UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science curriculum

Karel Liiv

Contact Automation For The ESTCube-2 Mission Control System

Bachelor's Thesis (9 ECTS)

Supervisors:

Umesh Anilchandra Bhat, B.Tech

Tõnis Eenmäe, MSc

Tartu 2017

Contact Automation For The ESTCube-2 Mission Control System

Abstract:

ESTCube-2 is the first student satellite project of the Estonian Student Satellite Foundation and it's objective is the technological demonstration of scientific payloads for upcoming missions. After a satellite's launch, the communication can be done through ground stations, but the contact time is very short over a limited area at a given point in time over the earth. The aim of this thesis is to design and implement a contact automation module which can schedule and execute automatic contacts with the satellite without minimal/no human intervention. The author aims to implement the first functional version of the required software module. This thesis explains the thought process, design and implementation of the software module, including the integration with the existing ESTCube Mission Control System—a combination of software and hardware that is used for monitoring and controlling the satellite after launch.

Keywords:

Space technology, mission control system, ESTCube-2, contact automation

CERCS:

P170 Computer science, numerical analysis, systems, control

Kontakti automatiseerimine ESTCube-2 missioonijuhtimissüsteemile

Lühikokkuvõte:

ESTCube-2 on Eesti Tudengisatelliidi Sihtasutuse satelliidi programm, mille eesmärgiks on tehnoloogiline testimine järgmiste missioonide tarbeks. Peale satelliidi orbiidile saatmist toimub suhtlemine satelliidiga maapealsete sidejaamade abil, aga kontakti aeg satelliidiga ühest maapealsest sidejaamast on väga limiteeritud . Käesoleva töö eesmärk on disainida ja implementeerida kontakti automatiseerimise tarkvaramoodul, mis võimaldab minimaalse inimese sekkumisega planeerida ja läbi viia kontakte satelliitidega. Esmane funktsioneeriv tarkvaramoodul on loodud autori poolt. Töös selgitatakse tarkvara disaini ja

implementatsiooni ning ühildamist olemasoleva missioonijuhtimissüstemiga, mis on kombinatsioon tark- ja riistvarast, mille abil jälgitakse ja juhitakse orbiidil olevat satelliiti.

Võtmesõnad:

Kosmosetehnoloogia, missioonijuhtimissüsteem, ESTCube-2, kontakti automatiseerimine

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of contents

Introduction **5**
Background **7**
ESTCube-2 7
ESTCube-2 Mission Control System (MCS) 7
Contact automation 8
Two-Line Element set format (TLE) 8
TLE Fetcher 9
Satellite-Location Predictor 10
Problem statement 10
Requirements **11**
Application program interface 11
Design **12**
Notion of a single command 13
Life cycle of a single command 14
Implementation **16**
The application 16
Applications program interface endpoints 16
Inserting and updating commands 17
Querying stored commands 18
Deletion of stored commands 19
Scheduled background jobs 20
Updating overpass times 20
Queueing commands before an overpass 22
Cleaning up after an overpass 23
Database 24
Ground Station Client simulator 25
Deployment to the ESTCube MCS server 27
Future work **29**
Conclusion **30**
References **31**
Appendices **33**
I. Source code 33
II. Sample ESTCube-1 overpass times 34
III. Sample TLE Fetcher output 35
IV. Licence 36

1. Introduction

ESTCube-2 is the first student satellite project of the Estonian Student Satellite Foundation and it's objective is the technological demonstration of scientific payloads for upcoming missions [1]. ESTCube-3, which will be partially based on ESTCube-2, is planned to be sent to the Moon orbit in the future [2]. A satellite mission comprises of several subsystems; some onboard and the rest on the ground (Earth), one of them being the Mission Control System (MCS) and ground station support.

The MCS is a combination of software and hardware that assists operators to track, command and communicate with spacecraft after their launch. The MCS can be used for uplink; sending commands to the spacecraft in orbit, and also for downlink; receiving data from the spacecraft.

The aim of this bachelor thesis is to create a software module which allows scheduling and perform contacts with the ESTCube satellite with minimal/no human interference. The current implemented solution requires too much manual intervention and therefore, the probability of missing contact windows due to human error is huge and the contact times are not used as optimally as they could have been.

There is also a probability for interruptions to occur during contact, which may result in loss of telemetry packets. Therefore, not all packets reach the target (*spacecraft in orbit*). New connections must be scheduled in order to catch the missing packets. For example, during the ESTCube-1 mission, only 85% of packets transmitted were received, even though multiple ground stations were tracking the satellite overpasses [3].

The ESTCube MCS contact automation software module is a network application that interconnects the authorised ground stations and tracks satellites and schedules communication packets in an optimal fashion so as to minimise delay during satellite overpasses. Another factor to consider would be automatic scheduling and compensation of lost data packets during transmission from the ground stations. Such an implementation would greatly reduce the burden of ground station personnel and simplify the work of satellite operators and thereby increasing the amount of information that could be exchanged with the satellite during the short time period available during the overpasses.

The deliverables of this bachelor thesis comprise of a Python[1] application, which after completed would be integrated into the ESTCube Mission Control System (MCS) hosted on Docker[2] platform as a service (PaaS) technology. The application communicates with multiple ground stations using RabbitMQ[3] messaging queue and stores relevant information in a PostgreSQL[4] database.

This thesis will only focus on storing and scheduling of commands for upcoming overpasses. The following functionalities are out of the scope of this thesis:

- On-ground testing of the scheduled commands
- Scheduling satellite downlink
- Specifying the execution order of commands
- Transferring and executing of commands

The source code for the application developed is attached in Appendix 1 as a ZIP file.

[1] https://www.python.org/
[2] https://www.docker.com/
[3] https://www.rabbitmq.com/
[4] https://www.postgresql.org/

2. Background

2.1 ESTCube-2

Estonia's third space program is going to be a 3-Unit CubeSat in-orbit technology demonstration platform [4]. A 3-Unit CubeSat is a satellite with standardised dimensions and weight. It’s structure is made of up three identical cubes connected in series where the volume of a single cube is approximately one liter with the dimensions of 10×10×10 cm [5]. The primary payloads for this space mission are to test deorbiting using plasma brake, electric solar wind sail and a highly integrated subsystem solution [1].

