Tartu University

Faculty of Science and Technology

Institute of Technology

Laimonas Suchockas

Automating data management system of 1.5-meter telescope at Tartu Observatory

Master's thesis (30 EAP)
Robotics and Computer Engineering

Supervisors:

MSc Tõnis Eenmäe
PhD Heleri Ramler

Tartu 2020

Resümee/Abstract

1,5-meetrise teleskoobi andmehaldussüsteemi automatiseerimine Tartu Observatooriumis

Lõputöö eesmärk on automatiseerida Tartu Observatooriumi 1,5-meetrise teleskoobi andmehaldussüsteem metaandmete tootmiseks. Hetkel ei ole peaaegu ükski kasutatud instrumentidest üksteisega ühendatud ning seetõttu on vaatlejal vaja teha palju manuaalset tööd. Nimelt tuleb vaatluse käigus vaatlusobjekti ja instrumendi konfiguratsiooni teave lisada kogutud andmete päistesse käsitsi, see on rutiinse ja eriti just öise töö tõttu üsna veaaldis.

Automatiseeritud süsteemi saavutamiseks kasutati modulaarset lähenemist ja astronoomias ning infotehnoloogias standardseid tehnoloogiaid nagu sõnumside järjekorrad (ingl message queue) ja ASCOM. Lisaks pidi süsteem töötama pea-aegu reaalajas, sest teave instrumentide kohta oli oluline, samas kuni mõne sekundi pikkused viivitused olid lubatud.

Lisaks uutele tehnoloogiatele tuli ajakohastada ka juba olemas olevaid tööriistu, kuna enamikul neist puudus tehnoloogiline tugi juba mitu aastat. Eritellimusel valmistatud tarkvara Astrolab, mis annab informatsiooni ja kontrollib teleskoopi, oli kirjutatud Python 2 programmeerimiskeeles ja paigaldatud Linuxi distributsioonile Ubuntu 10.10. Mõlemad neist on aegunud ja selleks, et luua uue andmehaldussüsteemi süsteemi komponente, tuli Astrolab portida programmeerimiskeele Python versiooni 3 värskeimale stabiilsele väljalaskele ning uuendada juhtarvuti operatsioonisüsteemi värskeima pikaajalise toega versioonile.

Lõputöö kirjeldab kõiki andmehaldussüsteemis kasutatavaid tehnoloogiaid, miks need valiti ja kuidas need implementeeriti. Valminud süsteem aitab kasutajal vaatlusi läbi viia lihtsamini, automatiseerides osa protsessidest.

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia; T125 Automatiseerimine, robootika, control engineering;

Märksõnad: ActiveMQ, ASCOM, arvutid, astronoomia, kontroll, tarkvara, teleskoop

Automating data management system of 1.5-meter telescope at Tartu Observatory

The aim of this thesis is to automate data management system of Tartu Observatory's 1.5-meter telescope which deals with metadata production. Currently, almost none of the instruments used for observations are connected to one another and that leads to a great amount of manual work, namely adding information about the instrument configuration into the headers of all collected individual data files.

In order to achieve an automated system, a modular approach was used and it was created with the industry standard technologies such as Messaging Queues and ASCOM. Additionally, the system had to be near real-time as it was important to have information of the instruments, though a delay of few seconds was allowed.

Besides the new technologies, old tools had to be updated as well, since most of them had not been supported for several years. A custom-made software Astrolab, responsible for providing information and control of the telescope, was written in Python 2 programming language and ran on Ubuntu 10.10. Both of these are obsolete and for the newly developed system components to work properly, the old ones needed to be updated as well.

This thesis describes all of the technologies used for the data management system. It provides the reasons behind each tool used as well as how they are implemented. The finished system helps the user observe stars more easily by automating some of the processes.

CERCS: T120 Systems engineering, computer technology; T125 Automation, robotics, control engineering

Keywords: ActiveMQ, ASCOM, astronomy, computers, control, software, telescope

Contents

Resümee/Abstract **2**

Abbreviations. Generic Terms **6**

1 Introduction **7**

2 Overview of control systems used in astronomy **8**
2.1 Custom systems of professional observatories 8
2.2 Messaging Queues 8
2.3 Technologies used in observatories 9
2.3.1 ASCOM standards 9
2.3.2 INDI 10
2.3.3 TANGO 10

3 Instruments and their control system at Tartu Observatory **11**
3.1 Overview of the observation process at Tartu observatory 11
3.2 Initial situation 11
3.2.1 Hardware situation 13
3.2.2 Software situation 13
3.3 The problem 14
3.4 Possible solution 14
3.4.1 Requirements for the system 14
3.4.2 The solution 15

4 Development of the automated data management system **16**
4.1 Converting Astrolab from Python 2 to Python 3 16
4.1.1 Analysis of Astrolab 16
4.1.2 Development environment for Astrolab 17
4.1.3 Python module 2to3 18
4.1.4 MySQL library 19
4.1.5 Trying to edit or select an object 19
4.1.6 Name 'open' is not defined 20
4.1.7 Testing on site 22
4.2 ActiveMQ module for Astrolab 22
4.2.1 Communication with ActiveMQ 22
4.2.2 Connection 23
4.2.3 Data format 23

4.2.4 Populating the element tree with data 24
4.3 RESTful service based on ASCOM ALPACA API 24
4.3.1 Benefits of ASCOM 25
4.3.2 Developed RESTful service 25
4.3.3 Camera control software 27

5 Results **29**

6 Analysis and discussion **30**

7 Conclusion **32**

Bibliography **34**

Appendix A Data sent through the system **36**

Appendix B Source Code **37**

Non-exclusive license **38**

Abbreviations and Generic Terms

ActiveMQ - Open source Message Broker written in Java.

API - An application programming interface which is a set of functions that acts as a way to access features of a service, application or an operating system

ASCOM - Open initiative providing common interfaces for astronomy equipment.

ASCOM Alpaca - Also known as ASCOM Remote, is a technology that allows ASCOM supported devices to connect remotely using RESTful services.

Astrolab - Specifically for Tartu Observatory's 1.5-meter telescope developed to control and have information about the telescope.

Dome - A building as well as an instrument that covers a telescope and protects it from weather conditions while allowing the temperature inside to be the same as outside.

Grating - An optical component that splits and diffracts the light into beams based on the wavelength. It is used in spectroscopy to analyse the spectral components of an object.

Messaging Queue - Software engineering components used for inter-process communication.

MySQL - An open-source relational database management system.

REST - Representational state transfer is a architectural style which defines constrains of a web service. The greatest benefit of this style is that it is stateless and the server does not retain any session information.

Telescope - An optical instrument which uses mirrors and lenses in a way that makes observed objects appear magnified.

Telescope position - Celestial coordinates where telescope is pointing at.

Virtual machine - An emulation of a computer system.

1 Introduction

In Tartu Observatory the 1.5-meter telescope system, consisting of telescope, dome, cameras, spectograph and other instruments, is controlled by different computers and controllers. One device stores and provides all information about the telescope's current orientation and the observed target. A few other devices independently control separate functions of the telescope and its instruments - such as the angle of the spectrograph grating and the fibre bundle illuminating the spectrograph slit by a wavelength comparison source light. However, most of the users' work is executed on a completely different computer where the observations are saved. Consequently, whenever a researcher observes a star and wants to save its recorded spectrum on the computer, the whole information about the observation itself has to be entered manually, which is prone to error.

A possible solution to this problem is to automate the data management by developing a system that would have a modular approach, where devices could be added and changed independently without breaking the system. This is achievable using current industry standards, such as various Messaging Queues (ActiveMQ [1], ZeroMQ [2], RabbitMQ [3], etc.) that are used e.g. in highly sophisticated telescopes such as High Accuracy Radial velocity for Planetary Search North (HARPS-N) [7] and Australian SKA Pathfinder [6]. The completed system is meant to be multi-platform, implying that to the system it should not matter what operating systems are running on the devices attached to it. Finally, one of the main requirements for the system is that it should be near real-time, allowing both: retrieval of status information of the telescope and its instruments, and possibly the control of some devices.

To achieve the final result it is required to fully analyse what currently exists in the system and how it is structured. After that the main goal is to convert the currently existing custom telescope control software Astrolab from officially obsolete Python 2 to the current stable version of Python 3 programming language. Additionally, Astrolab needs to have a new functionality written, specifically the ability to provide information to ActiveMQ service, for which the service needs to be set up and configured appropriately. After Astrolab and ActiveMQ are properly set up, it is important to write a script that takes information from the ActiveMQ and exposes it to a centralised device that works with all the information available in the system. This is required because the centralised device processes the information instead of each device having to do that, and this allows a modular and expandable approach to the system. It can be achieved by creating a REST interface to allow ASCOM-enabled [19] software to access remote devices. The finished project will provide a beginning for a user-friendly environment for data management and storage, with possible telescope control.

2 Overview of control systems used in astronomy

The industry is not fresh to the idea of automating observations, which means that ways of achieving an automated system exist. The strength of new instruments is that most of them already support tools and technologies that enable automation of systems. This reduces the need of manually doing it. However, there are still observatories that have created their own custom systems. This chapter describes technologies and approaches that are used in control systems in the field of astronomy.

