SOGS-API: An API for Satellite Data Retrieval

Markus Silvennoinen, 1800325
Master's Thesis in Computer Science
Supervisor: Dragos Truscan
Advisor: Timo Ryyppö (FMI)
Faculty of Science and Engineering
Åbo Akademi University
2021

Abstract

The Finnish Meteorological Institute (FMI) has a set of satellite data retrieval tools, which are used in SOGS-API. The full name is *Sodankylä Ground Station API*. Initially, the API had been put together quickly, and the problem was that it lacked features, contained bugs and had been implemented in a technology, which was about to be outdated. This thesis is about solving the problem by rewriting the API and adding the missing features and adapting existing ones. The goal is that the API is easy to use and access through SSH or web protocols, and it should be able to handle traffic and threats.

SOGS-API and its tools have been mainly implemented using Python 3 in a Linux environment. Most features of SOGS-API are used through a command line interface. Customer needs may change during and after the thesis, so it is important to keep the API maintainable. The API is tested and validated on a test server before it goes operational.

The following objective is formulated:

Rewrite SOGS-API to comply with new requirements, such that the API is reliable, secure and maintainable in order to let FMI provide third parties access to the organization's satellite retrieval services.

The rewriting of SOGS-API include adapting existing monitoring, visualization and reporting tools to work with the API. Some of the existing tools had not been designed for a wider audience and required safety and performance enhancements, and some had features that were removed because they were not essential for the API. The reliability and security of the API are evaluated with automated scripts, and by letting FMI employees test it. The API should be able to handle normal traffic, which is expected to be at most 1000 requests per minute, and most common threats, such as *password cracking* and *Distributed Denial of Service* attacks. The API has been implemented in a way that makes security breaches unfeasible. The maintainability of the API is preserved by following good programming practices. The retrieval services include tools for checking satellite overpass schedules, signal data and antenna status data.

Keywords: API, satellite data retrieval, product quality.

Contents

Preface

The work on the project started in December 2019, with a ten-month contract. In November, preparations for the project were made, such as the specification of the SOGS-API, receival of a work computer, and testing if the computer can be logged into and if it can be connected to the FMI intranet through VPN. Everything worked as expected.

I learned the basic infrastructure of FMI during my summer internship in Sodankylä. This was also the time when me and my manager discussed the possibility of writing my thesis for Finnish Meteorological Institute.

Acknowledgments

I would like to thank Finnish Meteorological Institute for this opportunity. I would like to express my appreciation to my manager and advisor Timo Ryyppö at FMI, for his guidance and advice, and for helping to keep the API working when a bug appears. Many thanks to my supervisor Dragos Truscan at Åbo Akademi University, for his time reading, correcting and guiding me. Finally, I would like to thank the love of my life, for her understanding and support.

Abbreviations

ACU	Antenna Control Unit
API	Application Programming Interface
CSV	Comma Separated Values
DOM	Document Object Model
FEP	Front-end Processor
FMI	Finnish Meteorological Institute
FTP	File Transfer Protocol
GPS	Geographic Point System
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
MOC	Ministry of Communication
PNG	Portable Network Graphics
SCC	Station Control Computer
SCP	Secure Copy
SCU	Servo Control Unit
SOGS-API	Sodankylä Ground Station API
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
UTC	Coordinated Universal Time
VPN	Virtual Private Network
XML	Extensible Markup Language

1. Introduction

This chapter contains the background for this thesis. It presents the main reasons and goals for this project. This master's thesis was carried out at Arctic Space Centre of the Finnish Meteorological Institute (FMI-ARC) located in Sodankylä, Finland. Most of the work was done remotely. The ground station operations team consists of the group leader, main operator, two system planners and a research engineer. The programming work in this project was mostly done by me. The basic design had already been done for the initial version of the API. During the development, some new design decisions were made on my part, which were approved by the manager.

Finnish Meteorological Institute (FMI) is a research and service institution under the Ministry of Transport and Communications of Finland. It is the largest institution related to space activities in Finland. FMI has two ground stations, one in its headquarters in Helsinki, and one in the Arctic Space Centre in Sodankylä.

Sodankylä Ground Station is a National Satellite Data Center. Located in the North, Sodankylä Ground Station is ideal for receiving data from polar orbiting satellites. The ground station has three antennas, an operations center and infrastructure for processing and archiving satellite data. Satellite orbits and Sodankylä Ground Station are discussed in more detail in chapters 2 and 3.

1.1 Problem Description

An application programming interface (API) is a set of routines, protocols and tools that abstract implementation and provide only the functions and objects that the developer needs [1] [2]. The meaning of the term *API* has changed slightly over the years, from being software libraries for e.g. hardware independent display in the 1960s [3], to modern web API services [2].

APIs are a useful way to provide functionality to client programs. They can make the development of applications faster and easier by abstracting away complicated procedures. For instance, the client can receive data from a satellite through an API without knowing anything about the communication between the satellite, antenna and ground station. A

concrete example of an API is the YouTube Data API, which provides functions for programmatically reading and editing resources, including videos, of a linked YouTube user account [4].

FMI provides the *SOGS-API* (Sodankylä Ground Station API) for accessing the organization's satellite data retrieval services. SOGS-API consists of several smaller programs. Two of the programs are services that provide the features that the customers can access. These are the *main program of SOGS-API*, which logs and sends antenna status data through UDP, and the *scheduling feature* that enables customers to upload satellite pass plans for antenna scheduling through HTTP requests. The rest are programs that support the operation and maintenance of the API.

The antenna status data that are sent to customers are the ACU metrics and S band modem metrics. ACU metrics are received from the antenna station control computer (SCC) through TCP. The ACU metrics are numbered parameters with short values, such as numbers. The meanings of these values are defined in the antenna manufacturer's manual for the interface. The raw metrics are time-consuming to interpret. Therefore, it is more common to let software read the metrics. Within FMI, tools for visualization and automated issue reporting have been made for the metrics coming from the SCC. Appendix A has an example of raw ACU metrics, and how its parameters are visualized.

S band is defined by IEEE as the part of the microwave band of the electromagnetic spectrum covering frequencies 2-4 GHz and is mainly used for satellite communication. An S band modem converts radio frequencies and processes signals.

The S band modem metrics are received through HTTP using a Python program that was delivered by the manufacturer of the S band modem. This program returns the S band modem metrics in JSON format. The modem metrics are human readable as is, but the most important values, signal strength (EbnoMeas), symbol lock and carrier lock are also plotted for an overpass, making it easier to spot issues in satellite data receival. Appendix B has an example of S band modem metrics and a plot. The plot shows that the signal strength was good and stable during the overpass. Sudden notches or bad signal strength during an overpass are signs of trouble.

Initially, SOGS-API had been put together quickly. It lacked features and contained bugs. It had a main program that logged and sent antenna status data through UDP, with basic exception handling, and a separate program for the scheduling feature, which had not been put into use. The main program had been written in Python 2, which was about to be outdated when this project started. During the project, the API was almost entirely rewritten, and Python 3 was used as the programming language. The related tools *Satellite*

Data Flow Monitor, *Acquisition Issue Reporter* and *Antenna Status Monitor* were also worked on during the project.

1.2 Purpose of the Thesis

The purpose of this thesis is to present SOGS-API. The thesis also serves as additional documentation for SOGS-API, along with the manuals. The purpose of the produced software is to serve FMI and its customers data for quality assurance and maintenance. The data can be used to check if there are issues in the receival of data coming from the satellite. This is important, because valid satellite data are valuable, and are used in e.g. research and to check the ice situation on the sea, which affects the work of the ice breakers.

1.3 Project Goals

The goal is to create a reliable, secure and maintainable API that provides access to the satellite data retrieval services. Only customers and staff have access to the API, and the API is mostly used with a command line interface. The SOGS-API is implemented so that it is easy to make changes later, because the customer's needs can change. These were the requirements during the project:

- to let the customers view antenna schedules without revealing what satellites other customers receive from,
- to schedule the customer's satellite passes,
- to provide information about antenna and component status to customers,
- to automatically check for faults and report them to FMI and customers, and
- to check the delivery of data.

The features and data that customers receive are customer dependent.

SOGS-API is evaluated through measurements of quality attributes. The most important quality attributes for the API are reliability, security and maintainability. Performance and availability to customers were also considered.

Reliability was tested with special scenarios, such as when the power goes out from the antenna, and fuzzing. During the development of the API, the API was in test use on a test server. The API on the operational server was also frequently updated during the project,

and it went through a multitude of special situations, either managing it or failing, and then being developed to manage it. Performance was also measured.

The API was in a secure environment, behind firewalls and with access limited to customers and staff only. The security measures in the new API were to have authentication for customer requests and the *Schedule Manager* tool, as well as local file verification for special commands, such as shutdown, to make misuse even more difficult. The API was further developed to detect and report attempts to crack customer IDs and passwords, and to block requests from IP addresses that show suspicious activity.

To make the API more maintainable, effort was put on code quality, documentation and usability for operators. Configuration files and design principles were used to add modifiability to the API.

Availability to customers was increased by using exception handling and tolerance, and by having tools for the riskiest situation for availability, an update. The effects of updating on availability were further reduced by agreeing on a time window, when the API is updated without notification, and that customers will be notified about updates that are done outside of the time window.

1.4 Benefits

The benefit of having the improved API for satellite data retrieval services is that it supports multiple customers, reports problems through email, and repairs itself. The benefit of the API being reliable is that it can handle several customers and satellites, and it will not break easily in exceptional situations. Security benefits mostly the FMI, by preventing unauthorized activity in the organization's servers, but also the customers, as there will not be anything malicious served through the services, and data will not fall into the wrong hands. Maintainability benefits the FMI by making it easier to correct or update the API. It is also a benefit for the customers, as corrections and updates can be made more quickly and reliably, keeping the services running.

1.5 Related Work

There are other satellite-related APIs [5], but these are not directly competing with SOGS-API, because SOGS-API is tailor-made for the FMI's satellite retrieval system and for FMI's customers.

Macy has done research regarding API security and writes about its challenges and so-

lutions in an article [6]. Macy points out that API security frameworks, adapters and toolkits cannot provide real security. According to Macy, technologies such as API security gateways with locked-down operating systems where no third-party code can be run are immune to vulnerabilities, even in a chipset.

Kleppmann has written about reliability, scalability and maintainability of applications and APIs [7]. Kleppmann writes that reliable software is expected to do what the user expects, tolerating mistakes, performing well enough for its task, and preventing unauthorized access and abuse. Kleppmann splits maintainability into three components: operability, simplicity and evolvability. The operability is dependent on the operational team and the ways routine tasks have been made easier in the software, as well as how well it is documented. Abstraction is a good way to reduce complexity in software, and this also benefits evolvability, which can be further improved with agile tools and patterns.

1.6 Thesis Structure

Chapters two to five review theory about satellites, overpasses, Sodankylä Ground Station, APIs, software architecture, quality attributes and metrics, as well as tools and technologies used. Chapter six presents SOGS-API. Chapter seven evaluates SOGS-API using quality attributes and chapter eight summarizes the result. At the end of the thesis there is a Swedish summary and the appendices.

2. Satellites

A satellite is an object that orbits another object, such as a planet, star or asteroid. The satellite can be natural, such as a planet or a moon, or artificial, which means any objects that has been intentionally placed in orbit. Usually, the word "satellite" refers to a machine that has been launched into orbit [8].

There are several types of satellites with different purposes. Astronomical satellites are used for observing distant planets and galaxies, while Earth observation satellites are used for non-military observations of Earth. Reconnaissance and weaponized killer satellites are used for military purposes. Communication and navigational satellites are used for telecommunication and GPS. Weather satellites are used for observing weather and climate.

2.1 Overpass

A satellite overpass occurs from the time when the satellite receiver acquires the signal from the satellite until the signal is lost. From the satellite receiver's perspective, the overpass roughly occurs while the satellite is above the horizon. This is illustrated in Figure 2.1.

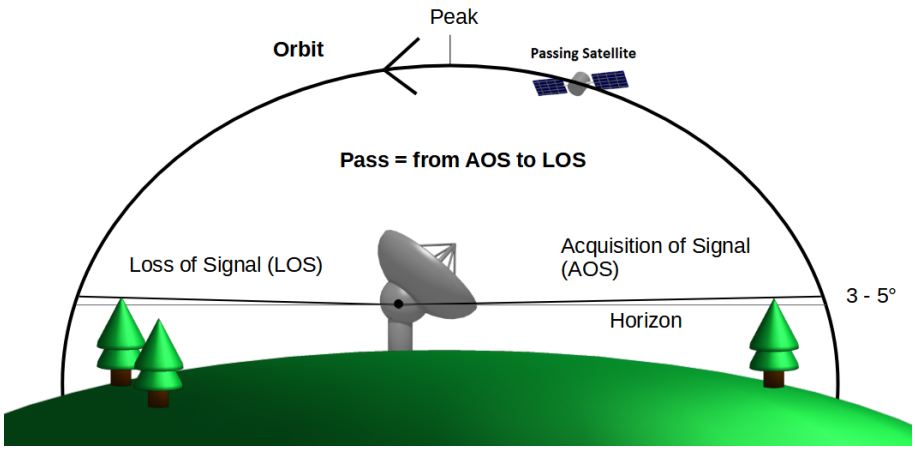

Figure 2.1: Overpass.

2.2 Polar Orbiting vs Geostationary Satellites

Polar orbiting satellites move from pole to pole, with an inclination between 60 and 90 ° to the Equator. The satellite will pass the Equator at a different longitude on each of its orbits. Figure 2.2 demonstrates how polar orbits are used when the inclination is close to 90 °.

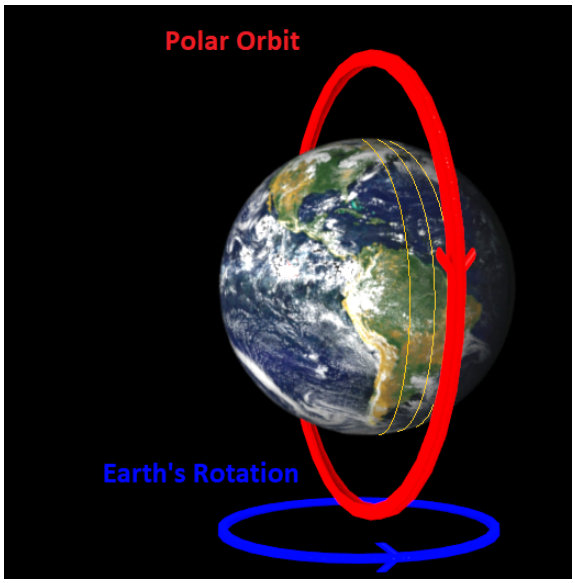

Figure 2.2: Polar Orbit.

A satellite in a geostationary orbit is 35,786 kilometers above Earth's equator, going in the direction of Earth's rotation, as Figure 2.3 shows. The satellite is always above the same location on Earth and can always be seen in the same location in the sky. This also means that Earth-based antennas do not need to rotate to track a geostationary satellite. Therefore, geostationary orbits are especially useful for communications satellites. Geostationary weather satellites are used for real-time monitoring and data collection and navigation satellites for providing a known calibration point and enhancing GPS accuracy.

Geostationary satellites are at altitude 35,786 km above the Equator, because this is where the satellite can stay in the same spot relative to Earth's surface, with minimal need for orbit corrections. The altitude of the geostationary orbit is calculated by setting the centripetal force (F_c) equal to the gravitational force (F_g):

$$F_c = F_g \tag{2.1}$$

$$m_s r\omega^2 = \frac{m_s M_E G}{r^2}, \tag{2.2}$$

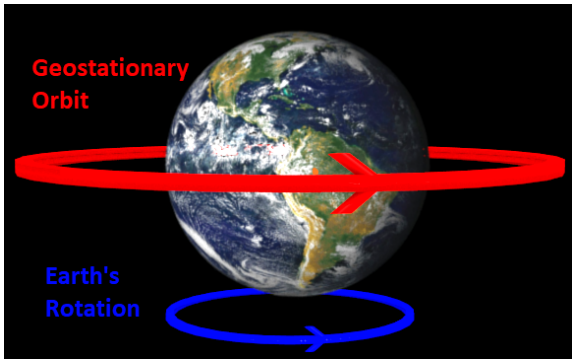

Figure 2.3: Geostationary Orbit.

where m_s is the mass of the satellite, M_E is the mass of the Earth ($5.988 * 10^24$ kg), G is the gravitational constant ($6.673 * 10^-11 Nm^2/kg^2$), r is the distance between the satellite and the center of the Earth, and ω is the angular velocity of the satellite. The angular velocity of the satellite must be the same as Earth's angular velocity. This can be calculated when we know that Earth rotates around its axis a full round (2π rad) in 23 hours 56 minutes and 42 seconds (86164 s).

$$\omega = \frac{2\pi}{T} = \frac{2\pi}{86164s} \approx 7.292 * 10^{-5} s^{-1} \tag{2.3}$$

By solving r from Equation 2.2 and subtracting Earth's radius from it ($R = 6.37 * 10^6 m$), we obtain an equation for calculating the satellite's altitude (h):

$$r = R + h = \sqrt[3]{\frac{M_E G}{\omega^2}} \tag{2.4}$$

$$h = \sqrt[3]{\frac{M_E G}{\omega^2}} - R \tag{2.5}$$

Polar orbiting satellites are on a much lower orbit than the geostationary satellites, e.g. Terra is at 700 km. Polar orbiting satellites are more suitable for taking pictures of the Polar regions. Geostationary satellites can also take pictures towards the poles, but these are taken from a steep angle, making the images inaccurate. The greater altitude of geostationary satellites also reduces the resolution accuracy of the images. Figure 2.4 illustrates the effects of satellite position on image quality.