The satellite would consist of various integrated electro-mechanical and electronic subsystems, most of them built in-house with the one of the important goals being to minimise mass and volume, so as to enable a similar satellite platform for upcoming space missions and experiments [1].

2.2 ESTCube-2 Mission Control System (MCS)

The ESTCube Mission Control System (MCS) is a combination of software and hardware designed for sending (uplink) and receiving (downlink) data to the ESTCube satellites in orbit. While most mission control systems for space missions require the physical presence of the satellite operators in the “mission control” room, the ESTCube MCS is a lightweight web-based command and control solution for satellites in orbit, enabling the operators to be anywhere on the planet and have the same operational capabilities.

In addition to monitoring and controlling the satellites, MCS also provides tools for automating and scheduling various operations of the MCS so that the operator does not always have to be present when a satellite is passing over a ground station. The automation minimises errors and increases the amount of data received from a satellite.

During the development phase, MCS provides a hardware test platform for all the subsystem teams. The hardware test platform comprises of the actual engineering model and simulates all the operations of the real satellite so that the components can be tested properly before launch. After launch, the hardware test platform exists as a testbed to check and determine the behaviour of the command before they are sent to the actual satellite in orbit.

2.3 Contact automation

Contact automation or satellite range scheduling is the process of scheduling the communication between a satellite and its ground station over blocks of time so that a satellite's support request gets always satisfied on a ground station overpass [6]. A ground station overpass is a period of time during which the satellite is in the communication range of its ground station. For low Earth orbit (LEO) satellites like ESTCube-2, this can occur up to ten times a day and the duration of a single overpass is approximately from 10 to 15 minutes [3, 4]. Appendix 2 contains a small sample of a satellite's (ESTCube-1) overpass times.

In order to schedule contacts with the satellite, a series of jobs have to be scheduled. To prioritise jobs in the set, each job is characterised by the following parameters:

- The processing time of job
- Set of ground stations available for processing the job
- Due date of the job
- Revenue of the job completion
- Size of data packets [7]

As ground stations and satellite have both unit capacity, meaning that they can only process a single job at a time, the precedence constraints between jobs and ground station setup times have to be taken into account for scheduling a job as well [7].

As the connection between a satellite and ground stations is very noisy and the signal's strength can be very low, it is very common that some data packets go missing during the connection. In order to retrieve the missing data packets, new jobs have to be scheduled for requesting them.

2.3.1 Two-Line Element set format (TLE)

The Two-Line Element set format is a data format developed by the North American Aerospace Defence Command[5] and is used to transmit a coded set of orbital elements that perfectly describe a satellite's orbit [8].

[5] http://www.norad.mil/

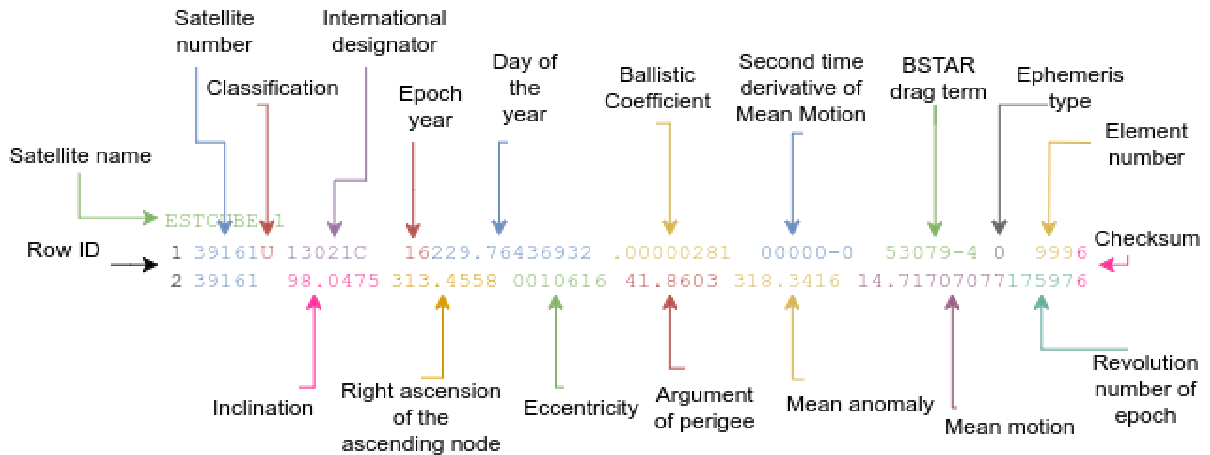

Figure 1: Definition of the two-line element format [9]

Two-line element format represents information about the orbit of a satellite, an example for ESTCube-1 with element explanations is shown in figure 1. The TLE contains the orbital data of the satellite, information on when these were obtained and meta information on the satellite, including the launch date and angle between the equator and earth plane. But as the satellite's orbit changes over time, TLE is up-to-date for only a limited amount of time.

Satellite TLE's are commonly used along with NORAD's SGP4/SDP4 orbital model to calculate the current position and velocity of the satellite [10]. To make calculations more accurate NORAD regularly calculates and publishes new TLEs for most of the public satellites in orbit.

2.3.2 TLE Fetcher

TLE Fetcher is a detached software module developed by the ESTCube-2 Mission Control System team. The aim of this software module is to fetch and serve the latest TLE for ESTCube satellites. In order to retrieve the freshest TLE for a satellite, the software module makes a request to Space-Track[6]. Space-Track is a system developed by the Science Applications International Corporation[7] for tracking all artificial earth satellites and space probes.

An example output of the TLE Fetcher application is provided in appendix 3.

