ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ & ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

Διπλωματική Εργασία

Πλατφόρμα απομακρυσμένου ελέγχου επιστημονικών οργάνων και συλλογής δεδομένων

Φοιτητής: Αθανάσιος Γεωργίου

ΑΜ: 50106664

Επιβλέπων Καθηγητής

Δρ. Ηλίας Σταύρακας

Καθηγητής

ΑΘΗΝΑ-ΑΙΓΑΛΕΩ, ΟΚΤΩΒΡΙΟΣ 2020

UNIVERSITY OF WEST ATTICA

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

Diploma Thesis

Platform for remote control of scientific equipment and collection of measurement data

Student: Athanasios Georgiou

Registration Number: 50106664

Supervisor

Dr. Ilias Stavrakas

Professor

ATHENS-EGALEO, October 2020

ΔΗΛΩΣΗ ΠΕΡΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ ΚΑΙ ΛΟΓΟΚΛΟΠΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπόγραφα ότι η παρούσα εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα αποκλειστικά και ότι είμαι ο αποκλειστικός συγγραφέας του κειμένου της.
Η εργασία μου δεν προσβάλλει οποιασδήποτε μορφής δικαιώματα πνευματικής ιδιοκτησίας, προσωπικότητας ή προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής ή λογοκλοπής.
Κάθε βοήθεια που έλαβα για την ολοκλήρωση της εργασίας είναι αναγνωρισμένη και αναφέρεται λεπτομερώς στο κείμενό της. Ειδικότερα, έχω αναφέρει ευδιάκριτα μέσα στο κείμενο και με την κατάλληλη παραπομπή όλες τις πηγές δεδομένων, κώδικα προγραμματισμού Η/Υ, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών που χρησιμοποιήθηκαν, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης, και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Επιπλέον, όλες οι πηγές που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης κατά τα διεθνή πρότυπα.
Τέλος δηλώνω ενυπόγραφα ότι αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της είναι προϊόν λογοκλοπής.

Ημερομηνία 2020-10-19

Αθανάσιος Γεωργίου

The project detailed in this thesis was completed in cooperation with the Institute of Astronomy, Astrophysics, Space Applications & Remote Sensing of the National Observatory of Athens. I am especially thankful for the help of Dr. Vassilis Amoiridis and of the ReACT team, for without their resources and guidance this thesis could not have been completed.

I would also like to thank my supervisor Dr. Ilias Stavrakas for guiding me along the steps of this thesis and Dr. George Hloupis, who along with my supervisor, helped me in the early stages of my academic career.

Περίληψη

Η διπλωματική αυτή εργασία περιγράφει τον σχεδιασμό μίας πλατφόρμας για τον έλεγχο απομακρυσμένου επιστημονικού εξοπλισμού και την συλλογή των μετρήσεων, καθώς και την πρωτότυπη υλοποίηση μίας τέτοιας πλατφόρμας. Το σύστημα είναι μία διαδικτυακή πλατφόρμα που αποτελείται από το backend με μία βάση δεδομένων, μία ιστοσελίδα και ένα λογισμικό προς εγκατάσταση σε κάθε επιστημονικό όργανο που θα βρίσκεται υπό την επίβλεψη της πλατφόρμας. Οι κύριοι στόχοι είναι η συνεχής παρακολούθηση του απομακρυσμένου εξοπλισμού και η συλλογή των αποτελεσμάτων των μετρήσεων.

Η διαδικτυακή πλατφόρμα έχει σχεδιαστεί χρησιμοποιώντας τεχνολογίες αιχμής, εστιάζοντας ιδιαίτερα στην επεκτασιμότητα και την αξιοπιστία. Το υποσύστημα αποθήκευσης των δεδομένων βασίζεται σε μία βάση αντικειμένων (object store), με την δυνατότητα επέκτασης της χωρητικότητας σε εξαιρετικά μεγάλο όγκο. Λόγο των συνθηκών που συχνά επικρατούν σε απομακρυσμένες εγκαταστάσεις, όπως για παράδειγμα αδύναμη σύνδεση στο διαδίκτυο ή συχνές διακοπές της ηλεκτροδότησης, ιδιαίτερη προσοχή απαιτήθηκε ώστε το σύστημα να είναι σε θέση να λειτουργεί αδιάλειπτα. Τέλος, το λογισμικό ελέγχου των οργάνων είναι επεκτάσιμο με την χρήση add-ons ώστε να υποστηρίζεται μεγαλύτερο εύρος εξοπλισμού.

Σύμφωνα με αυτό τον σχεδιασμό, υλοποιήθηκε ένα πρωτότυπο της πλατφόρμας και εγκαταστάθηκε πιλοτικά στο Παρατηρητήριο Κλιματικής Αλλαγής των Αντικυθήρων, σε συνεργασία με το Εθνικό Αστεροσκοπείο Αθηνών. Η πλατφόρμα ανέλαβε την συνεχή παρακολούθηση τριών επιστημονικών οργάνων. Η πειραματική αυτή εφαρμογή της πλατφόρμας πραγματοποιήθηκε με επιτυχία, καθώς πρόσφερε λήψη των δεδομένων σε σχεδόν πραγματικό χρόνο, καθώς και ειδοποιήσεις για την κατάσταση του εξοπλισμού. Στο τέλος παρατίθενται επίσης μερικές ιδέες για περαιτέρω ανάπτυξη της πλατφόρμας.

Abstract

This thesis describes the design of a platform for the remote control of scientific equipment and collection of measurement data, alongside with an implementation of a running prototype. The platform is a web application composed by a backend with its data storage system and a database, a website, and an application (agent) installed on each instrument for monitoring. The main functions are monitoring of the remote installation's status and uptime and collection of measurement data.

The platform is designed based on state-of-the-art technologies, focusing on expandability and reliability. The data storage subsystem is based on an object store, capable of scaling to extremely large amounts of measurement data. In regard to reliability, it was deemed necessary to take extra precautions to guarantee continuous operation in order to overcome challenges related to infrastructure problems of remote installations, such as weak internet connection or power grid outages. Finally, the software that monitors the instruments themselves is cross-platform expandable with add-ons to maximize the range of equipment it can support.

A prototype of the designed platform is developed and experimentally deployed in the remote climate change observatory of Antikythera, in cooperation with the National Observatory of Athens. The platform was tasked with the monitoring and data collection for three instruments and had to deal with the observatories', at the time of writing, frequent power cuts and unstable internet connection. The experiment was deemed successful, providing the researchers real-time notifications about the station's status, and dutifully fetching new measurements in near real-time. Closing, some ideas are presented about future work that could expand the platform so it can assume more responsibilities of station management, such as execution of common commands.

1. Introduction

Trends in many industries and sciences tend to align with the global increase of data generation. A study by IDC's concluded that from 2020 and beyond, the digital universe and the world's data will double every two years [1]. This trend has both created new industries, such as the Internet of Things, but also shaken existing ones like data analytics [2]. New opportunities for research have emerged, since the development of new technologies can enable research tasks that were not feasible with the tools the community had.

The scientific community and funding agencies have tried to tackle the problem of exponential data growth in many ways. An increasing number of institutions and agencies require the use of data management plans (DMP), even if their effectiveness is unproven [3]. In the meantime, many initiatives are trying to tackle the problem of data analysis, sharing and collaboration. The European Open Science Cloud (EOSC) initiative, funded by the European Commission in 2016, aims to provide scientists tools, resources, and guidance in navigating today's data-driven research.

On the topic of atmospheric research, a domain that traditionally maintains remote research stations, the pan-European initiative ACTRIS (Aerosol, Clouds and Trace Gases Research Infrastructure, founded in 2014) attempts to consolidate data collection efforts of European entities. While standards are defined and a centralized data center for data publication is available, each research station must create their own tooling to automate data collection and conform to the standards. This leads to the topic of this thesis, the creation of a research station management tool.

1.1. Aim of this thesis

In this thesis, the aim is to design a complete solution for managing remote research stations and observatories. This is a multi-disciplinary undertaking, requiring knowledge of software engineering and web applications, automation of embedded systems and "big" data management. The term big data is not used prematurely as a buzzword, but to indicate that the developed software should be able to manage datasets beyond what can normally fit in one hard disk or one server.

Firstly, the needs are identified and translated into requirements. The platform is designed as a web application, due to the versality the ecosystem offers, along with how well a web-app fits

into current-day workflows. By following industry standard practices, the developed tool aims to save researcher's time by removing friction from the measurement process. When required, open-source libraries and services are exclusively used to supplement the developed application.

After carefully designing the platform and selecting which technologies should act as the foundation, a prototype is developed. This prototype tests both the general idea and concept, but the specific design as well.

1.2. Motivation

The spark for this thesis was the instigation of the climate change observatory of the National Observatory of Athens at the island of Antikythera, a project which has received funding by the European Investment Bank in 2020 [4]. The station already operates with 5 instruments [5] and conducts near-24/7 measurements of atmospheric parameters. During the spring of 2020, the station participated in the European Aerosol Research Lidar Network's (EARLINET) campaign with aim to quantify the effects of COVID-19 prevention methods in the atmosphere [6]. The continuous operation of the instruments lets the station "catch" events such as transfer of aerosols from the wildfires of Canada during the summer 2019 [7] and volcanic dust from Etna's activity during May of 2019 [8].

The requirement of 24/7 operation and data collection is a significant undertaking. Given that the number of instruments will rise considerably in the near future due to the received funding, management workload will also raise, thus it is essential to develop the tools to aid in the operation of the observatory. The case study described in chapter 6 is an application of the prototype platform at the remote observatory of Antikythera.

1.3. Past work

By browsing atmospheric research station websites, one can come to the conclusion that no standard solution for managing infrastructure exists. Each station uses in-house tools to automate measurements and almost no information is publicly available. An exception to this is the ARM Research Infrastructure by the United States' Department of Energy [9]. The website presents a list of available instruments, accompanied by activity logs and handbooks. Measurement data is also available for downloading, often in a standard format. There are no details on how the infrastructure is automated and managed internally, but it is implied that procedures are automated. ARM's platform serves as a working example that the goal of this thesis is achievable and makes its benefits apparent.

1.4. Document Structure

This thesis is organized as described below.

In Chapter 2 there is a literature survey in order to determine how stations and research networks across the globe are managing infrastructure and data collection.

Chapter 3 contains an overview of the platform. Key goals are identified, and requirements set, as such to be in a position to design and implement the system in later chapters.

In Chapter 4 a detailed design of the platform is presented. Decisions are made in regards of how to implement certain features and what kind of technology was used to do so. Flowcharts and sequence diagrams of important procedures are also included.

Chapter 5 details the experience of implementing the platform, as outlined in the previous chapter.

Chapter 6 contains a case study conducted at the remote observatory of Antikythera using the prototype implementation of the previous chapter.

Finally, Chapter 7 concludes the thesis with a short review of the work done. Some ideas about future work are also included.

1.5. Definitions

Platform: The software designed to aid in research station management

Instrument: A single unit of research equipment, managed by the platform

Measurement: A single unit of data produced by an Instrument.

2. Literature Survey

The aim of this thesis is to design and develop a platform to aid in the management of existing and future research stations and observatories. Hence, this chapter contains a survey of literature related to research infrastructure automation, big data storage and application development, in order to determine the state-of-the-art of such platforms in a global scale. In section 2.1 three sample research entities are described (ARM Climate Research Facility, AERONET, Stromboli seismic network), all having remote installations that are automatically operated. Section 2.2 contains details about systems that are tasked with big data management. Two cases are studied, one involving storage of a extremely large amount of files (photos) and the second regards storage of scientific datasets and HPC storage.

2.1. Research Infrastructure Automation

The ARM Climate Research Facility is a program started by the United States Department of Energy in 1994 to increase the knowledge on atmospheric radiation and cloud interactions. It consists of multiple permanent sites as well as mobile platforms. In a journal article, James Mather and Jimmy Voyles [10] discuss the ever-evolving structure of ARM and briefly touch the subject of the data infrastructure. Local data collection systems are installed at every site, tasked with collecting measurement data from the instruments, cataloging the files, and performing some preliminary automated analysis. These data files are uploaded to the central Data Management Facility where they are processed into a standardized format (usually NetCDF). Lastly, the collected data is made available to the scientific community within a few days of collection through online user interfaces (e.g. https://www.arm.gov/data). These procedures are followed both for permanent static sites and mobile facilities, including during scientific campaigns. Data from nonstandard instruments, such as guest instruments during campaigns, are separately managed by the External Data Center.

In addition to the automated delivery and analysis described above, ARM offers the so-called Value Added Products (VAP). VAPs are products of automated processing of the collected data with aim to refine parameter estimates, derive higher-order parameters by combining multiple instrument data and to consolidate parameters for easier analysis. VAPs are generally based on algorithms developed by the scientific community and then further refined for operational use. The ARM program has contributed significantly to the atmospheric science community and the automated data collection and management procedures play an important role in guaranteeing the program's success.

AERONET (AErosol RObotic NETwork) is a global network of ground-based sun photometers, initiated by NASA in 1998 to monitor atmospheric aerosols. A journal article by Holben et al. [11] describes the network structure at its initial form, offering insight at how research infrastructure automations have evolved over the years. Fully automated instruments are installed globally at remote sites and carry out multiple measurements per day without human intervention. Each photometer consists of a robot arm holding a sun-tracking spectrometer. By measuring the spectral properties of sun radiation, it can estimate the atmosphere's optical depth[1] and by subtracting effects of known gases, aerosol optical depth (AOD). At the end of each measurement, the sensor head is automatically placed facing downwards to protect it from rain and debris. Data is transmitted from the photometer's memory either directly through an internet connection or through geostationary satellites and corresponding ground stations. Satellite communications offer extensive coverage of the Earth's surface, especially important considering the remote location of many stations and the availability of internet connectivity in 1998.

The data transmitted from the remote stations is processed using a UNIX-based system. The received instrument files include, besides the photometry measurements, timestamps, temperature, battery voltage, time of transmission and other metadata. In case a parameter is outside the recommended operational range, an automated email is sent by the processing system to the station's manager. The UNIX system is also programmed to automatically perform a variety of corrections, calibrations and data analysis procedures that stem of a series of published algorithms. Both the original data and the generated products are available at AERONET's online archive for download.

At the time of writing, AERONET still operates with more than 1000 remote sites registered [12]. Together with guides on photometer operations, at the network's website, software is available that connects to the instrument with a serial port (RS-232) and automatically uploads the data to AERONET through the internet.

Sensor arrays installed at remote locations are commonly used in seismology, an example being the seismic network at the Stromboli volcanic island [13] . W. D. Cesare et al. describe the network, which consists of 13 broadband seismic stations located across the island. Data is retrieved from the stations in two ways, either through a UHF radio link in case the station has a direct line-of-sight to one of the three receiver sections, or through a WiFi mesh network that

[1] Optical depth is a measure of the ratio of incident to transmitted radiation through the atmosphere (or another material)

covers the island. Three collection hubs exist on the island, tasked with this data retrieval process. At each of them, a high-availability system is configured to oversee data acquisition. If a computer fails, another takes control to continue data acquisition. From these collection hubs, the data is finally transmitted off-island.

While in this chapter a couple of research networks based on remote installations are explored, a common theme is that station management, automations and data retrieval are means to meet the ends of research work and thus, little to no details are available. It is clear that many of these networks, and probably others not mentioned in this chapter, share many common characteristics and are burdened by the same issues, however no literature exists detailing how to solve these problems. Every network and station approach these things differently, using in-house software.

2.2. Big data management

By the term “big data”, in the context of this thesis, we refer to large amounts of semi-unstructured data. This data occurs from hosting a variety of instruments, each measuring different parameters, in the same remote station. While the series of data produced by one instrument is structured, when dealing with several instruments, the files stop being homogenous. As such, any data archival solution should be designed to accommodate many different formats. It is not important that a system specifically supports “unstructured data” but a system that relies on a specific structure for storage would be unable to handle this kind of workload. An kind of such a storage systems are the commonly used filesystems, which store data as files, regardless of content.

The popular social network Facebook faces the challenge of storing massive amounts of user uploaded photographs. In a 2009 journal article by Doug Beaver et al. goes over the original design based on traditional filesystems, how it failed to handle the workload, and the applied alternative [14]. In the original system, each photograph is stored in its own file in commercial NAS appliances. All the filesystems exported by the NAS appliances are mounting over NFS on a set of servers called “Photo Stores”. When a user’s browser requests a photo, the request is routed to one of the Photo Store servers, which in turn derives where the photo file is stored (i.e. which NAS), reads the file and returns it. To read the photo file from a filesystem, each NAS must perform at least 3 disk operations. The first reads the directory metadata, the second

reads the inode[2] and finally, the third reads the file contents. These extra operations add latency, especially at the scale of Facebook's application.

The system designed to replace this filesystem-based approach is called Haystack. The Haystack architecture consists of the Directory and the Store. The Store is comprised by several servers with persistent storage. Instead of storing each photo in a separate file, Haystack relies

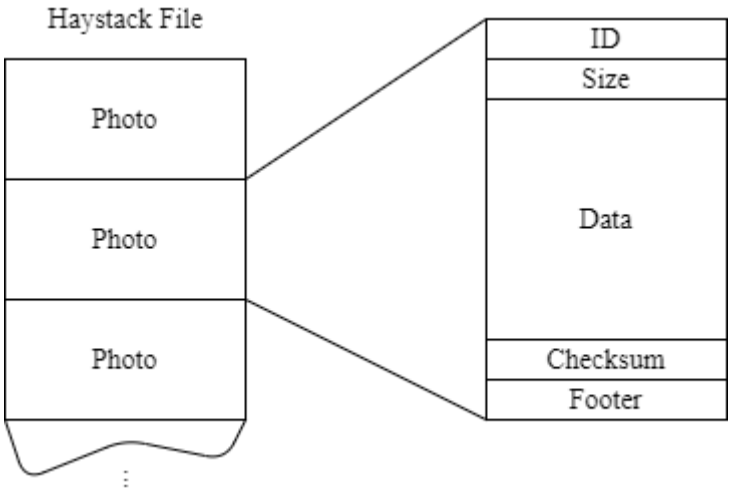

Figure 1: Simplified layout of a Haystack Store file.

on large files, called Haystack Files, that contain thousands of photos. A simplified layout of such a file is sketched in Figure 1. Each photo is assigned a unique ID and stored inside the file. Such Haystack files are treated as "volumes" and each of the is replicated in several physical servers for resiliency. The system creates an index containing the location (offset from start) of each photo in relation to the Haystack file's beginning. Using this approach, the server can read the photo using only one disks operation by using the in-memory index. In case the index is lost, it can easily be created anew by sequentially reading the whole Haystack File and noting where each photo begins. The Haystack Directory contains a mapping of photographs to their Haystack volume (i.e. Haystack file) and ID within that file.

