') {
            const connectionStrategy = {
                initialDelay: source.timeout
            }

            const options = {
                applicationName: source.key + "_Client",
                connectionStrategy: connectionStrategy,
                securityMode: nodeopcua.MessageSecurityMode.None,
                securityPolicy: nodeopcua.SecurityPolicy.None,
                endpointMustExist: false,
            };

            clients[source.key] = OPCUAClient.create(options);
        }
    })

    defineSourceEvents();
}

function defineSourceEvents()
{
    eqconfig.datasources.forEach(source => {
        if(source.type == 'opcua') {
            //Reconnection try event
            clients[source.key].on("start_reconnection", function() {
                console.log("[" + source.name + "]\t" + "Trying to reconnect...")
              });

            // Backoff Event
            clients[source.key].on("backoff", function(nb, delay) {
                console.log("[" + source.name + "]\t" + "  connection failed for
                    the", nb,
                    " time. Retry in ", delay, " ms");
            });

        }
    })
}

async function connectToSources()
{
    eqconfig.datasources.forEach(async(source) => {
        if(source.type == 'opcua') {
```

```
            try {
                // step 1 : connect to
                await clients[source.key].connect("opc.tcp://" + source.ip + ":" +
                    source.sourceport.toString());
                console.log("[" + source.name + "]\t" + "Connected.");

                // step 2 : Create session
                sessions[source.key] = await clients[source.key].createSession();
                console.log("[" + source.name + "]\t" + "Session created.");


            } catch(err) {
                console.log("[" + source.name + "]\t" + "Connection Error: ", err);
            }
        }
    })
}

async function fetchEqConfig()
{
    console.log("Fetching Equipment Configuration in 3s...")
    await sleep(3000); //give ex-ui webserver some time to start up
    while(true) {
        try{
            // Change "web" for localhost if you are not running in docker context!
            await fetch("http://localhost:8080/eq.json", { method: "Get" })
                .then(res => res.json())
                .then((json) => {
                    eqconfig = json;
                });
            //console.log(eqconfig);
            return;
        } catch (error) {
            if (error.name === 'AbortError') {
                console.log('Request was aborted. Trying again in 1s.');
                await sleep(1000);
                continue;
            }
            else if (error.name === 'FetchError') {
                console.log(error.message +  '. Trying again in 1s.');
                await sleep(1000);
                continue;
            }
            else {
                await sleep(1000);
                continue;
            }
        }
    }

}

async function parseSourceDataPoints()
{
    //For each opc ua data source, parse its folder structure and put it into a
        JSON object.
    //Normally the datapoints arrays in the eq.json have a depth of 1. For OPC UA,
        data points
    //are often categorized into parent nodes. This is respected here, with data
        points that
```

```
    //do not contain a "values" property.
    eqconfig.datasources.forEach(async(source) => {
        if(source.type == 'opcua') {
            datapoints[source.key] = [];

            opcuaobjects = await sessions[source.key].browse("ObjectsFolder");
            //opcuaobjects = opcuaobjects.toString().replace(/\/\*([\s\S]*?)\*\//g, "");
            //console.log(opcuaobjects.references)
            opcuaobjects.references.forEach(async(parentnode) => {
                if(parentnode.browseName.name != 'Server') {
                    //console.log(parentnode.browseName.name);

                    let index = datapoints[source.key].push({
                        name: parentnode.browseName.name,
                        key: source.name + "_" + parentnode.browseName.name,
                        datapoints: []
                    });

                    parentbrowseresult = await sessions[source.key].browse(parentnode.nodeId);
                    //console.log(parentbrowseresult.references);
                    parentbrowseresult.references.forEach(async(pointnode) => {
                        //Figure out the data type id
                        let dataTypeId = await sessions[source.key].getBuiltInDataType(pointnode.nodeId);
                        //Map the id to OPC UA data types and those to OpenMCT types
                        //For standard IDs, see https://reference.opcfoundation.org/Core/Part6/v104/5.1.2/
                        let format = ""
                        if(dataTypeId == 11) {
                            format = "float"
                        }
                        else {
                            format = "integer"
                        }

                        len = datapoints[source.key].length
                        datapoints[source.key][index - 1].datapoints.push({
                            name: pointnode.browseName.name,
                            key: pointnode.browseName.name,
                            values: [
                                {
                                    key: "value",
                                    name: "Value",
                                    unit: "",
                                    format: format,
                                    hints: {
                                        range: 1
                                    }
                                },
                                {
                                    key: "utc",
                                    name: " Time",
                                    format: "utc",
                                    hints: {
                                        domain: 1
                                    }
                                }
```