[6] https://www.space-track.org/
[7] http://www.saic.com/

2.3.3 Satellite-Location Predictor

The Satellite-Location Predictor is another detached software module developed by the ESTCube-2 Mission Control System team. The software uses satellite's orbital parameters and ground station location information for estimating satellite overpass times. For calculating the overpass times, the Satellite-Location predictor uses the Orekit[8] library. Orekit is a Java library which provides methods for building flight dynamics applications [11]. The library also contains methods for computing propagated coordinates with a SGP4 model [12].

For calculating upcoming overpass times a satellite's TLE and information about ground station's coordinates are required. In order to get the latest satellite TLE regular requests get made to the TLE Fetcher software module. As the satellite's TLE gets regular updates the overpass calculations can be done regularly with new outcomes. Therefore, in order to save calculation time the Satellite-Location Predictor's output is limited to calculate only configured amount of upcoming overpasses. Appendix 2 contains a small sample of contact times predicted by the Satellite-Location Predictor software module. As the calculated overpass times are only used for scheduling upcoming contacts then the accuracy is not a concern. Actual and accurate contact start and end times are calculated individually for each ground station prior to the scheduled contact.

2.4 Problem statement

The main problem is that the current solution requires human interaction and as the contact window for a single satellite is very limited and often happens at late hours the contact windows with the satellite may not be used as optimally as it could. Another problem is that the interruptions during the connection between a ground station and a satellite are very common. When an interruption happens the missing packets have to be scheduled for being retransmitted.

[8] https://www.orekit.org/

3. Requirements

Requirements for the Contact Automation Module for ESTCube-2 Mission Control System would be the following:

1. Built using the best practices of microservices architecture
2. Fetch the TLE information for the satellite
3. Predict the satellite passes
4. Schedule the commands according to priority
5. Update the priority queue in case of a manual override
6. Check for ground station availability
7. Schedule the commands according to ground station availability

3.1 Application program interface

Requirements for the application program interface would be the following:

1. The content type for all requests must be *application/json*
2. Correct HTTP request methods must be used
3. The content type of all responses must be *application/json*
4. Correct HTTP response statuses must be used
5. The application must be secured using authentication

4. Design

The contact automation with a satellite can be separated into two different parts—scheduling commands and executing commands. The scheduling part includes storing and prioritising commands for upcoming overpasses and is a part of the mission control system. The execution part is responsible for transferring the commands to the satellite and executing them. This thesis concentrates only on the scheduling part and is not responsible for the execution part. The execution part will be implemented as a part of the ground station software and will depend on the scheduling part. The dependencies between the different software modules are described using figure 2.

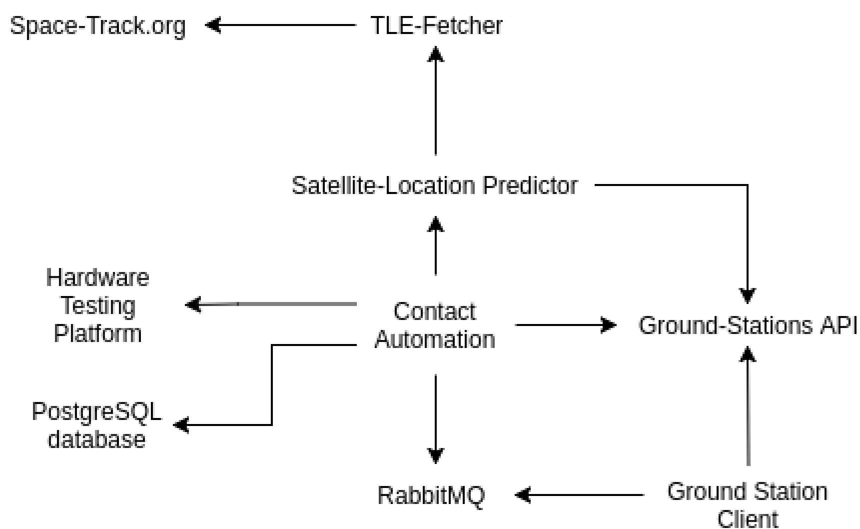

Figure 2: Dependency between different Mission Control System components

An arrow in figure 2 indicates that a component depends on another component (the dependent component being indicated by the direction of the arrow). For example, the Contact Automation module is dependent on the Satellite-Location Predictor module.

4.1 Notion of a single command

Each command in the system contains the following information:

- Unique identifier
- Target satellite
- Command body
- Command arguments
- Type
- Priority (low, normal, high)
- Status

The unique identifier is required in order to avoid a single command getting executed multiple times. Additionally, it adds the option of backtracking a command's status starting from the scheduling process up to its execution.

As the mission control system is not specific to a single satellite an identifier of the target satellite is required.

The command body and command argument are for now a textual representation of the command to be executed. In the future they may be replaced with actual packets that are transferred to the satellite.

The priority value will ensure that the commands with a higher priority will be transmitted before the commands with a lower priority. That way it is possible to later add commands for maintenance or other critical commands in a way that they still get executed first. The priority can not be used for controlling the execution order of commands; such a feature would need to be handled by a different mechanism which shall be added in the future.

The type field indicates whether the command is meant to be executed on the ground station or on the satellite. The type of a command can either be uplink, in which case the data is sent from the ground station to the satellite, or downlink, meaning the data is sent from the satellite to the ground station.

The status field helps the system keep track of a single command. It holds information on the current state of the command. The status of a command may either be untested, scheduled for

the upcoming overpass, transmitted to the ground station, finished, or failed. Under normal circumstances a command has to pass through all of the aforementioned states, excluding failed, before it can be counted as successfully executed.