This approach offered greater performance in comparison to using filesystems by avoiding the extraneous disk operations. It should be noted that while there are some operations that are difficult to perform with this system (e.g. photo deletion), they are infrequent and can be worked around without significant performance losses. While this workload might seem

[2] inodes (index nodes) are data structures in UNIX-style filesystems that describe files or directories. It holds related attributes, such as owner, permissions, and timestamps, as well as the list of disk blocks that make up the file.

completely different to storing instrument data, there are striking similarities that are discussed in Chapter 3.

Zenodo is an OpenAIRE project operated by CERN, providing an open repository for publications and scientific datasets [15]. The project relies on Invenio for its storage needs [16], an open-source software for abstracting storage and adding object-storage features, such as file versioning [17]. Invenio uses relational databases for metadata storage and stores raw data on a variety of supported backends, including filesystems, another object storage, etc. User-submitted datasets are assigned a Digital Object Identified (DOI) and are preserved indefinitely. Even if a new version is uploaded, the old one is kept available. Users can download datasets directly from Zenodo's website.

Besides Zenodo and Invenio, for other use-cases CERN employees CEPH [18], an open-source object storage system [19]. CEPH is used for its high scalability and resiliency, object storage features and API, as well as its capability for provide a POSIX-style filesystem. This means already-written applications that expect files and directories on disks can use the object storage without any changes. CEPH has been tasked with serving ~100 HPC nodes approximately 1PB of storage since 2016 and performance has been adequate.

3. Overview of the Platform

This chapter presents a detailed overview of the Platform, identifying key components, goals and requirements, so to provide us with the necessary information to design the platform during the next section.

3.1. Goals and Requirements

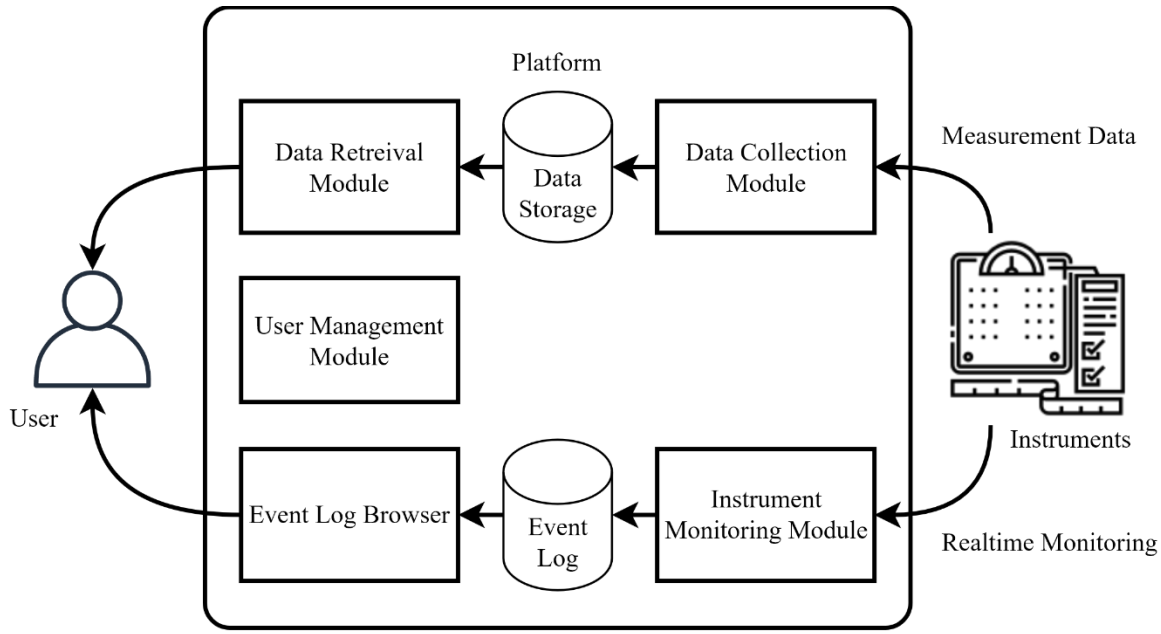

Figure 2: Simplified overview of the platform

As presented in 1.3, there is no standard for research infrastructure management software. The goal of this thesis is to fill this gap by designing a platform to be a one-stop-shop for remote research infrastructure management. This entails two main tasks, the first being measurement retrieval and data management, while the second involves infrastructure monitoring. Figure 2 presents the platform in a very simplified way. The design of the platform should take the physical limitations of remote infrastructure into account, such as unstable internet connectivity and/or power grid. This translates to increased focus on resiliency and redundancy. The next subsections will delve further into the requirements.

The platform can be divided into two main components. The first is the backend, the “heart” of the system. This component will be hosted in a datacenter, away from the research infrastructure it monitors. The backend consists of the web service, the orchestrator of all platform operations, and other supporting services such as message brokers and databases. On the other

hand, the component that monitors the instruments and oversees data retrieval will be installed as close to the instruments as possible. We will refer to this second component as the “Agent”. This compartmentalization increases reliability and simplifies design.

3.2. Instrument Support

To match the needs of modern research stations, the platform should be accommodating to the needs of a wide variety of instruments. In order to satisfy this requirement, basic functionality such as uptime monitoring and data collection should be instrument-agnostic, while keeping the option to add specialized features targeting specific instrumentation. There should be no upper limit to the number of supported instruments.

Often, instruments are operated using bundled (and sometimes proprietary) software. Attempting to replace this software would be a monumental task and, even if the reverse engineering efforts proved fruitful, would cause myriads of problems (e.g. loss of measurement guarantees, difficult maintenance, development time and cost). To avoid this, the platform should coexist with the bundled software, automating its input where possible and if it is required and retrieving the measurement results. Multiple methods of data ingress should be considered to maximize the range of supported instrumentation.

It is necessary to assume that each instrument will be, in one way or another, connected to a computer. This assumption enables us to stay out of the embedded development domain and create a general solution that runs on any computer connected to an instrument. It is improbable that the platform never encounters an instrument that functions entirely on its own, without a computer, but nonetheless a computer will always be used to collect the data. That computer will also run the platform’s software, intercepting events, and data. Directly interfacing the embedded platforms is, however, unavoidable for deploying monitoring for the instruments.

3.2.1. Monitoring

The platform’s monitoring features are key in reducing the workload associated with remote instrument management. In a basic level, this directly corresponds to monitoring the computer connected to the instrument. Essential metrics are resource consumption (processing and storage), network availability and reboot detection. This basic functionality will be available for all instruments, but specialized add-ons can be added to the platform to monitor specific instrument parameters. For example, it could measure the operating temperature of the measurement environment. In case the platform detects abnormalities in operation, the Principal Investigators (PI) associated with the instrument should be immediately notified to act.

The agent will be tasked to conduct the monitoring task. A persistent connection to the backend is required to relay all information related to the instruments. In case this connection is severed, the agent will temporarily store the information in a local database and attempt transmission as soon as connectivity is restored. The backend, on the other hand, will monitor this connection and keep track of any issues. If an instrument remains disconnected for a long period of time, it should be assumed that it, or the infrastructure supporting it, has failed in some way.

3.2.2. Event Log

Any events recorded in accordance with section 3.2.1 are stored in the *Event Log*. This shall function as an advanced version of the "logbook", a physical or virtual notebook used by researchers to track important events for an instrument. Events stored in the log can be retrieved by the users, optionally filtering by metadata (e.g. timestamp, severity, event type). That platform can process past events to create visualization, such as plotting the instrument's active time (or downtime) over a specified period. Users of the platform should be able to add entries to the logbook manually, as to keep a history of events that cannot be automatically detected, such as physical maintenance or measurement parameter adjustments.

Retention policies can be instilled so that past events of low significance are removed after some time. This reduces database burden while still keeping important information intact, while also making browsing easier by removing "noise".

In order to receive notifications of important events, the platform's users should be allowed to "subscribe" to certain events or instruments. A subscription should be able to target a specific instrument, events of a minimum severity, an event type, or combinations of these. At time of creation for a new event, all subscriptions should be checked, and the users should be notified appropriately.

3.2.3. Measurement Retrieval

The platform's agent works in tandem with the instrument's software to automatically retrieve the measurements. The instrument's software should be configured to automatically store measurement files in a directory and the agent can monitor that directory for changes, so when a new file is deposited by the instrument, it can be sent to the database.

New measurements are added to a local database by the agent as soon as they are created. This database is unique to each agent and keeps track of measurement transmission. If a connection to the platform is available, the local database is synchronized with the platform's main

database. Any failed transmissions will be repeated until the platform can prove that it received the correct file (e.g. using hash values). After successful transmission of a file, it can be removed from local storage.

3.3. Data Management

In order to support scientific work, the platform is tasked with the automatic retrieval and archival of measurement data. The platform does not process the data in any way, it only receives the files from the station and indexes them. However, it is important to follow good data handling and archival practices to help facilitate the use of the measurements.

3.3.1. FAIR

The FAIR principles [20], originally published in 2016, intend to set some universal standards in data management. Since modern research produces large amounts of data, it is crucial that this data is effectively managed. In this notion, the platform should adhere to the FAIR principles as much as possible, if applicable within the operating goals. FAIR stands for *Findable*, *Accessible*, *Interoperable* and *Reusable*.

The first pillar of FAIR is for the data to be *Findable*. Data files should be assigned a unique and persistent identifier to distinguish it from other files. Rich metadata should accompany the file, clearly and explicitly including the file's origin, type, contents and of course, the unique identifier. The measurement is always accompanied by metadata, recording the date of measurement, the instrument of origin and its status at that time. Any additional metadata provided by the instrument should be recorded.

The date should also be *Accessible* and thus be retrievable by their identifier with the use of a standardized protocol. The platform provides access to the data both with human and machine interfaces, using open protocols. User authentication is possible both interactively (i.e. by a human) and automatically (through a computer program) to allow mass file transfers. If a measurement is deleted, its metadata will be preserved as a persistent point of reference. Data discoverability is increased by allowing potential users to browse the database, increasing the chance the collected data gets discovered by potential users.

Interoperability demands that data is stored in a universally applicable and accessible format. The platform has no processing capabilities and thus cannot transform the data between formats. Best effort shall be made by the platform's users to only submit measurements in an

appropriate form. As aforementioned, computer interfaces will be available to aid in automatic retrieval of data from the platform.

Finally, *Reusability* is guaranteed by meeting domain-relevant standards, using open formats, and providing clear data usage licenses. None of these falls inside the scope of the platform and cannot be explicitly met, but the platform shall support the related metadata.

3.3.2. Inhomogeneity

To support as many instrument and measurement types as possible, the platform should use a database system capable of storing vast amounts of heterogeneous data. The platform shall treat all types of data as large binary objects (a commonly used term is BLOB) and make no assumptions about their size or format, even within the measurements of a single instrument. Transformations on the measurement data are not allowed, as they must be preserved in the original form. This excludes databases and storage systems that rely on specific structural features of the data to operate (e.g. relational databases).

For instruments that measure continuously and produce a timeseries in real-time (e.g. data loggers), as opposed to the creation of measurement batches, the data shall be combined into single files (e.g. hourly, daily) before submission to the platform. Native management of timeseries would require a secondary storage system and is considered out-of-scope for this project.

3.3.3. Archival & Expandability

The platform's data storage is an archive of the research station past and current activity. Modern computational resources and techniques make it feasible to process years of measurements, and thus it is valuable to keep as much data as possible. In addition, research stations are projects with high longevity, it is very probable that the data storage needs exceed the available resources from the initial investment.

The storage workflow consists of regular deposits of new data and mass retrievals of past measurements by users. Data files are always retrieved in their entirety, meaning that partial reads are unnecessary. Additionally, files are never modified and are rarely deleted. While the example of photo storage presented in section 2.2 might have seemed very different from the purpose of this platform, the workload as far as storage is concerned is very similar. Photos were stored or read in their entirety and modifications or deletions never happened.

To meet the aforesaid requirements, the data storage system should be easily expandable, without relying on specialized server hardware that might not be available down the road. High

reliability is also especially important, as downtime of the storage system causes a backlog of measurements waiting to be transmitted from the remote research station, a task that becomes increasingly more difficult the longer the system remains unavailable. The storage system design should also incorporate a level of resiliency against hardware errors, both to guard from data loss and to reduce potential downtime. Operations that are not required (partial reads, modifications, deletions) can be omitted or be unoptimized.

3.3.4. Indexing

It is essential that the data archive of the platform is easy to use and browse. The platform users should be able to browse the data based on (at least) the following parameters:

- Source Instrument
- Date of measurement
- Parameter

The implementation should use appropriate database indexing techniques to facilitate data browsing and retrieval. Batch retrieval of multiple files should only be constrained by the network speed and not by any kind of database querying.

3.4. Single Page Application

A single page application (SPA) is a website that after the initial page load, interacts with the user by replacing parts of the page using information dynamically requested from the web server [21]. This comes in contrast with traditional web sites, where each interaction generally causes a full-page reload. SPAs aim to be something closer to a native computer program than a website, by offering rich user interfaces and faster transitions.

The human interface for the platform is such a web application. This web application authenticates users and enables access to the instrument's status, the event log, and the measurements database. The SPA pattern is used to offer better user experience in pages that display complex data, such as measurements (with search/filtering functionality). Additionally, since an SPA requires an API to function and access resources, the same API can be reused to make the platform scriptable.

3.5. User Management

As with any web application, basic user management features are available. Users are authenticated using a password. The platform administrator can allow a user to access or modify an instrument's information.

3.6. Underlying platform support

The platform does not depend on specialized hardware features or underlying software stack. The backend can be hosted on any modern Linux server environment, while the agent runs on modern versions of the Windows and Linux operating systems. It is probable that an older version of these operating systems is encountered, especially on the agent's side. These older versions will be supported on best-effort basis. There is no technical modifications preventing the backend from running on other operating systems but no work will be done to ensure compatibility.

4. Platform Architecture

The goal pursued by the platform is the management of remote research infrastructure. In this chapter, the architecture of the platform designed to achieve this goal is presented. First, an overview of the platform is discussed, introducing the main components that comprise it. The obvious categorization is to divide the components between the backend and the instrument agent, as shown in Figure 3; however, it is simpler to treat related components as one entity, independently of their physical location.

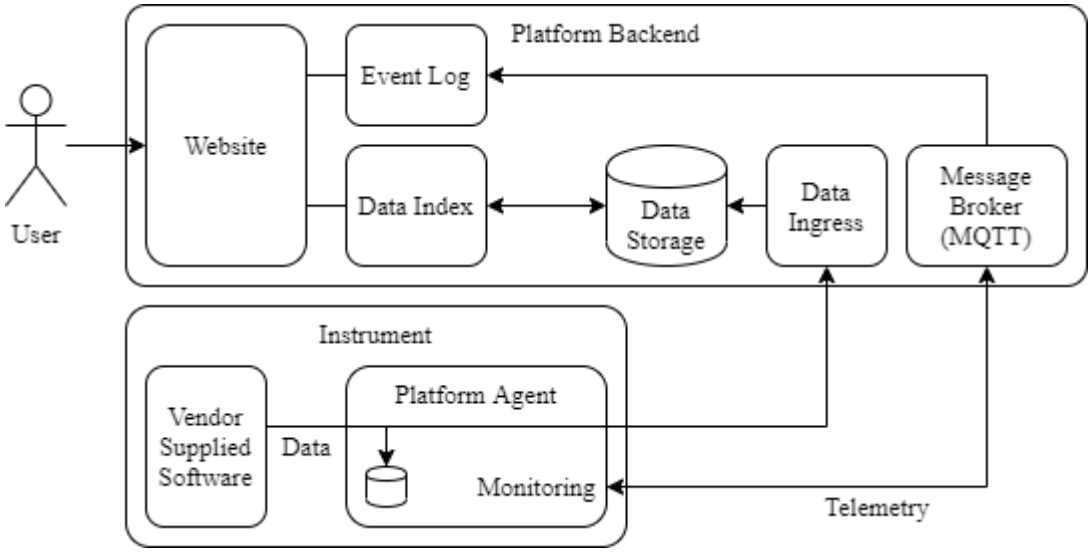

Figure 3: Overview of the platform's architecture, showing the main components

Thus, the main organizational element we use is the *Module*. Modules oversee one domain of functions and features and provide interfaces to interact with other modules. Modules can contain *Services*, a component tasked with one single job, *Interfaces* for communication and *External Components* which refer to databases or other software packages.

This chapter is closed by a description of the technologies picked to implement this design. The application meshes modern cloud application development with control of embedded devices.

4.1. Overview

As aforementioned in the chapter introduction, the platform consists of two main components: the backend and the agent. The platform’s backend is a web application tasked with managing the data storage, keeping the event log, communicating with the instruments, and providing machine and user interfaces for these features. It is hosted on a datacenter, possibly far away from the research station it manages. This simplifies design because the backend is not

subjected to the infrastructure issues that plague remote facilities, such as problematic connectivity and power. The agent, on the other hand, runs on the physical machine that controls each instrument. Its design is focused on high reliability and resiliency.

By breaking down the platform goals, the following modules are clearly defined:

- Data Storage Module: Tasked with storing and indexing of data.
- Instruments Module: Oversees instrument monitoring and orchestrates data retrieval. The agent is part of this module.
- Users Module: Handles authentication and authorization of users.

Each of the above modules will be discussed in the following sections. All these components have needs for operational data storage, such as user accounts, event log entries, etc. This data is not to be confused with measurements. To meet the application needs a relational database is used. Since the needs from this database are typical of a web application, it is not discussed extensively, and it is assumed to be accessible from every module and service (except the agent, due to the network barrier).

4.2. Data Storage Module

To meet the requirements set in chapter 3.3, the data storage system is designed on top of an Object Storage system. An overview of the module's components is sketched in Figure 4.