2.1 Custom systems of professional observatories

Most of the professional observatories around the world have automated at least some parts of their systems and all of the new telescopes are highly automated. Many small observatories make use of the commonly available hardware and software. For example, Australian SKA Pathfinder (ASKAP for short) telescope uses a Messaging Queue for communication between software components [6]. In their case every component used in the observatory is connected to the Messaging Queue which allows multiple devices to retrieve information about each component. Another professional system - High Accuracy Radial velocity for Planetary Search North (HARPS-N) located in La Silla, Chile, uses ActiveMQ to handle the exchange of data between processes [7]. HARPS-N system has a sequencer that automates the observations by providing templates from a scheduler software to the instruments that include a telescope, a spectrograph and other devices. Considering the importance of these telescopes in the world, their system architectures - specifically, connecting instruments together through a Message Broker - can be used as reference to automate Tartu Observatory's 1.5-meter telescope's system.

2.2 Messaging Queues

Messaging Queue (also known as Message Broker) is a component of software engineering that is used to enable communication between different processes. It works in such a way that a device (called publisher) uploads information (called a message) in a queue and then the publisher can disconnect. The message stays in the queue until another device (called subscriber) tries to access the message. In that instance the message is sent to the subscriber and is deleted from the queue. This is the main feature of how Message Brokers work. Some have optional functions such as keeping a log of messages or letting multiple subscribers receive a message.

For a specific Message Broker, there are many options such as Apache ActiveMQ [1], RabbitMQ [3], Microsoft Azure [4], Amazon Simple Queue Service [5] and many more Messaging Queue tools. To decide which one to use for the thesis, other telescope system designs were

analysed. Afterwards, options were evaluated based on available functions as well as other users' reviews.

First of all, it is important to note that one of the most popular Message Brokers used in telescope control and data management systems around the world is ActiveMQ. World-known telescopes such as GEMINI and HARPS-N [7] use ActiveMQ, while ASKAP chose to use another tool called Internet Communications Engine because it has a strict Interface Definition Language [6] which does not apply to this thesis. Taking that into account, ActiveMQ was already the likely choice as it would have more support towards Astronomy-related needs. However, other Messaging Queues were studied as well. For that it was important to specify the main requirements for a Messaging Queue. They include:

- Multi-platform - different OS need to be able to connect to the Messaging Queue through different languages.
- Stability - it is important for a Message Broker to be able to work throughout the night while observing objects in the sky.
- Industry support - it is important for a tool to be used in similar scenarios in the Industry.

Other options, apart from the previously mentioned ActiveMQ, were RabbitMQ and Kafka [10], while the rest were not considered as they are not available for free. To start with Apache Kafka, it has great scalability and it supports high volumes of data. However, that might as well be its greatest drawback as the use case in Tartu Observatory would not reach even half of what Apache Kafka supports because majority of the functions provided by Apache Kafka are not needed. Additionally, there were no found cases where Apache Kafka was used for Astronomy. Next, RabbitMQ is another open-source Messaging Queue used in a wide array of fields. One use case for RabbitMQ is The Gaia Added Value Interface Platform mission [9] by European Space Agency (ESA). Both RabbitMQ and the previously mentioned Apache Kafka are reported to be stable and support multiple OS and language connections. In the end, Apache ActiveMQ was chosen for its use in similar systems, support of message logging and a highly configurable connection scheme.

2.3 Technologies used in observatories

This section describes some of the technologies used in the amateur community. The section lists ASCOM, INDI and Tango, which all can connect hardware to software. The main distinctions being that ASCOM and INDI are primarily aimed for astronomy tools while Tango is used more broadly in control of scientific research equipment. For the thesis it is important to consider amateur tools for a professional research lab as well as professional solutions because in case of astronomy, the amateur technologies are mature enough to be used in professional applications. Additionally, Tartu Observatory already has one telescope that operates with devices and programs that are used in the community of amateur astronomers.

2.3.1 ASCOM standards

ASCOM [19], as described by the creators, is an open initiative that aims to decrease vendor dependency. This means that ASCOM is a way to standardise interfaces to a range of astronomy instruments. It is achieved by providing an application programming interface (API) that developers of both software and hardware can use for communication. The project started in the

late 90's and by now it is more widely used by the companies making astronomy instruments for amateur astronomy community. Initially, it was required for instruments to be directly connected to the computer with the software, however, from the year 2018 ASCOM Remote came out (later named ASCOM Alpaca) which enabled remote hardware connection, the requirement for the hardware being that it must provide an interface in the form of a RESTful web service. This way the computer can connect to the telescope or other ASCOM Alpaca supporting devices through the Internet or local network. The greatest benefit of ASCOM is that it enables Tartu Observatory and other small observatories to connect almost any instrument to their control systems with ease.

2.3.2 INDI

INDI [24] open-source library, similarly to ASCOM, provides a framework for astronomy equipment automation. It is often compared to ASCOM because its main goal is to connect hardware to software, allowing data exchange and hardware control. INDI is cross-platform, supporting a number of Operating Systems (OS) but at the time of writing this thesis it does not support Windows OS. The information sent between INDI and instruments is in Extensible Markup Language (XML) file format and the connected equipment can connect either directly to the system or remotely through a network.

2.3.3 TANGO

Tango [25] is an open-source toolkit oriented at controlling any hardware and software. The project is supported and used by Square Kilometre Array (SKA) [26] and other research institutions such as SOLARIS [27]. It is hardware independent and can be used in same fashion as both ASCOM and INDI. It acts as an intermediary between the connected instruments and software in order to create a system that supervises all of the connections. The scripts for each device that connect to Tango can be written in one of the three supported languages: Python, Java or C++.

3 Instruments and their control system at Tartu Observatory

This chapter describes the instruments, software and control situation of Tartu Observatory's 1.5-meter telescope. Additionally, it outlines what is needed for the system and finishes with a solution that was implemented for this thesis.

3.1 Overview of the observation process at Tartu observatory

For a software engineer embarking on a project that utilizes a telescope, it is important to familiarise oneself with how telescopes work and how to use them. The use case of a telescope system is observing celestial objects. The telescope system consists of several instruments, each having a specific function, such as focusing the light into a spectrograph that divides white light into a spectrum that a camera can capture. Whenever a telescope is mentioned in this system, it implies the whole system of instruments unless stated otherwise.

In order to observe a star in the sky, an observer uses a custom-made software called Astrolab to get the coordinates of the star from the database. Assuming that the star is visible in the hemisphere of the observation, the observer aims the telescope at the star using the same software. Because Earth is spinning around its axis, the coordinates of sky objects change and thus the telescope or the observer has to make sure that the telescope moves together with the star. That is needed because when taking an image of a star the exposure time sometimes lasts several minutes, where in one second an object's position in the sky changes by fifteen arc seconds[1]. For comparison, the width of the spectrograph slit is 2 arc seconds. The spectrograph captures images of spectra, which is used to analyse properties of light. This conveys information about the physical features of the star such as temperature.

3.2 Initial situation

In order to observe an object in the sky, be it a star or an asteroid, one needs to have quite a few tools. These would include: a telescope which allows more light to be collected; a focuser to be able to choose a precise focus point; a dome controller that primarily controls the rotation of the dome; a camera to take images; and other equipment. In an ideal scenario, there is a computer or a group of them that have information of and controls all the equipment present. However, that is not the case for the Tartu Observatory's 1.5-meter telescope.