```
                            ]
                        })
                    })
                }
            })
            //console.log(datapoints);
        }
    });
}


function setupSubscriptions()
{
    //Update eqconfig
    eqconfig.datasources.forEach( source => {
        if(source.type == 'opcua') {
            source.datapoints = datapoints[source.key];
        }
    })

    console.log(eqconfig);

    //console.log(eqconfig);
    eqconfig.datasources.forEach(source => {
        if(source.type == 'opcua') {
            const subscription = ClientSubscription.create(sessions[source.key], subscriptionparams);
            // EVENT DIEFINITIONS
            //Start Event
            subscription.on("started", function() {
                console.log("[" + source.name + "]\t" + "Subscribed with ID ", subscription.subscriptionId);
            })

            //Termination Event
            subscription.on("terminated", function() {
                console.log("[" + source.name + "]\t" + "Subscription " + subscription.subscriptionId + " terminated.");
            })

            //Keep Alive Event (Server cannot supply data, but wants to keep suscription alive)
            subscription.on("keepalive", function() {
                console.log("[" + source.name + "]\t" + "\"Keep Alive\" signal received.");
            })

            //Error Event
            subscription.on("error", function() {
                console.log("[" + source.name + "]\t" + "Error in subscription with ID " + subscription.subscriptionId);
            })

            //ADD SUBSCRIPTION FOR EACH DATA POINT
            source.datapoints.forEach(point => {
                if(point.values) { //Parent level node
                    const itemToMonitor = {
                        nodeId: "ns=1;s=" + point.key,
                        attributeId: AttributeIds.Value
                    }
```

```
                    const itemParams = {
                        samplingInterval: 33,
                        discardOldest: true,
                        queueSize: 1
                    }
                    const monitoredItem  = ClientMonitoredItem.create(
                            subscription,
                            itemToMonitor,
                            itemParams,
                            TimestampsToReturn.Both
                    );

                    var regex = /[+-]?\d+(\.\d+)?/g;
                    monitoredItem.on("changed", (dataValue) => {
                        //console.log(dataValue.value.toString().match(regex).map(
                            function(v) { return parseFloat(v); })[0]);
                        packet = {
                            value: dataValue.value.toString().match(regex).map(
                                function(v) { return parseFloat(v); })[0],
                            utc: Date.now(), //Use sourceTimeStamp in Future
                            key: point.key
                        }
                        destSockets[source.key].clients.forEach(client => {
                            client.send(JSON.stringify(packet));
                        })
                        // console.log(dataValue.toString());
                    });
                }
                else if(point.datapoints) { //sub datapoints of parent nodes
                    point.datapoints.forEach(subpoint => {
                        const itemToMonitor = {
                            nodeId: "ns=1;s=" + subpoint.key,
                            attributeId: AttributeIds.Value
                        }
                        const itemParams = {
                            samplingInterval: 33,
                            discardOldest: true,
                            queueSize: 1
                        }
                        const monitoredItem  = ClientMonitoredItem.create(
                                subscription,
                                itemToMonitor,
                                itemParams,
                                TimestampsToReturn.Both
                        );

                        var regex = /[+-]?\d+(\.\d+)?/g;
                        monitoredItem.on("changed", (dataValue) => {
                            //console.log(dataValue.value.toString().match(regex).
                                map(function(v) { return parseFloat(v); })[0]);
                            packet = {
                                value: dataValue.value.toString().match(regex).map(
                                    function(v) { return parseFloat(v); })[0],
                                utc: Date.now(), //Use sourceTimeStamp in Future
                                key: subpoint.key
                            }
                            destSockets[source.key].clients.forEach(client => {
                                client.send(JSON.stringify(packet));
                            })
                            // console.log(dataValue.toString());
```

```
                        });
                    })
                }
            })
        }
    })
}


async function main()
{
    await fetchEqConfig();
    await setupSourceClients();
    await connectToSources();
    await sleep(3000); //Fix this XX
    await parseSourceDataPoints();
    await setupDestServers();
    await sleep(5000); //Fix this XX
    //console.log(util.inspect(datapoints, {showHidden: false, depth: null, colors:
         true}))
    setInterval(() => {
        browseSocket.clients.forEach(client => {
            client.send(JSON.stringify(datapoints));
        })
    }, 500);
    await setupSubscriptions();

}

main();
```

C.5. Benchmark Adapter Code

Listing 5: Benchmark adapter, written in JavaScript. Acts as a dummy data source by supplying samples of a sine wave at a specifiable rate.