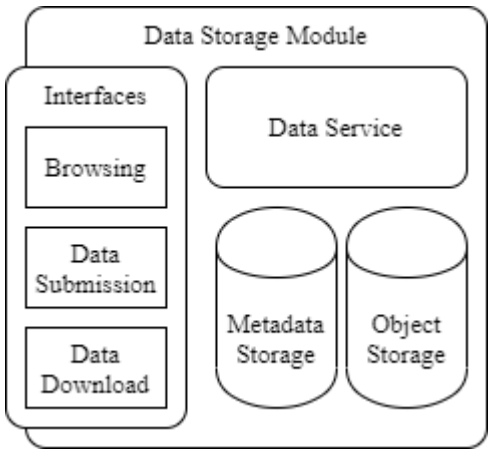

Figure 4: Overview of the Data Storage Module

4.2.1. Object Storage System

Object Storage systems existed as a concept since the early 1990's [22] and had already gained significant traction in the industry by 2007 [23]. As of now, they have become pervasive as

they can grow to the extreme scale required by modern web applications. In June of 2012, Amazon announced that its Object Storage product, called S3, has stored one trillion (10^{12}) objects [24].

Object storage is an abstraction layer above traditional block-based storage devices. Instead of storing files directly on the storage medium and representing them as an array of indices pointing to blocks, like we do with filesystems, object storage systems present themselves as a collection of objects. The application can no longer directly access files from the disk but must instead request them through the object storage's API. It is possible to encounter an object storage system that works on top of a filesystem, instead of raw disks, but the concepts remain the same (remember Haystack from section 2.2).

If the platform's data were stored using a traditional filesystem-based solution, problems would surface as the database grew. Since a single hard disk would be filled relatively quickly, a way to expand capacity would have to be used. A first idea is to rely on hardware or software RAID, to create resilient volumes by combining hard drives. This approach is very proven and works up to a specific scale but falls short when more storage than what a single server can support is required. Additionally, pushing the limits by creating an exceptionally large server can be expensive, requiring specialized controllers and reduces resiliency by making the storage server a single point of failure. Server vendors offer storage appliances (e.g. SANs) that can be expanded but they are expensive and can cause vendor lock-in.

An alternative solution would be to employ many server nodes and manually keep track of file locations. Files could be stored on multiple nodes for resilience, while the individual nodes could still use RAID to guard against hardware failures. Designing such a system is a massive undertaking and comes close to re-implementing an object storage system, thus is out of scope for this thesis.

By designing the platform on top of an object storage system, the issues mentioned above are eliminated. The storage system consists of several server nodes (at least one), which in turn contain an amount of storage (hard disks). The storage system automatically distributes load throughout the cluster (nodes and disks) by storing objects where it is deemed appropriate. Erasure coding [25] can be used to ensure resilience by storing pieces of an object across nodes and increase read performance by parallelizing disk access. This can additionally eliminate the need of RAID, simplifying the administration of the system and avoiding stressful RAID rebuilds.

Objects in such systems are uniquely identified by a variable called object ID. Given an object ID, the storage system looks up the appropriate server node (or nodes, in case the object is stored across a number of servers) to retrieve the item and return it to the client. Objects can also be accompanied by rich metadata. Since object storage systems are often used to store unstructured data, this being the case for this project as well, the addition of metadata can make the storage system self-explanatory. Some object storage systems rely on metadata servers to keep track of object and node relations, while others encode all the necessary information inside the ID.

The application, in our case being the platform's backend, can access the files through the storage system's API. This is a valuable abstraction layer because it allows for the storage system to be swapped with minimal changes to the application. Furthermore, it makes accessing the storage system over the network trivial, an essential feature for scaling the platform past one server node. A common API style encountered is called "S3-style", getting its name by Amazon's object storage product (S3), which pioneered use of object storage systems. Such S3-compatible systems should be able to replace S3 with minimal incompatibilities. A side effect of this API standardization is that the platform could also be adapted to use cloud storage, or even be hosted entirely in the cloud.

There are many implementations of object storage systems, both open-source and proprietary. Some of them were discussed in section 2.2. Given the extensive popularity of the last years, there is no shortage of choices in storage. Some commercial cloud solutions are Amazon S3 and Microsoft's Azure Storage. For local storage, there are both appliance offerings (such as Dell's ECS [26]), promising ease of use and management simplicity and software offerings (such as CEPH or MinIO). Such software offerings are often referred to as "Software-defined storage", since such systems support more kinds of storage than just objects (e.g. block storage through iSCSI [27]), therefore covering a wide range of storage needs without special hardware.

4.2.2. Data Service

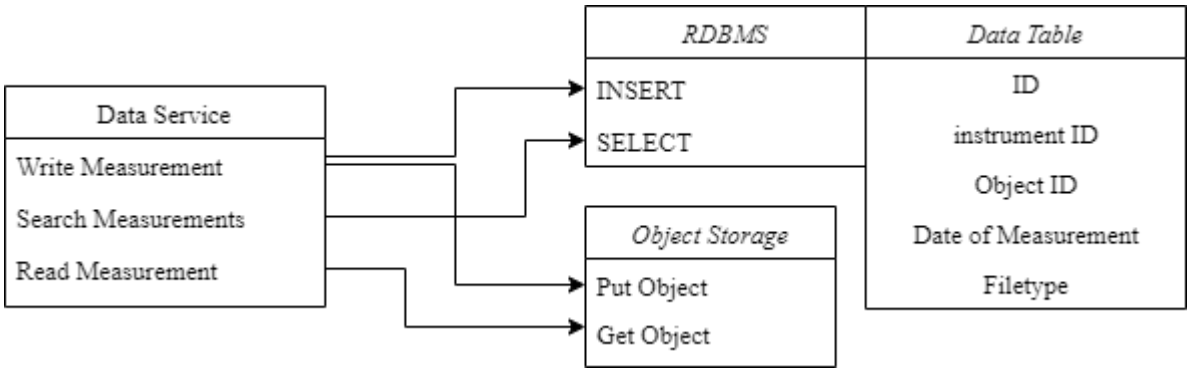

Figure 5: UML diagram showing Data Service and its dependencies

Through the Data Service, other modules can use the Object Storage system and access or store measurement data. This service acts as a proxy for the storage, abstracting objects into measurements and vice versa. All communications with the storage subsystem go through the network using an S3-style API and, therefore, the platform is independent of the individual storage system implementation. This abstraction layer is a simple interface for accessing the measurements, as shown in Figure 5. In reality, this is a very thin layer, merely keeping track of the many identification keys (instruments, measurements, objects) and the relations between them As mentioned in earlier chapters, file modification and deletion is not required and so such functions are missing from the data service.

4.3. Instrument module

The instrument module manages all actions and procedures related to instruments. This consists of managing the agents, handling status updates and monitoring, keeping the event log and orchestrating data retrieval. The components making up this module are show in Figure 6 and are analyzed in the following subsections.

4.3.1. Agent

The instrument agent is an application meant to be executed as “close” to the instrument as possible. This can translate to running directly on the instrument’s computer, or on another computer nearby. As discussed in 3.2, each agent is responsible to monitor the instrument’s status, detect any notable events and submit new measurements to the backend.

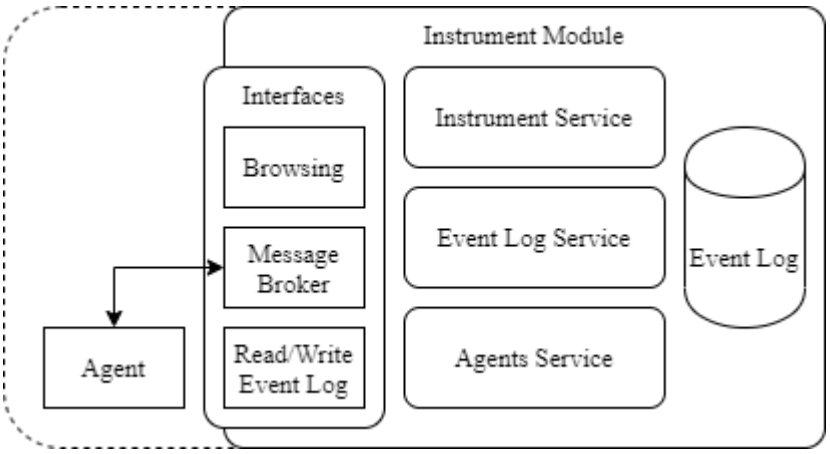

Figure 6: Overall architecture of the Instrument Module. The agent, while part of the module, is displayed as an external component to signify that it runs on a different machine.

To meet the first requirement, the agent runs a timer that periodically triggers the monitoring routines. First, general system attributes are checked, such as connectivity and computational resource utilization. Continuing, the agent runs any instrument-specific code available to acquire specialized metrics (e.g. operating temperature). Instrument-specific metrics are measured through the use of add-ons that only apply on a specific system.

Since the addons responsible for measuring specialized metrics are designed for the agent, the agent is also responsible for interpreting these results. In case of a detected anomaly, the platform is notified as soon as possible, to subscribed users can be notified. Regardless if the metrics are a cause of concern or not, they are sent to the platform and are made available through the website.

It is common wisdom in web application to distrust the client since it is almost always out of your control. Based on this, the decision to move all decision taking regarding potentially dangerous operations on the agent might seem flawed, but for the purposes of the platform, agents are trusted clients. The security model of this prototype does not account for any potential bad behavior of the agents, since they run on machines managed by the platform's administrator. Most of the security effort is focused in keeping unauthorized clients from interacting with the platform in the first place.

In any case, the collected metrics and events are submitted to the platform as soon as possible. In case of failure, they are stored and retransmitted as soon as the connection is restored. To aid in this, the information is timestamped at the time of collection and not at submission. A flowchart of the procedure is shown in Figure 7. Events, too, are stored and transmitted later if there is a problem with the connection.

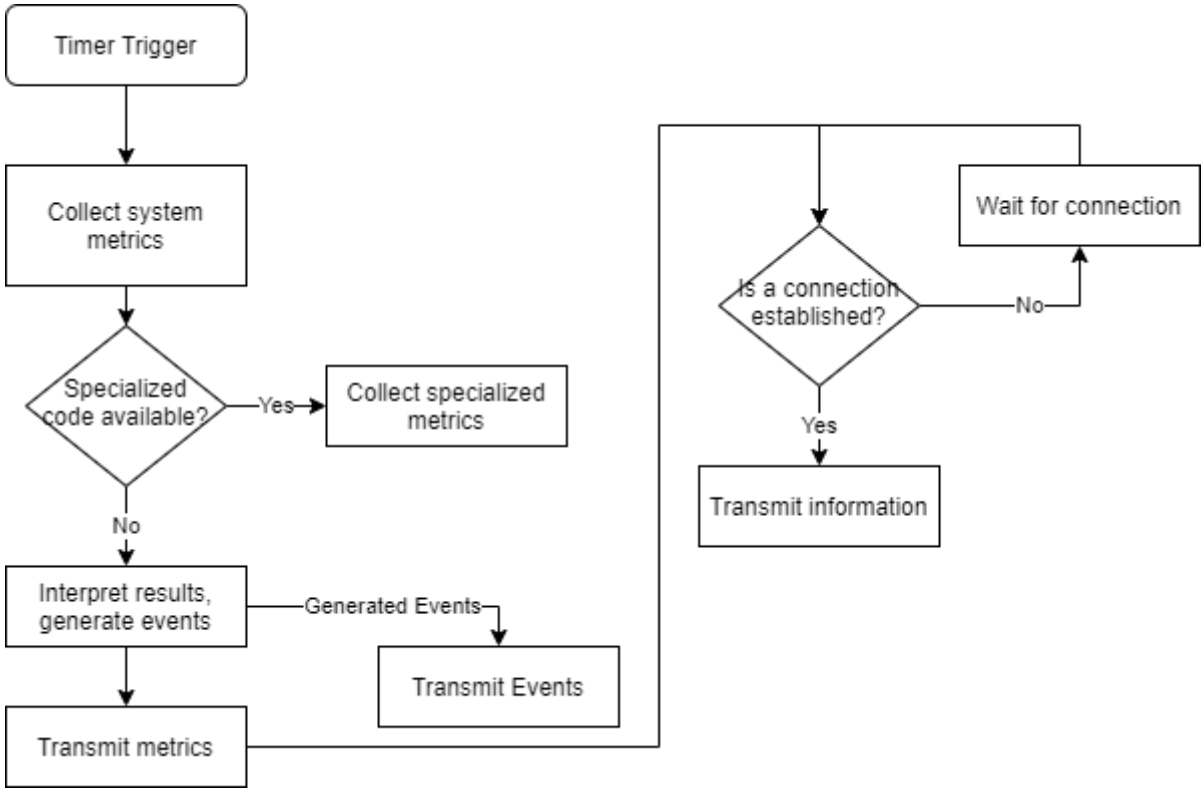

Figure 7: Flowchart of the message transmission procedure from the agent to the platform.

The agent communicates with the platform using the *Message Queuing Telemetry Protocol* (MQTT). MQTT is a message transport protocol based on the publish/subscribe pattern, built on top of TCP, utilizing a *Message Broker* to bridge the gap between publishers and subscribers [28]. Clients are connected to a Broker and are given the option to subscribe to certain *topics*. Any messages published to these topics will be relayed back to the subscribed clients. A persistent connection to the broker is always required. MQTT supports different levels of transmission guarantees, called *Quality of Service* level (QoS). When using the lowest, level 0, the communication relies only on TCP to guarantee delivery. Level 1 requires an acknowledgement by the broker to verify that a packet is received, while Level 2 uses a handshake procedure to guarantee single delivery of the message without duplicates. More details on how these QoS levels are used are available in section 4.4.

Interruptions to the persistent connection between the agent and the broker are used to monitor network connectivity issues. The platform's backend is subscribed to receive updates from all instruments and each instrument publishes information to a unique topic. QoS Level 1 is used to verify that the broker has received any transmitted messages and remove them from the transmission queue.

MQTT is focused on transport of small messages, the maximum message size being 256 MB. To transmit measurement data, files that are possibly larger than 256 MB, the agent uses HTTP after establishing a transaction through MQTT. A request is sent to the platform, containing information related to the transfer, specifically the filetype, its size and the hash value. The platform then returns a unique measurement ID and a secret key, required to submit the data through HTTP. This is further described in section 0.

MQTT support client authentication through a pair of values, akin to username and password. Therefore, to authenticate the agents, the platform assigns each a universally unique identifier (UUID) and issues secret keys. The UUID acts as the username, while the key acts as the password. Both variables are generated upon agent setup and stored in its config file. All messages sent by the agent must include both, otherwise it is rejected by the platform. The generation of these keys, as well as their verification, is handled by the Agents Service, which is sketched in Figure 8. Finally, to ensure integrity during transmission, the messages are encrypted using Transport Layer Security (TLS), as defined in the MQTT standard.

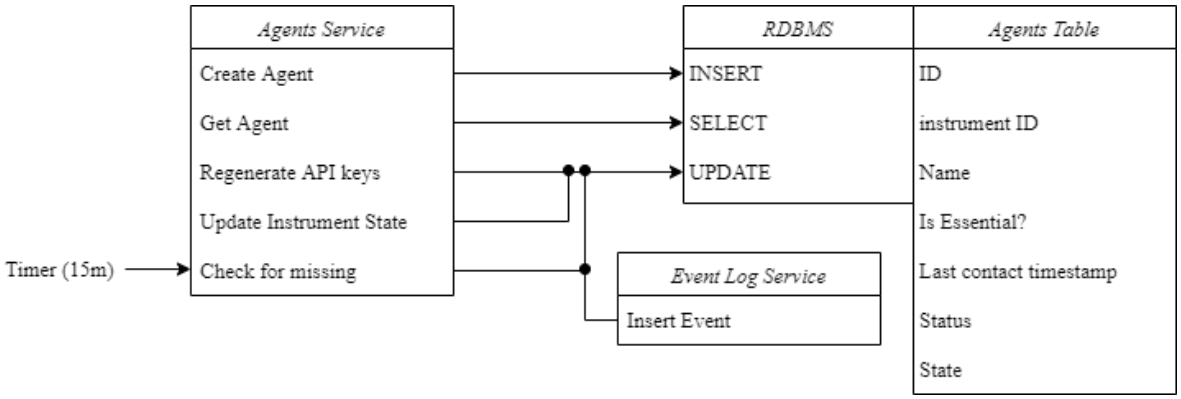

Figure 8: UML diagram for Agents Service and its dependencies.

The agent may send one of the following types of messages using MQTT:

- Status Update: Contains the current state of the instrument (utilization, uptime, specialized metrics)
- Event: Used to add a new event to the Event Log
- Begin Transaction: Requests the start of a data transfer transaction

Similarly, the platform may send messages to the agents. Only one message type is identified, the response to “Begin Transaction” that contains the measurement ID and the secret key.

4.3.2. Realtime monitoring

In section 4.3.1 we discussed how the agent oversees monitoring on the remote side. The platform, at its end, is always ready to accept MQTT messages from the remote instruments.

Receiving any message is considered a “sign of life” by the instrument. If 15 minutes have passed from the last communication, an instrument is declared as *Missing* and the users are notified. This is useful since issues with the remote infrastructure can interrupt operation without leaving a chance to communicate (e.g. power cuts). This is also added to the event log.

Each instrument is represented as a row in a table hosted by the relational database. A special column is used to store the instrument’s state. This is required because each different instrument uses different fields and data types to represent its current state. Upon receiving a status update message, the old state is overwritten with the new one.

The above operations are also managed by the Agents Service described in the previous chapter. This is because the platform allows more than one agent to be associated with each instrument in case a complicated system is managed by more than one computer.

4.3.3. Event log

The Event Log is stored in the relational database. Each event is marked by a severity level marking its significance (Table 1). Additionally, each event is categorized by a type and contains a free-form body. By checking the event type, a user (human or computer) can know what information to expect in the body field. Some event types are defined by the platform as universal (Table 2), while more can be defined for instrument-specific events.

Users are given the option to subscribe to be notified when new events are logged. To avoid the communications becoming overwhelming, the subscription can be instrument specific and target a minimum severity level or specific event types. These subscriptions are stored in a second table. Upon event insertion, the platform will iterate over subscriptions to compile a list of users that should be notified.

To oversee the described tasks, two services are implemented, Event Log Service and Event Subscription service.

Event Severity	Description
`INFO`	An event containing marking some notable, but otherwise unremarkable circumstance. No user action is required.
`NOTICE`	An important event that requires no user action.
`WARNING`	An event that could potentially be ignored, depending on the circumstances.
`ERROR`	An important event that stops the instrument or parts of the instrument from operating. User action is required.
`EMERGENCY`	A critical event that can cause instrumentation damage. User action is required.