4.2 Life cycle of a single command

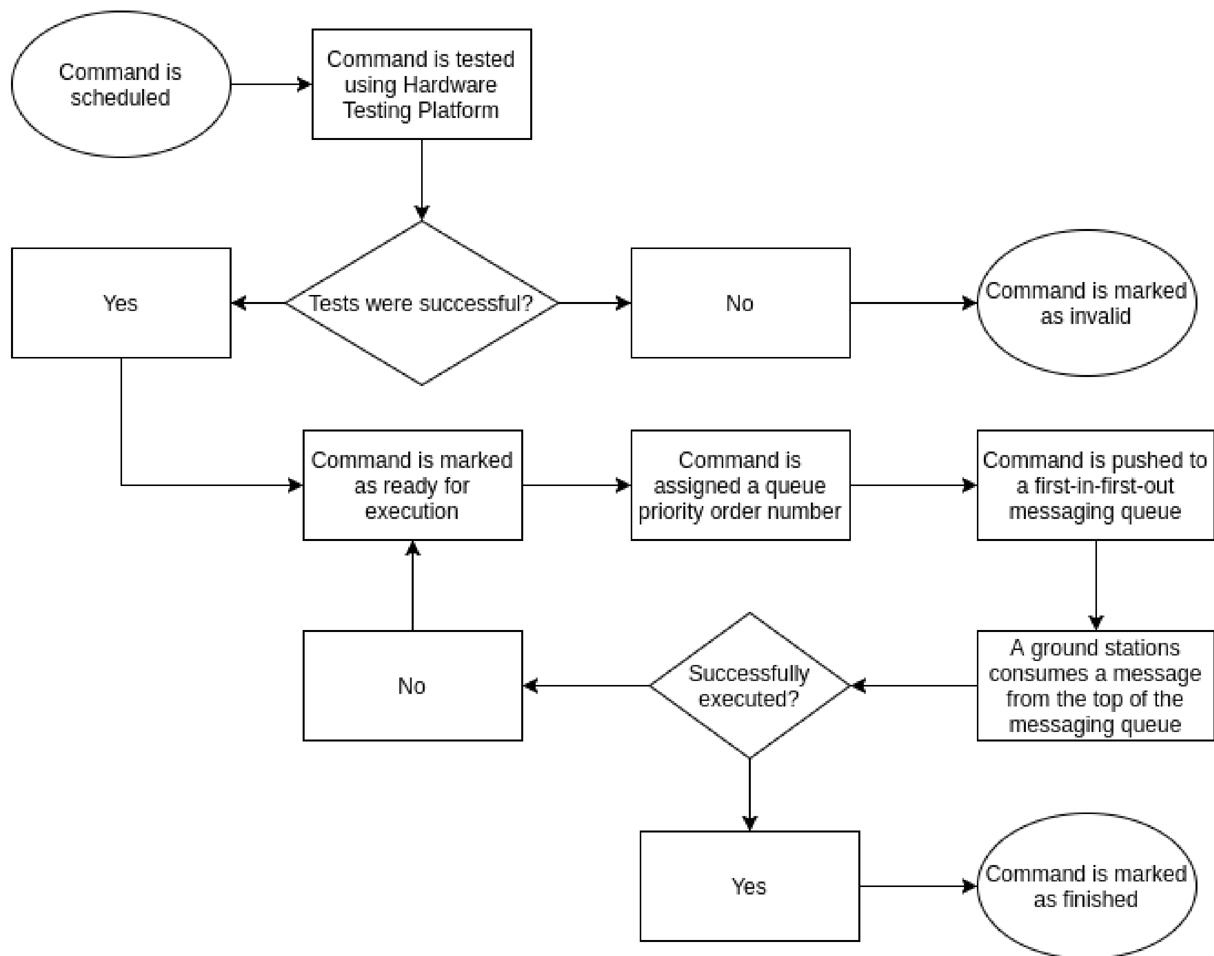

Figure 3: The flow of a single command from scheduling to execution

The life cycle of a single command is described in figure 3. The life cycle of a command starts with it getting stored for scheduling. The next step in a command's life cycle is testing. Each command is tested on ground using a satellite simulator before being transmitted to the satellite, thus avoiding the transmission of corrupted commands. As the testing platform is still work-in-progress, the testing of command is skipped for now. It shall be added in the future.

After testing of a command is finished, the command gets marked as ready for execution. Prior to the next satellite pass, the command is assigned a queue priority order number based

on the creation time of the command and the value of it's priority field. By taking this priority order number into account, the command is then pushed into a first-in-first-out messaging queue. During a pass, the ground station consumes messages from the messaging queue and tries to execute them one-by-one. If the execution of the command succeeds, it gets removed from the messaging queue, otherwise it remains in the queue to be re-executed later.

A command is marked as successfully executed if it no longer remains in the messaging queue after the pass. Otherwise, the command's status is set back to ready for execution and it will be retransmitted before the next pass.

5. Implementation

The MCS uses modern microservices architecture, meaning it consists of many independent applications which all communicate with each other over the network [13]. In MCS the microservices architecture is achieved using Docker technology. Each microservice is packaged into a Docker container and assigned a unique port number in the MCS context. This way each microservice is independently maintainable without breaking the whole system.

As the deliverable of this thesis, a contact automation module was designed and implemented. The module itself consists of multiple separate Docker containers - an application container, a database container and a messaging queue container. In addition, each container can initialize connections to other containers on the ESTCube Mission Control System server, for example to the Satellite-Location predictor or to the TLE Fetcher, in order to avoid multiple implementations of the same service.

5.1 The application

Contact Automation is an independent application that mainly takes care of two different things. It provides endpoints for storing and managing commands for upcoming overpasses and queues the stored commands for an upcoming overpass (depending on whether the command is marked as uplink or downlink).

The application itself is written using Flask[9], a lightweight web framework for Python. The framework provides methods for handling different HTTP requests and responding to them. SQLAlchemy[10], a SQL toolkit for Python applications which also provides an object-relational mapper, is used to simplify interfacing with the database. The RabbitMQ first-in-first-out messaging queue is used for queueing commands.