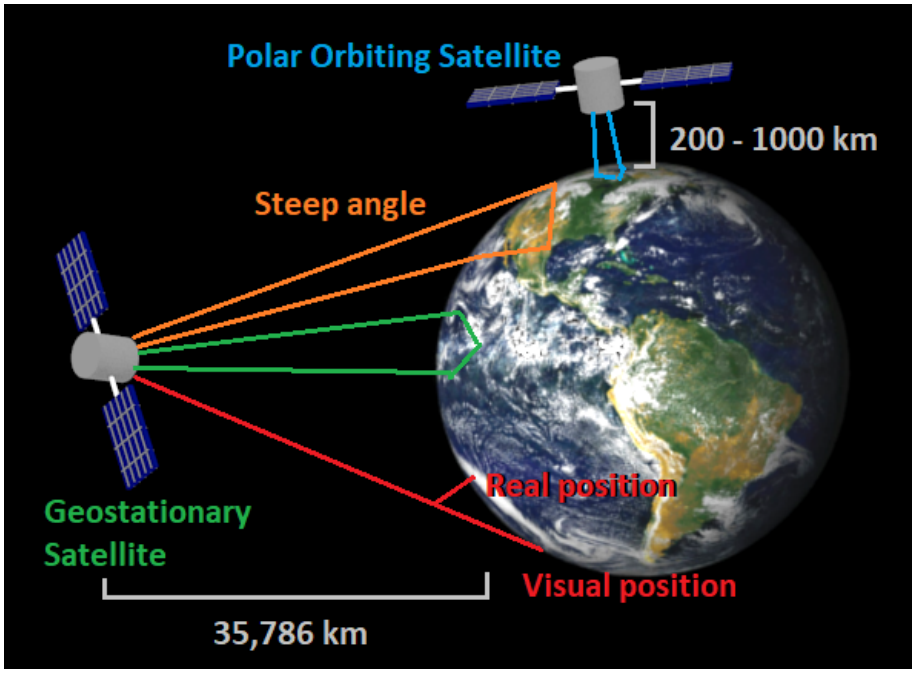

Figure 2.4: Satellite image quality between geostationary and polar orbiting satellites. The red line also shows a positioning error in steep angles.

3. Sodankylä Ground Station

3.1 History

FMI Arctic Space Centre (FMI-ARC) satellite operations started in 1998 and the processing of satellite data started in 2001. In 2003, the Sodankylä satellite operations expanded significantly, when the construction of the first satellite receiver was finished. The antenna received data from satellites Aqua and Terra, which belong to NASA's EOS series. The third EOS series satellite, Aura, was launched in 2004 and receival from it began the same year. EOS series environmental satellites can detect natural phenomena, such as volcano eruptions.

In 2009, the Finnish Government proposed that 3.2 million euro would be invested in updating infrastructure at Sodankylä Ground Station. The proposition was approved, and the infrastructure was updated to be able to receive from NPP/NPOESS (NPOESS Preparatory Project/National Polar-orbiting Operational Environmental Satellite System) satellites. A bigger investment was the construction of a new, bigger antenna in Sodankylä and updating the telecommunications connections to meet the increasing demands in transmission of data. The antenna was finished in time in spring 2011. Another similar antenna was finished at the end of 2016.

3.2 Ideal Location

Sodankylä is in the North (67.48 ° N, 26.53 ° E), which is ideal for receiving from polar orbiting satellites. Up to 10 out of 14 orbits for a satellite are within the visibility cone of Sodankylä per day.

Many of the polar orbiting satellites use the *Direct Broadcast* function when sending data. This function enables the continuous sending of real-time data of measurements. From Finland and its perimeter, data can be received in near real-time, making the data useful not only for research, but for operations too. Data can be received while the satellite is in the coverage area of the ground station. The duration of this state varies and is at most around 15 minutes, which is about 10% of a satellite's orbit.

3.3 Infrastructure

Figure 3.1 shows the infrastructure of Sodankylä Ground Station. FMI has three antennas on the site, and the building of the fourth has been planned. When an antenna receives a signal from a satellite, this signal is converted to a readable format by a demodulator. Then front-end processors send the data to further processing and storage and, finally, the data are sent to relevant stakeholders.

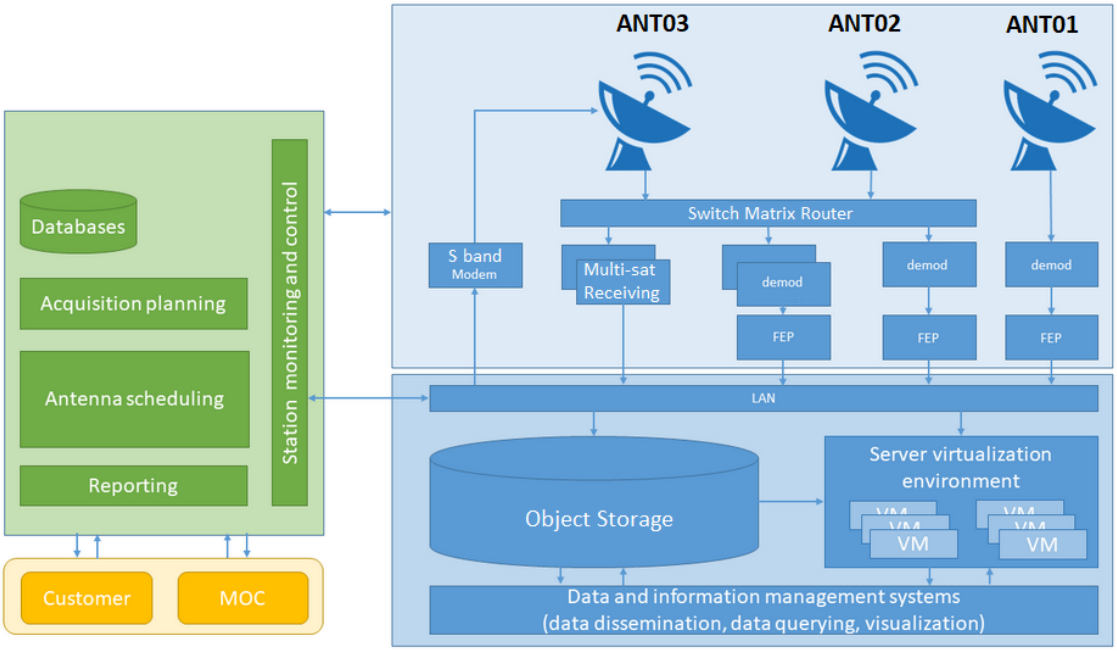

Figure 3.1: Infrastructure of Sodankylä ground station.

The *Antenna control unit* (ACU) is the key part of an *antenna control system* [9]. It is a computer and user interface that processes digital signals and performs calculations for the purpose of tracking a satellite. It also monitors and controls antenna brakes, interlocks and feed status. The ACU communicates with the *servo control unit* (SCU) [10], which is responsible for antenna movement. The block diagram in Figure 3.2 shows how the components in an antenna control system interact.

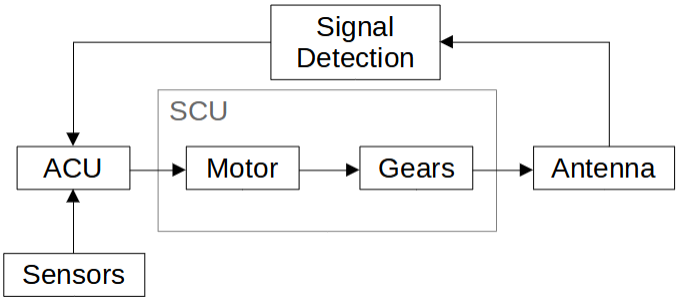

Figure 3.2: Antenna control system.

4. Software Architecture and Quality Attributes

This chapter presents software architecture and quality attributes. Quality attributes are non-functional requirements, which are more dependent on the architecture than functional requirements [11].

4.1 Software Architecture

Software architecture means the basic structure of a software system and the way it is built up [12]. Abstraction is a fundamental part of architecture and complex systems contain many levels of abstraction, with their own architectures [2].

4.1.1 History

In the 1960s, the challenges of developing large-scale software systems were recognized [13]. This led to research of *Software design* in the 1970s and resulted in the development of computer-aided software engineering tools [14]. In the 1980s, the research focused more on integrating design and implementation. At that time, there were also improvements in the way software systems are described and analyzed through formal descriptive techniques and sophisticated notions of typing. In the 1990s, the term *architecture* was used instead of *design* when the software design process contains notions of codification, abstraction, standards, formal training and style [15].

4.1.2 Design Principles and Patterns

Design principles and patterns have their roots in architecture, where they have been used to solve recurring problems. The same principle applies for design patterns in software architecture [16].

In the context of software architecture, a pattern is like a recipe for creating a desired set of interactions among objects [17]. The patterns are a useful way to abstract complex interactions between objects [2]. Design principles in software architecture are good guidelines

for implementing software and many design patterns follow at least some of these guidelines. Design principles encourage to write reusable code that is easy to manage [16]. SOGS-API does not use any clear patterns, but it does follow design principles.

4.2 Quality Attributes and Metrics

Quality attributes, or non-functional requirements, are measurable or testable properties of a system. They indicate how well the system satisfies stakeholders' needs. The most common important quality attributes in software-reliant systems are availability, interoperability, modifiability, performance, security, testability and usability. Other product quality attributes are variability, portability, development distributability, scalability, deployability, mobility, monitorability and safety. There are also non-product related quality attributes, such as conceptual integrity of the architecture, quality in use, and marketability. Sometimes, new quality attributes are defined, and for those, general guidelines, such as the one defined by Bass et al. pp. 196-199 [18], can be used.

Next, we will review metrics that have been used for evaluating SOGS-API.

4.2.1 Halstead Metrics

Halstead metrics is a method from the 1970s for measuring code complexity and maintainability. It takes the total number of operators, total number of operands, number of unique operators and number of unique operands. With these, program length, program vocabulary, volume, difficulty and effort can be calculated. From these, the time to implement and the number of delivered bugs can be estimated [19].

The *time to implement* (T) estimation is calculated with the formula:

$$T = \frac{E}{S} \tag{4.1}$$

where E is the effort and S is the Stroud number, which indicates the number of mental discriminations per second by the human brain, and this is set to 18 for software scientists [20].

The number of delivered bugs (B) is estimated with the formula:

$$B = \frac{E^{\frac{2}{3}}}{3000} \tag{4.2}$$

where E is the effort.

The most useful Halstead metrics are the volume, time to implement, and number of delivered bugs. For the volume, a guideline is that it should be at most 8000 for one file [19] and there should only be two bugs per file [21] [19].

4.2.2 IF Ratio

IF-ratio is a value for measuring legibility and structuredness of code. It is expressed as number of *if statements* per 1000 lines of code (IFs/kLOC). A good result is 20-30 IFs/kLOC, and 60 IFs/kLOC or above indicates that code is difficult to analyze. The latter is common in larger software systems [22].

4.2.3 Product Failure Rate

A widely used [23] metric for reliability is the product failure rate, which can be calculated with the formula:

$$\lambda = \frac{F}{N * T} \tag{4.3}$$

[24], where F is the total number of failures in all N installations during time period T, which is usually measured in days, assuming that the product is used an equal amount of time each day [23].

A more radical approach to test reliability is to try to cause the software to crash. One way to do this is through fuzzing. Fuzzing, or fuzz testing, is a method of software testing where bugs are detected with generated random input. The idea is that, with enough random input, there will be a sequence that leads to a crash. This method is cost-effective and suitable for released or near-released versions of software. [25].

A program that generates random input is called a *fuzzer*. There are different types of fuzzers, such as command-line, environment variable, file format, network protocol and web application fuzzers. There are also different kinds of fuzzing methods, such as random and mutating. The random method uses random data, while the mutating method starts with valid data and changes parts of it [26].

Fuzzing has proved to be an effective way to find vulnerabilities in software. For instance, Microsoft used fuzzing to detect vulnerabilities in Internet Explorer and MS Office [26]. Tools for doing fuzzing tests have been built, such as SAGE [27]. Any effective fuzzing tool should be able to reproduce results. Another desirable, if not required, feature is the documentation of the results [26].

4.2.4 Performance

There are many performance testing tools, such as LoadNinja, Apache JMeter and WebLOAD [28]. Performance testing can also be as simple as measuring execution times of programs or its components. As the execution time can vary between runs, it is a good practice to do several measurements and calculate the average time. Python programs can be profiled using the built-in module *cProfile* [29]. The module writes the execution times and number of module- and function calls when the program finishes running.

4.2.5 Security

Choosing the right security metrics can be challenging. Therefore, there are guidelines for choosing and even creating quality attributes related to security [18]. Security metrics should measure the security level through quantities, have reproducible results, be objective and unbiased and be able to measure progress towards a goal over time [30]. Examples of security metrics are *Number of Policy Violations* and *Percentage of Weak Passwords*.

5. Tools and Technologies Used

This chapter presents the tools and technologies that have been used during the development of SOGS-API. The API has been built with Python, HTML, CSS and JavaScript, while other technologies and tools have supported the development process.

5.1 Python

Python [31] is a multiparadigm high-level general purpose interpreted programming language created by Guido van Rossum in the early 1990s [32]. Python has features, such as block indentation, that makes the code more readable [33].

The Python debugger *pdb* is used for debugging Python programs. It is included in the Python standard library and provides basic debugging features such as breakpoints, stepping and checking the value of variables [34].

The Python module and linter tool *Pycodestyle* [35] (formerly PEP-8) is used for checking if the code complies with the style guide of PEP-8 [36].

The Python module *cProfile* [29] is used for profiling execution time and number of executions of Python programs, modules and functions. It is included in the Python standard library and can be used by running:

```
python -m cProfile yourprogram.py
```

The profiler can also be run from code:

```
import cProfile
import yourprogram
cProfile.run('yourprogram.afunction("arg")', 'optionaloutfile')
```

The execution of the program can be finished in the same ways as during normal execution. When the program finishes, the cProfile module prints or writes to file the execution times and number of executions of programs, modules and functions. The list will include both built-in and custom modules. As an example, here is the header line and an entry from the profile report of SOGS-API:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
18	0.000	0.000	0.001	0.000	apiutils.py:104 (sanitize)

In the column headings *ncalls* means the number of calls, *tottime* is the total time spent in a function excluding time spent in sub-functions, *percall* is *tottime* divided by *ncalls*, *cumtime* is the cumulative time spent in a function including sub-functions, the second *percall* is *cumtime* divided by the number of primitive (non-recursive) calls, and *filename:lineno(function)* provides the file line and function. The *cumtime* attribute is also accurate for recursive functions. If the *ncalls* attribute contains two values, the function has recursed. For instance, 4/1 means that the function has recursed four times and had one primitive call.

The *Miniconda* package and environment management system is used for handling Python environments. Miniconda is a minimal version of *Anaconda*, and contains the package manager *conda*, Python, packages that these depend on, and some other useful packages, such as *pip* and *zlib* [37].

5.2 Web Technologies

Web pages are built with HTML and styled with CSS. More advanced functionality can be programmed in JavaScript.

5.2.1 HTML

Hypertext Markup Language (HTML) is the core markup language of the World Wide Web. The language is based on XML, and the XHTML variant is actually pure XML, which is compatible with programs that use XML. Originally, HTML was designed to be a language for semantically describing scientific documents. Thanks to its general design, HTML has adapted to describe other types of documents and applications as well [38].

HTML is used for displaying information in a human friendly format in Web browsers. Web browsers receive documents from a web server and render them as the web page that the client sees. HTML consists of tags, describing the structure of the web page. Figure 5.1 shows the HTML and display of a very simple web page.

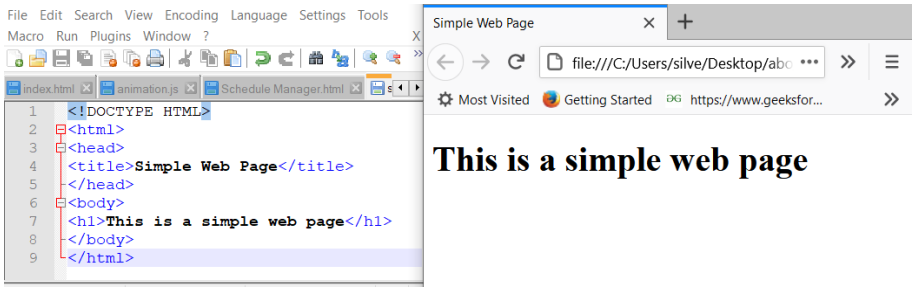

Figure 5.1: A simple web page and its HTML code.

5.2.2 CSS

Whereas HTML is the language for structuring a web page, Cascading Style Sheets (CSS) is the language for defining the presentation [39]. With CSS one can specify fonts, colors and layout to be used in any web page that uses the CSS document. The CSS language makes use of selectors, which makes it easy to apply styling for specific HTML tags. The following CSS sample would turn the text color of all h1 -tags brown:

```
h1 {color:brown;}
```

CSS can be written within the HTML file that uses it, or it can be in an external file that is loaded by HTML. The latter is more common, as it has the benefit of using the same CSS file for several HTML files.

5.2.3 JavaScript

JavaScript is a multi-paradigm prototype-based object-oriented programming language used mainly for adding dynamic features to a web page. There are also server-side versions of JavaScript, such as Node.js, which can handle requests, communicate with databases, and do file operations on the server [40]. Many libraries and frameworks have been made for JavaScript. Some popular ones are D3, jQuery, Angular, React, Ember.js, Node.js, and Vue.js.

Like CSS, JavaScript can be written directly in the HTML file or be loaded from an external file. JavaScript can manipulate the Document Object Model (DOM), changing the contents of the web page.

5.3 Git

Git [41] is a distributed version control system created by Linus Torvalds in 2005. Files are stored in Git repositories, which are like tree structures, divided into branches and

a trunk. Branches are created to separate the current state of the files from the trunk, until they are merged once again. Usually in a project, each developer is developing the software in his/her own repository. The updates to the software go to a master repository through a so-called *pull* request, where a developer has contacted the administrator of the master branch, that changes can be pulled from the developer's repository into the master. This way the pull request can be reviewed and discussed before the changes are merged with the master repository.

5.4 APIs

There are APIs for software libraries and frameworks, operating systems, databases, and the web. The popularity of web APIs has grown since 2005, when companies have recognized the benefits of providing an open platform. In 2013, there were over 9000 APIs [42]. In 2019, there were 22 000 web APIs and the number was growing fast [43].

The web APIs use standardized protocols, such as HTTP, for communication through Internet. A popular architectural style for web APIs is REST. REST stands for Representational State Transfer and contains six guiding constraints:

1. Client-server - separating user interface concerns from data storage concerns
2. Stateless - the requests are independent of server state
3. Cacheable - a label indicates if a response is cacheable
4. Uniform interface - following interface constraints:
    (a) identification of resources
    (b) manipulation of resources through representations
    (c) self-descriptive messages
    (d) hypermedia as the engine of application state
5. Layered system - component visibility is restricted by layers
6. Code on demand (optional) - client code can be extended by downloading and executing scripts or applets [2]

An API that complies to these constraints can be considered a RESTful API.