Table 1: List of event severity levels

Event Type	Severity	Description
`System/Reboot`	`ERROR`	Logged when an instrument's computer is rebooted.
`System/StateUpdate`	`INFO`	Logged every time the agent sends a state update
`System/WentMissing`	`NOTICE`	Logged when an instrument has had no contact with the platform for an extended period
`System/Disconnected`	`NOTICE`	Logged when an instrument is disconnected from the platform

Table 2: List of some event types defined by the platform. Note the use of a prefix (`System/`) to namespace events.

4.3.4. Data collection

The storage infrastructure described in 4.2 is responsible for the archival of measurements, the instruments module is responsible for getting the data onto the platform. The data transfer procedure begins by the agent sending a "Begin Transaction" message, signaling it has a new measurement to upload. The message includes information about the data file, its size, its type, and a SHA-256 hash value. The platform then registers a new measurement file (named Data Files) and responds to the agent with the new measurement ID, along with a secret key. Finally, the agent sends the file using HTTP. If the received file's hash does not match the original hash value, the HTTP response indicated that the agent should attempt a retransmission, otherwise

a response marking success is sent. The agent can optionally delete the measurement file after a successful transmission, to lesser the storage burden on the instrument computer.

The data transmission procedures should be carefully implemented to avoid occurrences of "infinite loops", such as infinite retransmissions of the same file. The number of failures is recorded and appropriate action should be taken (for example, re-calculate the file's hash and begin the submission process anew).

4.4. Procedures

In this section, various important procedures will be described in detail using flow charts and sequence diagrams. This is an important step before implementation as it lets us see if the designed architecture holds in practice.

4.4.1. Handle instrument status update

The instrument status update sequence is shown in Figure 9 above, together with the MQTT connection procedure. The sequence runs on a timer, triggering the agent to check the status of the instrument. If at this time the agent is disconnected from the broker, it attempts to reconnect, as shown in the figure. The MQTT broker verifies the credentials by calling one of Agent Service's functions and accepts connections only after a successful verification. After a connection, the agent transmits the information to the platform by sending a Status Update message. In case of failure, the information is stored, and transmission is re-attempted after a small delay. These periodic check-in messages are also important for detecting power outages since a powered off instrument cannot contact the platform.

The broker receives the Status Update message and relays it to the platform's Agents Service, which is subscribed to all instrument messages. Upon retrieval of the message, the service stores the new instrument state, overwriting the old one. In case the key is invalid, the update is rejected without notifying the agent. A low severity event is generated for this state update (not shown in the sequence diagram). In case the instrument was marked as "missing" before, receiving a status update will change its status back to OK.

In subsequence diagrams, the MQTT login procedures will not be shown, to simplify the figures. By delegating this functionality to the MQTT login, the individual procedures are simpler because they can assume the agent is authenticated.

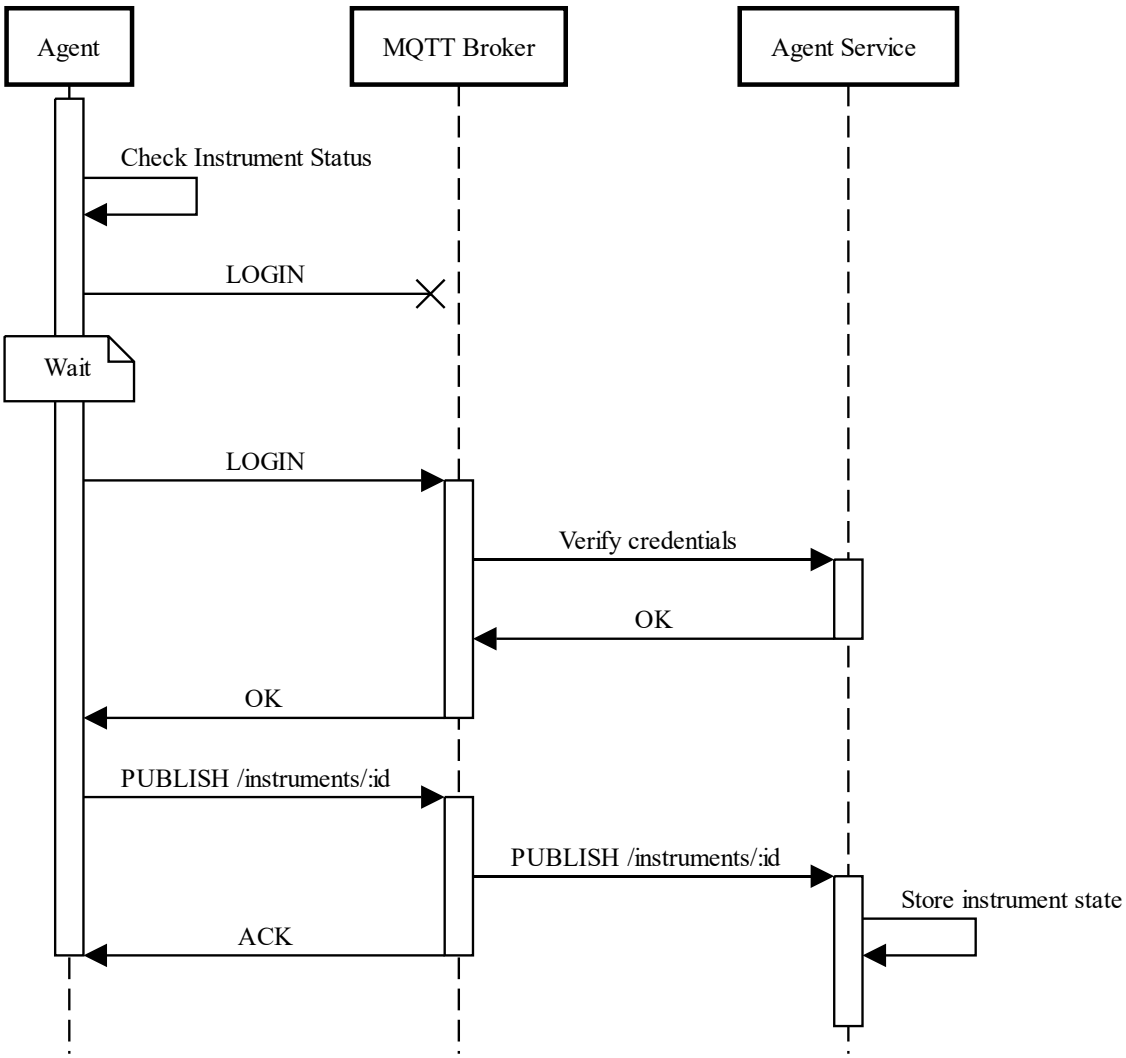

Figure 9: Sequence diagram showing the MQTT connection and instrument status update procedure. The MQTT connection is only repeated in case of an earlier disconnection.

4.4.2. Log events from agent

Like the procedure described in 4.4.1, logging new events from the agent begins by sending a message through MQTT, this time an Event message. The sequence is shown in Figure 10.

As before, the broker receives the message and relays to the Agents Service. After doing basic verification on the inputs, the message is forwarded to the Event Log service to get stored. Each time a new event is logged, the Agent Service updates the stored last contact time to the event's timestamp. This time is periodically checked to ensure no instruments go out of contact.

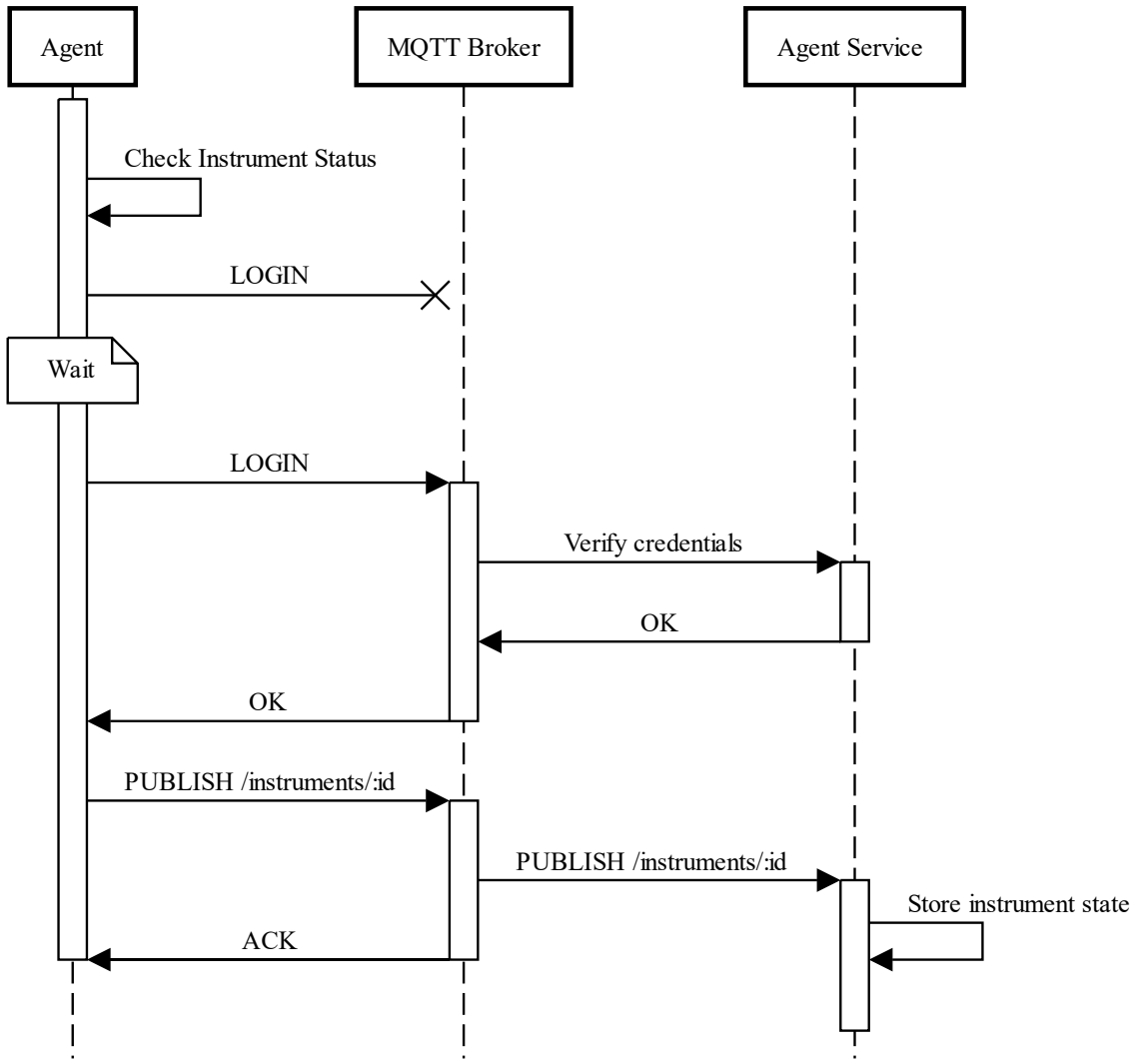

Figure 10: Sequence diagram showing the logging of an event from an agent. "S." stands for Service.

4.4.3. Power outage detection

To detect power outages, the platform periodically computes the elapsed time since the last contact of each instrument. If this interval is found to be greater than 15 minutes, the instrument is marked as Missing, a new event is created, and the subscribed users are notified. A flowchart depicting this task is shown in Figure 11. If an instrument is already marked as missing, the procedure is cancelled since it would generate repeated notifications every 15 minutes.

4.4.4. New measurement submission

The agent is continuously monitoring the instrument for the completion of new measurements. As soon as a new measurement file is created, the submission procedure is started. A sequence diagram of the procedure can be found in Figure 12. To make the implementation easier, at the

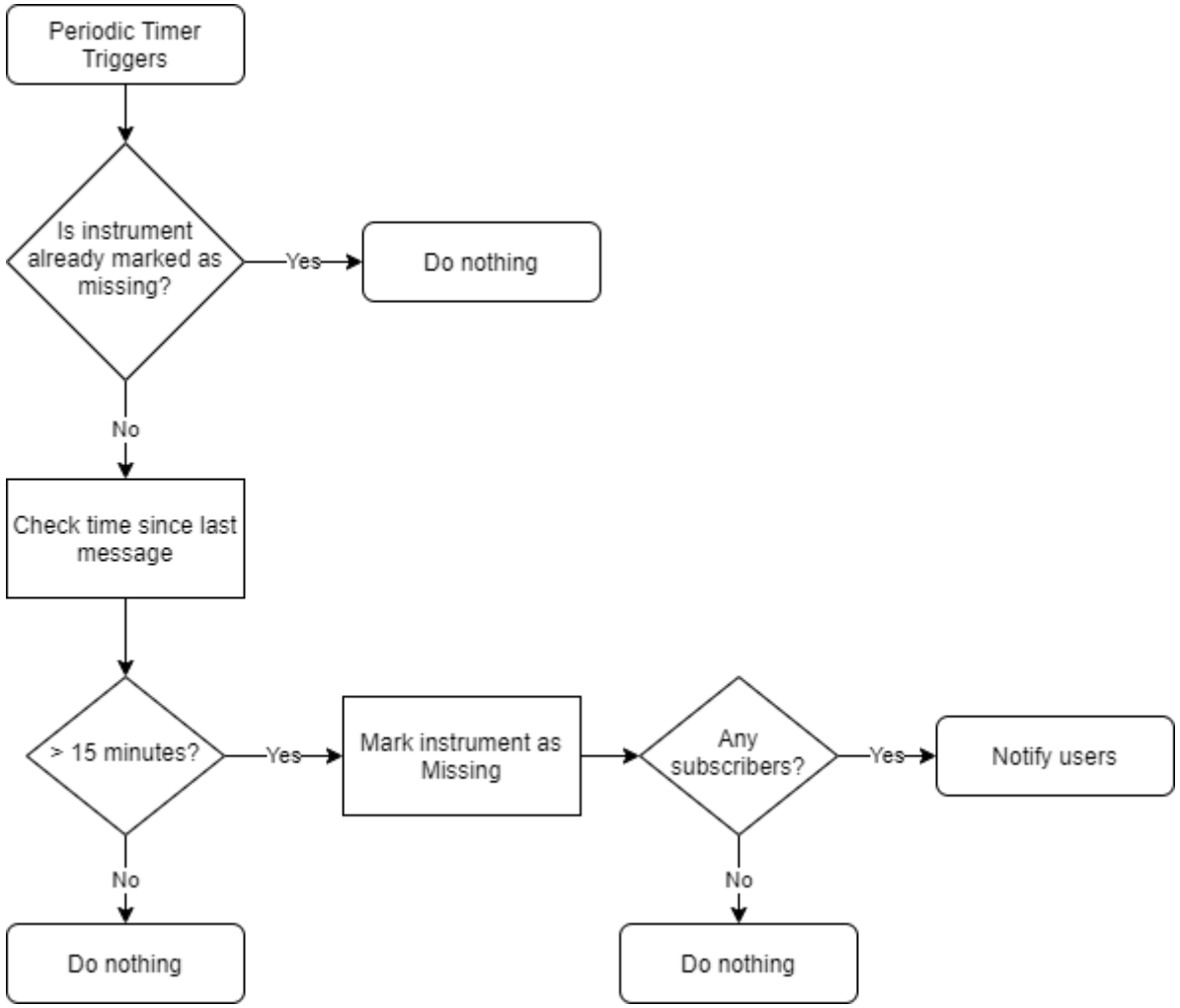

Figure 11: Flowchart of the missing-check login.

agent's side the procedure utilizes two services, one to watch for new measurements (File Watcher) and one to upload files (File Uploader).

The agent begins by calculating the file hash and storing a reference to this file in the local database. While the local database contains any files and a connection to the platform is available, the agent attempts to upload all files by requesting the start of a transaction through a MQTT message for each of them. The platform generated a unique ID for the measurement file and sends to the agent. Using the ID, the agent uploads the measurement file using a HTTP POST request. The platform then verifies the file by comparing its hash with the one submitted by the agent and, if there is a mismatch, it requests a re-upload through MQTT. If the file passes verification, it is stored in the object storage and the agent is notified through MQTT that the transaction has ended. The agent then marks the file as uploaded to avoid re-submissions and can optionally delete the file to conserve storage space.

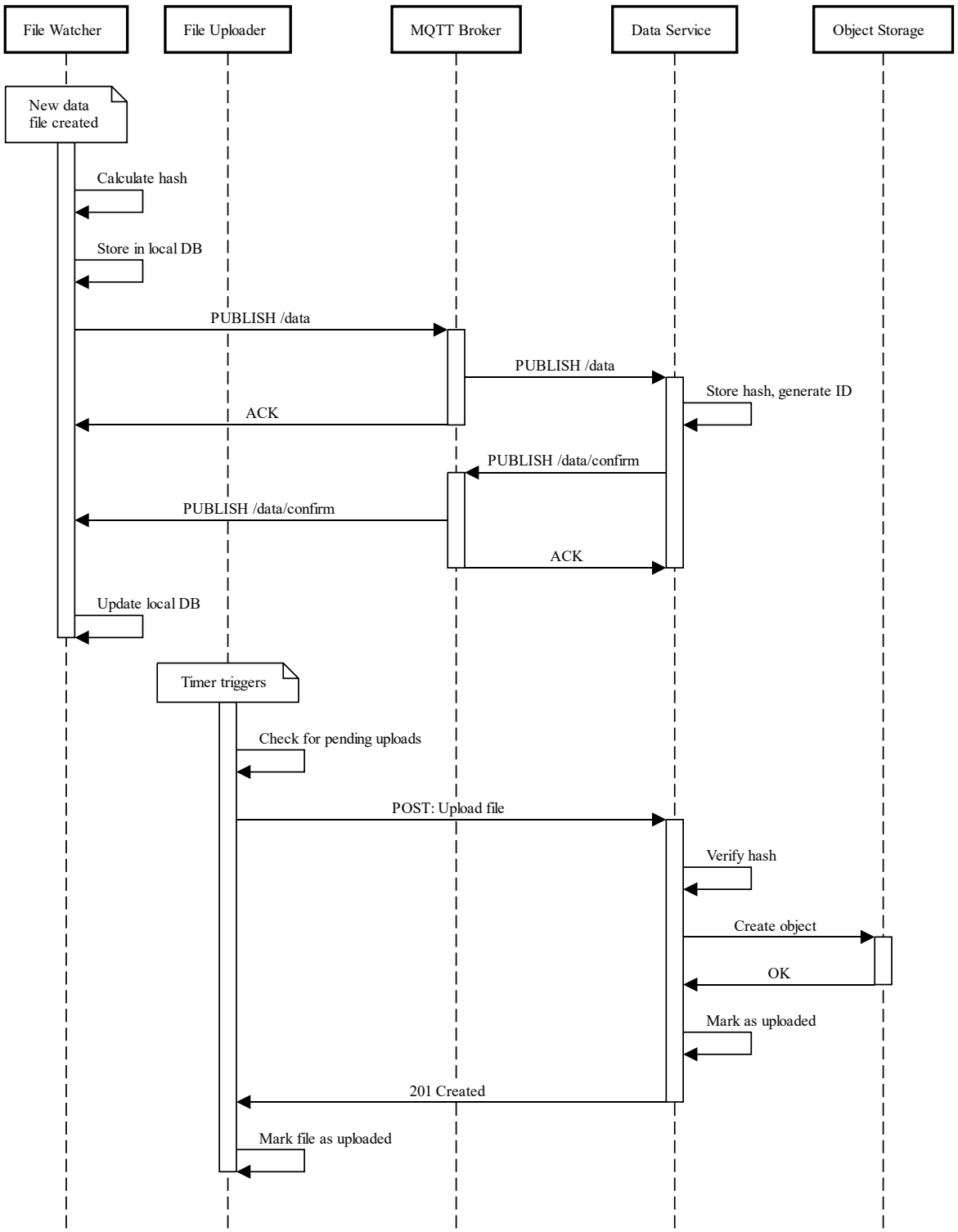

Figure 12: Sequence diagram showing the data submission procedure.