5.1.1 Applications program interface endpoints

The application provides several API endpoints for managing scheduled commands. The application endpoints can be divided into two groups—query and modification. For query endpoints the HTTP request method must be GET, for modification endpoints the request

[9] http://flask.pocoo.org/
[10] https://www.sqlalchemy.org/

method can be either POST, PUT, or DELETE, depending on the specific endpoint. All modification endpoints expect the request's body to be in a valid JSON format. The response type of all endpoints is JSON. The response body may contain either a confirmation of a successful operation or an error message indicating a failure. Appropriate HTTP response status codes are also used. All endpoints are also granted with unit-tests.

In order to protect the application from unauthorised users, a basic authentication is applied to all routes. The basic authentication is achieved using HTTP headers—the application expects each request to have an authorization header. If the header is not present, a login prompt is shown to the user. The credentials can be set using the configuration file in the project's root directory.

5.1.1.1 Inserting and updating commands

The endpoints for inserting a new command and for updating a command data are pretty similar. They both expect the request body to contain all the information about the command to be scheduled in JSON format and as a response, they both return the JSON representation of the stored command. In figure 4 the new command creation endpoint handler function is shown. This function expects the HTTP request method to be POST. or the command update endpoint the HTTP request method would have to be PUT.

```
@routes.route('/commands', methods=['POST'])
def post_command():
    """API endpoint for storing a new command"""

    data = request.get_json()
    command = Command()

    try:
        command.update(data)
        command.status = Status.UNTESTED

        db.session.add(command)
        db.session.commit()

        data = command.serialize()
    except (sqlexc.IntegrityError, sqlexc.DataError, ValueError):
        db.session.rollback()
        abort(200, 'Invalid data provided.')

    return jsonify({'data': data})
```

Figure 4: Command creation endpoint

First, the function parses the data from the request's body and then maps it to a *Command* model entity. The *Command* model entity is a Python representation of a single row in the *command* table in the database. For the command update endpoint the command to be updated would first be queried from the database and then the received data would be merged with the command queried from the database.

Before the command is stored in the database its status is set to untested. This is important because all commands have to be tested on a satellite simulator before they can be scheduled for execution.

Finally, the function tries to store the command's data into the database. If no error occurred during the database insertion or row update, a JSON representation of the *Command* entity is returned. Otherwise, an HTTP response with status 200 and a relevant error message is returned.

5.1.1.2 Querying stored commands

It is possible to query the information about stored a command via two endpoints. One endpoint returns a list of stored command and another one returns information about a single specific command. In figure 5 the listing endpoint function is shown. As it is a query endpoint the HTTP request method is expected to be GET.

As there are no limitations to the number of commands that can be stored in the system, the number of commands returned can be very large. Therefore it is possible to filter scheduled commands either by satellite, status, or subsystem. There are no restrictions to applying multiple filters simultaneously. The filters can be passed to the endpoint using the endpoint URL's query component.

```
@routes.route('/commands', methods=['GET'])
def commands():
    """API endpoint for listing stored commands"""

    satellite = request.args.get('satellite')
    status = request.args.get('status')
    subsystem = request.args.get('subsystem')

    query = Command.query.filter(Command.delete_time== None)

    if satellite:
        query = query.filter(Command.satellite == satellite)
    if status:
        query = query.filter(Command.status == status)
    if subsystem:
        query = query.filter(Command.subsystem == subsystem)

    data = []
    for row in query:
        data.append(row.serialize())

    return jsonify({
        'count': len(data),
        'commands': data,
    })
```

Figure 5: Commands listing endpoint

5.1.1.3 Deletion of stored commands

For cases where an unwanted command gets created, there is a data modification endpoint that allows the deletion of a single command. As shown in figure 6 the HTTP request method of this endpoint has to be DELETE.

```
@routes.route('/command/<int:command_id>',methods=['DELETE'])
def delete_command(command_id):
    """API endpoint for deleting a specified stored command"""

    command = get_object_or_404(Command, command_id)
    command.delete()

    db.session.commit()
    return jsonify({'data': 'Command successfully deleted.'})
```

Figure 6: Command deleting endpoint

First, the command to be deleted is queried from the database. If no command with such id is found, an HTTP response with status 404 is returned. After the command has been queried it

gets marked as deleted and the modified command is saved to the database. To indicate that the deletion of the command was successful a human readable success message is returned in the response body.

5.1.2 Scheduled background jobs

Another core functionality of the application is achieved through scheduled background jobs. For scheduled background jobs the APScheduler[11] library is used. Two different types of scheduled jobs are used—timed and regular background jobs. The execution of timed background jobs is started at the specified time and regular background jobs are executed regularly over blocks of time.

5.1.2.1 Updating overpass times

Fetching and storing the satellite overpass times is done by a regular background job. The background job is scheduled on application startup and executed regularly (the execution interval can be configured using the configuration file in the project root directory).

The background job first fetches information about the available ground stations from the Ground-Stations API. As the actual Ground-Stations application is still a work-in-progress an Apiary[12] project was created and used instead. Apiary is a tool for designing application program interfaces with an option to simulate the operation of the designed API.

[11] https://pypi.python.org/pypi/APScheduler
[12] http://apiary.io/

```
def update_overpass_times():
    """Inserts the satellite overpass times to the database

    Updates the overpass times in the database for each ground station registered
    in the Ground-Stations API
    """
    util.print_info("Trying to update overpass times")

    ground_station_library.GroundStation.api_url= app.config['GROUND_STATION_API']
    ground_stations = ground_station_library.get_ground_stations()

    for ground_station in ground_stations:
        util.print_info("Trying to update overpass times for {}".
                        format(ground_station.name))

        predictor = LocationPredictor(ground_station)
        predictor.fetch()
        predictor.filter()
        predictor.store_and_schedule()

    db.session.commit()

    util.print_info("Updating overpass times finished")
    return True
```