6. SOGS-API - Features and Services

6.1 Initial Version of SOGS-API

The initial version of SOGS-API sent ACU and S band modem metrics to a single customer through UDP. It had basic exception handling and logging. It also contained the scheduling feature, although it was not in use.

The system overview and a detailed overview of the initial version of SOGS-API is illustrated in Figures 6.1 and 6.2. These figures present the interactions between computers, programs, and people. The antenna and its components are to the left, the API and FMI operations are in the center, and the customers are to the right in the figure. Figure 6.1 also shows that the customers may send TCP requests to the S band modem without interacting with the API.

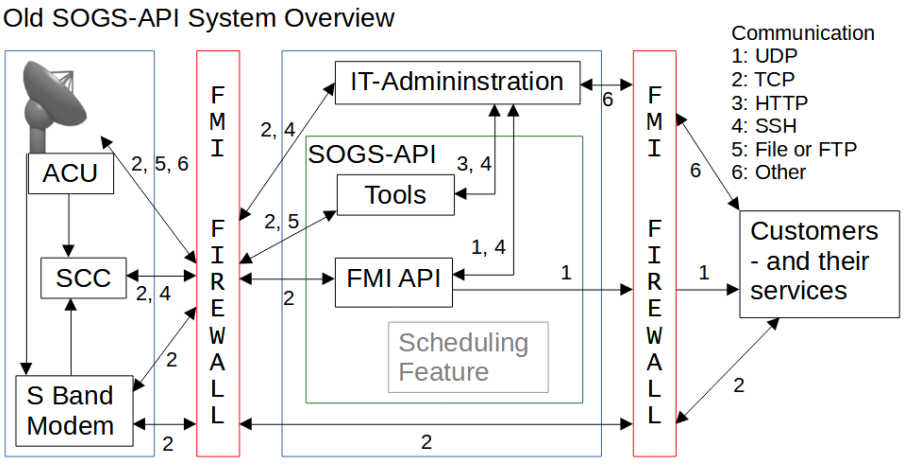

Figure 6.1: Old SOGS-API system overview.

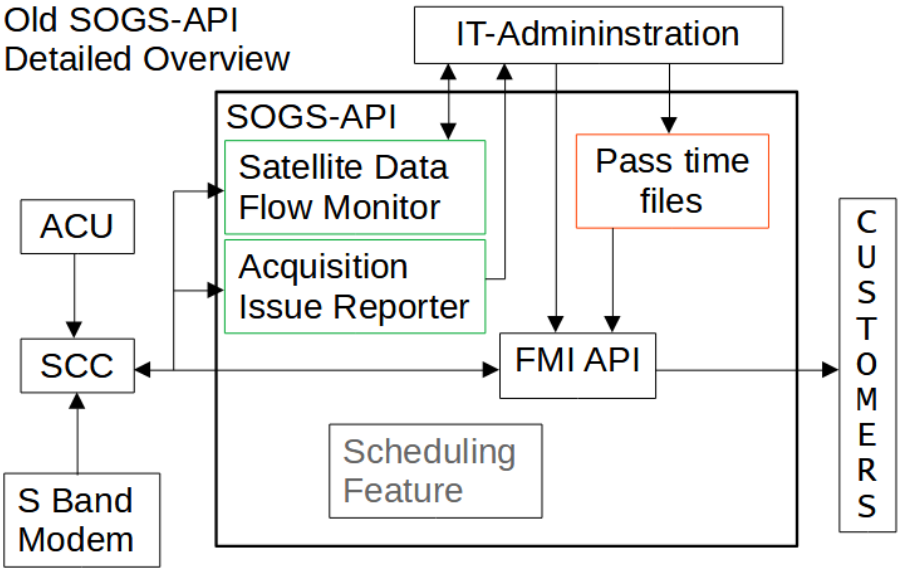

Figure 6.2: Old SOGS-API detailed overview. Related tools are marked with green border, and files with orange.

6.2 Issues

6.2.1 Initial Version

The initial version of SOGS-API had five issues. The main program was written in Python 2, which was about to be outdated. There was a bug in the antenna SCC server, which caused it to occasionally send thousands of empty messages per second and the logger of SOGS-API would log all the empty messages, filling the log file quickly. If SOGS-API lost contact to the antenna SCC server, it would endlessly try to reconnect and the only way to re-establish the connection was by restarting the API. The code that read overpass times could not handle overpasses that occurred during change of day. The API lacked desirable features, such as S band plots and reporting through email.

6.2.2 Issues Discovered During Development

At one point, the API started to shut itself down without notifying anything. When a week later it occurred the second time, there was an observable pattern. It happened after an S band CSV file was created, and before the data in the file was plotted and saved into a PNG file. We nicknamed this phenomenon *the CSV to PNG issue*. This issue initiated the creation of a Python script that revives the API if it goes down. It has been a very useful feature and made the issue much less severe. The only consequence with the issue was that the operator staff of the FMI did not receive the S band image through email

automatically. The image could be drawn afterwards, though, and tools were made for making it easier. The issue has been tracked since it first appeared, and it has not occurred again.

While working on the API shutdown function, some issues appeared. The first way that API processes were shutdown was gracefully through text files. The drawback with reading a shutdown command through a text file is the delay. Therefore, support for clean shutdown through a SIGINT (POSIX) signal was added. Unfortunately, the API reviver programs utilize sub-processing, which can block SIGINT signals, so it is not a reliable way to shutdown the API. In the end, there are three ways to shut down API processes, through file, through SIGINT, and by killing the process, and the shutdown through file is recommended because it is both reliable and clean.

There was a time when the API also reported when components went offline in the antenna, which at worst means that the power has gone out of the antenna. This feature was later moved to its own module and was disabled in the API, because it could prevent the API from doing its main tasks when checking if a component is back online. The feature was later used in another program.

6.3 Overview of the new SOGS-API

The system overview of SOGS-API is illustrated in Figure 6.3, and a detailed overview of the API is shown in Figure 6.4. The figures are organized in the same way as in Figure 6.1.

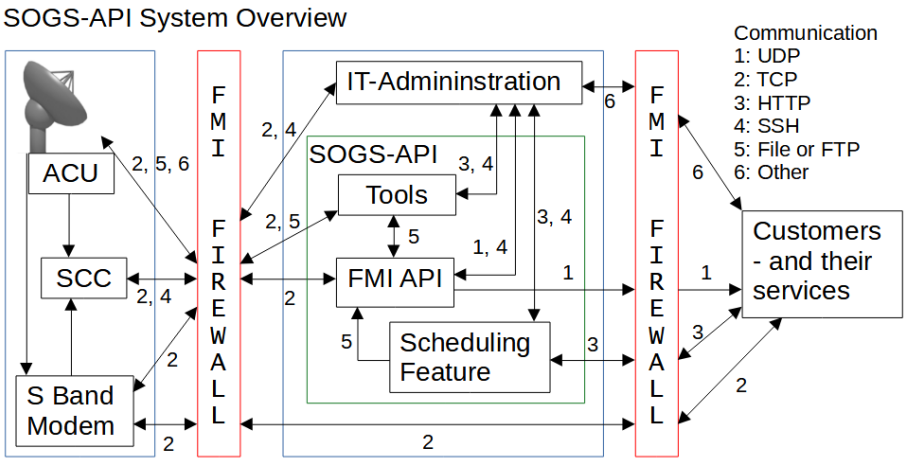

Figure 6.3: SOGS-API system overview.

There are actually two versions of SOGS-API, a development version running on a test

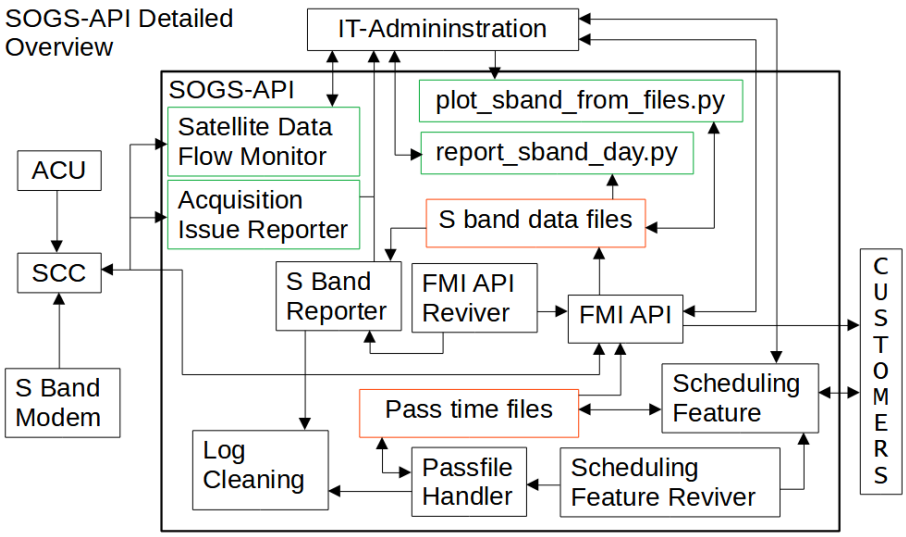

Figure 6.4: SOGS-API detailed overview. Related tools are marked with green border, and files with orange.

server, and an operational version running on an operational server. The versions are very similar and the key differences are in the configuration. The development version is in *debug* mode, sending emails only to developers, while the operational version sends emails both to developers and operators. The development version also has more content and tools to strip away unnecessary content before deploying a bigger update to the operational version.

6.4 Main Program

The main program of SOGS-API, also called FMI API, is a service that serves antenna status data to customers of FMI. Data is served every two seconds during a satellite overpass, and every half minute otherwise. The data flow during an overpass is illustrated in Figure 6.5.

Figure 6.6 shows the use case diagram of SOGS-API. A customer has access to the customer's overpass schedule and can by default also submit new overpass times and request changes. A customer receives antenna status data regularly and can be informed about problems. The FMI Staff can view, add and edit schedules and users, develop the API further and read a report about the signal strength from retrievals.

Table 6.1 compares features between the old and new version of the main program. *Read overpass times* means that the program reads the schedules of satellite overpasses from file. *Send ACU metrics* and *Send S band data* mean that the respective data are sent through UDP to customers. *Logging* is self-explanatory. *Exception tolerance* means that

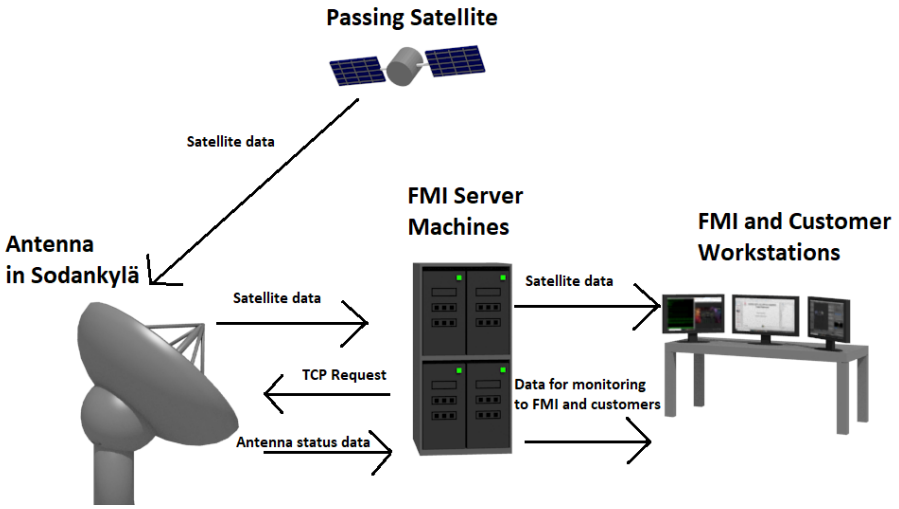

Figure 6.5: FMI API data flow.

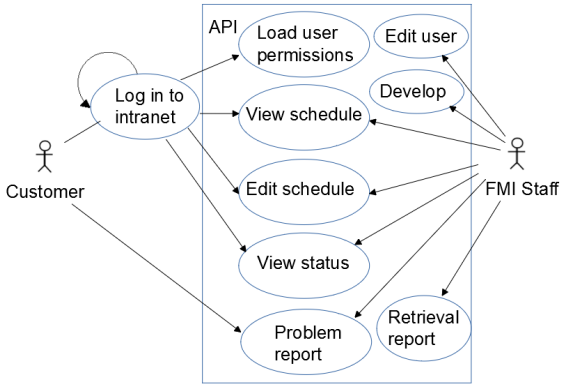

Figure 6.6: Use case diagram of SOGS-API.

exceptions are handled and the program keeps running, *Exception reporting* reports issues through email, and *Exception statistics* counts exception occurrences. *S band visualization* plots S band modem data and *S band reporting* sends daily reports with the plots through email. *Operation scripts* are shell scripts that make the starting, stopping and restarting of the program easier. *Update tools* are support scripts and environments for handling updates to the operational server. *Log cleanup* removes old log files, and *Log cut down* reduces log messages to the most critical ones when log file size has exceeded a configurable limit. *Runtime commands* are commands that can be given to the program through text files during runtime. *Self-repair restart* lets the program restart itself when exception handling cannot correct a recurring issue. If the program crashes for some reason, the *Revival* feature restarts the program and so does *Anti-jamming* when the program starts waiting for a TCP response it does not receive. *Supports multiple customers* and *Supports multiple satellites* are self-explanatory. *Supports multiple outputs* means that a customer can receive specific data in different destinations through UDP. *PEP-8 code*

style means that the Python code is following the PEP-8 style guide. *Wiki documentation* means that the program has documentation in FMI's Wiki.

Table 6.1: Features of the old and the new version of the main program

Feature	Old	New	Feature	Old	New
Read overpass times	✓	✓	Log cleanup	x	✓
Send ACU metrics	✓	✓	Log cut down	x	✓
Send S band data	✓	✓	Runtime commands	x	✓
Logging	✓	✓	Self-repair restart	x	✓
Exception tolerance	✓	✓	Revival	x	✓
Exception reporting	x	✓	Anti-jamming	x	✓
Exception statistics	x	✓	Supports multiple customers	x	✓
S band visualization	x	✓	Supports multiple outputs	x	✓
S band reporting	x	✓	Supports multiple satellites	x	✓
Operation scripts	x	✓	PEP-8 code style	x	✓
Update tools	x	✓	Wiki documentation	x	✓

Figure 6.7 shows the class diagram of the main program.

Figure 6.13 shows the state diagram of the main program. When the main program is started, modules and configuration parameters are loaded, and the program goes into *normal mode*. During *normal mode*, which is also called *Non-overpass mode*, the API sends antenna status data to customers and FMI logging services every half minute. The API also checks the satellite overpass schedule and eventual updates to user configuration. When the satellite is in range of the antenna, the SOGS-API switches to *overpass mode*. In this mode, the API sends ACU metrics and S band modem metrics data to customers and the FMI every two seconds. When the overpass ends, the API switches back to *normal mode*.

Figure 6.14 shows the state diagram of SOGS-API S band Reporter. The reporter starts by loading relevant modules and configuration. Then it starts a schedule, where it sends S band plots through email and cleans log files every day at 5:00 UTC. The reporter executes the log cleaning because it is convenient to do it once a day.

Figure 6.15 shows the state diagram of the FMI API Reviver. The reviver starts by loading the relevant modules and configuration. Then, every 10th second it reads a text file, checking if it should stop running. If not, it checks if the other API processes are running and revives them if necessary. It also detects the rare event when the main program jams. In that case, the reviver will restart the main program.

Customers and deliverables are defined in configuration files. If there are changes in customer or deliverable data, these can be updated in the running API through update

files, which the API reads every two minutes. This way the API does not need to be restarted every time there is a change in the configuration.

S band modem data are retrieved in JSON format through a Python script. The JSON is prettified and sent to the clients. Some important values are also extracted from the data, and these are stored in a CSV file and plotted with the Python module *Matplotlib* into a PNG image file.

When the API detects that it receives a series of empty messages from the antenna SCC, it reports it to the IT-administration through email and stops logging the empty messages. Temporary statistics of exception occurrences are also made and these are used to prevent reporting the same problem several times in a short time. There is also a check on log file size. If the log size exceeds a configured limit, the log messages are cut down to only the most severe. When the API encounters a problem that does not disappear, it will try to solve it by restarting itself. The limit for how many times the API will restart and how long it will wait between restarts are configurable.

6.5 Scheduling Feature

A service that receives and serves satellite pass plans through HTTP requests was put into use in a later state of the project. The Python 3 code for it already existed, and before launch, it was updated, cleaned up and tested. The pass plans were received from one of the customers and, initially, the service was named after that customer. Later, the service was renamed to *scheduling feature*, for the purpose of describing what it does, to avoid mixing program email messages with the customer's emails, and to have a name that can be used without exposing the customer's name. In the future, other customers may also post pass plans, and therefore, it is more appropriate to have a name that suits a wider audience. Because the scheduling feature may be used by multiple customers, a tool was created for making it easier to manage the overpass schedules. The tool is called Schedule Manager and its GUI is shown in Figure 6.8.

Table 6.2 compares features between the old and new version of the scheduling feature. *Receive/Serve overpass times* means that the program receives and serves overpass times through HTTP. *Overpass format validation* validates received overpass times. *Multiple passes in one request* means that customers can send multiple overpass times in one request. *Automated pass handling* creates the *Overpass history* by moving passed overpasses into weekly files in directories organized by status and year. *Supports multiple customers* and *Supports multiple satellites* are self-explanatory. *Misuse prevention* means that there is authentication, detection of suspicious activity, and blocking of attacks. *PEP-*

8 code style means that the Python code is following the PEP-8 style guide. *Logging* is self-explanatory. *Log cleanup* removes old log files, and also three-year-old history files. *Revival* revives a crashed program. *Exception tolerance* means that exceptions are handled without aborting the program, *Exception reporting* reports issues through email, and *Exception statistics* counts exception occurrences. Schedule Manager is the *GUI for managing schedules*. *Operation scripts* are shell scripts that make the starting, stopping and restarting of the program easier. *Runtime commands* are commands that can be given to the program through text files and HTTP requests during runtime. *Wiki documentation* means that the program has documentation in FMI's Wiki.