Any of the above steps can fail due to connectivity or power outages but can be repeated after connection is re-established. The procedure is fully asynchronous, meaning that all steps can occur in parallel for multiple files, or be interrupted and continued later. For simplicity, only the successful path is shown in Figure 12. It is crucially important to implement this procedure

with great care, adding the necessary guards so that the agent doesn't get stuck at any point (e.g. uploading the same file repeatedly).

4.5. Technology choices

There are countless ways to implement the platform. The constant rise of popularity of web applications the past 20 years has caused an abundancy of tools and frameworks focused on web development [29]. The amount of choices can cause paralysis but on the other hand, there is great potential for development. In this section some of the tools making up the platform will be discussed, why were they picked and how their advantages aids in the development of the platform. In all cases, the platform could have been implemented with different foundations, but it would look and feel quite different. There are a lot of "correct" choices.

4.5.1. Node.js

Node.js is an open-source, cross-platform runtime environment for JavaScript [30]. It can execute JavaScript code outside the browser. Based on the highly optimized V8 runtime (originally developed for the Chrome browser), it offers comparable performance to other web platforms [31] while keeping a very distinct advantage, the ability to use one language both on the server and the browser. Node sees extensive use in the industry, including by companies such as Microsoft and Paypal.

Node's architecture is based on the realization that a significant portion of a request's execution time, regardless of the application, is spent waiting for storage. By storage we can either refer to a file on a disk, or a database server, but nonetheless the fetching of a resource wastes a lot of time. To make use of this "dead time", node uses an event loop running in a single thread. Events (e.g. requests) are handled in first-come-first-served basis, up until an I/O call must be made (e.g. request a resource through the network). At that point, a non-blocking called is used so the event loop can process another event while waiting for results. This approach, besides producing great performance, it also simplifies development, since the programmers do not have to deal with multi-threading challenges, such as locking and concurrent access. If the application grows to the point where one thread cannot computationally manage the load, two or more copies of the application can be run simultaneously to handle requests. This is called horizontal scaling.

JavaScript (formally ECMAScript), the language that runs on node, is a high-level, dynamic, interpreted language, originally designed for web browser scripting. It is considered one of the core technologies of the World Wide Web, in the company of HTML and CSS. It was created

in 1995 by Brendan Eich for the Netscape web browser. Despite its potentially confusing name, the language has no relation with Java. JavaScript, and derivatives of it, quickly became the de-facto tool for browser scripting and standardization attempts started as early as the end of 1996. Early attempts did not prove fruitful but finally, in 2009, the ECMAScript 5 standard was released, the first ever widely accepted standard for JavaScript. Since then, ECMA has released yearly updates on the standard and JavaScript is considered a mature language.

In an attempt to improve the ergonomics of JavaScript, various languages were developed. CoffeeScript is a prominent example, a language with Ruby-based syntax that compiled to JavaScript code. Another such project is TypeScript, a superset of JavaScript with support for static typing. TypeScript's type system offers itself to editors and integrated development environments, improving support for problem identification before running the code. The platform was developed using TypeScript in order to benefit from better integration with editors and the static typing system, which can prevent errors before runtime.

On top of Node.JS, the Nest.JS framework will be used. Nest is a framework for building web application with Node. It is fully compatible with TypeScript and offers rigid foundations for web development. By acting as a level of abstraction above Node and commonly used libraries, Nest lets developers focus on the target product, while providing access to the underlying libraries if needed. The platform was originally written without Nest but after switching, the author found that productivity increased significantly by taking advantage of the extensive implemented tooling. Nest's way of organizing the application is also very similar to the organizational structure described earlier in section 4.1.

Specifically, apps developed with Nest are organized into *Modules*, with each module containing *Providers* and *Controllers* [32]. Controllers are responsible for handling incoming requests from clients and returning the appropriate responses (referred to as *Interfaces* before). By clients, we generally refer to HTTP clients and web browsers, but a controller is also tasked with handling requests through MQTT. Providers, on the other hand, provide specific services to other providers. This can seem confusing at first but imagining a UsersController that handles requests related to user accounts, a UsersService would carry out the requests, by talking to a database and making the necessary changes. Providers and controllers can depend on other providers, creating a dependency graph. This graph is automatically solved by Nest, so the developer does not have to painstakingly figure out in which way to initialize the components (this pattern is known as Dependency Injection or DI). Providers can be both private and public, referring to their visibility outside their module.

Finally, to handle operations related to the relational database the library TypeORM will be used. TypeORM, as someone might have guessed, is an Object-Relational Mapping tool that maps database rows to TypeScript objects. Database tables are represented with *Repositories*, which integrate well with Nest's architecture by acting as Providers. Additionally, it offers a programmatic query builder, so the programmer can avoid writing SQL using string manipulation. While there is notable aversion to ORM tools by the industry [33]–[35], there are undeniable benefits in using ORMs for mapping simplicity, easier cache control and development speed. At high throughputs, ORMs can prove a hurdle and reduce performance, but this will not be a problem for the prototype in question.

4.5.2. .NET core

.NET core (pronounced dot-net core) is another open-source framework for developing software [36]. It is the successor of .NET framework, originally targeting only the Windows operating system, now supporting all major platforms (Windows, Linux, and Mac). With nearly 20 years of market availability, the .NET ecosystem enjoys a diverse ecosystem of mature libraries and tools, both for desktop and web applications.

.NET is built on top of the Common Language Infrastructure (CLI) [37], an open specification for platforms and runtime environments that consist of compiling high level programming languages to intermediate code, in order to be executed by a cross-platform virtual machine [38]. This approach allows for applications to be developed in one or more high level languages and then executed on different platforms using the respective runtimes (readers with Java experience will find this familiar). .NET core specifically runs on the CoreCLR runtime.

Besides the runtime and virtual machine specifications, a critical component of the .NET framework is the standard library. The .NET standard is a specification of APIs that are intended to be available on all .NET implementations (read: runtimes). This way, applications developed using .NET core can not only run on multiple platforms but also have access to an extensive library with platform-independent behavior.

One of the main requirements for the agent is to run on both the Windows and Linux operating systems. By using the .NET core framework this can be achieved effortlessly, while still having access to the extensive tooling. By using .NET core's dependency injection framework, the agent's architecture can mimic the backend's (with Nest's DI) and reducing cognitive load in development.

4.5.3. Docker

Docker is an open platform for the development and deployment of software packages. Applications are packaged and executed in isolated environments called *containers*. This can be considered a form of OS-level virtualization since it partially replaces the need to use a hypervisor to isolate services in virtual machines. By doing away with the hypervisor resources, all services use the same base operating system and kernel, making better use of computational resources. Finally, Docker aids in the development process by simplifying the environment setup process. Instead of having to meticulously prepare a hosting environment for development, testing and finally production use, containers are easily created and remain consistent regard-

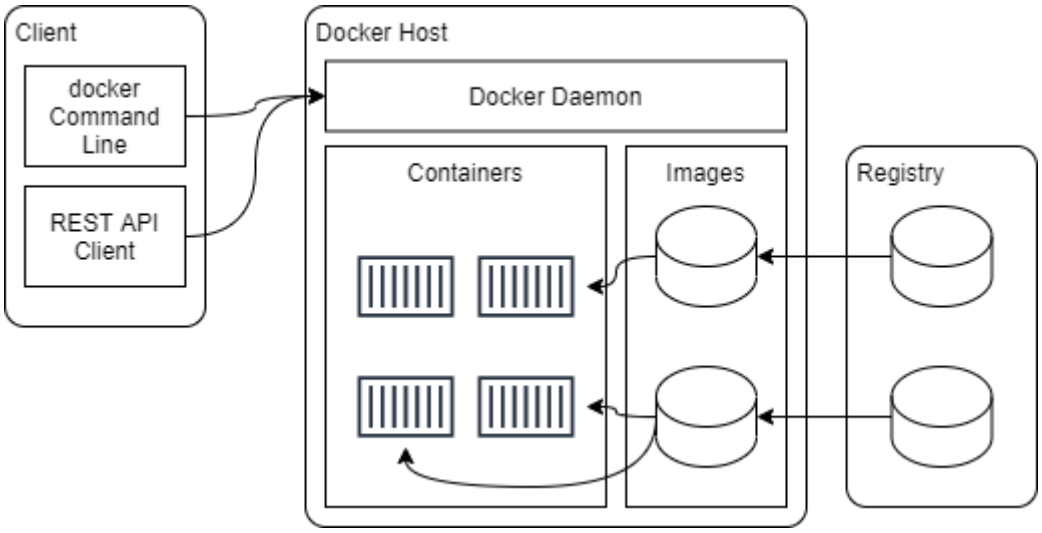

Figure 13: Overview of Docker's architecture.

less of where they run. This eliminates hard-to-troubleshoot problems relating dependencies. Docker is supported natively on Linux and, by the use of virtualization, on Windows.

Docker is based on central service called the *Docker Daemon*. This service is tasked with doing the heavy lifting of building and running the containers. It can be controller through a REST API or a UNIX socket. A command-line client for controlling the Daemon is also available, called *Docker Client*. The daemon and the client can communicate through the network, enabling easy administration of a remote host. The general architecture is also shown Figure 13.

Containers and the applications that run inside them have no access to the host's filesystem. Instead, they are based on *images*, which are read-only filesystems containing the necessary files and libraries to run the containerized application. Images can be created or downloaded from an image repository called *Registry*. Creation of an image consists of writing instructions on what to add into the filesystem, much like a recipe. These instructions are written using

*Dockerfile*s. Listing 1 shows a simple Dockerfile for creating a basic image. Images can be composed together through layering, creating new images based on existing ones.

```
FROM centos:8

WORKDIR /app
ADD ./dist/

CMD ["run-app.sh"]
```

Listing 1: `Dockefile` for creating a simple image, based on the image "`centos`", version 8. The `ADD` command will add the directory `./dist` to `/app` inside the container. When the container is started , the command set by `CMD` will be executed.

Applications, including the platform designed in this thesis, often require more than one separate software components to function, each with its own dependencies. The use of Docker is really beneficial in this case, as the individual services are isolated, avoiding complicated issues with dependency resolution. A development environment can be quickly created using *docker-compose*, a tool for starting multiple containers with one command. This reduces day-to-day infrastructure friction and helps the developers focus on important work.

The raising popularity of Docker has created a flourishing ecosystem based on platforms. Virtualization platforms have added support for running containers directly, avoiding the need to manually setup Docker hosts. Furthermore, based on the same container standard (called open container initiative, OCI), Docker alternatives are also available, keeping compatibility with existing images. A popular example is Podman [39], developed by Redhat, a container engine that is able to function without the use of a server daemon. Cloud providers such as Amazon and Google offer products related to the hosting of Docker containers.

The choice to use Docker in the development of the platform should not be a surprise, as it is a popular solution for dealing with hosting infrastructure. In development, all services are containerized, including the relational database and the object store. This is helpful while working on the platform, as you can get a working environment with only one command. In the final production, it is common not to containerize databases (and object stores), but the platform's backend can be delivered as a container for convenient installation and monitoring. It would be possible to finish this thesis without the help of Docker, but it would certainly be more tiring.

4.5.4. PostgreSQL

The vast majority of web applications must store some relational data and the designed platform is no exception. To fulfill this need, PostgreSQL was used as the RDBMS of choice. PostgreSQL is an open source object-relational database system that uses the SQL language. It has a strong reputation for its reliability, performance, and robust feature set. It runs on all major operating systems and enjoys a rich ecosystem of add-ons and tooling. A popular addon is PostGIS, adding geospatial features to the database.

PostgreSQL strives to comfort to the SQL standard, diverging only where it makes sense architecturally. It is an ACID-complaint database, meaning database transactions are guaranteed despite errors such as connectivity issues, crashes, and power outages. For resiliency and expandability, PostgreSQL can run in clustered mode, thus if one server fails, the rest can continue working.

The platform does not depend on any advanced SQL features, and thus, should work with any SQL standard compliant database. However, there is little reason to use something other than PostgreSQL, given its excellent credentials, standard conformity, and performance. To interface with Postgres, as mentioned in section 4.5.1, the TypeORM library [40] is used. An interesting side-effect of using an ORM is database agnosticism which makes it easy to replace PostgreSQL if the need to do so arises.

4.5.5. MinIO

MinIO is an object storage system with a S3-compatible API [41]. While a commercial offering is available, the base product is open-source and freely available. MinIO's feature-set is in no way lacking, offering support for continuous replication, erasure coding and multi-cloud deployments, while keeping a very distinct advantage, its ease of use. Starting a MinIO server only requires a single binary and a single command, instantly giving access to an object storage system.

MinIO also offers a client library SDK with bindings for JavaScript. This library aids in the fast development of applications on top of object storage but is in no way limiting the users to MinIO, as it supports the S3-style API. The object storage system of choice can later be swapped out without having to rework the application.

MinIO can operate with a single node, as already mentioned, but operation in a cluster is also possible. A MinIO server uses commodity hardware with locally attached hard disks (this

pattern is commonly referred to as JBOD). Being a fully symmetrical design, all nodes in the cluster are equal in capability. This is in contrast with other Object Storage systems (e.g. Ceph), that require metadata or auxiliary nodes to operate [19].

The platform uses MinIO primarily for its simplicity. As more research data is collected and the needs for storage grow, an incredibly careful analysis must be conducted to select the best solution. Some alternative choices besides MinIO are Ceph and OpenStack Swift, both mature packages offering object storage services. However, in the prototype stage of development, MinIO's ease-of-use minimizes the time investment required by the storage subsystem.

4.5.6. React

React is an open-source JavaScript library, developed by Facebook, for building user interfaces or interface components [42]. In simple terms, the most important aspect of React is the ability to create custom and reusable HTML components to build interfaces. Traditionally, websites are represented using the Document Object Model (DOM), a tree structure where each node is a part of the document [43]. DOM can be expressed using XML or, more commonly, HTML. Browsers offer APIs (in JavaScript) to enable programmatic manipulation of the DOM in real-time. Using these APIs works well for adding simple interactivity to a web page but as websites and web applications grew more complicated, it became increasingly hard to use direct DOM manipulations to swap out large parts of the website. This effect caused the fertile ground for libraries like React and Angular, libraries that offer alternative ways of manipulating the DOM, to take up ground. In the Model-View-Controller pattern, React takes care of the "View", allowing programmers to architect the rest of the applications as they see fit.

Internally, React maintains a "virtual DOM" tree, made out of React components. These components are conceptually similar to HTML components, in that they are nodes of a specific type, they accept "properties" and optionally have children nodes. In Listing 2 a simple application's structure is shown, made of two nested components. Each time new information should be rendered to the screen (which means, to the browser's DOM), React compares the new virtual DOM state with the previous one and, finally, makes targeted updates to the browser's

DOM. This tactic allows the developers to write code as if the entire page is rendered while the library only renders changes, providing a significant performance boost.

React comes with an extension of the JavaScript (and TypeScript) language called JSX (JavaScript XML). This extension adds support for writing HTML-style notation inside a JavaScript file and having it translated to React command that create the virtual DOM. An example of this translation is available in Listing 3. While use of JSX is optional, it really reduces code ver-

```
<PageContainer title="Hello!">
  <p>This is the content of PageContainer</p>
</PageContainer>
```

Listing 2: The structure of a React app with one component that contains an HTML element. The outer component, `PageContainer`, has one prop called title.

bosity and React is generally never used without it. Since JSX elements are in fact JavaScript objects, they can be stored in variables and manipulated like any other object. Readers familiar with the PHP ecosystem might be reminded of XHP, a similar project that adds HTML-style notation in PHP. XHP is also developed by Facebook.

```
// JSX
const heading = <h1 className="title">Hello world!</h1>;
// Compiled to JS
const heading = React.createElement(
  "h1",
  { className: "title" },
  "Hello world!"
);
```

Listing 3: Example of JSX code compiled to JavaScript.

React components can be represented with either a class or a function. In the first case, React defines a list of “Lifecycle” methods, functions that the class might implement and are called by the library when appropriate. For example, `render()` is called so the component can create the virtual DOM it wants to display. Another pair of common lifecycle functions is `componentDidMount()` and `componentWillUnmount()`, called when the component is firstly shown on the screen and before it is hidden. These methods can be used to fetch or free resources. Alternatively, components can be represented as functions, simply called every time the components get drawn on the screen. Functional components are simpler and easier to

understand, having only one code path instead of all the different lifecycle methods. In Listing 4, a simple component is written both as a class and as a function.

Data is React is passed from component to component through *props* (short for properties). Each component can pass some variables to its child components, but not to the parents. Each

```
class Heading extends React.Component {
  public render() {
    return <div>
      <h1>{{this.props.title}}</h1>
      <h2>{{this.props.subtitle}}</h2>
    </div>
  }
}

const Heading: React.FC = props => {
  return (
    <div>
      <h1>{{props.title}}</h1>
      <h2>{{props.subtitle}}</h2>
    </div>
  );
}
```

Listing 4: A simple component called Heading, implemented both as a class and as a functional component. Note that for the functional version, the JavaScript lambda syntax is used.

time a component's props are changed, it is rendered anew. To communicate with parent components, a *callback function* can be passed as a prop, which then the child component will call to signify and event or pass some data upwards the component tree. Internally, React components are allowed to store variables in the *state*. The state is managed by React and changes to it are only allowed through the `setState()` function. Each time the state is updates, React redraws the component and any children of it as needed.