```
var argv = require('minimist')(process.argv.slice(2));
const WebSocketServer = require('ws').Server;
const Math = require('mathjs');
const fs = require('fs');


// ENVIRONMENT
const EQFILEMOUNTPT = '/eqconfig';
// The equipment configuration is mounted into the adapters'
// docker container by docker swarm using the "configs" property
// in docker-compose.yml
const TASKSLOT = argv._[0];
//The task slot identifies the replica number of this adapter.
//If the task slot is 1, the adapter is supposed to ONLY handle
//the first tcp source in the eqconfig. This way, adapters can be
//scaled and distributed with number of data sources
const DESTPORT = 9000
// The destination port is identical for every adapter, because it
// is automatically mapped to an unused port in range specified in
// docker-compose.yml

var EQCONFIG = {};
var SRC = {};


// GLOBAL VARS
var destSocket = {};


// FUNCTIONS
function parseEqConfig()
{
    // Get equipment configuration file from file system
    // It has previously been mounted into the docker container by the "configs"
    // property in the docker-compose.yml

    let eqfile = fs.readFileSync(EQFILEMOUNTPT);
    EQCONFIG = JSON.parse(eqfile);
    // console.log(eqconfig)

    let bmcounter = 0
    EQCONFIG.datasources.forEach( (source, i) => {
        if(source.type == "benchmark" || source.type == "Benchmark" || source.type
            == "BENCHMARK") {
            bmcounter += 1;
            if(bmcounter == TASKSLOT) { //i.e. Only handle first tcp source if
                TASKSLOT=1
                SRC = source;
            }
        }
    });
}

function setupDestServers()
{
    destSocket= new WebSocketServer({   port: DESTPORT,
```

```
                                                host: "0.0.0.0",
                                                clientTracking: true });

    defineDestEvents();
}

function defineDestEvents()
{
    //WebSocket Server Listening Event
    destSocket.on('listening', () => {
        console.log("[" + SRC.name + "]\t" + "WebSocket Server is listening.");
    });

    //WebSocket Server Connection Event
    destSocket.on('connection', (ws, req) => {

        //Event: Connection established
        console.log("[" + SRC.name + "]\t" + "Successful Client Connection from: "
            +
                req.socket.remoteAddress);

        //Set up benchmark
        let time = Date.now()
        let timestamp = 0;
        let key = SRC.datapoints[0].key
        setInterval( () => {
            time = time + SRC.sampleinterval
            let packet = {
                value: 3 * Math.sin(0.125 * timestamp),
                utc: time,
                key: key
            }
            ws.send(JSON.stringify(packet));
            timestamp = timestamp + SRC.sampleinterval;
        }, SRC.sampleinterval);

        //Event: Client closes connection
        ws.on('close', () => {
            console.log("[" + SRC.name + "]\t" + "Connection with client at " +
                ws._socket.remoteAddress + " closed.");
        })

        //Event: Some error with client
        ws.on('error', () => {
            console.log("[" + SRC.name + "]\t" + "Error communicating with Client
                at" +
                ws._socket.remoteAddress);
        })
    });

    // WebSocket Server close event
    destSocket.on('close', () => {
        console.log("[" + SRC.name + "]\t" + "Websocket Server closed.")
    })
}

async function main()
{
    parseEqConfig();
    setupDestServers();
```

```
}


// ### SOFT SHUTDOWN ### //
process.on('SIGTERM', () => {
    console.log("Soft shutdown requested, bye...")
    // Soft close
    destSocket.clients.forEach((socket) => {
        socket.close();
    })
    // If a socket somehow stayed open after 5s, close it
    setTimeout(() => {
        destSocket.clients.forEach((socket) => {
          if ([socket.OPEN, socket.CLOSING].includes(socket.readyState)) {
            socket.terminate();
          }
        });
        process.exit(0)
      }, 5000);
})

main();
```

C.6. EQ Plugin Code

Listing 6: Equipment configuration plugin for OpenMCT, written in JavaScript.