Table 6.2: Features of the old and the new version of scheduling feature

Feature	Old	New	Feature	Old	New
Receive overpass times	✓	✓	Logging	x	✓
Serve overpass times	✓	✓	Log cleanup	x	✓
Overpass format validation	✓	✓	Revival	x	✓
Multiple passes in one request	x	✓	Exception tolerance	x	✓
Automated pass handling	x	✓	Exception reporting	x	✓
Overpass history	x	✓	Exception statistics	x	✓
Supports multiple customers	x	✓	GUI for managing schedules	x	✓
Supports multiple satellites	x	✓	Operation scripts	x	✓
Misuse prevention	x	✓	Runtime commands	x	✓
PEP-8 code style	x	✓	Wiki documentation	x	✓

Figure 6.9 shows the class diagram of the scheduling feature.

Figure 6.16 shows the state diagram of the scheduling feature. The program starts by loading modules and configuration, and by starting an HTTP server. When the server receives a GET request, it checks the address. If the address ends with *query<customer id>*, it serves overpass time data related to that customer in JSON format. The HTML GUI for the Schedule Manager tool is served if the end of the address points to it. Figure 6.17 shows the state diagram of the GET handler.

When the server receives a POST request, it first checks if the POST request is for adding pass plans or making a query to the Schedule Manager, by reading the end of the address. In the case of pass plans, it validates and saves the overpass times and responds with a report of successful and unsuccessful additions with error descriptions. If the request is a query to the Schedule Manager, the query is validated. If the query is valid, pass plans that match query criteria are returned in HTML tables. Figure 6.18 shows the state diagram of the POST handler. The program can also be shut down through a POST request, with a special command. For security reasons, there is also a shutdown confirmation through

a local text file. There are shell scripts that make the starting and stopping of the program easier.

Figure 6.19 shows the state diagram of the Passfile Handler. The program starts by loading modules and configuration. Then, every 10th second, it checks if there have been additions to the confirmed_passes.txt file, in which case they are added to scheduled_passes.txt. It also checks if there are rejected and scheduled overpass times that have passed and moves them into weekly history files. When the day changes, it will also check the log and history files. Half-year-old log files and three-year-old history files are removed. Passfile Handler is also responsible for creating the log history that is saving the daily logs into separate files. This is done by renaming *scheduling.log* to *scheduling.log.YYYY-mm-dd*, where *YYYY* is the year, *mm* is the month and *dd* is the day. A new *scheduling.log* file is created when the next logging event occurs. Finally, Passfile Handler checks its shutdown text file for a shutdown command, and if the command is present the program stops.

Figure 6.20 shows the state diagram of the scheduling feature reviver. The reviver starts by loading the relevant modules and configuration. Then, every 10th second it reads its shutdown text file, checking if it should stop running. If not, it checks if the other feature related processes are running and revives them if necessary.

6.6 Implementation

After three months of programming, the code was refactored [44] as the main program file, fmiapi.py had grown to 800 lines of code. A rule of thumb, taught in a software quality course, is that a function should contain 4 to 40 lines and a file 4 to 400 lines of code. If the code contains documentation or a larger data set, the lines of code in a file may exceed 400. First, the Python code was changed to comply with the guidelines of PEP-8, which consequently increased the number of lines to 940. Then all utility functions were moved into modules, with the benefit of also reusing some code, reducing the lines to 500.

During the project, the operational API was updated several times. Updating is a little challenging, because the customers will notice if the API is down for two minutes, so the updated API must work right away, or else there is the need for a quick roll-back. Another challenge arises from the fact that the development version of SOGS-API is slightly different from the operational version. Most of the differences are in the main configuration and customer configuration. Another small difference is that Slack messaging must be disabled on the operational side, because the firewall blocks outgoing HTTP POST

requests. This only requires the insertion of one line of code in module *apiutils.py*, which is imported by every program in the API. For long, the source of overpass times was different. On the development server, the satellite overpass times were read from a file generated by the scheduling feature and this was the planned way to read overpass data. Because the customer that would send the overpass times through the scheduling feature had difficulties implementing the sending of overpasses on their system, the overpasses were inserted manually into another text file. This difference did not only show in the configuration, but also in the module *overpassreader.py*, where three lines of code had to be disabled. These lines would split the overpass times of two different satellites into individual files. In a later state of the project, the overpass reader was updated to handle any pass plans received through the scheduling feature and from multiple customers. With this update, a function had to be disabled with one line of code on the operational side until the scheduling feature is being used.

To make the updating process easier, procedures and tools were made for deployment and testing. On the development server, two shell scripts were made to make the deployment easier, *cpdeploy.sh* and *zipdeploy.sh*. *cpdeploy.sh* copies all files and directories that go to the operational server into a directory called *sogsapi_deployment*. *zipdeploy.sh* compresses the directory into file *sogsapi_deployment.tar.gz*, which is easy to move to the operational server through SCP or FTP. On the operational server, a simulation environment was created for testing the API without affecting the running operational API. In case an update fails, there is a roll-back version of the API, which is independent of any other version of the API. In the developer's manual for the API, a section for updating the operational API was added, with step-by-step instructions for both smaller and bigger updates. Later during the project, an agreement was made to primarily make the updates in a defined time window.

6.7 Related Tools

During the project, other existing tools related to SOGS-API were developed. These are called *Satellite Data Flow Monitor*, *Acquisition Issue Reporter* and *Antenna Status Monitor*.

6.7.1 Satellite Data Flow Monitor and Acquisition Issue Reporter

Satellite Data Flow Monitor plots antenna attributes, such as signal strength, antenna elevation and azimuth from overpasses, and Acquisition Issue Reporter finds and reports issues in these attributes. The tools were improved by adding automated organization of

ACU log files and adapting the tools for this change. The organization of files improved the performance by reducing the time taken to find files and filtering them. Before this improvement, all files had to be read by the file-finding function, *glob.glob()*, and time filtering used binary search to find the times of the overpasses from the files. After the improvement, only directories that match the time and satellite are read and the detailed time filtering is done with an algorithm that deletes from the beginning and the end of the sorted list of files those files that are not within the time range. Testing has shown that this algorithm is faster than binary search when the lists are smaller. The execution times of the tests are shown in Table 6.3.

Table 6.3: Comparison of algorithms. Times are in milliseconds. *Algorithm 1* is *binary search* and *Algorithm 2* is *list edge removal*

Algorithm 1	Algorithm 2	Proportion	Plots	Date Filter	Satellite Filter
25.8147	24.8951	1.036938	100	None	None
8.2702	3.0999	2.667897	6	None	CSK 1
55.8059	28.1581	1.981872	100	All from May	HY2A, KOMPSAT-5, NOAA 20, NPP
24.7573	7.4102	3.340948	78	All from May	AQUA
33.3435	91.6104	0.363971	42	19 – 25 May	NOAA 20, NPP
10.0202	8.2411	1.215877	3	19 – 25 May	CSK 1
25.9385	32.0560	0.809160	100	None	None (Same as first one)

Figure 6.21 shows the state diagram of Satellite Data Flow Monitor. The program runs an HTTP server. It responds to GET requests with the HTML for the web interface. In the interface, a user can filter logs by start date, end date, satellite and antenna. The criteria are sent in a POST request and the server responds with plots for ACU log files that match the criteria. From the plots, it is easy to read if there are problems in signal strength or antenna movement. The program can be shut down using a special shutdown POST message and confirming the shutdown through a text file on the server. Shell scripts make the starting and stopping of the service trivial.

Figure 6.22 shows the state diagram of Acquisition Issue Reporter. The program runs a scheduled check for issues in new ACU log files and organizes the files into directories by year, month, antenna and satellite.

The class diagrams of Satellite Data Flow Monitor and Acquisition Issue Reporter are presented in Figures 6.10 and 6.11.

6.7.2 Antenna Status Monitor

The Antenna Status Monitor tool requests status information for antenna components from the SCC and displays the information in a human-friendly format. Appendix A demonstrates how the tool displays ACU metrics.

Figure 6.23 shows the state diagram of Antenna Status Monitor. The program runs an HTTP server, which serves a GUI for selecting antenna and components for status checking. Based on the parameters in an HTTP POST request, the program sends TCP requests to the target antenna's SCC for specific components. The TCP response messages are parsed into HTML code, which is sent to the client for display in the output area of the GUI. The program can be shut down with a SIGINT signal.

Figure 6.12 shows the class diagram of Antenna Status Monitor.

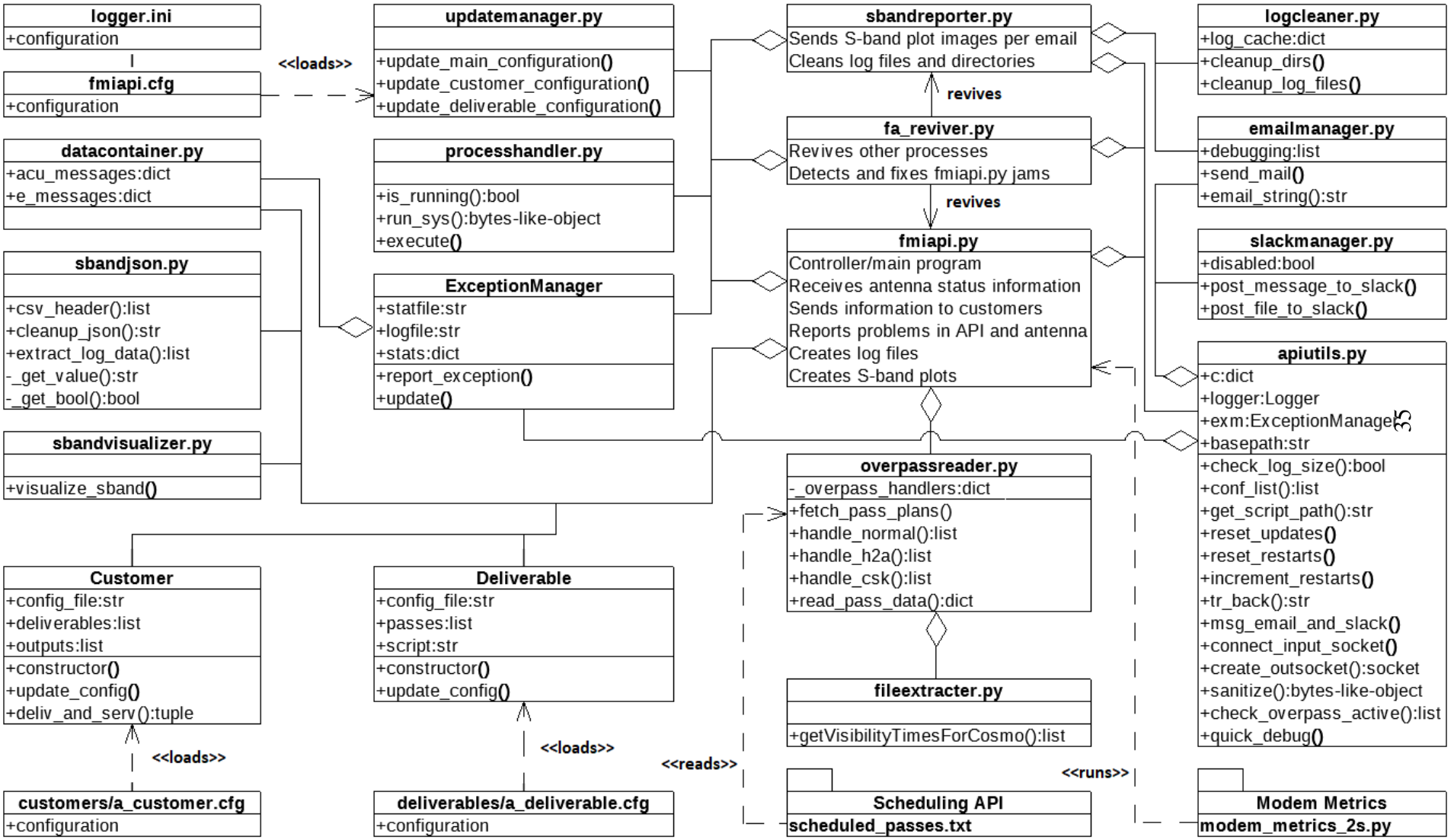

Figure 6.7: Class diagram of main program.

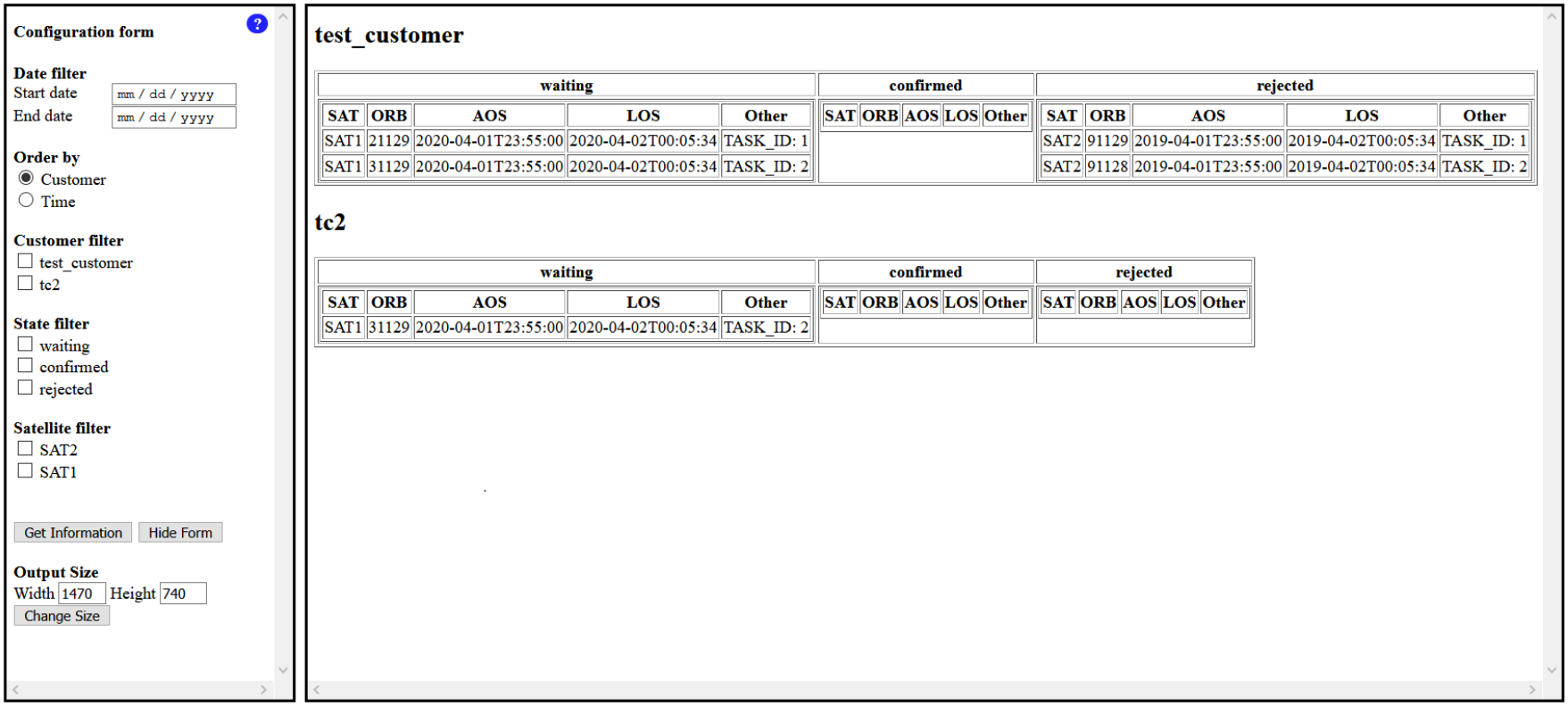

Figure 6.8: Schedule Manager GUI.

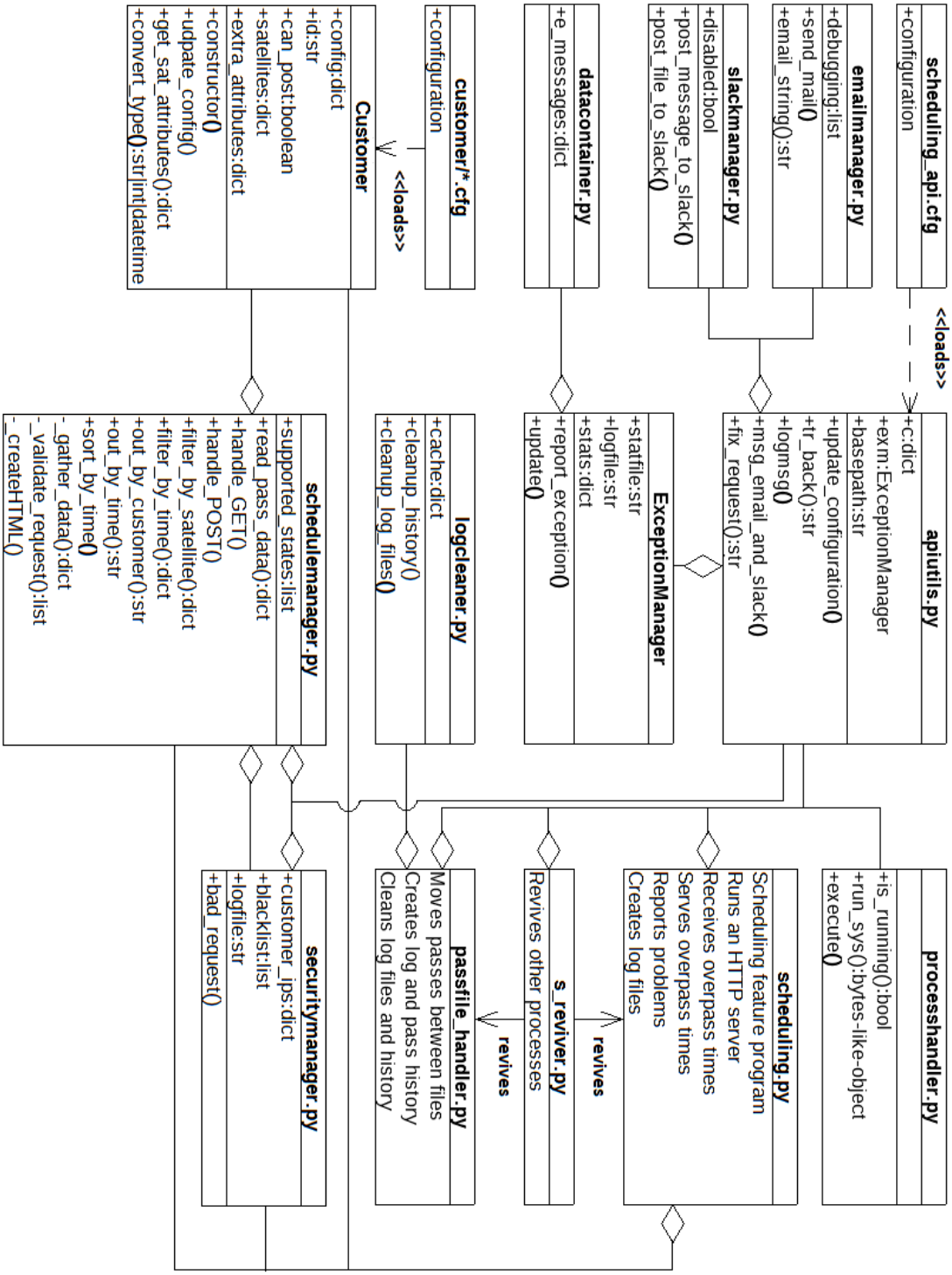

Figure 6.9: Class diagram of scheduling feature.