Figure 7: Updating the satellite overpass times

Then, as shown in figure 7, for each ground station an API call is made towards the Satellite-Location Predictor. As a response, the Satellite-Location Predictor returns a list of upcoming passes in a valid JSON format. First, the JSON is parsed and read by the Contact Automation application. For reading and parsing the response a utility method is called. The described utility method is shown in figure 8.

```
def read_remote_json(url):
    response = urlopen(url)
    data = response.read().decode('utf-8')
    json_data = {}

    try:
        json_data = json.loads(data)
    except ValueError:
        print_info('An error occurred while reading remote JSON data: {}'.format(data))

    return json_data
```

Figure 8: Reading JSON response from a URL

As the Satellite-Location Predictor returns multiple upcoming overpasses, an iteration over the results is done. Each result is checked against the database to determine whether it already exists in the database. As the fetched overpass times and their accuracy can change over time, two overpasses whose start times differ less than the configured minimum amount of seconds are considered to be the same. All newly found overpasses are stored in the database for future reference. In addition, two background jobs are scheduled for each newly found overpass—the first one is scheduled a configured amount of seconds prior to the overpass and another one after it. These background jobs handle the transferring of scheduled commands to the ground station for execution.

5.1.2.2 Queueing commands before an overpass

Prioritising and queuing commands prior to a satellite overpass is done using a timed background job. When the background job is executed, it first fetches the commands scheduled for this overpass from the database in a prioritised order.

```
def fetch_commands(self):
    """Fetches commands from the database in prioritized order"""
    query = Command.query.filter(Command.delete_time== None).\
        filter(Command.satellite == self.satellite).\
        filter(Command.status == Status.SCHEDULED).\
        order_by(sqlalchemy.desc(Command.priority)).\
        order_by(sqlalchemy.asc(Command.create_time))

    self.commands = query.all()
    print_info('Fetched {} commands from the database'.format(len(self.commands)))
```

Figure 9: Querying commands in prioritised order from the database

In figure 9 the function for querying commands in a prioritised order is shown. First the commands in the database are filtered by the satellite and their status. After the suitable commands have been filtered out from the database, an ordering operation is applied. The ordering—also shown in figure 9—ensures that commands with a higher priority and earlier creation time are preferred over other commands.

```python
def queue(self):
    connection = messaging_queue.get_connection()
    channel = connection.channel()

    # Ensuring that a queue for this satellite exists..
    channel.queue_declare(queue=self.satellite)

    for command in self.commands:
        command.status = Status.TRANSMITTED

        body = json.dumps(command.serialize())
        channel.basic_publish(exchange='',
                              routing_key=command.satellite,
                              body=body)

        print_info('Transmitted command with ID {}'.format(command.id))

    db.session.commit()
    connection.close()
```

Figure 10: Queuing commands prior to an overpass

After prioritising the commands a connection to a RabbitMQ messaging queue channel, which is unique for each satellite, is opened. This connection is opened using the Python pika [13] library. This library provides a pure Python implementation of the AMQP 0-9-1 protocol [15]. This is the connection protocol that is used for connection to the RabbitMQ messaging queue. After a connection to the messaging queue channel has been opened an iteration is done over the previously prioritised command, as shown in figure 10. Each command's status is set to transmitted and their JSON representation is published to the messaging queue. After the iteration over the commands is finished, changes on the command statuses are stored in the database and the messaging queue connection is closed.

5.1.2.3 Cleaning up after an overpass

Cleaning of the messaging queue, after a satellite overpass, is done in a timed background job. The cleaning process is important in order to avoid overloading the queue with low priority commands.

First, a connection to the relevant RabbitMQ messaging queue channel is opened. After the connection has been opened the program starts to consume messages from the queue until

[13] https://pypi.python.org/pypi/pika

there are none left. During the consumption of messages, each command's status is set from transmitted back to scheduled. It means that these commands will be transmitted again prior to the next overpass.

Once the messaging queue is empty, the connection to the messaging queue channel is closed. Finally, all commands that had their status set to transmitted prior to this overpass, but were no longer in the messaging queue, are marked as finished.

5.2 Database

For storing commands and information about overpasses a PostgreSQL database is used, which is a powerful open-source object-relational database system [10]. The database's design is pretty simple and is shown in figure 11. For *command* and *satellite_pass* tables soft deletion is used—table rows are only marked as deleted on the application level instead of being actually deleted. Soft deletes help avoid any possible data loss at a pretty small cost. The table *apsheduler_jobs* is used by the APScheduler library for storing the scheduled background jobs.

command	
id	integer
create_time	timestamp
update_time	timestamp
delete_time	timestamp
satellite	varchar
subsystem	varchar
command	varchar
parameters	varchar
priority	smallint
status	status
type	command_type
created_by	varchar

satellite_pass	
id	integer
create_time	timestamp
update_time	timestamp
delete_time	timestamp
satellite	varchar
ground_station	varchar
start_time	timestamp
end_time	timestamp

apscheduler_jobs	
id	varchar(191)
next_run_time	precision
job_state	bytea

Figure 11: Database structure

The *command* table stores information about all the commands that have ever been scheduled for satellite contacts. A scheduled command itself consists of a textual command name and

the relevant parameters. Each command also has a priority field, which is an integer value where a higher value indicates a higher priority. In order to avoid multiple executions of a single command, each row has a status field. The status field has the type *status* which itself is an enum type. The status can be either *untested*, *scheduled*, *cancelled*, *failed*, or *finished*. All commands with the status *failed* should be investigated and rescheduled by the command scheduler application, whenever possible. All other statuses are self-descriptive. In addition, each stored command has some other informative fields like the name of the operator who created the command, and the type of the command. The type of a command can be either *uplink* or *downlink*.

The *satellite_pass* table stores any relevant information about upcoming and past satellite overpasses. As for now we only care about overpass times, the rows in this table only contain the information about overpass start and end times. In the future, it is possible to store more parameters about a satellite pass.

5.3 Ground Station Client simulator

As for now the Ground Station Application is still work-in-progress, a ground station client simulator application was implemented in order to validate and test the scheduling process of this application. The aim of this application is to simulate the operation of a real ground station during a satellite overpass. The application is written in Python and is controlled through a command-line interface.