Angular and Vue are the main competitors of React, offering alternative ways to build front-end web applications. While Vue follows React's footsteps in being a lightweight "View" layer, Angular is a complete application framework, covering all three layers of MVC (model, view, and controller). Both Vue and Angular rely on templating to render HTML, in contrast to React's approach with JSX. React and Vue's ecosystems contain addon libraries for covering many of the functionality that comes built-in with Angular. Finally, all three frameworks enjoy a wide range of UI component libraries.

From these choices, React was picked for its very intuitive approach to templating (or lack of thereof) with JSX, extensive ecosystem and tooling and ease of use. As a matter of fact, all three major frameworks (and many more) would probably be as capable to handle the platform's frontend needs. To speed up prototyping, the Ant Design UI component library is used, an open-source library created by Ant Financial (commonly known as Alipay).

5. Implementation of a prototype

While in chapter 0 a high-level design is outlines, in this chapter technical details about the implementation of the prototype are presented. This chapter contains a lot of UML class diagrams in order to visualize the architecture of each module. There diagrams are not meant to be exhaustive and some minor connections or components might be missing. Their main purpose is to give a sense about how the different services and modules interact with each other, and not to be an flawlessly documentation of the modules.

5.1. Backend

As mentioned in section 4.5.1, the platform's backend is implemented using the Nest.JS framework. In Nest, apps are structured by modules, each offering controllers and services. The simplest module is `UsersModule`, tasked with user profile management. To serve as an example, Listing 5 shows the required code to declare this module. In the subsections below each of the platform's modules will be described in detail.

```
@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UsersService, AuthenticationService],
  controllers: [UsersController],
  exports: [UsersService, AuthenticationService],
})
export class UsersModule { }
```

Listing 5: Declaration of the Users Module. Nest uses a decorator construct to create modules by wrapping a given class.

Modules are created using decorators (through the `@` syntax), which are functions that wrap another entity. In this case, it is enough to wrap an empty class. Decorators are extensively used by Nest to create modules, services and controllers.

5.1.1. UsersModule

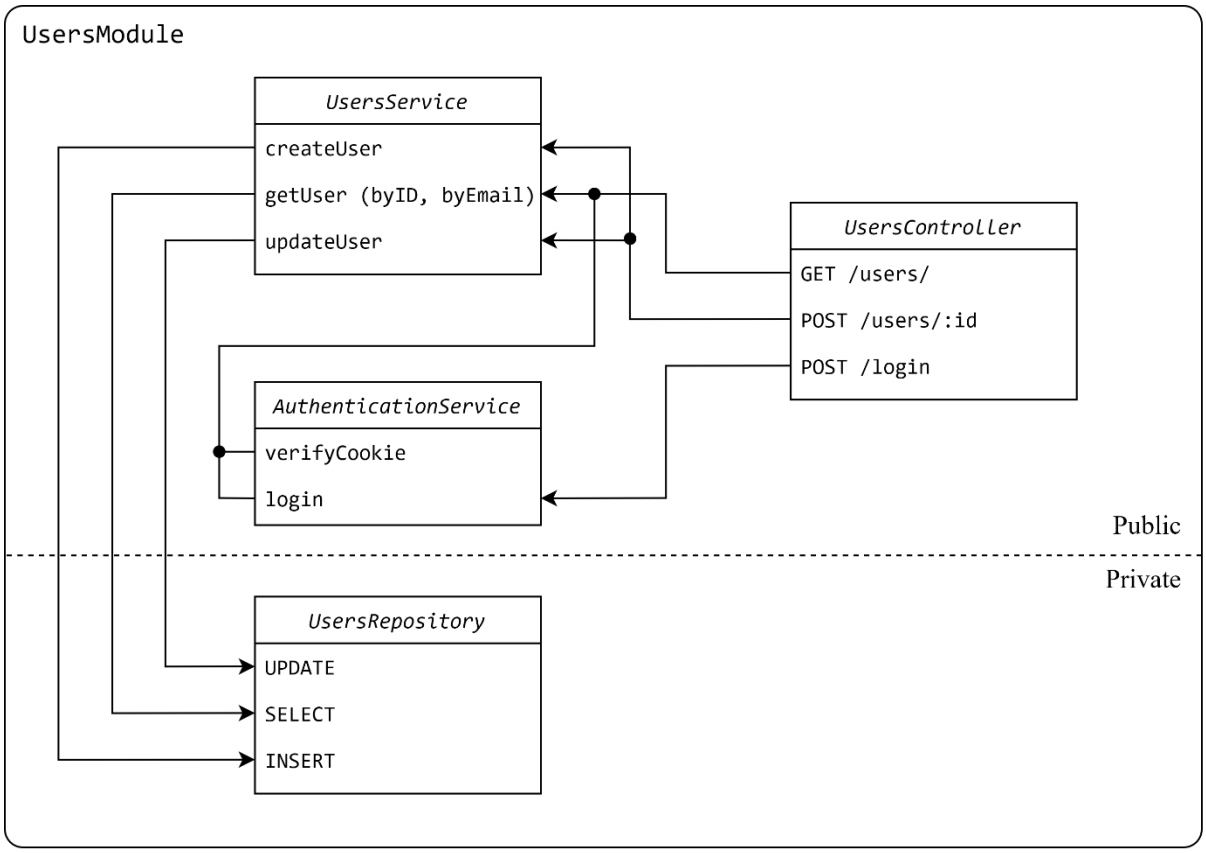

Figure 14: UML Diagram of `UsersModule`. Both Public and Private services and controllers are shown.

Starting with the simplest module, `UsersModule` is responsible for user accounts and authentication. A UML diagram of this module is shown in Figure 14. The Users Repository seen in the figure is created automatically by providing TypeORM an *Entity* definition, a class which field's are translated to database table columns. A sole controller is used since only three API endpoints are available, listed in Table 3.

API Endpoint	Description
GET /users	Returns all registered users
POST /users/:id	Create or update a user with a given id
POST /login	Login using email and password

Table 3: REST API Endpoints of UsersModule

All management operations are handled by `UsersService`, overseeing user profile updates, user creations and querying of users by other services. A supplementary service, `AuthenticationService`, is tasked with handling logins and cookie verifications. Most API endpoints are reserved for logged-in users, and thus require authentication. `AuthenticationService` can be used across all modules and controllers to fill this need.

Finally, the repository (`UsersRepository`) is the only private provider of the module. This is intentional, as allowing other modules and services to directly access the database is considered bad practice. By having all user-related function go through one of the appropriate services, we can be sure that the database is always in an acceptable state and user-related code remains inside the module, increasing maintainability. This pattern will be repeated throughout the platform.

5.1.2. InstrumentsModule

Much more interesting compared to the last one, this module is tasked with managing all instrument tasks and most importantly, status updates through MQTT. An UML class diagram of this module is available in Figure 15. In the figure, all repositories are omitted for simplicity, they are described however in Table 4.

Repository	Description
`InstrumentsRepository`	A remarkably simple table, simply holding instrument IDs and names.
`AgentsRepository`	Contains information about each agent. Last contact time, last reported status and which instrument is the agent associated with. The last status is stored inside a JSONB column that can accommodate arbitrary data.
`EventLogRepository`	Contains the event log for all instruments
`EventSubscriptions`	Manages the subscriptions of users to events

Table 4: Repositories Instruments Module

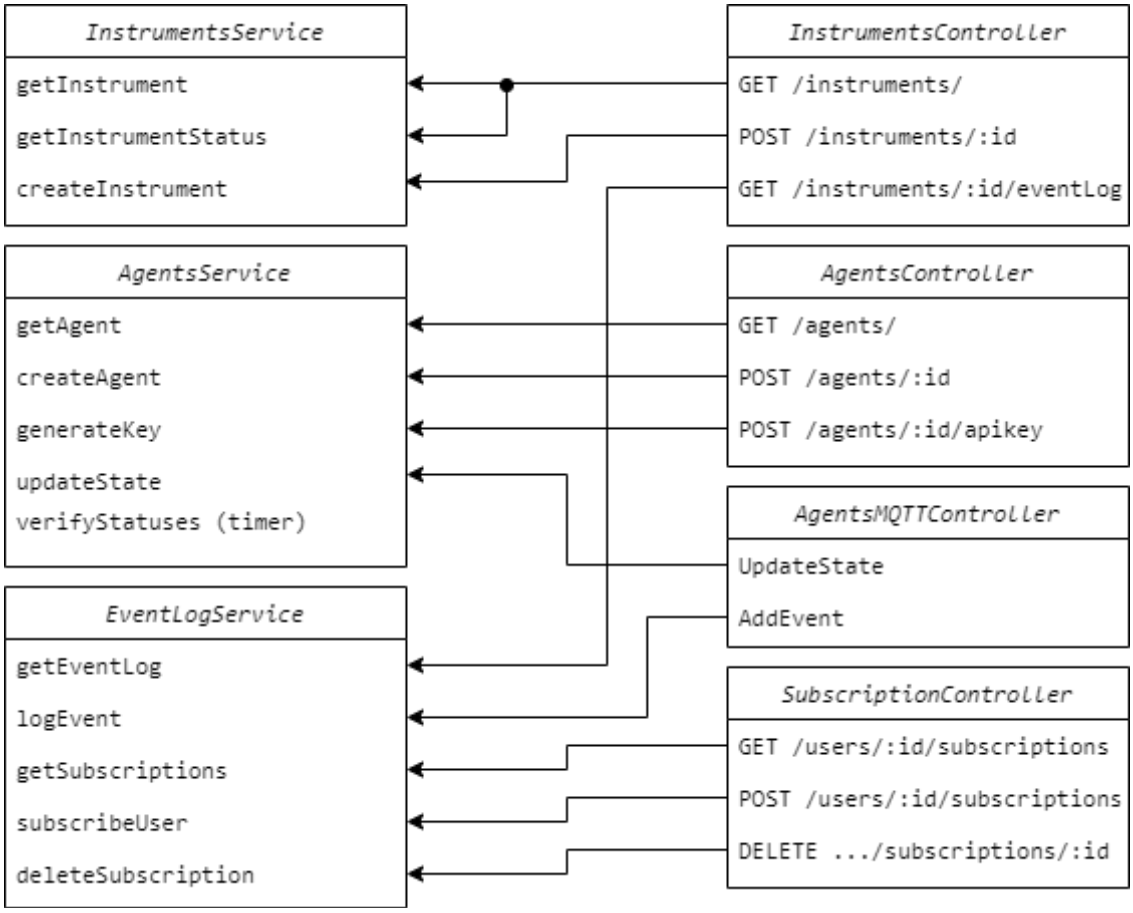

Figure 15: UML Diagram of `InstrumentsModule`. For simplicity, repositories are not shown.

While the obvious deduction would be that `InstrumentsService` is the most important provider in this module, most of the logic is inside `AgentsService`. For flexibility, the platform allows one instrument to be associated with more than one agent, since it is possible that many computers are needed to operate it. Most logic related to status updates should therefore refer to agent updates, making `AgentsService` the main provider of this module. To determine the instrument's status as a whole entity, all agents are considered. Individual agents can be marked as nonessential to signal that the instrument can function without it. Otherwise, if any agents report a degraded or failing status, the whole instrument is considered failing.

The `verifyStatuses` function runs on a timer (e.g. every 15 minutes) to check about disconnected instruments. If an instrument has had no contact with the platform for an extended period, it is marked as missing and users are notified (according to subscriptions). This task scheduling feature is a part of Nest.

The users are not notified directly by `AgentsService`, but instead an event is created using `EventLogService`, which in turns triggers the subscriptions. Every time an event is logged

(using `logEvent`), the list of subscriptions is filtered for any entries that match the new event and users are notified appropriately. Notifications are handled by a different service called `NotificationService` (not pictured in Figure 15) that supports E-Mail and Webhooks.

This module contains four controllers, one of them being unique in that it is tasked with MQTT messages instead of HTTP requests. However, the principle is the same. All endpoints are described in Table 5. While all endpoints could have been defined in one controller, it makes organizational sense to divide them in separate constructs, one controller for each service. Controllers mostly contain input validation logic and not much can be discussed about them.

API Endpoint	**Description**
`GET /instruments`	Returns the list of all instruments and their PIs
`POST /instruments/:id`	Edit an instrument's details or create a new one if `id` is omitted
`GET /instruments/:id/eventLog`	Fetch an instrument's event log. Can optionally filter for a specific period of time
`GET /agents`	Returns all agents, optionally filtered by an instrument
`POST /agents/:id`	Edit an agent or create a new one if `id` is omitted
`POST /agents/:id/apiKey`	Creates a new API key for an agent and returns it. Once retrieved, the key cannot be read again and must be created anew if required
`GET /users/:id/subscriptions`	Fetch the list of all subscriptions for a specific user
`POST /users/:id/subscriptions/:id`	Create a new subscription for a user
`DELETE /users/:id/subscriptions/:id`	Delete a subscription

Table 5: API Endpoints of `InstrumentModule`

5.1.3. DataModule

This module is an abstraction layer on top of the object storage, tasked with all data-related jobs. While crucially important for the platform to function, most of the hard lifting is done by the object storage system, leaving this module simple and easy to maintain. An UML diagram is available in Figure 16.

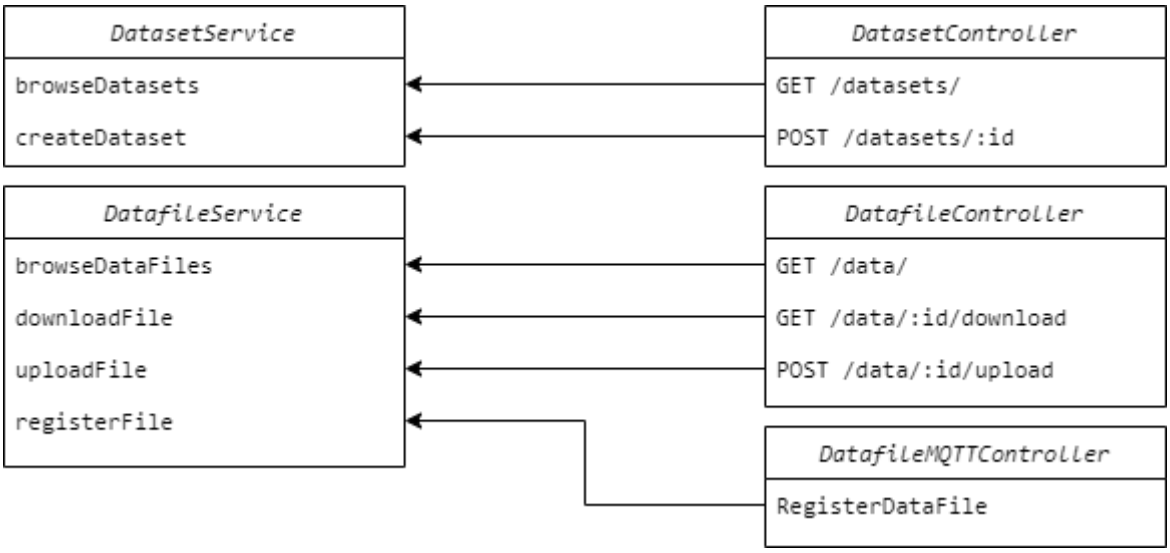

Figure 16: UML diagram of the DataModule. The repositories are omitted.

To accommodate the use case of an instrument producing two or more distinct datasets, all measurements are categorized into *Datasets*, which are owned by instruments. Datasets consist of *Data Files*, the internal name for measurements. Each Data File corresponds to one measurement file archived in the object store. To meet these needs, two repositories are required (not sketched in Figure 16): One for Data Files and one for Datasets.

While the module seems simple at first glance, a lot of responsibility falls onto the Data File Service, which is responsible with orchestrating the data uploading procedure described in section 4.4.4. It is the only service that makes use of an MQTT client to send messages to agents, in order to inform them about the new measurements' unique IDs. The service also communicates with the object storage system to read and write files. For efficient handling of the files, the service uses *pipes* to pass data from the storage to the client and vice versa. In the case of a download, instead of first loading the whole file in memory from the object storage and then passing it to the client, possibly consuming large amounts of system memory in the process, it streams the data to the client as it receives them.

5.2. Storage

The storage subsystem is based on MinIO, as already mention in section 4.5.5. To better simulate a large-scale deployment of the platform, MinIO is used in cluster mode with two nodes, each with four 1TB hard disk attached. Furthermore, each node uses 3 hard disks as data disks and the last one as parity disks. This is required because in cluster mode, MinIO automatically uses erasure coding. By using this technique, each object (each file) will be spread across 4 disks, allowing the system to lose up to 2 disks or a whole node without data loss. Storage effectiveness can easily be calculated:

$$\text{Storage Efficiency} = \frac{\text{Data Disk \#}}{\text{Total Disk \#}} = 2\frac{3}{4} = 75\%$$

MinIO also offer a JavaScript library for conveniently communicating with S3-style object stores without having to implement each used API call. The platform uses this library to communicate with MinIO.

5.3. MQTT Broker

Initially, the platform relied on the Mosquitto MQTT Broker [44] for its message relay needs. Mosquitto is a well-known open-source broker, written in the C language. Unfortunately, during testing, while Mosquitto displayed excellent performance, it would frequently cause clients to temporarily disconnect. While the clients would immediately reconnect, this caused a false alarm disconnect notification to be sent. This issue occurred even when the client was running on the same machine as the broker but only if the client was a Windows machine. This issue was also reported by other users online [45]. Finding no workaround, the broker was swapped out for the community edition of HiveMQ, another open-source broker written in the Java language [46].

HiveMQ additionally offers a Software Development Kit (SDK) for creating authentication addons. Since MQTT allows client authentication, using a pair of values akin to username and password, HiveMQ was programmed to only accept connections from clients with valid agent IDs and API keys. To facilitate communications between HiveMQ and the platform, an `AgentAuthenticationController` was added to the `InstrumentsModule` (section 5.1.2). Finally, the broker was configured to use Transport Layer Security (TLS) to encrypt messages while in transit using a self-signed certificate. This certificate was installed on the clients so they can also verify the integrity of the broker.