[1] 1 arc second = 1/3600°

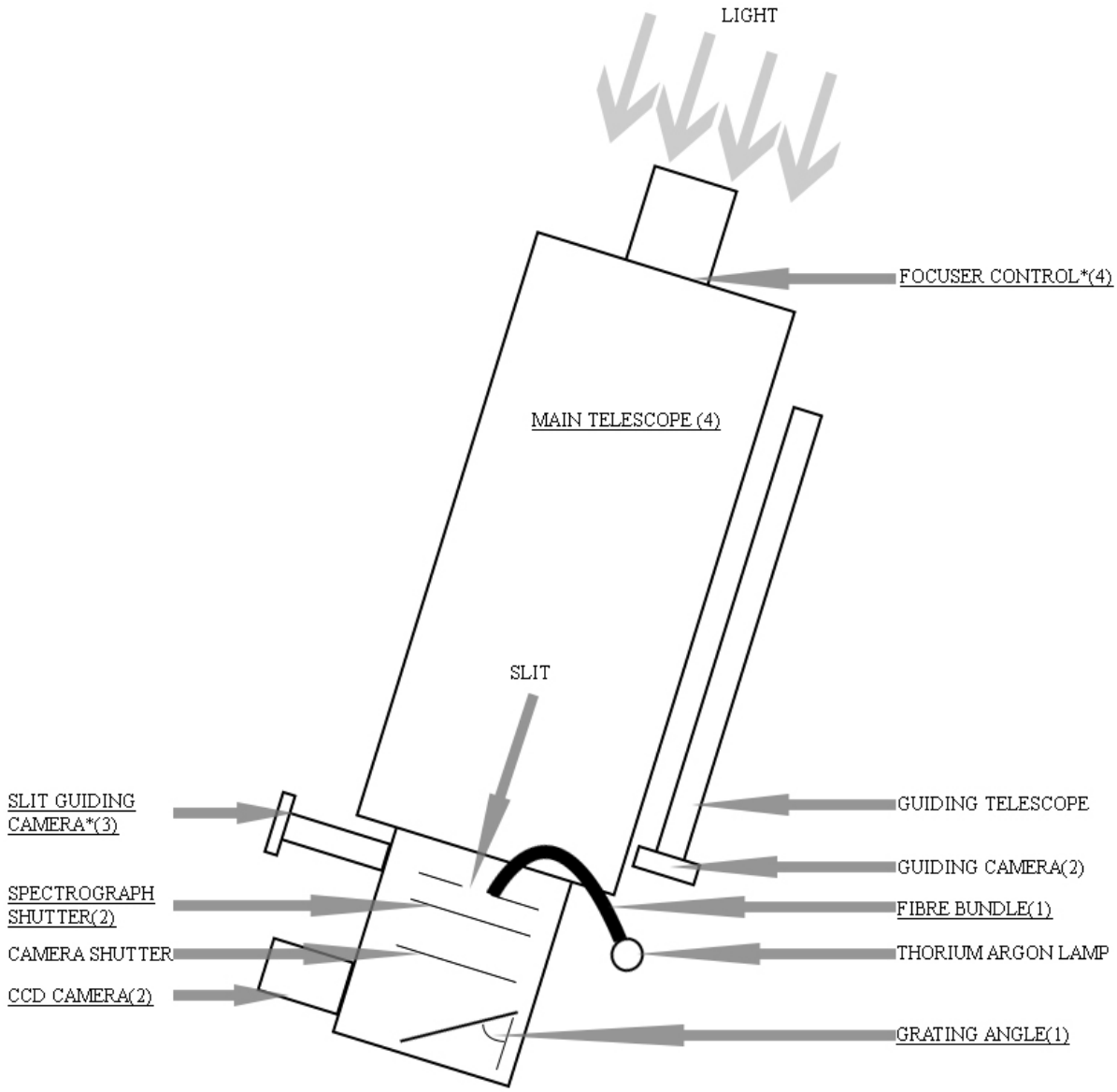

Figure 3.1: Components of the telescope and spectrograph at Tartu Observatory. Underlined components can be controlled either by software or with physical switches. The components with an asterisk (*) can be controlled by both. The numbers by the components refer to the computer responsible for the control: (1) micro-controller with a touch-pad (2) camera control computer (3) slit guiding computer (4) telescope control computer.

3.2.1 Hardware situation

There are multiple components necessary for the observational operations. Most of these components are fully self-contained and do not provide any information to other devices. Grating control, which is a controller for an optical component that diffracts the light into spectrum and directs it at the camera, is one of the more informative devices. It has a screen that displays at which angle the grating is turned. However, the display has no information on how coarse the grating is because specific grating has to be changed physically and only a person that has inserted the grating into the slit knows which specific grating is being used. Other than that there are devices like fibre bundle that are controlled manually and have no interfaces other than an observer's own eye. Examples of such devices are the two pointing cameras that allow an observer to accurately point the telescope at an object as the automatic star tracking is not precise enough. It is worth noting that all alterations applied to the telescope guiding system using the precise control are reflected in the Astrolab program. An example that is not reflected in any way is opening and closing the dome: the system has no way of knowing whether the dome shutter is closed or not. The telescope as well as the spectrograph with most of the components present in Tartu Observatory are displayed in figure 3.1

3.2.2 Software situation

In order to control the system there are 4 micro and personal computers running different Operating Systems (OS) and each computer is dedicated to control one or two components:

1. Micro-controller with a touchpad that allows control of grating angle and fibre bundle.
2. Personal computer that has a software primarily controlling the CCD camera.
3. Personal computer displaying the camera output from the slit guiding camera.
4. Another personal computer running Linux OS which has a custom-made program Astrolab to control the telescope.

As described in figure 3.1 these computers control the system together. The computers run different OS, for example, while the second and the third computer have Windows installed, the fourth one has Linux. The reason for this is that there are programs that run on only one of them. Specifically, on Linux there is a custom-written software called Astrolab. This software is made specially for Tartu Observatory's 1.5-meter telescope. The tool connects to the telescope, the focuser and the dome that are all used for observing the sky objects and displays their information. Additionally, Astrolab is used to direct the telescope at an object. The celestial coordinates and names of these objects are stored in a MySQL database on the same computer.

Astrolab is supposed to act as software that displays all information about every component in the system as well as controlling it. However, it only contains information from the telescope, the focuser and the dome. Additionally, the program can set a location in the sky at which the telescope is pointing. Other than that Astrolab has a connection to a database which contains the origin location of the observable objects as well as telescope logs. The issue that comes forth is that the database is custom-made and contains only the objects that astronomers on site enter. There is no retrieval of information from astronomical databases.

Furthermore, some of the components are controlled via physical switches and do not display any information to a user, for example, camera shutter or thorium argon lamp.

3.3 The problem

It might not seem like a huge issue that some of the devices are controlled only manually or even only physically but that strains an observer who is a human being that tires, especially working through whole nights. The main issue with this type of system comes from the use case. To be precise, the main use of a telescope is to observe the sky and in order to do that, images are taken. Each image has to have the metadata about scientific and calibration observations. Therefore, whenever an observer finishes taking an image and wants to save it, they have to type in all of the information manually. That is highly prone to error, not only because sometimes the observer is tired, but also because there is information that are of similar format to one another.

3.4 Possible solution

During this thesis a possibility for data acquisition software to use information from listed components was created. Additionally, any device has a capability to be connected to the system, in turn being connected to the device used for data acquisition with some further development. That is possible because the system is meant to be modular and expandable, allowing any device that is also added in the future to be connected.

3.4.1 Requirements for the system

A newly developed system must fulfill a set of requirements. These defined after analysis of the existing architectural design is completed. For the data management system that was being developed for the Tartu Observatory's 1.5-meter telescope, the main following requirements were created with the help of supervisors:

1. Multi-platform
2. Near real-time
3. Modular system
4. Using industry standards

It is important for the system to uphold these requirements for various reasons. Multi-platform system would run smoothly even when devices with different OS are connected to it. In case the system is not multi-platform, a vast number of devices would not be able to connect, and upgrading the system would be difficult.

The requirement for the system to be near real-time comes from the use case. Whenever an observer has finished taking an image of an object in the sky and wants to save the recorded file, it is important to receive the current information about all the configuration and statistics of the equipment. However, considering the fact that this information does not change frequently, real-time is not required as it only introduces unnecessary complexity to the system. Therefore a delay of few seconds to the system is not an issue.

Considering the amount of devices used during observations, it is important to make sure that they all connect seamlessly, independent of other connected equipment. That is important because in case something unexpected happens with one device, others should still be connected to the data management system. However, the biggest benefit of the system being modular is

that connecting new devices is simple. As long as the new equipment is set up correctly, it will work without the need to drastically change something in the system.

The fourth requirement of using industry standards increases the system's ease of use and prepares the system to be upgraded more easily later on in its life cycle. Furthermore, a tool being a standard signifies the tool's quality and support. Some of the possible industry standards include but are not limited to ActiveMQ and ASCOM.

3.4.2 The solution

In the final system created during this thesis, there are 3 main components:

- Astrolab for Python 3
- ActiveMQ
- ASCOM Alpaca drivers

Each of the components can be categorised into a client or a server. Clients are devices such as a telescope, a focuser, a dome, a weather station and any other device that sends or receives information to and from the telescope control room. In this particular case the client is Astrolab. It creates an XML file with all the required information from the connected devices and sends it to ActiveMQ every time Astrolab updates its GUI. However, in order to implement this functionality, Astrolab is required to be upgraded to Python 3 programming language because previously used Python 2 is obsolete and should not be used for developing new functionality.

ActiveMQ, being a Message Broker, acts as an asynchronous relay of data. The devices that retrieve the information from the ActiveMQ can do this whenever required. Additionally, a benefit of using ActiveMQ is that multiple devices can be both uploading and/or retrieving information. This implementation, written in Python, is made for both cases. The upload of information is accomplished by Astrolab while a separate custom Python program retrieves the information. The greatest advantage of using ActiveMQ in the system is that it enables the system to be modular.

The ActiveMQ data, described in appendix 7, is propagated through a RESTful web service which is written based on ASCOM API. The use of ASCOM allows to connect currently existing and future hardware to the system with ease. In order to connect two computers in Tartu Observatory as part of Master's project, a custom Python program was written. The program retrieves the information from ActiveMQ and processes it into a required format. Considering that ASCOM is just an interface standard, it propagates the data further by creating a web service that is based on REST architectural style.

Afterwards, a camera control software that acts as a client connects to the created RESTful web service using ASCOM Alpaca interface. Assuming that the software supports ASCOM API, it retrieves data and processes it for an observer. This way all information about the instrument status is available for the user in one place. More importantly, because the data reaches the camera control software, a saved file of an observation can be automatically filled with that data.

The thesis focuses on creating the functionality. The objective of the work is to create a functional system, security comes second. Until greater security measures as encryption and authentication are implemented, the system works in an enclosed environment.

4 Development of the automated data management system

This chapter describes what tasks were completed for this thesis in order to create an automated data management system for Tartu Observatory's 1.5-meter telescope.