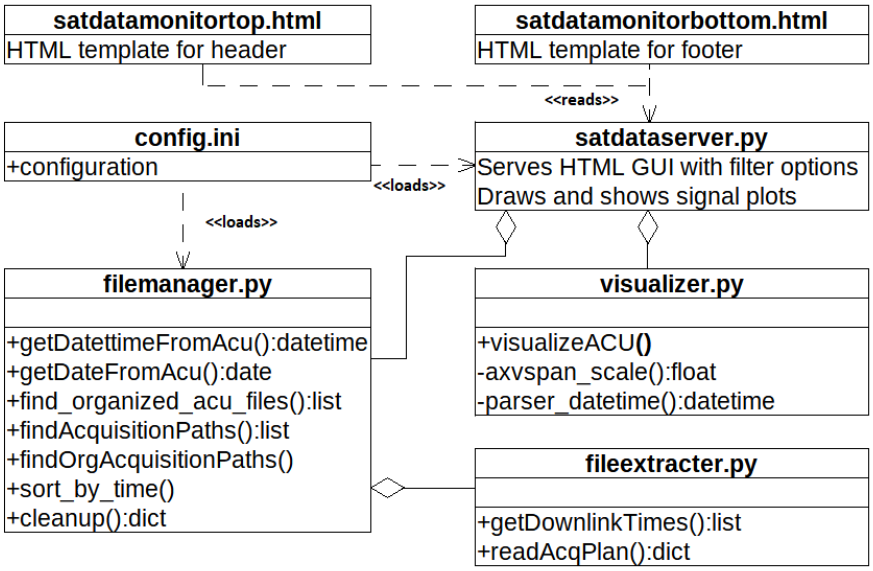

Figure 6.10: Class diagram of Satellite Data Flow Monitor.

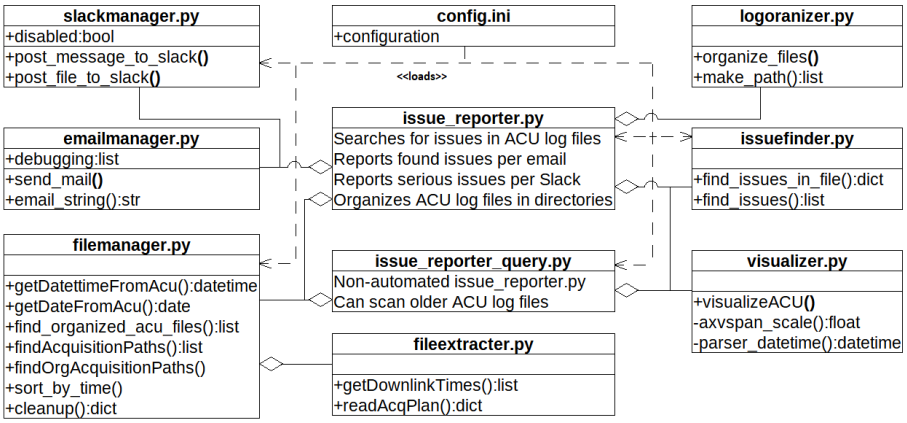

Figure 6.11: Class diagram of Acquisition Issue Reporter.

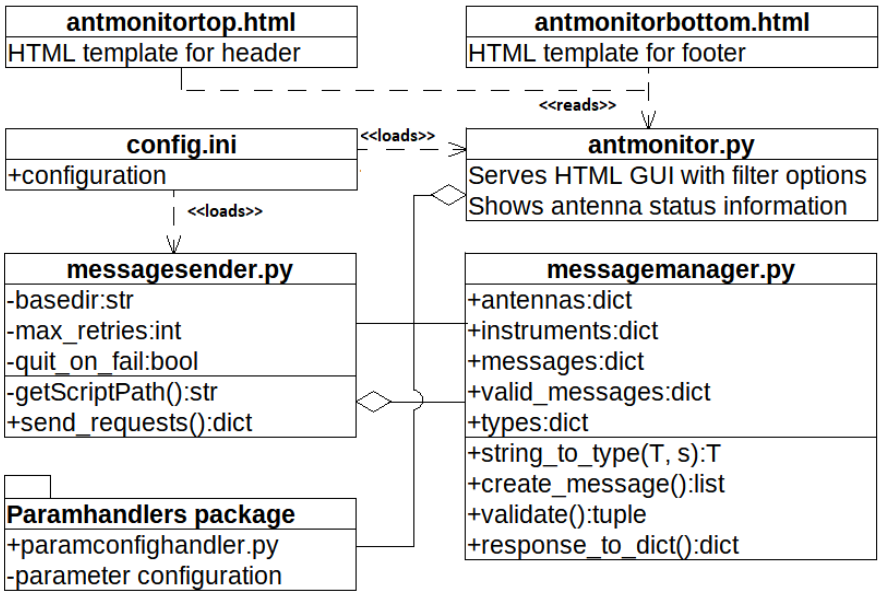

Figure 6.12: Class diagram of Antenna Status Monitor.

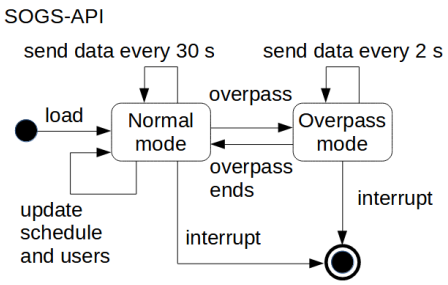

Figure 6.13: State diagram of main program.

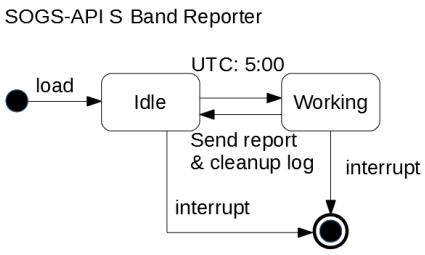

Figure 6.14: State diagram of SOGS-API S band Reporter.

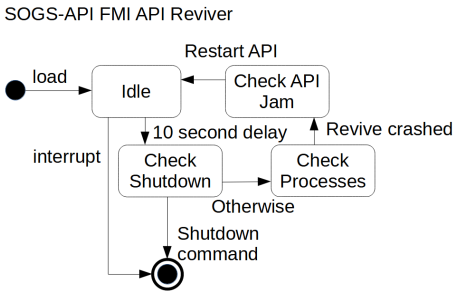

Figure 6.15: State diagram of FMI API Reviver.

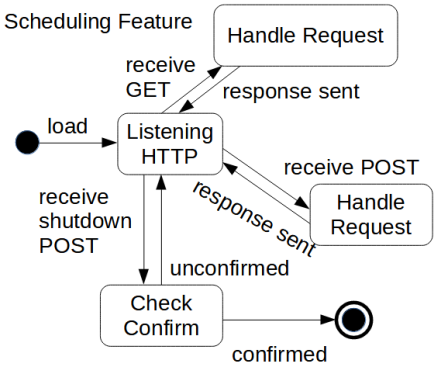

Figure 6.16: State diagram of scheduling feature.

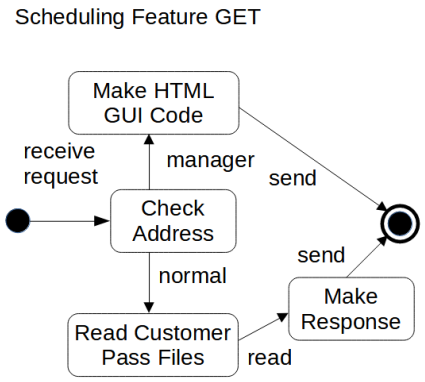

Figure 6.17: State diagram of scheduling feature GET handler.

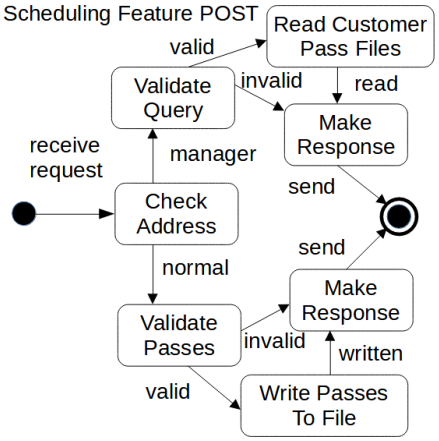

Figure 6.18: State diagram of scheduling feature POST handler.

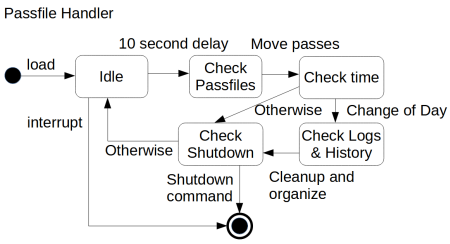

Figure 6.19: State diagram of the Passfile Handler.

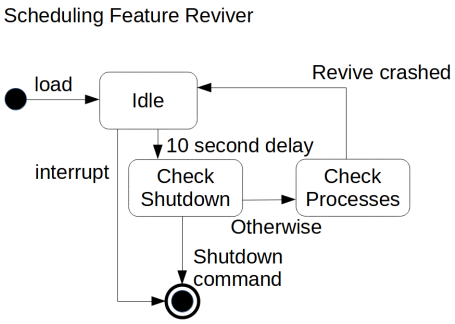

Figure 6.20: State diagram of the Scheduling Feature Reviver.

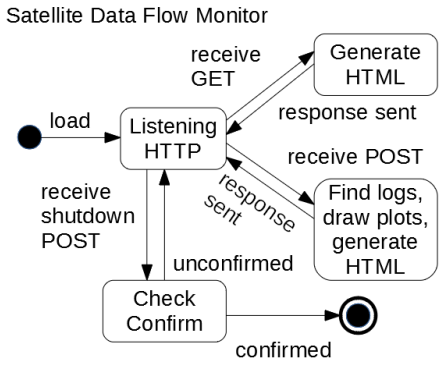

Figure 6.21: State diagram of Satellite Data Flow Monitor.

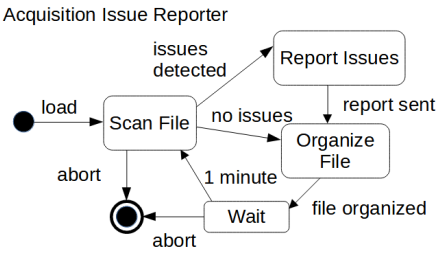

Figure 6.22: State diagram of Acquisition Issue Reporter.

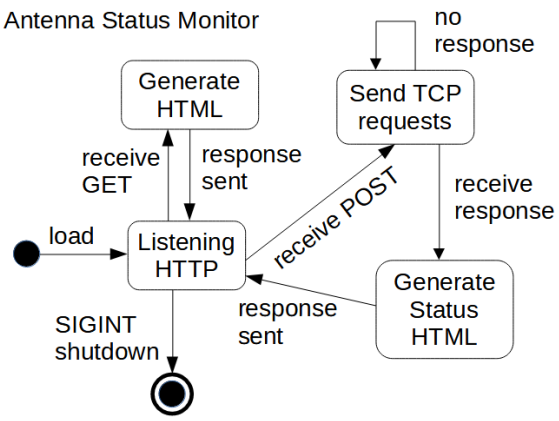

Figure 6.23: State diagram of Antenna Status Monitor.

7. Evaluation

In this section, we present the evaluation of SOGS-API. The API has been evaluated through the measurement of quality attributes on module level and as a system.

7.1 Reliability

Reliability was measured by recording the results of special scenarios and input. Both random and mutating fuzzing was used to test various input to the scheduling feature. The main program was not tested with fuzzing, because it only receives input from the antenna and the input is easy to validate.

The fuzzers were tailor-made with Python. The random fuzz contained random characters, and the mutator changed parts of a valid JSON formatted request. During the first random test, the need to update the handling of faulty JSON data was detected in the new scheduling feature, because a previous update had made the program respond with an empty message instead of the JSON format error. The problem was solved and everything else worked properly.

The behavior of the main program was tested when there were problems in the connection or the antenna. The old version did not handle the connection problems well, either crashing or filling the log file quickly. The new version handles these scenarios properly and will report bigger problems through email.

Table 6.1 shows the results of the reliability tests of old and new version of SOGS-API:

Table 7.1: Reliability test, old vs new SOGS-API

Scenario	Old	New	Scenario	Old	New
Faulty input	Pass	Pass	Fuzz test	Pass	Pass
Problem in antenna	Fail	Pass	Problem in connection	Fail	Pass

Product failure rate was calculated. The metric assumes that the product is used an equal amount of time each day, and this suits SOGS-API well, because the services are run-

ning constantly and the number and occurrence of overpasses vary only slightly. The old version of the main program failed on average once in three months, due to a problem in antenna or connection. This affected both the installation on the test server and the installation on the operational server. Using Formula 4.3 this makes the failure rate 2 / (2 * 90) = 1 / 90 = 0.01111. The new version of the main program handles problems in antenna and connection, and has yet not failed during operation, having a failure rate of 0. Both the old and the new scheduling feature programs have never failed, so their failure rate is also 0.

7.2 Security

The old version of the API was built for a single customer, and virtually there were no threats for it. The new version of the API supports multiple customers, which makes security measures more important. For instance, customers should not be able to access the pass plans of other customers. The security of the scheduling feature is of interest because it is the only program in the API that receives external user input.

Security was measured through scripts and tools. An attempt to crack Customer IDs were made by randomly generating a million strings. The crackability of the address and password of the Schedule Manager tool was measured using an online tool. When conducting the tests, it was assumed that the attacker would have gained access to the system through the hijacking of a customer's workstation.

In the first test a Python script was used for attempting to crack the ID of customers. Ten hexadecimal values of 32 characters were generated using md5 hashing to simulate the customer IDs. It was assumed that the attacker would have gained the knowledge that customer IDs are hexadecimal values of 32 characters, and only these kinds of IDs would be used in the brute-force attack. Still, none of the IDs were cracked in a million attempts, and it is no surprise, as possible combinations are 32^{16} = 1,208,925,819,614,629,174,706,176. In reality, the ID can be any random text with various length, making it even more difficult to crack the IDs of customers. Short IDs, or IDs that are otherwise easy to crack with the help of a dictionary, are not accepted.

In the second test, the attacker would try to find the address of the Schedule Manager tool. The Schedule Manager tool should have a very restricted audience, so it is better that it is challenging to find the address. For this test, an online password strength testing tool called *How Secure Is My Password?* [45] was used. It told that the address could be found instantly. The reason was that the unique part of the address was the word "manager". The address was made longer and more difficult to find by using a combination of words, and

the online tool estimated that it would take one day to guess the new address. The ideal address is one that is memorable, yet difficult to find or guess.

In the third and final test, the password strength of the Schedule Manager tool was measured. Again, the aforementioned tool was used, and the result was that it would take a computer 300 years to crack the password.

Detection of suspicious activity was added to the scheduling feature, making brute-force cracking techniques ineffective by blocking the attacker's IP-address.

The results of the security tests can be expressed as security metrics *No. of Customer IDs Cracked in a Million Attempts* = 0, *Schedule Manager Web Address Guessability* = 1 day *Schedule Manager Password Strength* = 300 years for a computer to crack

7.3 Performance

During the time of writing, there are a few users of the API, but in the future this number will likely grow. Therefore, the performance was measured to see if there are performance issues. Performance of the API was measured with the Python module *cProfile*, and by writing timestamps to file. The cProfile data showed no performance problems in any part of the API. When putting the scheduling feature under heavy load, an increase in request handling time was observed. The first request was handled almost immediately, and the final 2000th request had a 0.09 second delay. Because a single customer is sending weekly around 35 pass plans, a single instance of the API should be able to handle 100 customers without performance issues. If the requests of each customer are spread evenly over the week, a single instance of the API may even be able to handle 1000 customers.

7.3.1 Main Program with Modules

fmiapi.py cProfile times for idle and overpass states are shown in Tables 7.2 and 7.3. Most of the functions and modules are executed within one millisecond, which is good. The functions and modules that have longer execution times or more calls, are explained here.

The modules *apiutils*, *customer* and *slackmanager* have execution times between 0.15 and 0.2 seconds, and the reason is that they load modules that take some time to load, such as *socket* and *configparser*, and they are being used by many other modules. The module *sbandvisualizer* takes 0.507 seconds to load, and the reason is in the loading of matplotlib, and a large plotting function. During the overpass test, it took the function *sbandvisualizer.visualize_sband* 1.005 seconds to create and save the S band plot into a

file. The module *fmiapi* is the main program, and it shows that the total execution time was 239.711 seconds for the idle state test case and 240.426 seconds for the overpass state test case.

The *apiutils.sanitize* function was executed 18 times during idle state and 244 times during overpass state. It is a light-weight sanitizer for ACU metrics, which arrive through TCP every 30 seconds during idle state, and every two seconds during an overpass. The sanitizer also handles other metrics and messages coming through the TCP connection.

During the overpass, *processhandler.run_sys* was called 241 times. This function can run other python programs or shell scripts, and in this case it runs *modem_metrics2s.py*, which fetches the S band modem metrics. The *sbandjson* module has functions for handling these metrics, which explains the 100+ calls to its functions.

During the overpass test, the *fmiapi.cleanup* function is executing for 1.001 seconds. This is because the function sends a TCP request to the antenna SCC to stop sending ACU metrics every two seconds, waits for one second to let the message arrive before closing the connection.

The *apiutils.conf_list* was executed 17 times. This function is used on startup when reading multiline comma-separated lists from configuration files.

sbandreporter.py cProfile times for idle and report states are shown in Tables 7.4 and 7.5. The modules and functions used by the S band reporter have fast execution times. An interesting observation is that the loading of *processhandler.py* takes less time after each run, which indicates that the module is cached.

fa_reviver.py cProfile times for idle and revive states are shown in Tables 7.6 and 7.7. The execution times for the modules and functions used by the reviver are fast. Like S band reporter, the loading of *processhandler.py* takes less time after each run.