```
// EQ plugin
// Author: Antonio Steiger
// Last Updated: 17.08.2022
// Description: Gets the current equipment configuration file and adds all
// corresponding objects to the openmct tree.
gi
const pluginName = "WARR_EQ"
let eqconfig = {};
let host = "";

export default function () {
    return function install(openmct) {

        //This install script HAS TO BE CODED SEQUENTIALLY. This means if you want one function to be called
        //after another, you have to call that function within the first one.

        console.log("[" + pluginName + "]" + " Installing...");
        console.log("[" + pluginName + "]" + " Adding data source folder...");
        addDataSourceFolder();

        console.log("[" + pluginName + "]" + " Getting equipment configuration");
        getEqConfig();
    }
}

function addDataSourceFolder()
{
    // Add "Data Sources" Root Folder
    openmct.objects.addRoot({
        namespace: 'datasources',
        key: 'datasources'
    });
    openmct.objects.addProvider('datasources', {
        get: function (identifier) {
            return Promise.resolve(
                {
                identifier: identifier,
                name: 'Data Sources',
                type: 'folder',
                location: 'ROOT'
            })
        }
    });
}

function getEqConfig()
{
    // Get equipment configuration file from webserver
    host = window.location.host;
    //console.log(host);
    $.getJSON('http://' + host + '/eq.json', (data) => {
        eqconfig = data;
        //If there are opcua data sources in the eqconfig, expand it dynamically
        //Else all objects in the eqconfig can already start being added to OpenMCT
        for( const source of eqconfig.datasources) {
```

```
            if (source.type == 'opcua') {
                console.log("[" + pluginName + "]" + " OPC UA data source detected...")
                addOpcUaSourcesToEq();
                break;
            }
            else {
                addDataSources();
            }
        }
    })
}

function addOpcUaSourcesToEq()
{
    let wsocket = new WebSocket('ws://localhost:9999'); //XX ip

    wsocket.onmessage = function (msg) {
        let hierarchy = JSON.parse(msg.data);
        //console.log(hierarchy);
        wsocket.close();
        eqconfig.datasources.forEach( source => {
            if(source.type == 'opcua') {
                source.datapoints = hierarchy[source.key];
            }
        })
        //console.log(eqconfig);
        console.log("[" + pluginName + "]" + " OPC UA hierarchies fetched...")
        addDataSources();
    }
}

function sleep(ms)
{
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}

function addDataSources()
{

    console.log("[" + pluginName + "]" + " Adding Data Sources...")
    // For each data source
    eqconfig.datasources.forEach( source => {
        // Register an object provider for its folder object
        openmct.objects.addProvider(source.key, {
            get: function (identifier) {
                return Promise.resolve(
                    {
                    identifier: identifier,
                    name: source.name,
                    type: 'folder',
                    location: 'datasources:datasources'
                })
            }
        });
    });

    //Register a composition provider for the data sources folder
```

```
    //This lets OpenMCT know that the data sources folder shall contain
    //a subfolder for each data source
    openmct.composition.addProvider({
        appliesTo: function (domainObject) {
            return domainObject.identifier.namespace === 'datasources' &&
                   domainObject.type === 'folder';
        },
        load: function (domainObject) {
            return Promise.resolve(
                eqconfig.datasources.map(function (s) {
                    return {
                        namespace: s.key,
                        key: s.key
                    };
                })
            )
        }
    });

    addDataPoints();
}

function addDataPoints()
{
    // Add data point type to openmct
    eqconfig.datasources.forEach(source => {
        openmct.types.addType(source.key + '_datapoint', {
            name: source.name + 'Data Point',
            description: 'A single' + source.name +
                'data point. Can represent a float, integer, string and more.',
            cssClass: 'icon-telemetry'
        });
    });


    //for each data point of every data source, register an object provider
    eqconfig.datasources.forEach(source => {
        source.datapoints.forEach(point => {
            //Differentiate between parent data points and child data points
            //child
            if(point.values) {
                openmct.objects.addProvider(point.key, {
                    get: function (identifier) {
                        return Promise.resolve(
                            {
                            identifier: identifier,
                            name: point.name,
                            type: source.key + '_datapoint',
                            telemetry: {
                                values: point.values
                            },
                            location: source.key + ':' + source.key
                        })
                    }
                });
            }
            //parent
            else {
                //Register folder object provider
                openmct.objects.addProvider(point.key, {
```

```
                    get: function (identifier) {
                        return Promise.resolve(
                            {
                            identifier: identifier,
                            name: point.name,
                            type: 'folder',
                            location: source.key + ':' + source.key
                        })
                    }
                });
                //For all childs of folder, register data point provider
                point.datapoints.forEach(subpoint => {
                    openmct.objects.addProvider(subpoint.key, {
                        get: function (identifier) {
                            return Promise.resolve(
                                {
                                identifier: identifier,
                                name: subpoint.name,
                                type: source.key + '_datapoint',
                                telemetry: {
                                    values: subpoint.values
                                },
                                location: point.key + ':' + point.key
                            })
                        }
                    });
                })
                //Register Composition provider for folder
                openmct.composition.addProvider({
                    appliesTo: function (domainObject) {
                        return domainObject.identifier.namespace === point.key &&
                            domainObject.type === 'folder';
                    },
                    load: function (domainObject) {
                        return Promise.resolve(
                            point.datapoints.map(function (sp) {
                                return {
                                    namespace: sp.key,
                                    key: sp.key
                                };
                            })
                        )
                    }
                });
            }
        })
    });