4.1 Converting Astrolab from Python 2 to Python 3

An ActiveMQ module for the already existing Astrolab has to be created in order to develop the automated, modular and expandable data management system. However, an issue rises from the fact that the original Astrolab version of was developed in Python 2. This version of Python is obsolete and as of January 1st, 2020 Python 2 is not supported. It is a problem for multiple reasons. Firstly, no system should be left without support as it could be rendered unusable. Above all, from the data management system's point of view, it is beneficial if every component in the system is written in same programming language as much as possible because it helps with compatibility issues, it is also much easier for anyone in the future to create updates for the system. Because of these reasons, developing a system in an unsupported language is not preferable, therefore, Astrolab had to be upgraded from Python 2 to Python 3 in the scope of this thesis. The task of upgrading Astrolab to Python 3 can be split into several parts which are described in details in the following sections.

4.1.1 Analysis of Astrolab

First of all, an analysis of the existing software is required. The process includes understanding libraries, structure of the code, software design schemes, and other details. The initial analysis of the system was done when the employees of the observatory were observing the sky. The study of the use case made it clear which functions were used more often and which ones were important. Additionally, it became clear what information was stored in a database, what information was taken from the equipment and what was calculated in the program. For example, telescope and focuser position and temperature was taken from the equipment as this information was constantly changing, while certain information about sky objects was stored in a local database. However, the database stored certain information such as the coordinates of the objects only in a format which describe celestial coordinates that correspond to January 1st, 2000. These coordinates change slowly over time because of precession of Earth rotational axis. Based on that information the program had to calculate the current coordinates of the selected object for this data to be usable for telescope pointing. The user interface of Astrolab can be seen in figure 4.1, in it the object, telescope positioning and other information is represented.

Apart from observing someone else use Astrolab, a virtual environment was set up with all the

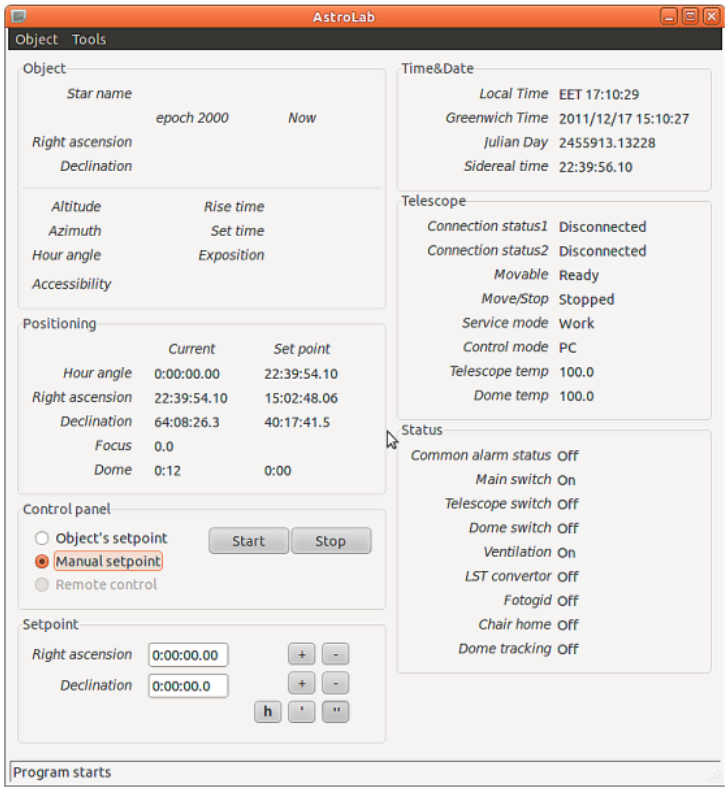

Figure 4.1: Astrolab UI. The left top Object panel presents information about a selected object that is being observed. Lower in the middle left panel is a positioning panel that describes the current position of the telescope, focuser and the dome, as well as a chosen set point that the instruments are moving towards. Underneath is the control of the telescope panels. On the right side, current time and date can be found, as well as various information about the status of the telescope and other devices.

needed components for Astrolab to work. All of the features of Astrolab worked well in a virtual environment except for those that communicated with the equipment. In order to simulate this part there was a need to simulate all equipment including programmable logic controllers and modbus TCP communication protocol [12]. After the evaluation of the duration of the simulation, it was decided that it is simpler to connect the virtual system to the telescope on site whenever required.

4.1.2 Development environment for Astrolab

One important point to note is that there was a requirement to keep Astrolab as close to the original as possible. This requirement came from the people working at Tartu Observatory and it had to be upheld. In order to develop the changes for Astrolab, a development environment had to be set up. This included having Astrolab on an Linux OS and having Python 3 with appropriate libraries that are only needed for Astrolab that is being updated.

Firstly, to create the environment, the first step that was takes was setting up a virtual machine. It enables quick reconfiguration if needed and allows the main environment to still be used as required, practically having two machines to work with. The virtual machine player of choice was VMware Workstation 15. This specific virtual machine player was chosen because it was a

familiar tool and there was no requirement set which virtualisation system to use.

Secondly, on a single OS it is possible to have several releases of Python of different versions with multiple versions of different libraries. That can affect the development of the system because there are less controlled variables and what might work on an OS that has multiple Python versions, might not run on OS that has only Python 3.8. Therefore, it is important to create an environment for Python where every Python module exists because it is required in the final system. For that, Python has a useful module called *venv*. It provides a way to establish an independent Python environment for modules and programs inside a running operating system. Additionally, with this tool it is simple to set up a finalised environment as it is easy to export a list of installed libraries that are in the *venv* development environment.

4.1.3 Python module 2to3

Upgrading programs from Python 2 to Python 3 is not a simple task. However, since the time Python 3 was first released a lot of people have already gone through this task and the Internet is full of helpful tips. Probably the greatest advice from other people who upgraded completely different systems is to look into Python 3's native program called 2to3 [13]. This program comes as a script whenever any version of Python 3 is installed. The way it works is that it reads Python 2 code and applies a number of fixers to it. These fixers are functions that look for specific syntax in the code and replace found cases with a new syntax. These syntax changes might be as simple as *import ConfigParser* being changed into *import configparser*, but the changes might be a little bit more complicated, for example, *keys = map(lambda x: x[0], desc)* being changed to *keys = [x[0] for x in desc]*.

All changes made by 2to3 script can be summarised in a few types.

- Converting all *print* calls to *print()*. For example, instead of *print(cur,task)* writing *print((cur, task))*. It is done like this because whenever 2to3 detects *from __future__ import print_function* compiler directive, 2to3 interprets *print* as a function. Therefore, even if there were already multiple sets of brackets, 2to3 would still add another set. This change can be turned off by setting appropriate flags but it is a safer option to leave the function, knowing that the result will have less bugs.
- Fixes *import* statements. For instance, *from Exceptions import ClosingException* is changed to *from .Exceptions import ClosingException*. In this case, Exception is a custom script found in the same folder as the file where these import statements were changed.
- Renaming *long* to *int*. In Python 3 some of the variables types were removed in order to streamline the development with Python.
- Specifying output of a function as a *list*, whenever functions such as *dict.keys()* are used. This is done because in Python 3 *dict.keys()* returns a *view* object instead of *list*.
- Other changes that appear only once or twice. Such as a change to a constant's name from *os.makedirs(logPath, mode=0777)* to *os.makedirs(logPath, mode=0o777)*, renaming libraries to have all lower case letters like *import ConfigParser* to *import configparser*.

Running the native Python 3's script 2to3 gave a solid start for Astrolab's upgrade to Python 3. The result that came out of 2to3 was a program that compiled itself and started up. The issue was that a user was not actually able to use the program, as clicking anything on the user interface would result into large number of errors.

4.1.4 MySQL library

The first error that surfaced after running Astrolab's code through 2to3 was that the previously used *mySQLdb* library did not exist anymore. The library had been used to connect to the database were the information about celestial objects and telescope logs were stored. Generally, the fact that a library is not installed is not an issue considering that it can simply be installed but in this case *mySQLdb* could not have been installed for Python 3. That is because the developers of this module have abandoned the project and therefore Python 3 had no releases of this library. For this reason, a new library that would communicate with a MySQL database had to be chosen.

After looking through most of the available options, 3 libraries stood out as possible options:

- *mysqlclient* - C-based wrapper requiring *mysql-connector-c* C library.
- *mysql-connector-python* - Python-based client developed by Oracle.
- *PyMySQL* - Fully Python-based MySQL client.

Each of the mentioned libraries have their benefits and drawbacks. Starting with *mysqlclient*, it is a C-based library which runs smoothly on CPython, which is an implementation of Python that compiles the code into bytecode and then interprets it. However, this library is the fastest of the 3 only in an appropriate environment based on the benchmark tests done by the community [14] [15] [16]. In comparison, *mysql-connector-python* is the slowest of the three but not much slower than *PyMySQL*. While *PyMySQL* is released under the MIT license[1] and in turn *PyMySQL* has a strong support from the community. Additionally, both *PyMySQL* and *mysql-connector-python* are fully Python-based and require no additional setup for the system.