7.3.2 Scheduling Feature

Figure 7.1 shows the time between handled requests in the scheduling feature during a concurrency performance test. The test was conducted with two client scripts that were run simultaneously. They send 1000 unique requests each. The time of receiving a request and writing the pass to a file was recorded into a text file. The times and order of execution was then analyzed, and the result was that all requests are handled without conflict, but the time it takes to handle a request increases slightly with each request, which indicates that requests end up in a queue until they are processed.

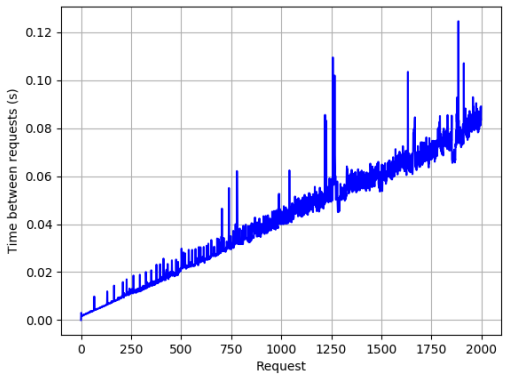

Figure 7.1: Time between handled requests during concurrency performance testing of scheduling feature.

A general performance test was also conducted. Table 7.8 shows the execution times, when all requests are accepted, and when they are rejected.

The Python module *cProfile* was used for getting execution time of programs and their functions during different states. The results in the tables show there is no overhead. Tables 7.9, 7.10, and 7.11 show the execution times of the scheduling feature, its modules and functions during idle state, and when it handles an HTTP POST and GET request. Everything executes fast. Even the HTTP request handling finishes in 0.005 seconds.

passfile_handler.py cProfile times for *idle* and *handling* states are shown in Tables 7.12 and 7.13. The execution times for *passfile_handler.py* show nothing out of the ordinary. All functions perform fast.

s_reviver.py cProfile times for *idle* and *revive* states are shown in Tables 7.14 and 7.15. The reviver for the scheduling feature has very similar execution times as the reviver for the main program. There are no signs of performance issues.

7.4 Maintainability

Maintainability is a quality attribute that consists of attributes modifiability, testability and understandability. These can be further broken down into augmentability, structuredness, communicativeness, accessibility, self-descriptiviness, conciseness and legibility [46].

SOGS-API utilizes configuration files, which contributes to self-descriptiviness, augmentability and structuredness. Responsibilities have been divided into modules, adding structuredness and self-descriptiviness. The code has been written following the style

guidelines of PEP-8, which contributes to legibility. The code was checked with the linter tool *pycodestyle* for compliance with PEP-8. The new versions of the main program and scheduling feature pass the *pycodestyle* test, while the old version of the main program has 61 style flaws, and the scheduling feature has 349. Descriptive variable and function names also add to legibility, and when the context is clear, shorter names have been used, making the code more concise. The code has also been made more concise by using aliases for longer module names, such as *overpassreader*.

The FMI operators have full access to the API and its files, and there are tools and documentation that increase communicativeness. The API also informs about errors and their severity through email, and through Slack when a connection through the firewall is established.

One metric that Pizka and Deißenböck recommends for legibility and structuredness is the IF-ratio expressed as number of *if statements* per 1000 lines of code (IFs/kLOC) [22]. This was counted in the main program and scheduling feature separately by creating and using a Python script that extinguished an *if statement* from "if" used in comments and identifiers. The result for the main program and its modules were 40 IFs/kLOC, and files containing most of the *if statements* were the main program *fmiapi.py* (27.5 %) and the runtime configuration reader *updatemanager.py* (15.5 %). The result for the scheduling feature and its modules were 56 IFs/kLOC, and the files that contained most of the *if statements* were the scheduling feature program *scheduling.py* (30.7 %), and the controller for the Schedule Manager, *schedulemanager.py* (29.8 %).

According to Pizka and Deißenböck, a good result is 20-30 IFs/kLOC, and 60 and above means the code is difficult to analyze, and this is common for legacy software that have had quick additions of features over the years. The IFs/kLOC values of SOGS-API are on the larger side, which indicate that the code could be refactored. A positive fact both in the main program and the scheduling feature is that most of the *if statements* are found in two files, opposed to being spread over several files. Tables 7.16 and 7.17 show the number of *if statements* in each file. Because requirements for the SOGS-API can change, it is important to do the updates in a sophisticated way, or else the structuredness of the API will deteriorate.

The IF-ratio was also calculated for the old SOGS-API. The main program had ten *if statements* in 270 lines of code, resulting in 37 IFs/kLOC, and scheduling feature had 13 in 195, resulting in 67 IFs/kLOC. The difference in IF-ratio between the old and new scheduling feature can be explained with the reduction of *if statements* in the validation of satellite pass plan posts through the utilization of configuration and iteration.

Code complexity was measured using Halstead metrics. The measurement was done by customizing the script *commentedCodeDetector.py* by Borowiec [47] to check Python source code. Borowiec's script does not only find commented code, but also computes Halstead metrics when run with option *-fm*. The script did not compute the *time to implement* and *number of delivered bugs* estimations, nor did it provide recommended values. These were calculated manually. All source code for the SOGS-API was put into one file with Bash command *cat *.py > source.py*, and the script analyzed it. Halstead metrics of the old and new versions of the main program are compared in Table 7.18.

It is recommended that the volume for one file would be less than 8000 [19]. The new main program and its modules are in 27 Python files, making the upper boundary 216000. The old main program is a single Python file, making its boundary 8000.

One file should contain less than two bugs [21] [19]. This means the new main program should contain less than 54 bugs, and the old less than two.

Halstead metrics of the old and new versions of the scheduling feature are compared in Table 7.19. The new scheduling feature contains 14 Python files, making the upper boundary for its volume 112000. The old scheduling feature contains a single Python file, making its boundary 8000. The new scheduling feature should contain less than 28 bugs, and the old less than two.

Looking at the volume, the scheduling feature is complex. Despite being larger than the scheduling feature, the main program has a volume that is within the recommended boundaries. This is because the main program is split into more files than scheduling feature. Scheduling feature could also be split into more files, and this would even raise the boundary of delivered bugs above 28.

Table 7.2: Main Program cProfile results during idle state, around 4 minutes of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
18	0.000	0.000	0.001	0.000	apiutils.py:104 sanitize
11	0.000	0.000	0.001	0.000	apiutils.py:118 check_overpass_active
10	0.000	0.000	0.000	0.000	apiutils.py:21 check_log_size
1	0.000	0.000	0.192	0.192	apiutils.py:3 <module>
17	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
1	0.000	0.000	0.000	0.000	apiutils.py:54 reset_updates
1	0.000	0.000	0.004	0.004	apiutils.py:92 connect_input_socket
1	0.000	0.000	0.192	0.192	customer.py:1 <module>
4	0.000	0.000	0.000	0.000	customer.py:15 <listcomp>
1	0.000	0.000	0.000	0.000	customer.py:19 Customer
2	0.000	0.000	0.001	0.000	customer.py:20 __init__
2	0.000	0.000	0.001	0.000	customer.py:24 update_config
4	0.000	0.000	0.000	0.000	customer.py:9 deliv_and_serv
1	0.000	0.000	0.000	0.000	datacontainer.py:7 <module>
1	0.000	0.000	0.000	0.000	deliverable.py:10 <module>
1	0.000	0.000	0.000	0.000	deliverable.py:15 Deliverable
9	0.000	0.000	0.003	0.000	deliverable.py:16 __init__
9	0.000	0.000	0.003	0.000	deliverable.py:20 update_config
1	0.000	0.000	0.033	0.033	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
9	0.000	0.000	0.001	0.000	fmiapi.py:102 check_for_configuration_updates
1	0.004	0.004	239.711	239.711	fmiapi.py:14 <module>
1	0.000	0.000	0.000	0.000	fmiapi.py:56 cleanup
4	0.000	0.000	0.006	0.001	overpassreader.py:105 read_pass_data
5	0.000	0.000	0.000	0.000	overpassreader.py:123 <listcomp>
2	0.000	0.000	0.000	0.000	overpassreader.py:11 fetch_pass_plans
1	0.000	0.000	0.000	0.000	overpassreader.py:3 <module>
2	0.000	0.000	0.004	0.002	overpassreader.py:63 _handle_h2a
1	0.000	0.000	0.002	0.002	overpassreader.py:84 _handle_csk
1	0.000	0.000	0.009	0.009	processhandler.py:16 is_running
1	0.000	0.000	0.000	0.000	processhandler.py:4 <module>
1	0.000	0.000	0.000	0.000	sbandjson.py:29 <module>
1	0.000	0.000	0.000	0.000	sbandjson.py:56 csv_header
1	0.000	0.000	0.507	0.507	sbandvisualizer.py:24 <module>
1	0.000	0.000	0.159	0.159	slackmanager.py:2 <module>
1	0.000	0.000	0.015	0.015	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.000	0.000	updatemanager.py:9 <module>

Table 7.3: Main program cProfile results during an overpass, around 4 minutes of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
244	0.002	0.000	0.009	0.000	apiutils.py:104 sanitize
123	0.002	0.000	0.002	0.000	apiutils.py:118 check_overpass_active
123	0.001	0.000	0.002	0.000	apiutils.py:21 check_log_size
1	0.000	0.000	0.192	0.192	apiutils.py:3 <module>
17	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
1	0.000	0.000	0.000	0.000	apiutils.py:54 reset_updates
1	0.000	0.000	0.001	0.001	apiutils.py:92 connect_input_socket
1	0.000	0.000	0.000	0.000	customer.py:1 <module>
4	0.000	0.000	0.000	0.000	customer.py:15 <listcomp>
1	0.000	0.000	0.000	0.000	customer.py:19 Customer
2	0.000	0.000	0.002	0.001	customer.py:20 __init__
2	0.000	0.000	0.002	0.001	customer.py:24 update_config
4	0.000	0.000	0.000	0.000	customer.py:9 deliv_and_serv
1	0.000	0.000	0.000	0.000	datacontainer.py:7 <module>
1	0.000	0.000	0.000	0.000	deliverable.py:10 <module>
1	0.000	0.000	0.000	0.000	deliverable.py:15 Deliverable
9	0.000	0.000	0.003	0.000	deliverable.py:16 __init__
9	0.000	0.000	0.003	0.000	deliverable.py:20 update_config
1	0.000	0.000	0.033	0.033	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
133	0.002	0.000	0.006	0.000	fmiapi.py:102 check_for_configuration_updates
1	0.090	0.090	240.426	240.426	fmiapi.py:14 <module>
1	0.000	0.000	1.001	1.001	fmiapi.py:56 cleanup
4	0.000	0.000	0.014	0.003	overpassreader.py:105 read_pass_data
6	0.000	0.000	0.000	0.000	overpassreader.py:123 <listcomp>
2	0.001	0.000	0.001	0.001	overpassreader.py:11 fetch_pass_plans
1	0.000	0.000	0.000	0.000	overpassreader.py:3 <module>
1	0.000	0.000	0.008	0.008	overpassreader.py:35 handle_normal
2	0.000	0.000	0.004	0.002	overpassreader.py:63 _handle_h2a
1	0.000	0.000	0.000	0.000	overpassreader.py:84 _handle_csk
1	0.000	0.000	0.010	0.010	processhandler.py:16 is_running
1	0.000	0.000	0.000	0.000	processhandler.py:4 <module>
241	0.008	0.000	35.933	0.149	processhandler.py:42 run_sys
1	0.000	0.000	0.000	0.000	sbandjson.py:29 <module>
120	0.000	0.000	0.001	0.000	sbandjson.py:29 _get_value
240	0.000	0.000	0.001	0.000	sbandjson.py:41 _get_bool
1	0.000	0.000	0.000	0.000	sbandjson.py:56 csv_header
120	0.001	0.000	0.003	0.000	sbandjson.py:62 extract_log_data
120	0.001	0.000	0.002	0.000	sbandjson.py:78 cleanup_json
1	0.000	0.000	0.500	0.500	sbandvisualizer.py:24 <module>
1	1.005	1.005	1.005	1.005	sbandvisualizer.py:41 visualize_sband
1	0.000	0.000	0.159	0.159	slackmanager.py:2 <module>
1	0.000	0.000	0.026	0.026	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.000	0.000	updatemanager.py:9 <module>

Table 7.4: S band reporter cProfile results during idle state, around 10 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.156	0.156	apiutils.py:3 <module>
3	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
1	0.000	0.000	0.038	0.038	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.000	0.000	logcleaner.py:3 <module>
1	0.000	0.000	0.010	0.010	processhandler.py:16 is_running
1	0.000	0.000	0.031	0.031	processhandler.py:4 <module>
1	0.000	0.000	10.616	10.616	sbandreporter.py:3 <module>
1	0.000	0.000	0.000	0.000	sbandreporter.py:45 <listcomp>
1	0.000	0.000	0.001	0.001	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.001	0.001	updatemanager.py:9 <module>

Table 7.5: S band reporter cProfile results while sending S band report and cleaning log files, around 10 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.197	0.197	apiutils.py:3 <module>
3	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
1	0.000	0.000	0.032	0.032	emailmanager.py:17 <module>
1	0.000	0.000	0.048	0.048	emailmanager.py:32 send_mail
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.009	0.009	logcleaner.py:15 cleanup_dirs
1	0.000	0.000	0.000	0.000	logcleaner.py:3 <module>
1	0.000	0.000	0.000	0.000	logcleaner.py:34 cleanup_log_files
1	0.000	0.000	0.009	0.009	processhandler.py:16 is_running
1	0.000	0.000	0.025	0.025	processhandler.py:4 <module>
1	0.000	0.000	10.250	10.250	sbandreporter.py:3 <module>
1	0.000	0.000	0.000	0.000	sbandreporter.py:45 <listcomp>
1	0.000	0.000	0.002	0.002	sbandreporter.py:62 _find_images
1	0.000	0.000	0.079	0.079	sbandreporter.py:88 job
1	0.000	0.000	0.001	0.001	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.001	0.001	updatemanager.py:9 <module>

Table 7.6: Fmiapi reviver cProfile results during idle state, 40 - 50 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.197	0.197	apiutils.py:3 <module>
2	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
1	0.000	0.000	0.035	0.035	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.001	0.001	42.133	42.133	fa_reviver.py:11 <module>
7	0.001	0.000	0.094	0.013	processhandler.py:16 is_running
1	0.000	0.000	0.098	0.098	processhandler.py:4 <module>
1	0.000	0.000	0.001	0.001	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.001	0.001	updatemanager.py:9 <module>

Table 7.7: Fmiapi reviver cProfile results while reviving the main program, 40 - 50 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.191	0.191	apiutils.py:3 <module>
2	0.000	0.000	0.000	0.000	apiutils.py:35 conf_list
1	0.000	0.000	0.000	0.000	apiutils.py:45 get_script_path
2	0.000	0.000	0.107	0.054	apiutils.py:84 msg_email_and_slack
1	0.000	0.000	0.037	0.037	emailmanager.py:17 <module>
2	0.000	0.000	0.107	0.054	emailmanager.py:32 send_mail
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.001	0.001	45.256	45.256	fa_reviver.py:11 <module>
7	0.002	0.000	0.113	0.016	processhandler.py:16 is_running
1	0.000	0.000	0.092	0.092	processhandler.py:4 <module>
2	0.000	0.000	0.016	0.008	processhandler.py:51 execute
1	0.000	0.000	0.001	0.001	updatemanager.py:21 update_main_configuration
1	0.000	0.000	0.001	0.001	updatemanager.py:9 <module>

Table 7.8: Scheduling feature performance test. Times are in seconds. Exetime 1 to 3 are for accepted requests, and R1 to R3 are for rejected

Requests	Exetime 1	Exetime 2	Exetime 3	Exetime R1	Exetime R2	Exetime R3
100	0.948262	1.111406	1.277754	1.533243	1.231364	1.159823
1000	37.75298	36.90509	35.59452	63.85035	87.72144	79.15281

Table 7.9: Scheduling feature cProfile results during idle state, between 15 and 25 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.159	0.159	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
4	0.000	0.000	0.001	0.000	apiutils.py:67 logmsg
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.009	0.009	processhandler.py:16 is_running
1	0.000	0.000	0.026	0.026	processhandler.py:4 <module>
1	0.000	0.000	0.000	0.000	schedulemanager.py:10 <module>
1	0.000	0.000	20.740	20.740	scheduling.py:13 <module>
1	0.000	0.000	0.000	0.000	scheduling.py:119 SchedulingHTTPRequestHandler
1	0.000	0.000	0.000	0.000	securitymanager.py:2 <module>

Table 7.10: Scheduling feature cProfile results while handling a customer POST request, between 15 and 25 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.170	0.170	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
6	0.000	0.000	0.001	0.000	apiutils.py:67 logmsg
1	0.000	0.000	0.000	0.000	apiutils.py:82 fix_request
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.012	0.012	processhandler.py:16 is_running
1	0.000	0.000	0.027	0.027	processhandler.py:4 <module>
1	0.000	0.000	0.000	0.000	schedulemanager.py:10 <module>
1	0.000	0.000	18.550	18.550	scheduling.py:13 <module>
3	0.000	0.000	0.000	0.000	scheduling.py:211 read_pass_data
1	0.000	0.000	0.005	0.005	scheduling.py:263 do_POST
1	0.000	0.000	0.006	0.006	scheduling.py:320 __init__
1	0.000	0.000	0.000	0.000	scheduling.py:119 SchedulingHTTPRequestHandler
1	0.000	0.000	0.005	0.005	scheduling.py:80 _validate_request
1	0.000	0.000	0.000	0.000	securitymanager.py:2 <module>