5.4. Agent

The agent had to meet a lot of requirements, including same behavior across platforms, MQTT over TLS support and a pluggable architecture to facilitate adding more instruments. Using .NET core takes care of the first requirement (with some care) and provides the tools to deal with the third. For MQTT support, the MQTTnet library is used [47].

The agent is based on top of ASP.NET's dependency injection framework. While ASP (which stands for Advanced Server Pages) is .NET's web application framework and has nothing to do with what the agent's goals, it offers a DI framework, not unlike the one offered by Nest, that can be used in any application. Using the service pattern again, the agent is split into services with specific goals. The UML diagram of Figure 17 sketches out the main components.

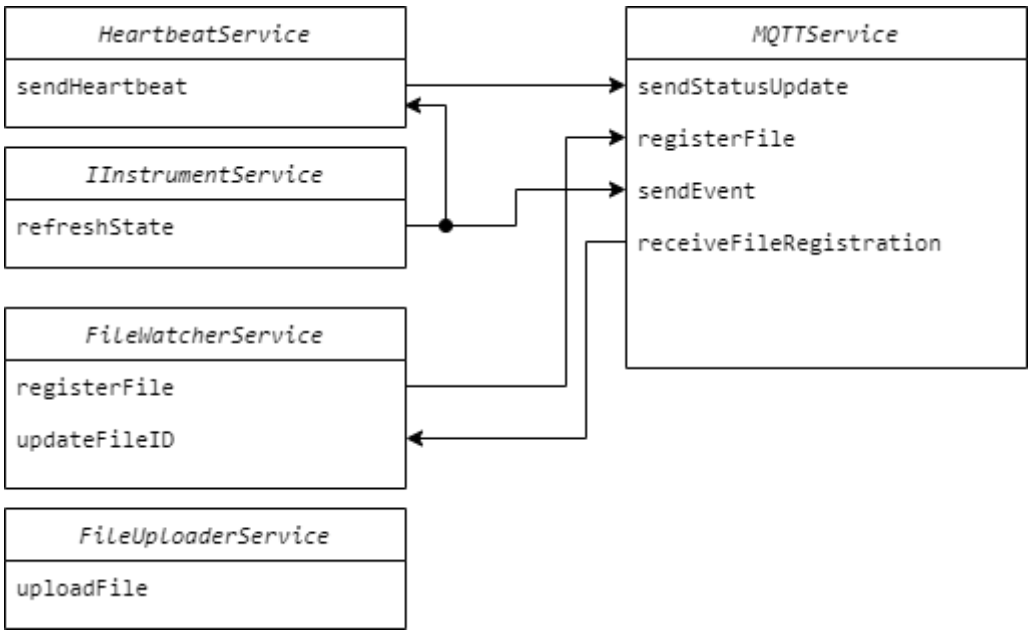

Figure 17: UML diagram of the agent, showing the services but omitting other, less important, components.

All communications through MQTT are managed by the creatively named `MQTTService`. Internally, the service keeps a queue of messages to be sent to the platform. If at any moment, the connection is severed and the messages cannot be sent, they remain in the queue, which is emptied as soon as it becomes possible again.

`HeartbeatService` is responsible for periodically checking the instrument's status and informing the platform. By default, only statistics about the computer are collected, but the service has an optional dependency of the type `IInstrumentService`. This *interface* defines which functions should be implemented by addons to read specialized parameters from

instruments. This way, if such an addon exists, it is automatically resolved by the DI framework and passed to `HeartbeatService`, otherwise it continuous with an "empty" instrument service.

Data management is done by two services, `FileWatcherService` and `FileUploaderService`, both depending on a local database. An SQLite database is used, in tandem with .NET's Entity Framework, to manage the list of files currently on system. By continuously monitoring the output directory of the instrument, the Watcher service detects new files as soon as they are created, hashes them, and adds them to the local database. If a connection to the platform is available, the file uploader service then attempts to sync the database by following the procedure described in 4.4.4, otherwise the files remain until connection is re-established. Both of these services rely on `MQTTService` to send messages to the platform, while is `FileWatcherService` special in that it is the only service to also receive messages. This is achieved through a "listener" interface, which the MQTT service can attach onto.

Finally, all the above are parametrized using a configuration file to allow use of the same executable on all instruments. Through configuration the correct instrument addon can be enabled and parameters about check-in intervals and measurement directories are set. Great care has been taken to make sure that no platform-specific code is used in the agent, and thus all the functions above behave the same on both Windows and Linux. The agent supports being executed as a Windows Service for background use, as well as being a Linux daemon.

5.5. Website

The website is written using React, as mentioned in section 4.5.6. Using react-router [48], the de-facto routing[3] library for React, the app is organized into semi-independent tabs. This is a common design used for administration panels such as this one. A sidebar panel is used for presenting the navigation choices and the content is presented on the right. A mockup of this design is shown in Figure 18. The left panel is displayed at all times, while react-router determines which component to display on the right. Screenshots of the implementation are shown in Figure 19.

[3] Routing in the context of single page applications refers to rendering the correct page depending on the current URL. For example, www.example.com/home and www.example.com/library should display different content, commonly achieved by using different React components. This is achieved in react-router by assigning a component to each URL, the library automatically renders the correct one as the user navigates.

Following the described design, each page is a React component. To make it easier to add pages, a general `PageContainer` component is used that offers the common functionality used by pages, such as the header bar. By relying on such reusable components, the website's design and user experience remains consistent across all pages. This is especially important when working with a team because different developers might work on different pages.

Since the website is a SPA, it can be treated as a "static" website and served by any webserver (e.g. Apache or nginx). The webserver should be configured to redirect any requests for resources (API calls) to the platform's backend.

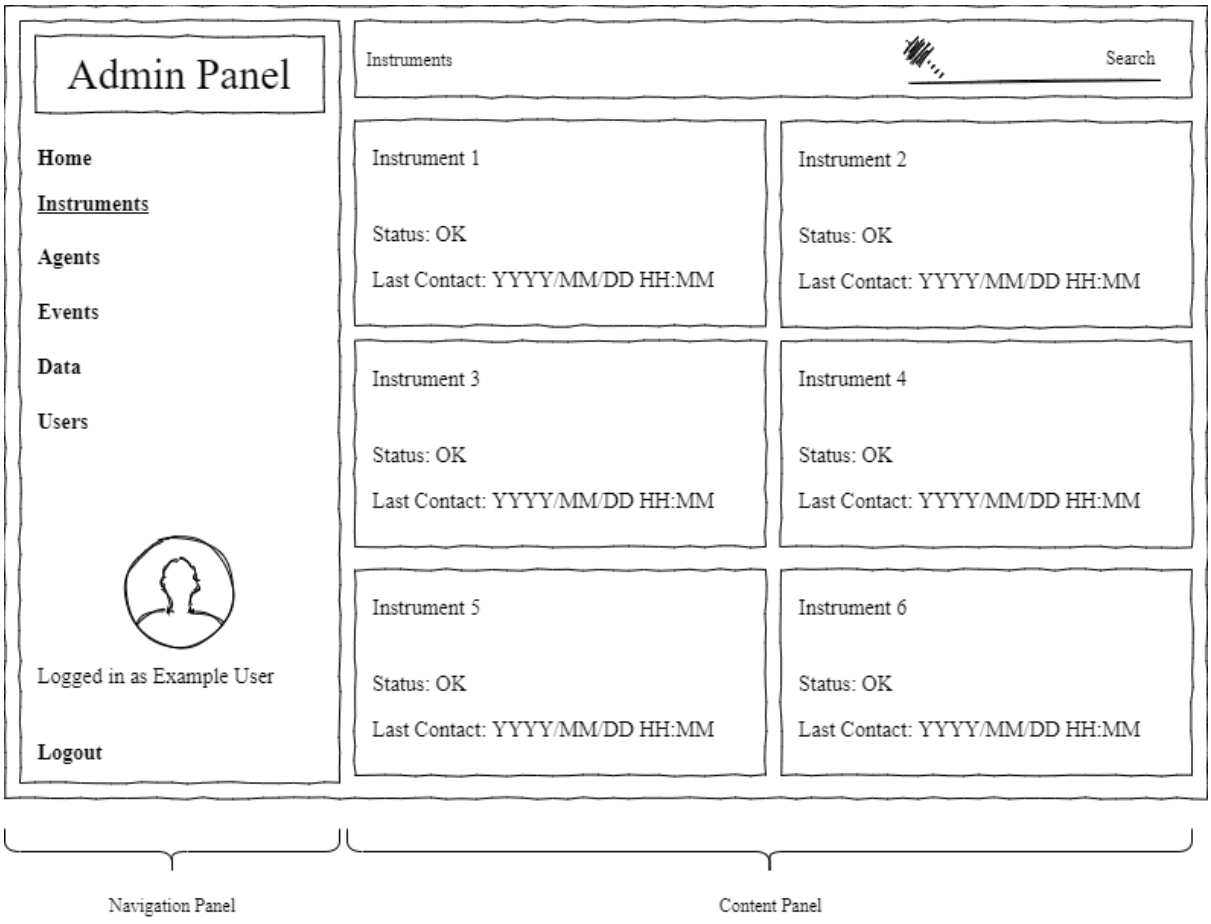

Figure 18: Mock-up of the website's design. A navigation panel is displayed on the left while the selected page is presented on the right.

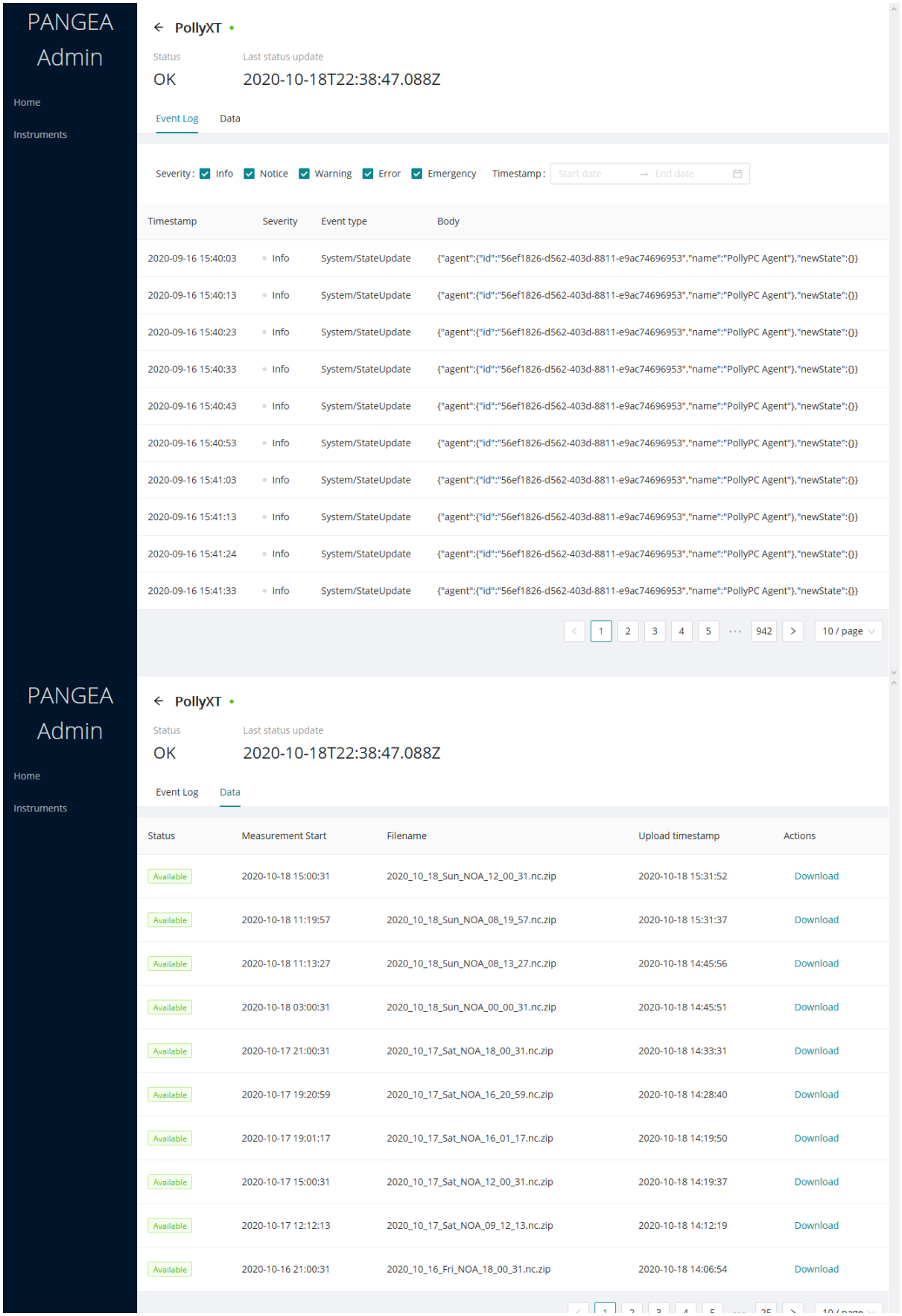

Figure 19: Screenshots of the website's implementation

6. Results/Case study

In order to evaluate the developed platform and its effectiveness, it was installed at the Antikythera Climate Change Observatory. At the time of writing, the remote infrastructure is connected to the internet through an unreliable line and the island's power grid often causes outages. The experiment of using the developed platform to monitor the infrastructure took place during the Summer of 2020. The platform remained in use after the end of the experiment since it has proven itself useful and plans to further develop it started.

6.1. Hosting Infrastructure

The platform was hosted on servers provided by the National Observatory of Athens. The infrastructure consists of a virtualization cluster (based on KVM and oVirt) and secondary NAS. The cluster is made of four nodes and a shared storage appliance, connected to each node with redundant SAS lanes. All network traffic goes through a shared 1Gbit network, including traffic to the NAS. The cluster's architecture is sketched out in Figure 20.

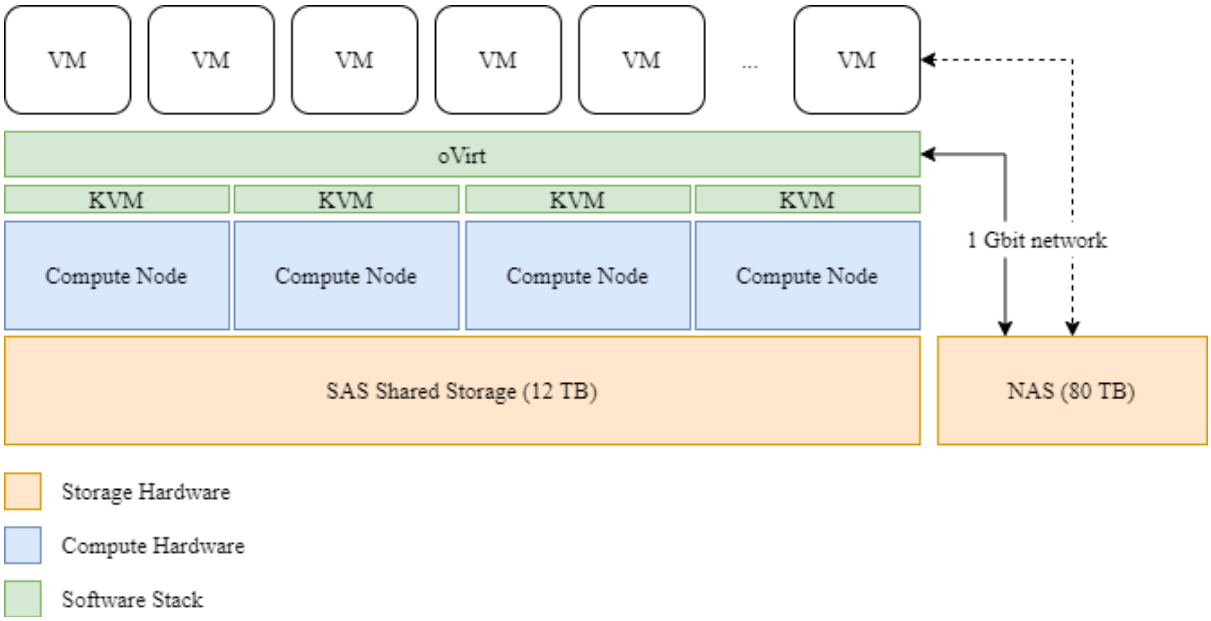

Figure 20: Hosting infrastructure at NOA

While the storage layout is not ideal for an object storage system due to lack of direct access to disks and no available nodes for redundancy, the hosting infrastructure is more than capable to accept the prototype. If the object storage system prove reliable, future hardware installations will take it into account and better accommodate it.

Virtual machines (VMs) are used to host each of the platform's components. Starting with the relational database, two VMs are reserved to run PostgreSQL for high availability. Discussing

PostgreSQL replication and load balancing options could be an entire thesis on its own, so only the selected solution is detailed. The first VM runs the primary database while the second is configured as a “Hot Standby”. The primary database is used for all transactions and internally mirrors them to the secondary. If at any point the primary database fails for any reason, the secondary takes over and the platform continues to operate normally. While the secondary could always be configured to be available for reading data from the database, taking some load off the primary, this is deemed unnecessary for the scale of this application.

Similarly, two VMs are used to host the object storage system. Each VM has four virtual hard disks for a total of 4TB storage. The physical data is stored on the same physical machine (the storage appliance), negating all benefits of MinIO in clustered mode. This might seem pointless but running the platform as if dedicated infrastructure exists can provide valuable insight into the administration needs it will require in the future. If a component proves to be a burden, requiring constant maintenance, its use should be re-evaluated.

The platform’s backend is also running on two VMs, configured as Docker hosts, with each host running the backend container once. In front of these two hosts, another VM running the Apache webserver [49] in load-balancing configuration is used. Each time a user sends a request for the platform, the webserver redirects the request into one of the two hosts, spreading the computational load between them. The webserver uses a “health-check” to determine if the application nodes are functioning correctly. The health-check consists of making a request to the application and checking that the response is what expected. In case of abnormal behavior or failure to respond, the specific node is taken out of the system and stops being used to serve users [50].

Finally, the MQTT broker also runs on the webserver VM. This final machine is made highly available through the virtualization cluster’s hypervisor. The VM is constantly monitored by the hypervisor through health-checks and if a crash is detected, the VM is forcefully restarted. In case the whole virtualization server fails, one of the others in the cluster will automatically start a new VM to replace the webserver and the MQTT broker.