In the end, the choice is up to the use case. Since the database is used rarely, only when loading and saving objects of the sky, the difference between approximately 1 second (using *mysqlclient* and a *CPython* implementation) and approximately 2 seconds (using either *mysql-connector-python* or *PyMySQL*) is negligible and would require too much development time. Therefore, taking into account the earlier mentioned fact that *PyMySQL* is a preferable library based on its license, the chosen MySQL driver is PyMySQL. Afterwards, Astrolab's source code was changed appropriately to use the chosen library instead of the old unsupported *mySQLdb*. Implementing these changes took some time as the code was unfamiliar but the bugs were solved and a MySQL library update was successful.

4.1.5 Trying to edit or select an object

Another issue that created a lot of problems and made Astrolab unusable was happening when a user tried to select or edit a sky object from the database. The error was *TypeError: can't concat str to bytes* and it would appear in a line *args='name' : (name + '%')* while args is described as *name = name.encode('utf-8')* where name comes as an input when calling this function and is simply *name = ""*. A pseudo code describes the situation more clearly:

```
name = ''
name = name.encode('utf-8')
args = {'name' : (name + '\%')}
```

[1] MIT license: https://opensource.org/licenses/MIT

As mentioned above, this error caused great problems, as seemingly *name* is supposed to be a *string* but based on the error it clearly was not *string*. The first straightforward solution was to delete the line where *name* is encoded into *utf-8*, however, that did not help. After some time spent analysing the documentation and community forums, it was found to be some issue with encoding in Python 3. It was further discovered that Python 3 has several different *string* types. Based on how a variable is created, it can be saved as either a *string* or a *byte* variable [17]. Theoretically, *name = ""* should create a *string*, as it is treated as a *str()* constructor with only one input. Based on the documentation, using *name.encode('utf-8')* results in a variable that is in *bytes* which would explain the initial error but it does not explain why the error persisted after deleting the line. In the end, after a considerable amount of debugging this error, the working solution was to convert the percent sign from *string* to *bytes* like so: *args='name' : (name + b'%')*. This way both sides of addition are in *bytes* format and works as intended.

There were also issues like certain inputs not being supported in certain library functions. As an example, using *wxPython* for creating GUI, a function *ListCtrl.SetStringItem()* that used to have input named *col* does not have it anymore. Therefore, it had to be changed appropriately. In order to accomplish that, the function had to be dissected into which inputs represented what and then had to be translated into a new syntax of the upgraded function. Luckily, the only change in this specific case was that *col* was renamed to *column*. Additionally, in the newer *wxPython* versions there are no *SetStringItem()*, *SetIntItem()*, nor any similar setter for a specific variable type. Instead there is only *SetItem()* which does not depend on a variable type that is being set. These functions had to be changed appropriately all throughout the Astrolab's code.

Another issue related to Astrolab's and database's communication came up whenever a user tried to select an object: *hMain = wx.FlexGridSizer(1,2) arguments did not match any overloaded call*. Evidently, the object constructors, which are a type of subroutine in programming that are used to create an object, got changed in the new version of wxPython, which led to analysing all the available constructors of this function in order to find one that seemingly accomplishes the same goal as before upgrading Astrolab to Python 3. The available constructors for *wx.FlexGridSize()* are as follows [18]:

- wx.FlexGridSizer (int cols, int vgap, int hgap)
- wx.FlexGridSizer (int cols, gap=wxSize(0, 0))
- wx.FlexGridSizer (int rows, int cols, int vgap, int hgap)
- wx.FlexGridSizer (int rows, int cols, gap)

This is a GUI function that lays out components in a two-dimensional table. Therefore, it makes no difference which constructor should be used as long as the outcome is similar to how it used to look before upgrading to Python 3. Assuming that the original variables were specifying the count of rows and columns, the simplest constructor to use is the third one *wx.FlexGridSizer (int rows, int cols, int vgap, int hgap)*. Here, *vgap* and *hgap* correspond to the height and width of the *wx.Size* object and define the size of the padding between the rows and columns in pixels. Based on this information it is safe to assume that the default values for *vgap* and *hgap* can be equal to 1, therefore the chosen solution is to use *hMain = wx.FlexGridSizer(1,2,1,1)*.

4.1.6 Name 'open' is not defined

One of the major changes from Python 2 to Python 3 was found with a bundle of repeating errors. Astrolab uses *logging* library to log what functions are called. A part of the logged

information is all of the devices disconnecting from Astrolab when it is turned off. The way this logging is implemented is that an inbuilt *logging* library's function that logs information is called from inside a *__del__(self)* function from each of the classes that control appropriate devices. The errors at hand were:

```
File "DeviceManager.py", line 128, in __del__
File "logger.py", line 28, in closeLog
File "/usr/lib/python3.7/logging/__init__.py", line 1116,
 in _open

NameError: name 'open' is not defined
```

Based on these errors it was apparent that there was something different with Python itself rather than Astrolab's source code and clearly a more drastic change needed to be implemented. The issue at hand was that an in-built Python function does not work. The first thought why it might have been happening was that the environment may have been set up incorrectly, but after reinstalling the environment the errors persisted. After a long research it was found that *__del__()* function is run at a different time compared to before. Since Python 3.4, the *__del__()* is called whenever interpreter shuts down and because of that the module's global variables are set to *None* before the module itself is released. Because of this drastic change in the interpreter, the previously used way of calling *logger* library's *closeLog()* function from *__del__()* is not viable.

Any solution to this problem required a change of where and when the logging takes place. In order to achieve that there is a need to pay attention to a few factors. First is the fact that the *logging* functions still need to be called at the same time as before, therefore, a deep analysis needs to be carried out on how the same result can be achieved. Second- the initial requirement of upgrading Astrolab and keeping everything as close to the original source code as possible. Based on these points it was required to find a solution that achieves the same result with as little of new code lines as possible.

Eventually, it was found that Python 3 has a module called *atexit* which registers and unregisters cleanup functions. Functions registered this way are then automatically executed when the interpreter normally terminates, which is the same time as when the __del__() ran. Using this module it is possible to make following changes to the code:

```
instead of:
    def __del__(self):
        closeLog(self.logger)

it is possible to do:
    def cleanup(self):
        ...
        closeLog(self.logger)
    def __init__(self):
        ...
        atexit.register(self.cleanup)
```

With this change in place, the errors disappeared and the log file was again logging information appropriately as intended. While implementing these changes, lines of code that tried to clean up previously unused variables were found. These lines were erased as well.

4.1.7 Testing on site

By this point everything seemed to be working correctly and since the updates were developed on a virtual machine away from the actual telescope equipment, it had to be brought to the control room and tested on site with all of the devices connected. To make the testing easier, the physical computer that was hosting the virtual machine was connected to the network that had access to the telescope hardware controller, this way making sure that on the virtual machine Astrolab was able to communicate well with all of the devices. This way there is no need to set up the environment on physical computers until it is clear that everything works.

The results of this testing were satisfactory as it worked as intended. All of the changes so far were successful and there were no new errors.

4.2 ActiveMQ module for Astrolab

After Astrolab was updated to Python 3, it is possible to start working on uploading information to ActiveMQ. In order to do that, a module for Astrolab needs to be written, one that would be permanently connected and continuously push information onto a queue. One of the first ideas on how to achieve that was to push log files onto ActiveMQ. However, this idea was rejected after noticing that logs are not storing the required information and the data that is logged is not frequent enough. Additionally, storing information from logs is not recommended as it is better to take the information from the original source and not from a third party source. For that reason a custom module that would transfer information directly was developed.

4.2.1 Communication with ActiveMQ

Each script that would upload information to ActiveMQ would take the information from the device and send it to ActiveMQ. However, the problem with writing custom scripts is that for each piece of equipment a new script requiring a lot of work and knowledge about the device is required. This can be partially avoided by implementing modules of code into the pre-existing code of each device, reducing the required programming knowledge. In turn this makes the system more modular because it is easier to add a new device to it. Overall, custom code still needs to be written for each device but now the final model of the system includes more than just ActiveMQ.

When it comes to communication, Apache ActiveMQ supports a multitude of programming languages such as C, C++, JAVA, Python, .Net and many others. For the purpose of this thesis Python was chosen for various reasons. Mainly, one of the already existing tools on site is written in Python. By adding a new module to this tool, unnecessary complexity can be avoided, since everything is in the same programming language. Additionally, Python is highly supported on many devices and it is lightweight, which is beneficial for the system considering it has to be multi-platform and modular. It is also important to note that it has a strong community support and a huge variety of libraries.

After choosing the programming language, it is important to figure out a specific library and a protocol through which the communication is enabled. There are 2 options:

- STOMP
- OpenWire

Both of these work on Python. However, what sets them apart is that OpenWire is Apache's specific wire transport protocol, while Stomp.py is a Python library that does not depend on a specific Message Broker. If it happens that ActiveMQ fails the requirements and a different Messaging Queue is chosen, all scripts can be left as they are if they are written in Stomp.py.

Additionally, it is also important to figure out what configuration to use for data transport. Apart from STOMP, there are plenty of other options such as Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), peer-to-peer (P2P), etc [11]. However, the chosen Stomp.py is more than satisfactory even though by default it sends everything in plain text. The advantages of it far outweigh this. To begin with, it is possible to set up Stomp.py to send information over Secure Sockets Layer (SSL). Moreover, the data management system is not open to the Internet and is only reachable through local network.