Table 7.11: Scheduling feature cProfile results while handling a customer GET request, between 15 and 25 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.163	0.163	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
4	0.000	0.000	0.000	0.000	apiutils.py:67 logmsg
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.010	0.010	processhandler.py:16 is_running
1	0.000	0.000	0.027	0.027	processhandler.py:4 <module>
1	0.000	0.000	0.000	0.000	schedulemanager.py:10 <module>
1	0.000	0.000	22.853	22.853	scheduling.py:13 <module>
3	0.000	0.000	0.005	0.002	scheduling.py:211 read_pass_data
1	0.000	0.000	0.000	0.000	scheduling.py:221 <listcomp>
1	0.000	0.000	0.005	0.005	scheduling.py:243 do_GET
1	0.000	0.000	0.005	0.005	scheduling.py:320 __init__
1	0.000	0.000	0.004	0.004	scheduling.py:43 _handle_kompsat
1	0.000	0.000	0.004	0.004	scheduling.py:69 _handle_paz
2	0.000	0.000	0.000	0.000	scheduling.py:74 datetime_str_for_response
1	0.000	0.000	0.000	0.000	scheduling.py:119 SchedulingHTTPRequestHandler
1	0.000	0.000	0.000	0.000	securitymanager.py:2 <module>

Table 7.12: Passfile handler cProfile results during idle state, around 10 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.212	0.212	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
2	0.000	0.000	0.000	0.000	apiutils.py:67 logmsg
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	10.226	10.226	passfile_handler.py:12 <module>
2	0.000	0.000	0.001	0.000	passfile_handler.py:34 split_passed_passes
1	0.000	0.000	0.000	0.000	passfile_handler.py:91 add_confirmed
1	0.000	0.000	0.008	0.008	processhandler.py:16 is_running
1	0.000	0.000	0.026	0.026	processhandler.py:4 <module>

Table 7.13: Passfile handler cProfile results while handling an overpass, around 10 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.225	0.225	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
2	0.000	0.000	0.000	0.000	apiutils.py:67 logmsg
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
1	0.000	0.000	0.000	0.000	logcleaner.py:3 <module>
1	0.000	0.000	11.446	11.446	passfile_handler.py:12 <module>
2	0.000	0.000	0.003	0.002	passfile_handler.py:34 split_passed_passes
1	0.000	0.000	0.001	0.001	passfile_handler.py:91 add_confirmed
1	0.000	0.000	0.009	0.009	processhandler.py:16 is_running
1	0.000	0.000	0.026	0.026	processhandler.py:4 <module>

Table 7.14: Scheduling feature reviver cProfile results during idle state, 40 - 50 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.195	0.195	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
1	0.000	0.000	0.034	0.034	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
11	0.002	0.000	0.116	0.011	processhandler.py:16 is_running
1	0.000	0.000	0.196	0.196	processhandler.py:4 <module>
1	0.000	0.000	45.903	45.903	s_reviver.py:10 <module>

Table 7.15: Scheduling feature reviver cProfile results while reviving the feature, 40 - 50 seconds of execution time. Times are in seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	0.000	0.000	0.000	0.000	apiutils.py:19 conf_list
1	0.000	0.000	0.194	0.194	apiutils.py:2 <module>
1	0.000	0.000	0.001	0.001	apiutils.py:29 update_configuration
1	0.000	0.000	0.033	0.033	emailmanager.py:17 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:18 <module>
1	0.000	0.000	0.000	0.000	exceptionmanager.py:24 ExceptionManager
1	0.000	0.000	0.000	0.000	exceptionmanager.py:25 __init__
11	0.002	0.000	0.121	0.011	processhandler.py:16 is_running
1	0.000	0.000	0.194	0.194	processhandler.py:4 <module>
1	0.000	0.000	0.004	0.004	processhandler.py:51 execute
1	0.002	0.002	44.855	44.855	s_reviver.py:10 <module>

Table 7.16: IF count on main program and its modules

apiutils.py: 7	logcleaner.py: 4
customer.py: 0	modem_metrics_2s.py: 0
datacontainer.py: 0	overpassreader.py: 7
deliverable.py: 0	plot_sband_from_files.py: 7
emailmanager.py: 4	processhandler.py: 2
emailtest.py: 0	report_sband_day.py: 0
exceptionmanager.py: 6	sbandjson.py: 3
extslacktest.py: 0	sbandreporter.py: 4
fa_reviver.py: 13	sbandvisualizer.py: 8
fileextracter.py: 0	sccerrorhandler.py: 1
fmiapi.py: 39	slackmanager.py: 3
hardrestartfmiapi.py: 0	testpass.py: 0
init.py: 0	updatemanager.py: 22
total: 142	lines of code: 3515

Table 7.17: IF count on scheduling feature and its modules

apiutils.py: 8	passfile_handler.py: 12
customer.py: 1	processhandler.py: 2
datacontainer.py: 0	s_reviver.py: 8
emailmanager.py: 4	schedulemanager.py: 34
exceptionmanager.py: 6	scheduling.py: 35
init.py: 0	securitymanager.py: 3
logcleaner.py: 7	slackmanager.py: 3
total: 114	lines of code: 2191

Table 7.18: Main program Halstead metrics, old vs new

Metric	Old	New
Operators count	1309.00	11414.00
Distinct operators	37.00	49.00
Operands count	949.00	8144.00
Distinct operands	205.00	1277.00
Program length	2258.00	19558.00
Program vocabulary	242.00	1326.00
Volume	17880.79	202872.49
Difficulty	85.64	156.25
Effort	1531337.30	31698311.00
Time to implement	85074 s = 23 h	1761017 s = 489 h
Number of delivered bugs	4	33

Table 7.19: Scheduling feature Halstead metrics, old vs new

Metric	Old	New
Operators count	760.00	8610.00
Distinct operators	31.00	49.00
Operands count	553.00	5921.00
Distinct operands	164.00	873.00
Program length	1313.00	14531.00
Program vocabulary	195.00	922.00
Volume	9988.42	143110.34
Difficulty	52.27	166.17
Effort	522047.45	23780332.08
Time to implement	29003 s = 8 h	1321123 s = 366 h
Number of delivered bugs	2	28

8. Conclusion

The existing SOGS-API was rewritten to comply with new requirements, such as support for multiple customers, better handling of faults, and sending S band plots and error reports through email. During development, the need of additional features, such as revival from crash, was recognized and implemented. Schedule Manager, log cleaner and other support tools were not required, but were implemented to make maintenance of the API easier.

8.1 Measurement Results

SOGS-API meets all the functional requirements described in the introduction. It does also satisfy all the non-functional requirements, although there is room for improvement on the complexity. Tests show that the API can handle a large number of customers, and if ever needed, more instances of the API can be made.

8.2 Further Development

The API can and will be developed further. The earliest change will be to split the scheduling feature program into one or two more modules, reducing the complexity of the largest files. This I will do myself at the end of this project. At the point, when HTTP POST messages can be sent from the operational server to Slack through the firewall, this feature will be enabled. Also, when the scheduling feature is taken into real use by the customers, a few changes should be made in the main program to make it read overpass times from the scheduling feature and not from a file elsewhere. The Schedule Manager tool can be further developed to enable easier editing of pass plan data, instead of just showing them with filtering and ordering options.

9. Swedish Summary - Svensk sammanfattning

SOGS-API: Ett API för satellitdatamottagning

Applikationsprogrammeringsgränssnitt (API) är behändiga för att lägga till funktionalitet i klienters program. Meteorologiska Institutet (MI) erbjuder *SOGS-API* för tillgång till organisationens satellitdatamottagningstjänster. SOGS-API existerade innan detta projekt började, men det hade lagts samman hastigt och saknade funktioner. Målet är att utveckla ett tillförlitligt, säkert och underhållbart applikationsprogrammeringsgränssnitt. Endast kunder och anställda har tillgång till API:et, som används främst via kommandotolken. SOGS-API implementeras med tanke på att det kan behöva vidareutveckling, eftersom kunders krav kan förändras. Då projektet startade, var de funktionella kraven:

- att låta kunder se sina satellitövergångstider utan att se andras
- att planlägga kunders satellitövergångstider
- att förmedla statusinformation om antennen och dess komponenter till kunder
- att automatiskt finna fel och rapportera dem till MI och kunder, samt
- att granska förmedlingen av data.

Funktioner och data beror på kunden.
För att försäkra att det nya API:et är bättre än det förra, skrivs programkoden mer läsbart enligt Pythons stilguide PEP-8, inklusive kommentering och dokumentering. API:et testas grundligt, även för speciella situationer, såsom för elavbrott i antennen.

Förbättringarna utvärderas med mätningar av kvalitetsattribut, d.v.s. icke-funktionella krav. API:ets viktigaste kvalitetsattribut är tillförlitlighet, säkerhet och underhållbarhet. Tillgänglighet för kunderna beaktades också. Tillförlitlighet testades med specialsituationer samt fuzz-testning. Under projektets gång kördes SOGS-API på en testserver. Även det operativa API:et uppdaterades ofta, och det gick igenom flera speciella situ-

ationer, som det antingen hanterade väl, eller som det misslyckades med och därefter utvecklades för att klara av att hantera det. Även prestandan mättes. API:et kördes i en säker miljö, bakom brandmurar och med begränsad tillgång. Det nya API:ets säkerhetsåtgärder var autentisering av kunders förfrågningar och planläggningsverktyget *Schedule Manager*, samt verifiering av specialkommandon, såsom avstängning, via en lokal textfil. API:ets säkerhet förbättrades ytterligare med funktioner för att upptäcka och rapportera försök till missbruk, såsom brytning av kunders identifikationsnycklar eller lösenordet till planläggningsverktyget. API:et blockerar IP-adresser, som påvisat misstänkt aktivitet. För att göra API:et mer underhållbart, lades prestation på kodkvalitet, dokumentering och användbarhet för operatorer. Konfigurationsfiler och designprinciper användes för att förbättra modifierbarheten. Tillgängligheten till kunder stärktes med undantagshantering och tolerans, samt med verktyg för dess största hot, uppdatering. Uppdateringens effekt på tillgängligheten minskades ytterligare genom att komma överens om ett tidsintervall då API:et kan uppdateras utan att behöva kontakta kunderna, och att kunderna alltid kontaktas i god tid före uppdateringar som äger rum utanför den bestämda tiden.

Förutom SOGS-API, förbättrades ett par verktyg relaterade till API:et, såsom de jag hade gjort under sommarpraktiken hos MI. Programmeringsarbetet under projektet utfördes för det mesta av mig. Grunddesignen hade redan gjorts till första versionen av API:et. Under utvecklingen, fattade jag några nya designbeslut vilka accepterades av min chef.

Nyttan med att ha ett API för satellitdatamottagningstjänster är att det gör det lättare för kunderna att komma åt data om mottagningen. Dessa data kan utnyttjas för felgranskning av satellitdata. Detta är viktigt, eftersom korrekta satellitdata är värdefulla, och används bland annat i undersökningar och vid granskning av havets issituation, vilket i sin tur inverkar på isbrytarnas arbete. Nyttan med att API:et är tillförlitligt är att det kan hantera flera kunder och satelliter, och tål specialsituationer. Säkerhet är av största betydelse för MI, då det förhindrar missbruk i organisationens servrar, samt kunderna, då inget skadligt serveras åt dem, eller data faller i fel händer. Underhållbarhet är till nytta för MI, genom att förenkla reparation och uppdatering av API:et. Det är också till nytta för kunderna, eftersom reparationer kan göras snabbare och pålitligt, och således hålls tjänsterna igång.

Den första versionen av SOGS-API kunde frekvent läsa data från antennen med TCP/IP, och skicka det vidare till kunder med UDP. SOGS-API tog emot en lista över satellitövergångstider, och då en övergång skedde, läste och sände API:et antenndata varannan sekund. API:et använde en enda konfigurationsfil och det skrev händelser i dagliga loggar. API:et hade skrivits i Python 2, vilket var nästan föråldrat då förbättringsprojektet satte

igång.

Första ändringen i SOGS-API var uppdateringen av koden till Python 3. Andra var organiseringen av kundernas konfigurering i separata filer. Därefter förbättrades loggningen genom att samla S-band-modemdata, och programmera API:et att märka när antennens server inte fungerar som den ska. Det fanns ett fel i servern, vilket orsakade att tusentals tomma meddelanden sändes per sekund och det tidigare API:et fyllde loggfilen med varje meddelande. Det nya API:et märker när det kommer en serie med tomma meddelanden, och meddelar detta till IT-administrationen via e-post och Slack meddelande och slutar logga tomma meddelanden efter att tre tomma meddelanden mottagits. Senare utvecklades mer avancerad loggning och rapportering, som även för statistik över felsituationer och undviker att skicka samma meddelanden flera gånger inom en kort tid. Även loggfilens storlek granskas. Ifall storleken på dagens loggfil överskrider en konfigurerad gräns, loggas endast de viktigaste meddelandena.

När det i API:et uppstår ett problem som inte försvinner, försöker det reparera det genom att starta om sig själv. Hur många gånger API:et startas om och hur länge det väntar mellan omstart är konfigurerbart. Vid ett tillfälle stannade det operativa API:et utan att meddela varför. Då samma fenomen upprepades en vecka senare, upptäcktes ett mönster. Felet uppstod efter att data för S-band-modemet sparats i en CSV-fil, och innan kurvan sparades i en PNG-fil. Detta problem döptes till *CSV -> PNG-problemet*, och satte igång skrivandet av ett program som återupplivar API:et ifall det stannar. Det har visat sig vara en mycket nyttig egenskap, som gjort liknande fel mindre allvarliga. Enda möjliga konsekvensen med felet vore att MI:s operatorer inte automatiskt skulle få S-band-modemdatakurvan skickad till sin e-post. Bilden går att rita i efterskott, och verktyg för detta gjordes också. CSV -> PNG-problemet har iakttagits sedan det först upptäcktes, men det har inte upprepats.

En annan förbättring var sättet som API:ets processer stängs av på. Först gjordes det möjligt att stänga av processerna rent via textfiler. Nackdelen med att läsa ett *stäng av*-kommando från en textfil är dröjsmålet. Därför utvecklades möjligheten att stänga av via SIGINT-signal. Tyvärr använder återupplivarprogrammet subprocesser, vilket kan blockera SIGINT-signaler. Därför bevarades både textfilsmetoden och SIGINT för avstängning av API:et. Även möjligheten att tvinga till avstängning förenklades, men rekommenderas att användas bara när de rena metoderna inte fungerar. Under projektets gång förbättrades andra verktyg relaterade till SOGS-API. Dessa kallades *Satellite Data Flow Monitor* och *Acquisition Issue Reporter*. Verktygen förbättrades genom tillägget av automatisk organ-

isation av satellitövergångslogfiler. Detta förbättrade prestandan genom att minska på söktiden bland filer.

SOGS-API är ett API som skickar antennens situationsdata till MI:s kunder. Då API:et startas, läser det dess moduler och konfiguration och går till *normalläge*. I *normalläge*, vilket också kallas *icke-övergångsläge*, skickar API:et statusdata var 30 sekund till kunder och MI:s loggningsservice. API:et granskar även övergångstidslistan och eventuella uppdateringar i kundernas konfiguration. När satelliten är ovanom antennen, övergår SOGS-API till *övergångsläget*. I detta läge sänder API:et atennkontrollenhetens lägesdata samt S-band-modemets data till kunder och MI varannan sekund. När satellitövergången är förbi, övergår API:et tillbaka till *normalläge*.

Processen i SOGS-API som skickar S-band-modemdatakurvor per e-post heter *S-band Reporter*, och kallas rapporteraren. Rapporteraren startar med att ladda relevanta moduler och konfiguration. Därefter startar den en process, där den skickar S-band-modemdatakurvorna och städar loggfilerna varje dag klockan 5:00 global tid, vilket i Finland är sju eller åtta på morgonen, beroende på om det är sommar eller vinter. Rapporteraren kör loggstädningen eftersom det är behändigt att göra en gång per dag.

Processen *FMI API Reviver* i SOGS-API återupplivar andra processer. Återupplivaren startar med att ladda relevanta moduler och konfiguration. Var 10 sekund granskar den sedan en textfil för dess eget avstängningskommando. Ifall den inte ska stängas av, granskar den om de andra processerna i API:et är igång, och återupplivar dem vid behov. Återupplivaren märker även om huvudprogrammet har fastnat, och isåfall startar den om huvudprogrammet.

Kunder och produkter definieras i konfigurationsfiler. Om det görs ändringar i kund- eller produktdata, kan dessa uppdateras medan API:et är igång, via uppdateringsfiler som API:et läser varannan minut. Således behöver API:et inte nödvändigtvis startas om efter en konfigurationsuppdatering.

S-band-modemdata tas emot i JSON-format via ett Python-skript. JSON-texten förfinas och skickas till kunderna. Några viktiga värden läses också och sparas i en CSV-fil, varefter de ritas i en kurva med Python-modulen *Matplotlib* i en PNG-fil.

Scheduling feature är ett program i SOGS-API som togs i bruk i ett senare skede av projektet. Ur kundens synvinkel är scheduling feature en egenskap i API:et. Python 3 kod till

scheduling feature existerade före projektet, och koden uppdaterades, renskrevs och testades innan det sattes igång. Satellitövergångstiderna skickades av en kund, och tidigare var programmet döpt efter denna kund. Senare döptes API:et om till *scheduling feature*, ett namn som beskriver vad programmet gör och som inte avslöjar kundens namn eller gör det svårt att urskilja e-postmeddelanden som sänts av programmet från de som sänts av kunden. Senare kan det hända att andra kunder också skickar satellitövergångstider, och då är ett generellt namn mer lämpligt.

Verktyget *Schedule Manager* gjordes för lättare granskning av satellitövergångstider då flera kunder använder *scheduling feature*. Det är ett HTML gränssnitt, som ger filtrerings- och organiseringsalternativ för visning av övergångstider.

Tillförlitlighet mättes genom att samla resultat från specialsituationer. *Scheduling feature* gick igenom Fuzz-testning, och hela SOGS-API klarade av speciella situationer, såsom elavbrott och uppdateringar i antennen.

Säkerheten mättes med skript och verktyg som försöker gissa kunders ID, samt *Schedule Manager*-verktygets adress och lösenord. Då kundernas ID testades, användes tio stycken 32 teckens hexadecimalvärden, och inte ett enda ID gissades rätt på en miljon försök, fastän "attackeraren" visste detta. I verkligheten kan ID innehålla text av varierande längd, vilket gör gissningen ännu svårare. Onlineverktyget *How Secure Is My Password?* användes för värdering av gissbarheten av *Schedule Manager*-verktygets adress och lösenord. Enligt verktyget gissade den adressen genast. Adressen gjordes längre och svårare, varefter det skulle ta en dag för en dator att gissa adressen. Att gissa lösenordet skulle kräva 300 år för en dator. Senare gjordes en funktion som märker försök till missbruk och blockerar attackerarens IP-adress.