As shown in figure 12, the simulator application expects three command line arguments—the name of the satellite, overpass start time, and the end time of the overpass to be simulated. The application starts the simulation process on the specified start time and ends the simulation process when the specified end time has passed.

```
$ python3 client.py "ESTCUBE 1" 18:28:23 18:32:32
Starting Contact Automation client simulation..
Trying to consume a message
Received: b'{"priority": 3, "subsystem": null, "id": 1, "type": "downlink",
"status": "transmitted", "satellite": "ESTCUBE 1", "parameters": "", "created_by":
"Karel Liiv", "command": "test"}'
Simulating an overpass command execution started
Simulating an overpass command execution finished
Failed to consume a message.
Trying to consume a message
Received: b'{"priority": 3, "subsystem": null, "id": 1, "type": "downlink",
"status": "transmitted", "satellite": "ESTCUBE 1", "parameters": "", "created_by":
"Karel Liiv", "command": "test"}'
Simulating an overpass command execution started
Simulating an overpass command execution finished
Successfully consumed a message.
No messages found.. Waiting some seconds..
Trying to consume a message
No messages found.. Waiting some seconds..
Trying to consume a message
No messages found.. Waiting some seconds..
Trying to consume a message
No messages found.. Waiting some seconds..

…

Simulating an overpass has ended.
```

Figure 12: Work of the Ground-Station simulator client

When the overpass simulation starts the simulator client starts consuming messages from the messaging queue for the specified satellite. For each message consumed from the messaging queue the simulator pauses for 5 seconds. This pause should simulate the process of executing a single command.

```
class RandomExecutor(Executor):
    def execute(self):
        """Simulates the execution of the command

        Returns:
            bool
        """
        print('Simulating an overpass command execution started')
        sleep(5)
        print('Simulating an overpass command execution finished')
        return random.randint(0, 1) == 1
```

Figure 13: Simulating a single command execution

As shown in figure 13, a failure which may happen during the execution of a command is simulated using a random number generating function. If the simulation of a command execution succeeds, an ACK response is sent to the channel indicating that the message in the queue was consumed successfully. If the simulation fails, a NACK response is sent to the messaging channel to indicate that the message was not successfully consumed. Messages that receive a NACK response are added back to the messaging queue by the RabbitMQ queue service.

5.4 Deployment to the ESTCube MCS server

Along with the developed application, a *Dockerfile* containing all the required commands for assembling a Docker image is provided [11]. A Docker image consists of all the required files and software for running the application contained within it.

```
FROM python:3.4
MAINTAINER Karel Liiv "karel.liiv@estcube.eu"

COPY ./requirements.txt /usr/src/app/
RUN pip3 install --no-cache-dir -r /usr/src/app/requirements.txt

COPY . /app
WORKDIR /app

ENTRYPOINT ["python3"]
CMD ["main.py"]
```

Figure 14: Contact Automation *Dockerfile*

In figure 14 the *Dockerfile* used for building the Contact Automation Docker image is shown. The first line of the *Dockerfile* declares the base image used for building the image. By using Python's official base image, the installation process of Python is skipped. After that the *Dockerfile* states that the Python libraries required by the application should be installed using the *pip3* command. Then the content of the current folder is copied from the host machine into the image. The two final lines of the *Dockerfile* specify that the program python3 with the argument main.py will get executed when a container created from the image is started.state that whenever a container created from this image is being started, execute *main.py* file using *python3* command.

A Docker image can be built from the *Dockerfile* using the *docker build* command.

```
$ docker build -t estcube/contactautomation:latest application/
```

Figure 15: Building the Contact Automation Docker image

As shown in figure 15 the *docker build* command is a pretty simple command. A tag name for the image to be built can be passed to the command using the -*t* flag. The last argument of the command has to be the location of the *Dockerfile* used by the building process. In our example, it is located in the *application* directory.

A Docker container can be created from the image previously built using the *docker run* command. The *docker run* command takes multiple arguments. An example command for running the Command Scheduler container is shown in figure 16.

```
$ docker run --name contactautomation \
-e DB_SERVICE='contactautomation_db' \
-e TLE_FETCHER_API='http://tlefetcher:8002/' \
-e RABBITMQ_HOST='rabbitmq' \
-e SATELLITE_LOCATION_PREDICTOR_URL='http://satellitelocationpredictor:8003/'\
-d -p 8006:8006 \
--link contactautomation_db:contactautomation_db \
--link tlefetcher:tlefetcher \
--link satellitelocationpredictor:satellitelocationpredictor\
--link rabbitmq:rabbitmq \
estcube/contactautomation
```

Figure 16: Starting the Contact Automation Docker container

As shown in figure 16, multiple environment variables can be passed to the container using the -*e* argument. The argument -*d* states that the container is started in the background. Mapping a port from the host machine to toe container is done using the -*p* argument. Containers that are required by the Contact Automation container are linked using the --*link* argument. Multiple links can be created using this parameter, as also seen in figure 16. The last argument of the *docker run* command is the name of the image to use.

6. Future work

The current solution meets the requirements that were set in the beginning, but the application is not yet a usable by a satellite operator. Therefore future improvements have already been planned for this software component.

Firstly, the current solution is missing the on-ground testing part, as described in Chapter 4.2. The integration with the hardware testing platform will be added to the implemented software module as soon as the development of the hardware testing platform has finished.

Secondly, the data pushed to the messaging queue might change in the future. This future improvement might be required because the Ground Station software is still work-in-progress. The finalised method of communication will be implemented as soon as the Ground Station software is finished.

Thirdly, the current solution has an option to sort commands by priority, but an option for specifying the execution order of commands should be added. A potential solution to this would be allowing to assign a prerequisite command to each command. The command could then be executed only after the prerequisite command has been successfully executed.

Finally, to make the current solution user-friendly and usable by satellite operators, a graphical user interface is required, including the integration with the MCS. The graphical user interface must be first designed and then a mock-up of it has to be created.