4.2.2 Connection

ActiveMQ's connection, as described in section2.2, can be one of two types. It can either retrieve the information from ActiveMQ or upload information. In this case Astrolab only needs to upload all of the information about the telescope, the focuser and the dome. Therefore, a module has to format a message appropriately and upload it onto ActiveMQ's queue.

ActiveMQ module is written in Python, similarly to Astrolab. STOMP.PY is used for the purpose of connecting to ActiveMQ. For connection, the usual information is required, such as IP and PORT, as well as authentication information so that only authorised devices are able to connect to ActiveMQ. Another important information when uploading to ActiveMQ is to specify a queue name to which the information should be uploaded.

4.2.3 Data format

Whether the information is uploaded in plain text or in cipher, it still has to follow certain formats. While ActiveMQ does not require a specific messaging configuration, the data management system as a whole needs a unified messaging format. Extensible Markup Language(XML) was chosen for this purpose. The main benefit of XML is that each field can have custom tags that describe the information. Additionally, by using XML it is possible to create an element tree structure from the information in the messages. This way it is simple to arrange the data.

A great alternative to XML is JavaScript Object Notation(JSON). In most cases it is vastly superior over XML as it has a smaller overhead and is more straightforward. However, this greatest benefit of JSON is also partly why XML was chosen over JSON in this case. XML allows specific data formatting, for example, to use naming and tagging to format the data more specifically. For example, specifying a variable's measurement unit in a *tag* header while *name* header specifies what the variable is.

In order to create an XML in Python, *ElementTree* module can be used. By using this module it is possible to create a container object that stores hierarchical data structures. The type of the object is a mixture between a dictionary and list type, which cannot be sent over STOMP.py to ActiveMQ. Considering that *ElementTree* module is a part of a larger *xml* module, it has appropriate functions to turn the object into a string. The *ElemenetTree* object needs to be created from the root element and branched out as required. For example, by using the following code, it is possible to create an element structure demonstrated in fig. 4.2.3

```
data = ET.Element('Data')
domeItems = ET.SubElement(data, 'Dome')
focItems = ET.SubElement(data, 'Focuser')

domeCurrPos = ET.SubElement(domeItems, 'CurrentPosition')
domeTargPos = ET.SubElement(domeItems, 'TargetPosition')

focPos = ET.SubElement(focItems, 'Position')
focTemp = ET.SubElement(focItems, 'Temperature')
```

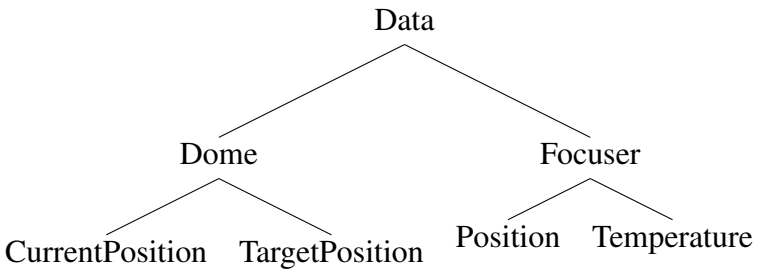

Figure. 3.4.5: Element Tree object created in the example

4.2.4 Populating the element tree with data

When the element tree structure is complete, it is populated with the data. The majority of the required information could be obtained from the GUI of Astrolab. By reverse engineering, it was possible to find how the code retrieves the data from the devices using modbus protocol [12]. Thus, the ActiveMQ module takes the values from the devices and assigns them to appropriate elements in the Element Tree every time the module function is called.

Additionally, some of the information that is required to be accessible in the data management system is not available directly in Astrolab. Therefore, the values had to be calculated based on the information available to the ActiveMQ module. As an example, the telescope position was required to be both in topocentric and horizonal coordinate systems, however, Astrolab only provided in the topocentric coordinate system.

Last but not least, a few more details about the module need to be noted. Astrolab is sending data to AMQ nearly continuously This means that every few seconds a new XML file is sent to a Messaging Queue. The interval of how often this is done was chosen to match the GUI's refresh speed. Uploading information to ActiveMQ more frequently introduces an unnecessary data overload and doing it less frequently than Astrolab can introduce inaccuracies. Therefore it is favourable for the two to be synchronised. To enable a reliable work of Astrolab, a switch to turn the ActiveMQ module on and off was implemented. The switch would automatically turn off the ActiveMQ module when a connection could not be established, but now a user of Astrolab has the capability to manually turn the module on and off at will.

4.3 RESTful service based on ASCOM ALPACA API

This section talks about an important part of the data management system that is ASCOM [19]. The tool is a standard interface for connecting different astronomy equipment together. It was

used in the data management system as an API to connect the devices together and enable the system to be expandable. The section focuses on ASCOM's functionality and implementation in the created system.

ASCOM, specifically ASCOM Alpaca, was decided to be used in the created system, fulfilling all of the requirements that were raised at the start of the system's creation. Firstly, the system can be multi-platform as there is no requirement for a device other than it being able to run a RESTful service. Secondly, the system can also be modular, as there are no dependencies between devices. The data management system and ASCOM do not need to have the devices connected simultaneously, and can work with any combination of instruments connected to the main computer.

4.3.1 Benefits of ASCOM

ASCOM introduces some significant advantages to the data management system, all of which fulfill more than one requirement of the system. By using ASCOM it is possible to connect practically any device. The device needs to follow a provided API [20] which describes the functions used to connect the specific instrument. Because of this it is technically possible to connect a device running on any platform and written in any programming language. Especially, ASCOM Alpaca, which is a part of ASCOM standard and allows connection of devices using REST. Having the ability to connect a device without any additional wiring is a huge plus for a system as it allows a wider range of instruments to be connected. For example, devices that do not have an Ethernet interface can still be connected. Additionally, one of the greatest advantages of using ASCOM is that it provides an option for an observer to choose a software for observing objects rather than having to use multiple programs that come with the hardware.

ASCOM functions by sending information from an instrument to a telescope control software, as well as receiving commands from the software. This software, generally, would be implemented on a device which would allow it to be connected directly, however, based on the way that ASCOM Alpaca is implemented it is technically not required to have the REST web service directly on the instrument. In this specific case of creating a system for Tartu Observatory, where Astrolab already exists and has all of the required information from three devices, ASCOM completes the chain by providing a way for the information to reach the user.

4.3.2 Developed RESTful service

In order to connect the user with the available information on ActiveMQ it is required to create a web service following ASCOM API which would constantly be retrieving the data from ActiveMQ. Each time a user requests information from the ASCOM RESTful server, the server automatically asks ActiveMQ for the newest set of data. Therefore, the created RESTful service has to consist of two modules: The first module being responsible for retrieving and processing the XML files from the Message Broker; and the second one handling the requests from the user.

The first part of the web service is receiving data from ActiveMQ. By specifying the IP, the port and the specific name of the queue in the script. ActiveMQ sends all of the messages from that queue to the script which requested the information. All that is left for this script to have is a listener that would sort errors from appropriate messages and process the received XML files. For processing the received data, the *ElementTree* module that Astrolab uses to create

these messages was used. As the exact format of the message is known (more information in the 4.2.3 section ”Data Format”) it can be split into separate arrays for easier handling of the information. These arrays are stored globally, therefore, the other script that is handling the service part can access them.

The second part of the web service is creating the web service itself. Like the rest of the data management system, it is written in Python and because the ASCOM Alpaca requires so, it is in REST architectural style. This means that the service itself:

- Is stateless, which means that the data is not stored in the service but comes from ActiveMQ.
- Uses HTTP, allowing standard GET, PUT methods to be used for accessing and publishing the information [21].
- Is accessed by using Unified Resource Identifiers (URI).

By following these REST properties as well as ASCOM Alpaca API [20], the following approach to propagating the information was recognised:

- For each variable there must be an associated URI which would return that variable.
- URI's must abide by the following format:

  `http://IP:port/api/v1/device_name/device_number/method_name`

  As an example, the first focuser's position would be:

  `http://localhost:5900/api/v1/focuser/0/position`

  where ”localhost” is the IP and 5900 is the port of the web service.
- The result must be returned in JSON format and the structure must be:
  - Value - the value of the requested method.
  - Client Transaction ID - supplied by the client and keeps track of transactions.
  - Server Transaction ID - unique for each client transaction, helping to sort log entries.
  - Error Number - follows the standard for error numbering and returns 0 when transaction is successful and other appropriate error in case such is encountered.
  - Error Message - the error message in case Error Number is not 0.
- The returned status code can only be one of 3 values:
  - 200 - when the request is handled successfully.
  - 400 - if the request path cannot be interpreted.
  - 500 - unexpected error.

  The figure 4.2 illustrates a flow diagram of handling the status code. The diagram is provided by the developers of ASCOM [22].