Prestandan mättes med Python-modulen *cProfile* och genom användning av tidsstämplar. Exekveringstiderna visade inga problem med prestandan i normala situationer. Då *scheduling feature* sattes under hård trafik, observerades en liten fördröjning i handlingen av förfrågningar. Fördröjningen är så liten, att den skulle märkas först när tusentals kunder använder API:et.

Underhållbarhet består av modifierbarhet, testbarhet och begriplighet, och dessa kan brytas ned till förstärkbarhet, struktur, kommunikation, tillgång, självbeskrivning, koncishet och läsbarhet. API:erna använder konfigurationsfiler, vilket främjar självbeskrivbarhet, förstärkbarhet och struktur. Ansvaret har delats in i moduler, vilket lägger till struktur

och självbeskrivbarhet. Koden följer stilguiden PEP-8, vilket främjar läsbarheten. Stilen granskades med verktyget *pycodestyle*. Verktyget hittade 61 fel i gamla SOGS-API och 349 i gamla Scheduling API. Beskrivande variabel- och funktionsnamn förbättrade också läsbarheten, och där kontexten är klar användes kortare namn för att göra koden mer koncis. Längre modulnamn förkortades med alias för att också förbättra koncisheten. Operatorerna hos MI har full tillgång till API:erna samt deras filer, och det finns verktyg och dokumentation som främjar kommunikationen. IF-förhållandet mättes, och det visade sig vara ganska högt för båda API:erna. I båda fallen fanns den goda sidan att största delen av *if-satserna* fanns i två filer, istället för att vara utspridda. Också *Halsteads*-värden mättes, och resultatet var att det finns behov att dela Scheduling API i fler moduler för att minska på komplexiteten.

Lösningen uppfyller alla de funktionella, och icke-funktionella, kraven, men det finns utrymme för förbättring vad gäller komplexiteten. API:et kommer att vidareutvecklas. Den första förändringen kommer att vara delningen av Scheduling API i fler moduler, för att minska på komplexiteten. När det går att skicka Slack-meddelanden genom brandmuren, kommer denna egenskap att tas i bruk. När Scheduling API börjar användas av kunderna, behöver några ändringar göras i SOGS-API så att den läser övergångstider från Scheduling API istället för en fil annanstans. *Schedule Manager*-verktyget kan vidareutvecklas så att det tillåter editering av övergångstiderna.

Bibliography

[1] Redfox Languages, *Redfox Master - Online Dictionary*, https://client.redfoxsanakirja.fi/en_US/dictionary, [Online and accessed 25-September-2020], 2020.

[2] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD thesis, University of California, Irvine, 2000.

[3] I. Cotton and F. Greatorex, Eds., *AFIPS 1968 Fall Joint Computer Conference*, San Francisco, California: Association for Computing Machinery, 1968.

[4] Google LLC, *YouTube Data API*, https://developers.google.com/youtube/v3/getting-started, [Online and accessed 11-December-2020], 2020.

[5] D. Berlind, W. Santos, and K. Sundström, *ProgrammableWeb - Category: Satellites*, https://www.programmableweb.com/category/satellites/api/, [Online and accessed 10-December-2019], 2019.

[6] J. Macy, "API security: whose job is it anyway?" *Network Security*, vol. 9, no. 1, 9:6–9:9, 2018. [Online]. Available: https://doi.org/10.1016/S1353-4858(18)30088-6.

[7] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[8] S. May, *What Is a Satellite?* https://www.nasa.gov/audience/forstudents/5-8/features/nasa-knows/what-is-a-satellite-58.html, [Online and accessed 5-December-2020], 2017.

[9] A. Mulla and P. Vasambekar, "Overview on the development and applications of antenna control systems," *Annual Reviews in Control*, vol. 41, pp. 47–57, 2016, ISSN: 1367-5788. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367578816300153.

[10] P. Zhang, "CHAPTER 8 - Industrial process controllers," in *Advanced Industrial Control Technology*, P. Zhang, Ed., Oxford: William Andrew Publishing, 2010, pp. 307–344, ISBN: 978-1-4377-7807-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781437778076100087.

[11] L. O'Brien, L. Bass, and P. Merson, "Quality Attributes and Service-Oriented Architectures," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2005.

[12] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond, Second Edition*. Boston: Addison-Wesley, 2010.

[13] P. Naur and B. Randell, Eds., *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968*, Brussels: NATO, Scientific Affairs Division, 1969.

[14] A. Case, "Computer-Aided Software Engineering (CASE): Technology for Improving Software Development Productivity," vol. 17, no. 1, 1985. [Online]. Available: `https://doi.org/10.1145/1040694.1040698`.

[15] D. Perry and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 4:40–4:52, 1992. [Online]. Available: `https://www.ics.uci.edu/~taylor/classes/221/Wolf-Perry-1992.pdf`.

[16] E. Freeman, K. Sierra, B. Bates, and E. Robson, *Head First Design Patterns*. O'Reilly Media, 2004.

[17] J. Coplien, *Idioms and Patterns as Architectural Litterature*. IEEE Software, 1997.

[18] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Third Edition*. Boston: Addison-Wesley, 2012.

[19] Verifysoft Technology, *Halstead Metrics*, `https://www.verifysoft.com/en_halstead_metrics.html`, [Online and accessed 8-September-2020], 2017.

[20] R. Al-Qutaish and A. Abran, *Software Metrics and Software Metrology*. Hoboken, NJ: John Wiley & Sons-IEEE, 2010.

[21] Lite Solutions, *Halstead*, `https://objectscriptquality.com/docs/metrics/halstead`, [Online and accessed 10-September-2020], 2020.

[22] M. Pizka and F. Deißenböck, "How to effectively define and measure maintainability," *ResearchGate*, vol. 1, no. 1, 1:5–1:8, 2007. [Online]. Available: `https://www.researchgate.net/publication/250188492`.

[23] P. Jalote, B. Murphy, M. Garzia, and B. Errez, "Measuring Reliability of Software Products," Microsoft Corporation, One Redmond Way, Redmond, WA 98052, Tech. Rep., 2004, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2004-145.pdf.

[24] K. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications, Second Edition*. Hoboken, NJ: John Wiley & Sons, 2002.

[25] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance, (2nd ed)*. Norwood, MA: Artech House, 2018.

[26] M. Sutton, A. Greene, and P. Amini, *Fuzzing Brute Force Vulnerability Discovery*. Boston: Addison-Wesley, 2007.

[27] P. Godefroid, M. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *ACM Queue*, vol. 10, p. 20, 2012.

[28] Sayantini, *Top 10 Performance Testing Tools – Your Ultimate Guide to Testing*, https://www.edureka.co/blog/performance-testing-tools/, [Online and accessed 22-September-2020], 2020.

[29] Python Software Foundation, *The Python Profilers*, https://docs.python.org/3/library/profile.html, [Online and accessed 15-Ferbuary-2021], 2021.

[30] G. Yee, *Computer and Information Security Handbook, Second Edition*. Ottawa, Canada: Carleton University, 2013.

[31] Python Software Foundation, *Python*, https://www.python.org/, [Online and accessed 20-January-2020], 2020.

[32] G. van Rossum, *A Brief Timeline of Python*, http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html, [Online and accessed 6-October-2020], 2009.

[33] D. Kuhlman, *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. http://www.davekuhlman.org, 2009.

[34] Python Software Foundation, *The Python Debugger*, https://docs.python.org/3/library/pdb.html, [Online and accessed 6-October-2020], 2020.

[35] J. Rocholl, F. Xicluna, and I. Lee, *pycodestyle's documentation*, https://pycodestyle.pycqa.org/en/latest/intro.html, [Online and accessed 31-March-2020], 2016.

[36] G. van Rossum, B. Warsaw, and N. Coghlan, *PEP 8 – Style Guide for Python Code*, https://www.python.org/dev/peps/pep-0008/, [Online and accessed 21-March-2020], 2013.

[37] Anaconda Inc., *Miniconda*, https://docs.conda.io/en/latest/miniconda.html, [Online and accessed 6-October-2020], 2017.

[38] WHATWG, *HTML Standard*, https://html.spec.whatwg.org/multipage/introduction.html, [Online and accessed 7-March-2021], 2021.

[39] W3C, *HTML & CSS*, https://www.w3.org/standards/webdesign/htmlcss.html, [Online and accessed 20-March-2021], 2021.

[40] MDN, *What is JavaScript?* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction\#what_is_javascript, [Online and accessed 5-December-2020], 2017.

[41] S. Chacon and B. Straub, *Pro Git, Second Edition*. New York: Apress, 2020.

[42] A. DuVander, *9,000 APIs: Mobile Gets Serious*, https://www.programmableweb.com/news/9000-apis-mobile-gets-serious/2013/04/30, [Online and accessed 25-September-2020], 2013.

[43] W. Santos, *APIs show Faster Growth Rate in 2019 than Previous Years*, https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17, [Online and accessed 25-September-2020], 2019.

[44] W. Griswold, "Program Restructuring as an Aid to Software Maintenance," PhD thesis, University of Washington, Washington, 1991.

[45] Small Hadron Collider, *How Secure Is My Password?* https://howsecureismypassword.net/, [Online and accessed 1-September-2020], 2020.

[46] B. Boehm, T. I. Systems, and Energy, *Characteristics of Software Quality*, ser. Notas de Matematica. North-Holland Publishing Company, 1978, ISBN: 9780444851055. [Online]. Available: https://books.google.com/books?id=Cdm0AAAAIAAJ.

[47] D. Borowiec, *Commented Code Detector*, https://github.com/dborowiec/commentedCodeDetector, [Online and accessed 8-September-2020], 2013.

Appendices

A. ACU Metrics: Raw vs Visualized

ISTT:1001:INST 0:DATA<TM ST 2021 049 03:11:26.059,11001 IN 0,501 UN 459008,574 ST ,575 FL 0.000000,576 IN 0,577 ST NONE,578 IN 0,579 FL 0.000000,502 UN 9218,503 IN 0,504 FL 181.001205,508 FL 181.001205,512 FL 0.000000,516 IN 0,520 IN 0,524 UN 0,528 FL 181.062302,505 FL 9.950500,509 FL 9.950500,513 FL 0.000000,517 IN 0,521 IN 0,525 UN 0,529 FL 181.062302,506 FL 0.061100,510 FL 0.061100,514 FL 0.000000,518 IN 1,522 IN 0,526 UN 512,507 FL 1.000000,511 FL 1.000000,515 FL 0.000000,519 IN 0,523 IN 0,527 UN 4096,532 UN 268435456,533 FL 182.189804,534 FL 16.709499,535 FL 182.189804,536 FL 16.709499,553 FL 0.000000,554 FL 0.000000,537 FL 0.000000,538 FL 0.000000,620 FL 0.421100,621 FL 17.000000,539 FL 0.421100,540 FL 0.186200,541 FL -0.058000,542 FL 21.021400,543 FL 0.000000,544 FL 0.000000,625 FL 10.700000,626 FL 10.800000,627 FL 10.900000,628 FL 11.100000,545 FL 17.000000,546 FL 10.000000,547 FL 10.000000,548 FL 0.000000,549 FL 0.000000,550 FL 0.000000,630 FL 40.700001,631 FL 40.799999,632 FL 40.900002,633 FL 41.099998,600 ST 10|0|0|0|0|0|0|0|0|0|0,601 ST 19|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0,602 ST 20|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0,603 ST 20|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0,604 ST 16|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0,605 ST 3|9|9|9,610 UN 0,611 UN 0,612 UN 0,613 UN 0,614 UN 8192,615 UN 0,584 ST 23|1|0|0|3|0|0|0|0|0|1|1|0|0|0 UN 0,573 IN 0,580 IN 3,581 IN 0,552 IN 1,585 IN 181,551 IN 0,900 IN 0,901 FL 0.000000,905 FL 0.000000,910 FL 0.000000,911 FL 0.000000,912 FL 0.000000,913 IN 1,400 IN 0,592 IN 0,593 FL 9.900000,594 FL 0.180000,595 FL 0.190000,596 FL 0.019000,660 IN 0,661 FL 1.000000,662 FL 60.000000,663 FL 0.100000,664 FL 0.100000,665 FL 0.100000,813 FL 0.000000,814 FL 0.000000,1050 IN 2,120 DL 0.001,1051 IN 0,1052 FL 0.000000,1053 FL 0.000000,1054 FL 0.000000,1055 FL 0.000000,1056 FL 0.000000,1057 FL 0.000000,1058 FL 0.000000,1101 FL 182.189804,1102 FL 16.709499,1111 FL 246.690994,1112 FL 180.941193,1113 FL 181.001205,1114 IN 1,1121 FL -2.258100,1122 FL 9.700500,1123 FL 9.950500,1124 IN 1,1131 FL 1.226400,1132 FL 0.061100,1133 FL 0.061100,1134 IN 2,622 IN 1,920 IN 0,921 ST NONE>

ACU Unit Flags
Software Fault: **OK**
System Mode: **Manual**
Control Mode: **Remote**
Time Sync Daemon: **OK**
Time Sync Time Status: **OK**
Time Sync Offset: **OK**
GPS: **GPS not used**
PPS: **PPS not used**

ACU Pedd Flags
Coordinate System: **earth**
Readout Format: **uncorrected**
Pedestal Power: disable
Feed Power Not used: disable
Walkbox: **Walkbox not connected**
HPA 1 Inhibit: off
HPA 2 Inhibit: off
Autotrack Inhibit: **on**
Time Synchronization Status: **sync**
Time Synchronization Source: off

ACU 10 Hz Flags
Fault: **Ok**
Warning: **Ok**
Topocentric AZ Mode: Inactive
Topocentric AZ Mode: Inactive
Topocentric AZ Summary Interlock: Clear
Topocentric EL Summary Interlock: Clear
Slave Data Dropout Alarm: Clear
Receiver Selection Mode: **Manual**

Acu View Program Track: off
ACU Ped Power Status: disable

ACU CDF Status
S11: **enable**
S12: disable
S13: disable
S14: **pend_on**
S15: disable
S21: disable
S22: disable
S23: disable
S24: disable
S25: **enable**
S31: **enable**
S32: disable
S33: disable
S34: disable
S35: disable
A7: disable
A8: disable
B7: disable
B8: disable
C7: disable
C8: disable
D7: disable
D8: disable

ACU Fault Status List
No Ped Power Fbk: **Clear**
ESTOP: Base: **Clear**
Base Door: **Clear**
MCS Unconfigured: **Clear**
MCS Config Error: **Clear**
MCS Local Lockout: **Clear**
MCS Phase: **Clear**
Misc MCS Fault: **Clear**
No MCS Comm: **Clear**
Bad MCS Comm: **Clear**

ACU Warning Status List
Power Summary: **Clear**
X-Band RHC Data LNA : **Clear**
X-Band TRK LNA : **Clear**
X-Band LHC Data LNA : **Clear**
S-Band RHC Data LNA: **Clear**
S-Band LHC Data LNA: **Clear**
ACU Outputs #1-8: **Clear**
ACU Outputs #9-16: **Clear**
ACU Autotrack Self Test: **Clear**
ACU to MCS 100Hz Sync (J3-B2): **Clear**
ACU Internal 3.3V Power: **Clear**
ACU Internal 5.0V Power: **Clear**
ACU Overtemp (70°C): **Clear**
AZ Motor 1 Overcurrent: **Clear**
AZ Motor 2 Overcurrent: **Clear**
EL Motor 1 Overcurrent: **Clear**
EL Motor 2 Overcurrent: **Clear**
TRAIN Motor 1 Overcurrent: **Clear**
Misc MCS Warn: **Clear**

Figure A.1: ACU metrics visualized

B. S Band Modem Metrics JSON


```
"targets": {
    "RxIn0/SampledComplex32IfQueued0": {
        "QItemsDiscarded": 0
    },
    "BpskDemod0/BpskDemodV27Fft0": {
        "PowerMeas": 108.6841888427734,
        "ViterbiLock": true,
        "EbnoMeas": 10.35342979431152,
        "SymbolLock": true,
        "SymbolRate": 1562500.0,
        "CarrierLock": true
    },
    "Tlm0/CcsdsAosTransferFrameToTsBits0": {
        "InvalidVcdus": 0,
        "CcsdsVcdus": 0
    },
    "RxIn0/Fmc150ComplexSampleCollector2\_0": {
        "CurrentGain": 62.0,
        "Fmc150InputFilter": "FIR\_12MHZ",
        "SignalClip": 0.0
    },
    "Tlm0/SwFrameSync0": {
        "CheckFrames": 0,
        "FrameLength": 1024,
        "LockFrames": 0,
        "LostLockCount": 0,
        "FsLockState": "INPUT\_SHUTDOWN",
        "SlipFrames": 0,
        "VerifyFrames": 0,
        "TotalFrames": 31
```

```
    },
    "Tlm0/CcsdsAosTransferFrameTx0": {
        "DiscardedFrames": 0,
        "TotalFrames": 0,
        "CcsdsInsertZoneFills": 0
    },
    "TxOut0/FreqControl0": {
        "ExtCenterFreq": 2245000000.0
    },
    "TimeAndRef0/NetPanelTimeProcessor2\_0": {
        "Synchronized": true
    },
    "TimeAndRef0/Fmc150Monitor0": {
        "Alarm": false
    },
    "Tlm0/CcsdsRs255223Decoder0": {
        "TotalBits": 0,
        "TotalFrames": 31,
        "CorrectedBits": 0,
        "CorrectedFrames": 0,
        "UncorrectableFrames": 0
    },
    "RxIn0/FreqControl0": {
        "ExtCenterFreq": 2245000000.0
    },
    "TxOut0/Fmc150ComplexSampleSender2\_0": {
        "ManualGain": 26.0
    },
    "TxOut0/CarrierMixer0": {
        "EnableCarrier": true,
        "EnableModulation": true
    }
}
```

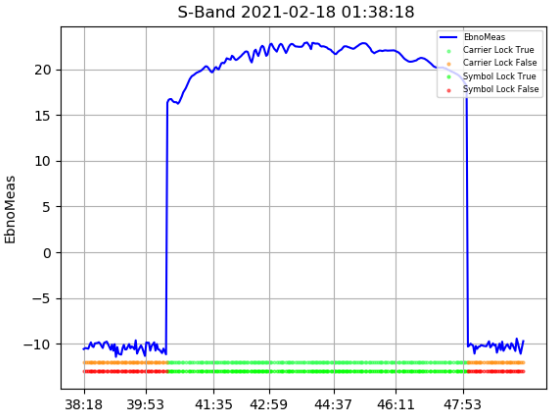

Figure B.1: S band plot