6.2. Supported Instruments

The observatory hosts (at the time of writing), three instruments with 24/7 operation. For two of these instruments, an addon was written to enable advanced features, while for the third it was deemed not worth the effort, since the instrument’s health is currently monitor through

other means. In the next sections some details about each instrument are discussed and how it got incorporated into the platform.

6.2.1. PollyXT

Figure 21: The PollyXT system at the Antikythera Observatory. Photo source: ReACT Photo Archive

The observatory is home to a third-generation PollyXT multiwavelength Raman polarization lidar [51] (a nighttime photo is displayed in Figure 21). Lidars[4] were among the first applications of lasers after 1962, with the first published results using an atmospheric lidar being in 1963 [52]. Since then, lidars have followed improvements in laser and electronics and have provided invaluable insight into the atmosphere. Lidars, along with radars, are one of the backbones of atmospheric research. In principle, all atmospheric lidars operate by casting a laser beam into the atmosphere and, using a telescope, measuring the backscattered photons. While modern systems do a lot more (e.g. rely on doppler broadening to measure temperature and wind speed, like the European's Space Agency Aeolus satellite [53]), the focus of this document is how to monitor such a system, not its functionality.

PollyXT specifically uses photomultipliers to detect backscattered photons. An FPGA is used to count the number of photons per laser-shot and an embedded Linux computer does temporal averaging on the data. Finally, a general-purpose computer can read the data from the

[4] While LiDaR stands for "Light Detection And Ranging", it is often used as a word instead of an acronym, similarly to laser and radar.

embedded one using Ethernet [51]. Control of the components is achieved through serial connections (RS-232) to the general-purpose computer.

At this computer, the measurements are collected and post-processed, before being saved as NetCDF files by the instrument's software. The platform's agent continuously monitors the output directory for these NetCDF files and uploads them to the datacenter. The laser's cooling water temperature is measured through a serial port and periodically reported to the platform. Lastly, the ambient temperature of the container housing the lidar is measured with a networked thermometer and included in the status updates.

6.2.2. Sun Photometer

Figure 22: Photo of the CE318 photometer installed at the Antikythera Observatory. The platform has been painted black to avoid reflections.

The Antikythera observatory, being part of NASA's AErosol RObotic NETwork (AERONET), hosts a CIMEL Sun Photometer (CE318). The photometer consists of a robot arm holding a sun-tracking spectrometer. By measuring the spectral properties of sun radiation it can estimate the atmosphere's optical depth[5] and by subtracting effects of known gases, aerosol optical depth

[5] Optical depth is a measure of the ratio of incident to transmitted radiation through the atmosphere (or another material)

(AOD) [11]. A photo of the instrument installed at the Antikythera Observatory is shown in Figure 22.

Being in operational use since before 1997, the CE318 series of sun photometers feature an extremely high level of operational autonomy. The instrument takes measurements on a schedule and transmits them through RS232 to a connected general-purpose computer. In case of a connection failure to the computer, measurements can be stored for multiple days and transmitted later. Provided software runs on the connected computer to store the data in "K7" files that are periodically uploaded to AERONET's servers. In case the internet connection is severed, the files are uploaded as soon as the connection is re-established.

Given the instrument's high levels of automation, there is not much to be monitored by the platform. Simply checking whether the instrument's computer, and subsequently, software, are running correctly is sufficient. The file uploading routine already in place in the context of AERONET was not replaced by the platform's, since it performed very well on its own and due to the fact that the raw instrument measurements are of low interest without AERONET's automated products.

6.2.3. Electric Field Mill

Figure 23: Photo of the Electric Field Mill at the Antikythera Observatory. The instrument has been since moved to another part of the island.

In order to measure atmospheric electric fields, a JCI 131 FM field mill is used at the Antikythera Observatory. A field mill operates by measuring the charge on a metal plate inside a

shielded housing. To measure changes in the atmosphere's electric field, a rotating shutter is used to block the sensor plate before each measurement. While the shutter is open, the sensor plate charges up from the electric field and when the shutter closes, the charge is measured and the plate is subsequently discharged [54]. The instrument's output is a differential voltage signal, measured by a Pico Technology ADC-20 data logger.

The data logger's software can be configured to continuously measure the output voltage and store it in hourly files. These data files are in a binary format but specifications are available from the manufacturer [55]. The agent is programmed to read the PLW files and upload the measurements in a human readable format (CSV).

7. Conclusions

In this final chapter, the main contributions of this thesis are presented, alongside some conclusions about the work done. Some notes and ideas about future work to expand the functionality of the platform are also presented.

7.1. Thesis Contributions

Despite major research infrastructures having highly automated procedures for conducting measurements and monitoring the equipment, no standard toolset exists for handing these tasks. The aim of this thesis was to design such a tool and implement a prototype. The proposed design is generalized enough to be able to handle the requirements of different research stations without needing to rewriting parts of the code.

The approach the author took involved implementing the platform as a web application, accompanied by a client (agent) to monitor each instrument, provided many advantages. Focusing on the web application aspect, such an application is easily accessible to the users through a web browser. This fits well into modern day-to-day workflows, allowing the users to monitor the infrastructure using any consumer device (computer, tablet, or phone). Files can be browsed directly through the website, avoiding the need to use a secondary application (e.g. FTP Client) to access the repository.

Regarding the agent, the cross-platform application that is designed and prototyped in this thesis offers great flexibility. It can be used to monitor different kind of equipment; no technical reason exists that prevents its application on things other than instruments. Using the add-on system, it can be parametrized to monitor the entirety of a station's infrastructure, including supporting components such as climate control units.

Finally, the prototype was deployed at the National Observatory of Athens' climate change observatory of Antikythera with success. The platform continuously monitored three instruments and due to the convenience, measurements collected before its development have been retroactively added to its archive. At the time of writing, the archive is approximately 100GB and is effortlessly managed by the platform. The application was well-received by the observatory's operators and further development is planned to expand the available functionality. Some ideas are presented in the next section.

7.2. Future Work

While the platform proved useful in everyday management operations of a remote research infrastructure, there is still much area left uncovered. Firstly, the data transfer procedure could be improved in regard to reliability and efficiency. Currently the agent will attempt to upload a file using a HTTP request and will keep repeating the upload if it fails. An improved approach could divide the files into small fixed-size chunks (e.g. 1MB each), hash them individually and upload them one-by-one. By making smaller uploads, a smaller piece of the file will need to be retransmitted in case of failure. This is similar to the mechanism the popular file transfer protocol BitTorrent uses to exchange files [56].

The platform already communicates with the instruments through a bi-directional channel (the MQTT broker) and the agent already contains addons with instrument-specific code. A logical next step would be to implement instrument control commands so that the users can execute common tasks from the website. Some commands could be “start measurement”, “restart system”, etc. The effectiveness of such a feature relies on the manual implementation of commands across the set of instruments. Nonetheless, it could prove useful, especially in cases of “stubborn” instruments or software that do not support any kind of automation on their own. Most of the common operations could be made available to authorized users through the website, delegating the need to connect to the instrument through alternative means (e.g. remote desktop) only in specific situations.

A feature that fits well within the platform’s automation spirit is automatic processing of data. Processing pipelines could be defined by the platform’s users that are triggered by new file arrivals. The results of these pipelines will also be stored in the data archive and links will be created between the original files and the products. This feature could be used to aid in data browsing by automatically creating figures and plots or create whole new datasets by combining measurements with modeling results. By taking advantage of the cloud technologies utilized to implement the prototype, a system that executes processing scripts on their own container can be utilized, guaranteeing a pristine environment each time a new file is processed. Each processing step will be monitored, and all the output archived, so in case of failure a user can deduct what went wrong and optionally repeat the procedure.

Finally, since the platform keeps the station’s data archive, a data browsing portal could be developed to allow internal and external users to browse the datasets and, optionally, download data to use for their own research. Such a portal could cooperate with the platform to create

personalized repositories with the requested data for download. By using a second website, all the “gruesome” details of station management can be hidden away from external visitors which are only concerned with the data. Such a portal would track visitation statistics, data download views and counts, etc. This could provide insight in how the data is used and which datasets are popular in the scientific community.

8. References

[1] J. Gantz and D. Reinsel, 'THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East', p. 16, 2012.

[2] S. Yin and O. Kaynak, 'Big Data for Modern Industry: Challenges and Trends [Point of View]', *Proceedings of the IEEE*, vol. 103, no. 2, pp. 143–146, Feb. 2015, doi: 10.1109/JPROC.2015.2388958.

[3] N. Smale, K. Unsworth, G. Denyer, and D. Barr, 'The History, Advocacy and Efficacy of Data Management Plans', Scientific Communication and Education, preprint, Oct. 2018. doi: 10.1101/443499.

[4] 'Modern research on the Greek island where computers were born', *European Investment Bank*. https://www.eib.org/en/stories/greece-climate-change-infrastructure (accessed Oct. 11, 2020).

[5] 'ReACT - Equipment'. https://react.space.noa.gr/index.php/infrastructure/equipment (accessed Oct. 11, 2020).

[6] 'COVID-19 campaign'. https://www.earlinet.org/index.php?id=covid-19 (accessed Oct. 11, 2020).

[7] 'ReACT - Aerosols From Canadian Fires Stretch from Alaska to the Mediterranean'. https://react.space.noa.gr/index.php/news-events/82-aerosols-from-canadian-fires-stretch-from-alaska-to-the-mediterranean (accessed Oct. 11, 2020).

[8] 'ReACT - Aerosol particles from Etna are monitored with the PollyXT lidar system of NOA-ReACT over Antikythera'. https://react.space.noa.gr/index.php/news-events/83-aerosol-particles-from-etna-are-monitored-with-the-pollyxt-lidar-system-of-noa-react-over-antikythera (accessed Oct. 11, 2020).

[9] 'ARM Research Facility'. https://www.arm.gov/ (accessed Oct. 06, 2020).

[10] J. H. Mather and J. W. Voyles, 'The Arm Climate Research Facility: A Review of Structure and Capabilities', *Bull. Amer. Meteor. Soc.*, vol. 94, no. 3, pp. 377–392, Mar. 2013, doi: 10.1175/BAMS-D-11-00218.1.

[11] B. N. Holben *et al.*, 'AERONET—A Federated Instrument Network and Data Archive for Aerosol Characterization', *Remote Sensing of Environment*, vol. 66, no. 1, pp. 1–16, Oct. 1998, doi: 10.1016/S0034-4257(98)00031-5.

[12] 'AERONET Data Display Interface - WWW DEMONSTRAT'. https://aeronet.gsfc.nasa.gov/cgi-bin/draw_map_display_aod_v3?long1=-180&long2=180&lat1=-90&lat2=90&multiplier=2&what_map=4&nachal=1&formatter=0&level=3&place_code=10&place_limit=0 (accessed Oct. 13, 2020).

[13] W. D. Cesare *et al.*, 'The Broadband Seismic Network of Stromboli Volcano, Italy', *Seismological Research Letters*, vol. 80, no. 3, pp. 435–439, May 2009, doi: 10.1785/gssrl.80.3.435.

[14] 'Needle in a haystack: efficient storage of billions of photos', *Facebook Engineering*, Apr. 30, 2009. https://engineering.fb.com/core-data/needle-in-a-haystack-efficient-storage-of-billions-of-photos/ (accessed Oct. 13, 2020).

[15] 'Zenodo - Research. Shared. - About'. https://about.zenodo.org/ (accessed Oct. 14, 2020).

[16] ‘Zenodo - Infrastructure’. https://about.zenodo.org/infrastructure/ (accessed Oct. 14, 2020).

[17] ‘Infrastructure architecture — Invenio 3.3.0 documentation’. https://invenio.readthedocs.io/en/latest/architecture/infrastructure.html (accessed Oct. 14, 2020).

[18] Dan van der Ster, ‘Building Scale-Out Storage Infrastructures with RADOS and Ceph’, presented at the XLDB 2017: 10thExtremely Large Databases Conference, Clermont-Ferrand, France, Oct. 11, 2017, Accessed: Oct. 14, 2020. [Online]. Available: https://indico.in2p3.fr/event/14490/contributions/56328/attachments/44366/54972/vandester-ceph-XLDB-2017.pdf.

[19] ‘Intro to Ceph — Ceph Documentation’. https://docs.ceph.com/en/latest/start/intro/ (accessed Oct. 05, 2020).

[20] M. D. Wilkinson *et al.*, ‘The FAIR Guiding Principles for scientific data management and stewardship’, *Scientific Data*, vol. 3, no. 1, Art. no. 1, Mar. 2016, doi: 10.1038/sdata.2016.18.

[21] ‘SPA (Single-page application)’, *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Glossary/SPA (accessed Oct. 10, 2020).

[22] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, ‘Object storage: the future building block for storage systems’, in *2005 IEEE International Symposium on Mass Storage Systems and Technology*, Jun. 2005, pp. 119–123, doi: 10.1109/LGDI.2005.1612479.

[23] ‘EMC Marks Five Years of EMC Centera Innovation and Market Leadership’. https://corporate.delltechnologies.com/en-us/newsroom/announcements/2007/04/04182007-5028.htm (accessed Oct. 03, 2020).

[24] ‘Amazon S3 – The First Trillion Objects’, *Amazon Web Services*, Jun. 12, 2012. https://aws.amazon.com/blogs/aws/amazon-s3-the-first-trillion-objects/ (accessed Oct. 03, 2020).

[25] C. Huang *et al.*, ‘Erasure Coding in Windows Azure Storage’, 2012, pp. 15–26, Accessed: Oct. 03, 2020. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.

[26] ‘Dell EMC ECS Object Storage’. https://www.delltechnologies.com/sr-me/storage/ecs/index.htm (accessed Oct. 10, 2020).

[27] ‘Ceph iSCSI Gateway — Ceph Documentation’. https://docs.ceph.com/en/latest/rbd/iscsi-overview/ (accessed Oct. 10, 2020).

[28] OASIS Open, ‘MQTT Specification, Version 5’. 2019, Accessed: Oct. 04, 2020. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[29] M. Jazayeri, ‘Some Trends in Web Application Development’, in *Future of Software Engineering (FOSE ’07)*, May 2007, pp. 199–213, doi: 10.1109/FOSE.2007.26.

[30] ‘Node.JS Repository’. https://github.com/nodejs/node (accessed Oct. 04, 2020).

[31] K. Lei, Y. Ma, and Z. Tan, ‘Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js’, in *2014 IEEE 17th International Conference on Computational Science and Engineering*, Dec. 2014, pp. 661–668, doi: 10.1109/CSE.2014.142.

[32] ‘Documentation | NestJS - A progressive Node.js framework’, *Documentation | NestJS - A progressive Node.js framework*. https://docs.nestjs.com (accessed Oct. 06, 2020).

[33] ‘Comment on HackerNews by the author of Sequel, an ORM tool for Ruby’. https://news.ycombinator.com/item?id=14662068 (accessed Oct. 07, 2020).

[34] ‘The case against ORMs | Korban.net’. https://korban.net/posts/postgres/2017-11-02-the-case-against-orms/ (accessed Oct. 07, 2020).

[35] ‘The case against ORM Frameworks in High Scalability Architectures - High Scalability -’. http://highscalability.com/blog/2008/2/2/the-case-against-orm-frameworks-in-high-scalability-architec.html (accessed Oct. 07, 2020).

[36]. ‘NET Core’. https://dotnet.microsoft.com/ (accessed Oct. 04, 2020).

[37] cartermp, .‘NET architectural components’. https://docs.microsoft.com/en-us/dotnet/standard/components (accessed Oct. 05, 2020).

[38] ECMA International, ‘Standard ECMA-335: Common Language Infrastructure (CLI)’. 2012, [Online]. Available: https://www.ecma-international.org/publications/standards/Ecma-335.htm.

[39] *containers/podman*. Containers, 2020.

[40] *typeorm/typeorm*. typeorm, 2020.

[41] M. Inc, ‘MinIO | Enterprise Grade, High Performance Object Storage’, *MinIO*. https://min.io (accessed Oct. 05, 2020).

[42] ‘React – A JavaScript library for building user interfaces’. https://reactjs.org/ (accessed Oct. 10, 2020).

[43] ‘DOM Standard’. https://dom.spec.whatwg.org/#what (accessed Oct. 10, 2020).

[44] R. A Light, ‘Mosquitto: server and client implementation of the MQTT protocol’, *JOSS*, vol. 2, no. 13, p. 265, May 2017, doi: 10.21105/joss.00265.

[45] ‘Frequent disconnect/connect issues even on localhost · Issue #289 · chkr1011/MQTTnet’, *GitHub*. https://github.com/chkr1011/MQTTnet/issues/289 (accessed Oct. 09, 2020).

[46] *hivemq/hivemq-community-edition*. HiveMQ - Enterprise MQTT Broker, 2020.

[47] Christian, *chkr1011/MQTTnet*. 2020.

[48] *ReactTraining/react-router*. React Training, 2020.

[49] ‘Welcome! - The Apache HTTP Server Project’. http://httpd.apache.org/ (accessed Oct. 11, 2020).

[50] ‘mod_proxy_hcheck - Apache HTTP Server Version 2.4’. https://httpd.apache.org/docs/2.4/mod/mod_proxy_hcheck.html (accessed Oct. 11, 2020).

[51] R. Engelmann *et al.*, ‘The automated multiwavelength Raman polarization and water-vapor lidar Polly^{XT}: the neXT generation’, *Atmos. Meas. Tech.*, vol. 9, no. 4, pp. 1767–1784, Apr. 2016, doi: 10.5194/amt-9-1767-2016.

[52] U. Wandinger, ‘Introduction to Lidar’, in *Lidar: Range-Resolved Optical Remote Sensing of the Atmosphere*, C. Weitkamp, Ed. New York, NY: Springer, 2005, pp. 1–18.

[53] A. Stoffelen, G. J. Marseille, F. Bouttier, D. Vasiljevic, S. de Haan, and C. Cardinali, ‘ADM-Aeolus Doppler wind lidar Observing System Simulation Experiment’, *Quarterly Journal of the Royal Meteorological Society*, vol. 132, no. 619, pp. 1927–1947, 2006, doi: 10.1256/qj.05.83.

[54] Mission Instruments, ‘Electric Field Mill Operation’. Accessed: Oct. 11, 2020. [Online]. Available: http://www.missioninstruments.com/pdf/fm_op_rev1d_0106.pdf.

[55] J. Bronks, ‘PicoLog User’s Guide’, p. 140.

[56] The BitTorrent community, ‘BitTorrent Specification/Metainfo File Structure’. https://wiki.theory.org/BitTorrentSpecification#Metainfo_File_Structure (accessed Oct. 11, 2020).