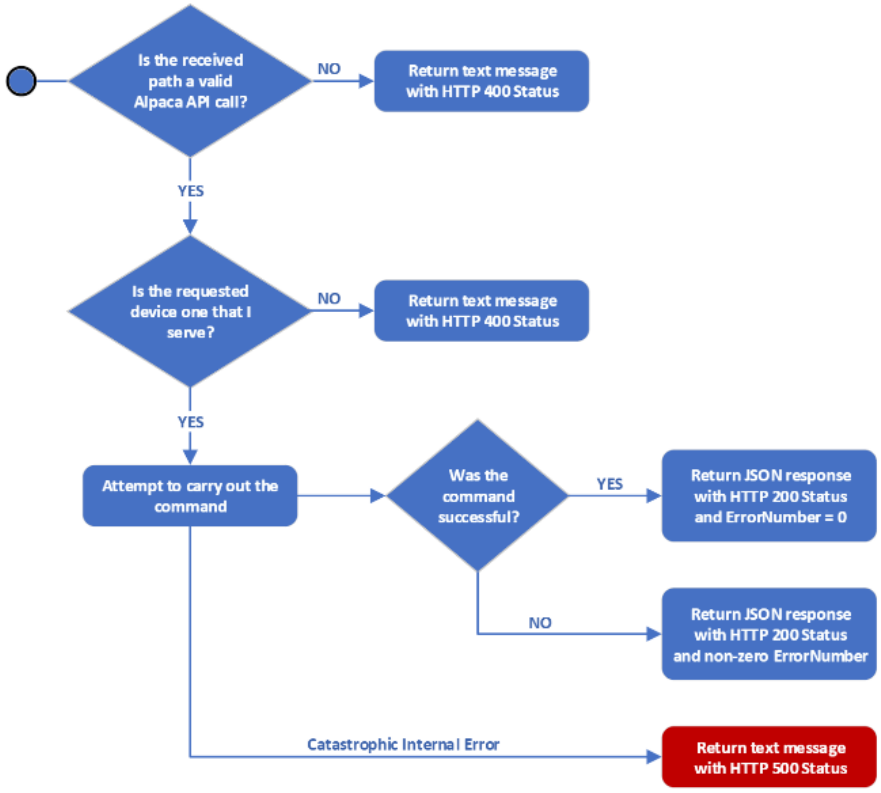

Figure 4.2: The flow diagram [22] presenting the proper handling of response status.

It is worth noting that the data management system focuses on receiving the data and, therefore, only required GET methods were implemented in this RESTful service based on ASCOM API. However, the API has a mandatory PUT function for every device. The function is for connecting and disconnecting the device. The method is mostly responsible for setting the software values to ensure that the hardware is connected. The data management system's implementation does not allow to connect to the hardware directly, thus, an empty method is used to walk around the issue of the function being mandatory as it does not have much of an impact for the system.

ASCOM RESTful service currently propagates three devices (the telescope, the focuser and the dome). Every device has common functions, such as device name and driver version, as well as device-specific functions. For example, a focuser has a method that tells the temperature of the focuser, while a telescope has a function returning the current Right Ascension value.

4.3.3 Camera control software

Allowing a flow of information through a RESTful server is only one half of what ASCOM is. The other half of ASCOM is considered the front end which tells the server which commands to execute. The implementation of this is up to the software developers creating applications for sky object observation. Such software most often involves control of camera and the connected devices. As ASCOM has become much more popular nowadays, most of the available software already comes with appropriate ASCOM implementation. Thus, the software only needs to fulfill the requirements that the observers have.

In order to handle ASCOM implementation, the computer running the camera control software has to have ASCOM drivers installed. These can be downloaded for free from the ASCOM website [19] and are straightforward to install. Additionally, since ASCOM Alpaca was used in the data management system created for this thesis, its drivers had to be also installed on the same computer. Other than that, the specific camera control software is irrelevant as long as the developers have made it available to connect devices via ASCOM. Choosing a specific software to control the devices is out of the scope of this thesis and is completely dependent on the requirements of the observers at the Tartu Observatory. On top of that, the specific program does not impact the automation of the data management system.

5 Results

During the thesis three main tasks were raised and all were accomplished:

- Astrolab was upgraded to Python 3 programming language using Python 3.8 as a current stable version.
- ActiveMQ 5 service, along with a module for Astrolab were implemented in order to upload the data that is described in appendix 7. The priority for the thesis was functionality and not security.
- ASCOM API-based RESTful interface was created, it retrieves information from the ActiveMQ service and enables an image acquisition program to access it.

A modular and expandable automated data management system was created successfully. Its Unified Model Language (UML) deployment diagram is shown in figure 5.1. Additional devices can be added to this system with ease.

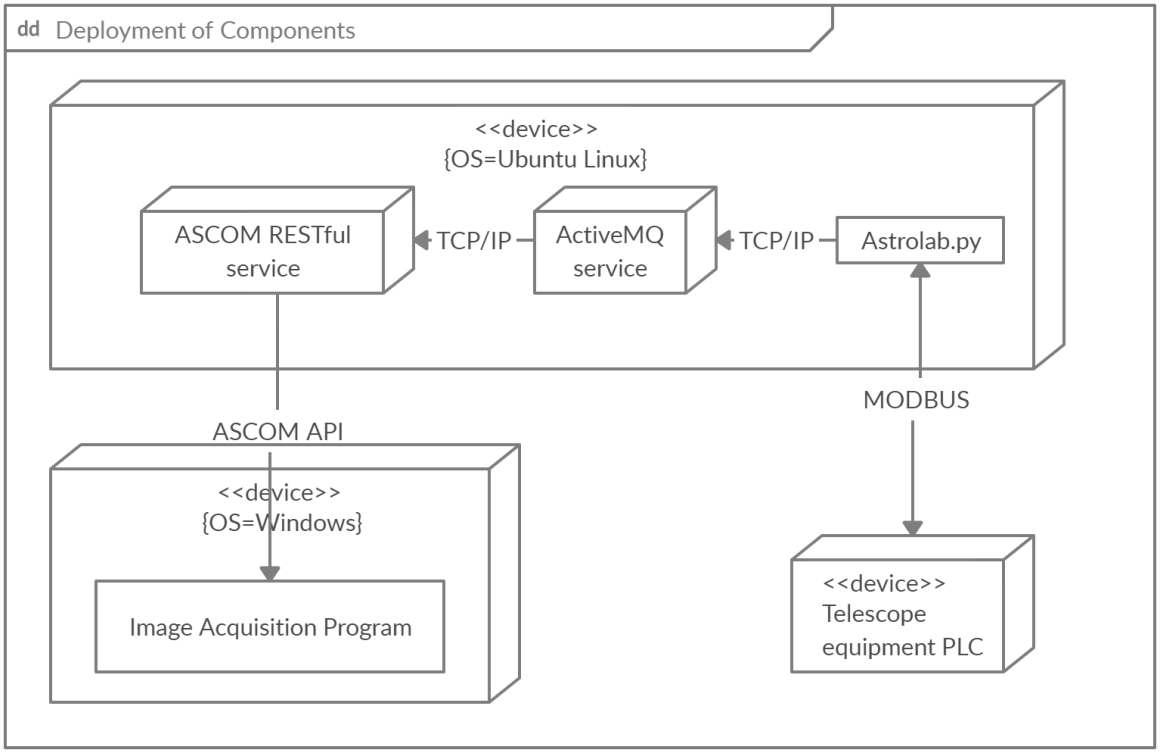

Figure 5.1: UML deployment diagram of the final system. The arrows represent data flow direction

6 Analysis and discussion

The main objective of this thesis was to establish a beginning for a data management system for Tartu Observatory's 1.5-meter telescope. The new system used ActiveMQ in conjunction with ASCOM in order to help automate data transfer from the telescope, focuser, and dome to data acquisition software. The new system had to fulfill several requirements. Firstly, it had to be near real-time, allowing a delay of at most few seconds. Almost as importantly, it had to be modular so that instruments could be added and changed easily without affecting the system itself. On top of that, a multi-platform system was required to allow more flexibility for the connected devices, and also the possibility to connect more instruments in the future. Last but not least, the system had to use technologies that are industry standards to guarantee the stability of the system for a long period of time.

The early design of the automated data management system relied heavily on ActiveMQ alone. The processing of information and communication with ActiveMQ was planned to be handled by custom Python programs. Later, ASCOM was found to achieve the needed goal of connecting the devices to the camera control software. It was decided to keep ActiveMQ nonetheless, as it provides an ability to connect multiple devices at once while making sure that in case connection to one of the instruments is lost, the camera control program does not crash in the middle of an observation. It could be argued that keeping ActiveMQ in the system is redundant because ASCOM by design takes care of the connections between the instruments and the camera control software. That is a well founded reasoning as the custom ASCOM script might be written directly in Astrolab. However, in this case ActiveMQ was chosen due to the mentioned improved software stability, along with a preference of the observers in Tartu Observatory to keep ActiveMQ in the system.

Considering the fact that there are several computers in Tartu Observatory's 1.5-meter telescope's control room, technologies used in the system were chosen to be deployed on these computers. Astrolab needs to be deployed on an Linux computer, and an image acquisition software has to run on a Windows machine because the ASCOM drivers, required for the image acquisition software, are only available on Windows OS. ActiveMQ service, as well as the RESTful web service written based on ASCOM API, can technically be installed anywhere on the network. For the data management system both of them were chosen to be installed on the machine with Linux OS. However, it does not create much difference whether it is on Linux or Windows OS- what is of importance in this case is how accustomed the users are to managing services on each platform. It is important to ensure that after the thesis is completed there is someone capable of performing maintenance on the system. Additionally, both of them could be running on a completely separate dedicated machine such as Raspberry PI. This way the services would be independent from the main machines that are used for observation, reducing the risk of the services going down. However, by introducing another device to the data management system, the complexity increases a lot, and the stability of the whole system is more

difficult to achieve as there are more devices that might fail.