    //One hierarchy down, register
    //For every data point parent, register a composition provider

    //For every data source folder, register a composition provider
    eqconfig.datasources.forEach(source => {
        openmct.composition.addProvider({
            appliesTo: function (domainObject) {
                return domainObject.identifier.namespace === source.key &&
                    domainObject.type === 'folder';
            },
            load: function (domainObject) {
                return Promise.resolve(
```

```
                    source.datapoints.map(function (p) {
                        return {
                            namespace: p.key,
                            key: p.key
                        };
                    })
                )
            }
        });
    })

    addTelemetryProviders();
}

function addTelemetryProviders()
{
    var listener = {};
    var sockets = {};

    // For each Data Source
    eqconfig.datasources.forEach(source => {
        // Set up WebSocket
        //XX Below line needs a fix for docker swarm. Perhaps get one swarm IP from eqconfig instead of localhost
        sockets[source.key] = new WebSocket('ws://localhost:' + source.destport.toString());
        sockets[source.key].onmessage = function (msg) {
            let datapoint = JSON.parse(msg.data);
            if (listener[datapoint.key]) {
                listener[datapoint.key](datapoint);
            }
            if(datapoint.key == '_CONN_') {
                //CONN.notify(source.key, datapoint.value);
            }
        };
        // Add Telemetry Provider
        openmct.telemetry.addProvider({
            canProvideTelemetry(domainObject) {     //PFC_Provider can only provide telemetry to pfc objects
                return domainObject.type === source.key + '_datasource';
            },
            supportsSubscribe: function (domainObject) {
                return domainObject.type === source.key + '_datapoint';
            },
            subscribe: function (domainObject, callback) {
                listener[domainObject.identifier.key] = callback;
                return function unsubscribe() {
                    delete listener[domainObject.identifier.key];
                    delete sockets[domainObject.identifier.key];
                };
            }
        });
    })
}
```

C.7. Exemplary Docker Compose File

In the following example, note the interplay between replica numbers and port mappings across the different services. The Netdata service is deployed globally, i.e., on every swarm node, because it is a monitoring tool. Furthermore, note the usage of the "TASKID" environment variable in the entry point definition of each adapter service. Lastly, note the declaration of the equipment configuration file and how it is assigned to the different adapters.

Listing 7: An example of a docker-compose file automatically created from an equipment configuration by the start script.

```
version: '3.9'
services:
  web:
    image: 127.0.0.1:5000/exui_web
    build:
      context: ./webserver
      dockerfile: Dockerfile
    deploy:
      replicas: 1
    ports:
    - "8080:8080"
    tty: true
    configs:
    - eqconfig
  netdata:
    image: 127.0.0.1:5000/exui_netdata
    build:
      context: ./adapters/netdata
      dockerfile: Dockerfile
    ports:
    - "20001-20001:19999"
    cap_add:
    - SYS_PTRACE
    security_opt:
    - apparmor:unconfined
    volumes:
    - netdataconfig:/etc/netdata
    - netdatalib:/var/lib/netdata
    - netdatacache:/var/cache/netdata
    - /etc/passwd:/host/etc/passwd:ro
    - /etc/group:/host/etc/group:ro
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /etc/os-release:/host/etc/os-release:ro
    deploy:
      mode: global
  benchmark:
    image: 127.0.0.1:5000/exui_benchmark
    build:
      context: ./adapters/benchmark
      dockerfile: Dockerfile
    deploy:
      replicas: 4
    ports:
    - "9001-9004:9000"
    tty: true
    configs:
    - eqconfig
```

```
    entrypoint:
    - /bin/sh
    - -c
    - node adapter.js $$TASKID
    environment:
      TASKID: '{{.Task.Slot}}'
  tcp:
    image: 127.0.0.1:5000/exui_tcp
    build:
      context: ./adapters/tcp
      dockerfile: Dockerfile
    deploy:
      replicas: 2
    ports:
    - "9005-9006:9000"
    tty: true
    configs:
    - eqconfig
    entrypoint:
    - /bin/sh
    - -c
    - node adapter.js $$TASKID
    environment:
      TASKID: '{{.Task.Slot}}'
configs:
  eqconfig:
    file: /home/toni/Documents/exui/config/eq_example.json
volumes:
  netdataconfig: {}
  netdatalib: {}
  netdatacache: {}
```