It was decided that the information about the three instruments - the telescope, the focuser and the dome - would be sent in a single XML format file as a structured element tree. This would minimise the amount of different connections between Astrolab and ActiveMQ because by combining everything into a single file, the network is not as packed. However, it can be argued that it might be more useful to separate each device into a separate message, allowing a greater flexibility for managing the data. On top of that, the RESTful service, that sends data further, accepts only separate connections for each device as that is the requirement in ASCOM API. That is a possible improvement for the future. Additionally, the RESTful web service should be split into separate services based on the instruments, this way enabling greater stability by reducing the chance of software crashes.

The created system is modular and expandable, which allows Tartu Observatory to connect other devices that are used for observations, such as grating angle controller, slit guiding camera control computer and other components. Furthermore, instruments - for instance a weather station - that previously were not used for observations can be added as well. Additionally, since the data management system uses ActiveMQ, some of the data can potentially be used by other departments in Tartu Observatory. For example, the connected weather station information can be useful for remote sensing as well as astronomical observations.

The thesis focused on creating functionality of the system and the only measure of security that is implemented is that the system runs in an enclosed environment- a local network. Therefore, it would be a sensible improvement to apply additional security measures. Such would include but would not be limited to: encapsulating data in a Secure Sockets Layer (SSL) tunnel; proper authentication of the devices that are connected to the system.

7 Conclusion

Tartu Observatory's 1.5-meter telescope required a lot of manual work to observe celestial objects: from controlling each instrument separately to inputting every detail to a saved file about the instruments post-observation. The work of this thesis sought to automate some data management processes. By using industry standard technologies like Messaging Queues and ASCOM standard to bridge a gap between different instruments, it was possible to achieve a modular system. The modular system ensured devices functioned properly regardless of their number and allowed new devices to be easily connected.

During the course of the Master's thesis the system that follows these guidelines was created. To achieve that, the old custom-made software Astrolab, that was displaying information about the few instruments in Tartu Observatory's 1.5-meter telescope, had to be updated from Python 2 to Python 3 and also had to have new features - such as connection to ActiveMQ, a chosen Messaging Queue that helps delivering messages from different devices - implemented. A custom ASCOM-based RESTful web service was also developed for enabling image acquisition programs to be connected. By having this in place, the need to manually enter information about the devices was removed.

There were several options when choosing each technology, which were compared and the most suitable ones were selected to be used in the end. On top of that, there were issues with setting up the system and upgrading the old tools. Problems like modules that became unavailable were solved by choosing similar modules, and the errors that were stopping the system from working were fixed.

The finished result can be improved by adding other important devices to the system of Tartu Observatory's 1.5-meter telescope. Additionally, the data management system could be improved even further— directly connecting the instruments to the image acquisition system instead of having several tools relaying information through a chain. To conclude, the aims that were raised at the start of the thesis were reached successfully.

Acknowledgements

First and foremost, I am grateful to my supervisors Tõnis Eenmäe and Heleri Ramler for overseeing my work through the Master's thesis. Their guidance allowed me to find solutions that I had overlooked. On top of their help I am grateful to Heleri Ramler for translating the Abstract to Estonian language. Additionally, I would like to thank Tartu Observatory for allowing me to work on one of their telescopes for the thesis. It was a great opportunity to learn more about astronomy. Lastly, I am thankful to Ketevani Eliosidze for proof reading the thesis.

Bibliography

[1] Apache ActiveMQ Message Broker hrefhttps://activemq.apache.org/https://activemq.apache.org/ 19.05.2020, 19:12 (UTC).

[2] ZeroMQ Message Broker https://zeromq.org/ 19.05.2020, 19:18 (UTC).

[3] RabbitMQ Message Broker https://www.rabbitmq.com/ 19.05.2020, 19:16 (UTC).

[4] Microsoft Azure https://azure.microsoft.com/en-us/ 19.05.2020, 19:29 (UTC).

[5] Amazon Simple Queue Service https://aws.amazon.com/sqs/ 19.05.2020, 19:34 (UTC).

[6] Juan C. Guzman, Ben Humphreys, "The Australian SKA Pathfinder (ASKAP) Software Architecture", *Software and Cyberinfrastructure for Astronomy*, **7740**, 2010, DOI:10.1117/12.856962

[7] D. Sosnowska, M. Lodi, X. Gao, N. Buchschacher, A. Vick, J. Guerra, M. Gonzalez, D. Kelly, C. Lovis, F. Pepe, E. Molinari, A. C. Cameron, D. Latham, S. Udry, "HARPS-N: software path from the observation block to the image", *Software and Cyberinfrastructure for Astronomy II*, **8451**, 2012, DOI:https://doi.org/10.1117/12.926208

[8] SIMBAD Astronomical database http://simbad.u-strasbg.fr/simbad/ 07.05.2020, 14:13 (UTC).

[9] Daniel Vagg, Derek O'Callaghan, Fionn Ó hÓgáin, Sheila McBreen, Lorraine Hanlon, David Lynn, and William O'Mullane, "GAVIP: A Platform for Gaia Data Analysis", *Software and Cyberinfrastructure for Astronomy*, **9913**, 2016, DOI:10.1117/12.2233619

[10] Apache Kafka distributed streaming platform https://kafka.apache.org/ 19.05.2020, 19:20 (UTC).

[11] ActiveMQ transport configuration options https://activemq.apache.org/activemq-connection-uris 19.05.2020, 19:14 (UTC).

[12] The modbus organization homepage http://www.modbus.org/ 19.05.2020, 18:45 (UTC).

[13] Documentation of Python 3.x native program 2to3 https://https://docs.python.org/2/library/2to3.html 19.04.2020, 17:52 (UTC).

[14] Methane's MySQL driver Benchmarking using Python 3.4 https://gist.github.com/methane/90ec97dda7fa9c7c4ef1 22.04.2020, 14:37 (UTC).

[15] PyMySQL's own evaluation and comparison between different MySQL drivers https://wiki.openstack.org/wiki/PyMySQL_evaluation 22.04.2020, 14:38 (UTC).

[16] Karoly Nagy's benchmarks of *mysql-connector* and *MySQLdb* drivers http://charlesnagy.info/it/python/python-mysqldb-vs-mysql-connector-query-performance 22.04.2020, 14:42 (UTC).

[17] Python 3 documentation on Standard Types https://docs.python.org/3/library/stdtypes.html 24.04.2020, 02:05 (UTC).

[18] wxPython API documentation https://wxpython.org/Phoenix/docs/html/ 25.04.2020, 16:30 (UTC).

[19] ASCOM 20.05.2020, 10:52 (UTC). https://ascom-standards.org/

[20] ASCOM API https://ascom-standards.org/api/ 07.05.2020, 17:28 (UTC).

[21] Ed. R. Fielding, Ed. J. Reschke, RFC 7235, Hypertext Transfer Protocol (HTTP/1.1) DOI:https://doi.org/10.17487/RFC7235

[22] Peter Simpson, Bob Denny, Daniel Van Noord, *ASCOM Alpaca API Reference*, 2020.

[23] Kirill Trusov, *Software Astrolab 1.0 User Manual*, Tallinn, 2011.

[24] INDI Open Astronomy Instrumentation https://indilib.org/ 19.05.2020, 16:27 (UTC).

[25] TANGO https://www.tango-controls.org/ 19.05.2020, 16:28 (UTC).

[26] Square Kilometre Array organisation public website https://www.skatelescope.org/ 19.05.2020, 16:31 (UTC).

[27] SOLARIS National Synchotron Radiation Centre https://synchrotron.uj.edu.pl/en$_GB/start$ 19.05.2020, 16:33 (UTC).

Appendix A Data sent through the system

The appendix presents the information that is automatically sent through the system in an Element Tree structure, ordered based on the devices.

- Data
  - Observable Object
    - Name
    - EP2000 Right Ascension
    - EP2000 Declination
    - Current Right Ascension
    - Current Declination
    - Altitude
    - Azimuth
    - Hour Angle
  - Focuser
    - Temperature
    - Position
  - Dome
    - Temperature
    - Current Position
    - Target Position
    - Slaved
  - Telescope
    - Current Hour Angle
    - Current Right Ascension
    - Current Declination
    - Altitude
    - Azimuth
    - Target Hour Angle
    - Target Right Ascension
    - Target Declination
    - Tracking
    - Slewing
    - Sidereal Time
    - UTC Date

Appendix B Source Code

The source code is available in the attached archive files and on gitlab.ut.ee:

Astrolab for Python 3: https://gitlab.ut.ee/laimonas.suchockas/astrolab-for-python-3.git

ASCOM ALPACA-based RESTful server: https://gitlab.ut.ee/laimonas.suchockas/ascom_restful_server.git

Non-exclusive licence to reproduce thesis and make thesis public

I, Laimonas Suchockas

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   "Automating data management system of 1.5-meter telescope at Tartu Observatory"

   supervised by MSc Tõnis Eenmäe and PhD Heleri Ramler

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Laimonas Suchockas
24.05.2020