7. Conclusion

This thesis presented the results of creating the first functional version of a contact automation module for the ESTCube-2 Mission Control System. The developed system allows scheduling and storing of commands for upcoming satellite overpasses. For each command it's status is tracked and therefore the system can also reschedule failed commands. The software module was also successfully integrated with the existing mission control system using a microservices architecture. The integration with the mission control system is described in Chapter 5.4.

As some of the mission control system components that are required by the Contact Automation module were still in development, the functionality of the developed modules was verified and tested using a simulator program also implemented as a part of this thesis. The workings of the simulator program are described in Chapter 5.3.

All of the requirements specified in Chapter 3 have been fulfilled, however the implemented application is not yet usable by satellite operators. The future work that needs to be done is described in Chapter 6.

8. References

[1] "ESTCube-2," [Online]. Available: http://www.estcube.eu/en/estcube-2 [Accessed 10.05.2017].

[2] M. Noormaa, A. Slavinskis "ESTCube team makes first contract for building space cameras," [Online]. Available: https://www.ut.ee/en/news/estcube-team-makes-first-contract-building-space-cameras [Accessed 10.05.2017].

[3] A. Slavinskis *et al.* "ESTCube-1 In-Orbit Experience and Lessons Learned", *IEEE Aerospace and Electronic Systems Magazine*, August 2015, vol. 30, no. 8, pp. 12-22, 2015

[4] H. Ehrpais, I. Sünter, E. Ilbis, *et al.* "ESTCube-2 mission and satellite design", *The 4S Symposium 2016*, p. 1, 2016

[5] Elizabeth Mabrouk "What are SmallSats and CubeSats?", [Online]. Available: https://www.nasa.gov/content/what-are-smallsats-and-cubesats [Accessed 10.05.2017]

[6] T.D. Gooley, J.J. Borsi, J.T. Moore "Automating Air Force Satellite control Network (AFSCN) scheduling", *Mathematical and Computer Modelling*, vol. 24, no. 2, p. 92, 1996

[7] N. Zufferey, P. Amstutz, P. Giaccari "Graph colouring approaches for a satellite range scheduling problem", *Journal of Scheduling*, August 2008, vol. 11, issue 4, p. 264, 2008

[8] E.-I. Croitoru, G. Oancea "Satellite tracking using NORAD two-line element set format", *Scientific Research & Education in the Air Force - AFASES . 2016*, vol. 1, p. 424, 2016

[9] "Definition of Two-line Element Set Coordinate System", [Online]. Available: https://spaceflight.nasa.gov/realdata/sightings/SSapplications/Post/JavaSSOP/SSOP_Help/tle_def.html [Accessed 10.05.2017]

[10] "TLE Format", [Online]. Available: http://www.celestrak.com/columns/v04n03/ [Accessed 10.05.2017]

[11] "Orekit", [Online]. Available: https://www.orekit.org/ [Accessed 10.05.2017]

[12] "SGP4 (ORbit Extrapolation KIT 8.0 API)", [Online]. Available: https://www.orekit.org/static/apidocs/org/orekit/propagation/analytical/tle/SGP4.html [Accessed 10.05.2017]

[13] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis "Microservices in Practice, Part 1", *IEEE Software*, January/February, vol. 34, issue 1, p. 95, 2017

[14] "Pika Python AMQP Client Library", [Online]. Available: https://pypi.python.org/pypi/pika [Accessed: 10.05.2017]

[15] "PostgreSQL", [Online]. Available: https://www.postgresql.org/about/ [Accessed 10.05.2017]

[16] "Dockerfile reference", [Online] Available: https://docs.docker.com/engine/reference/builder/ [Accessed 10.05.2017]

Appendices

I. Source code

The source code for the application developed can be found in the attached archive file.

II. Sample ESTCube-1 overpass times

Below is a list of example ESTCube-1 overpasses from between the 23th and 24th of April 2017. The overpass times are estimated by the Satellite-Location Predictor application.

satellite	ground_station	start_time	end_time
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 10:10:49	2017-04-23 10:24:15
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 11:47:51	2017-04-23 11:59:50
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 13:24:57	2017-04-23 13:33:16
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 15:01:25	2017-04-23 15:05:49
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 16:34:55	2017-04-23 16:41:01
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 18:07:39	2017-04-23 18:17:58
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 19:42:07	2017-04-23 19:55:04
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 21:19:00	2017-04-23 21:31:57
ESTCUBE 1	Tõravere - ES5EC	2017-04-23 22:59:20	2017-04-23 23:07:54
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 07:26:11	2017-04-24 07:33:14
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 09:01:43	2017-04-24 09:14:26
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 10:38:32	2017-04-24 10:51:47
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 12:15:37	2017-04-24 12:26:36
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 13:52:39	2017-04-24 13:59:31
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 15:28:23	2017-04-24 15:32:32
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 17:01:08	2017-04-24 17:08:40
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 18:34:19	2017-04-24 18:45:44
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 20:09:30	2017-04-24 20:22:48
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 21:47:20	2017-04-24 21:59:31
ESTCUBE 1	Tõravere - ES5EC	2017-04-24 23:29:47	2017-04-24 23:34:16

III. Sample TLE Fetcher output

Below is an example output of the TLE Fetcher software module.


```
{
    "ESTCUBE 1": {
        "satelliteNumber": 39161,
        "classification": "U",
        "launchYear": 2013,
        "launchNumber": 21,
        "launchPiece": "C",
        "ephemerisType": 0,
        "elementNumber": 999,
        "epoch": "2017-04-20T21:22:38.905",
        "meanMotion": 0.0010703716628493328,
        "meanMotionFirstDerivative": 3.7202716501808145E-15,
        "meanMotionSecondDerivative": 0.0,
        "eccentricity": 8.525E-4,
        "inclination": 1.7109026324987415,
        "perigeeArgument": 5.8493470696821035,
        "rightAscensionOfTheAscendingNode": 3.449442553702813,
        "meanAnomaly": 0.43521530227730604,
        "revolutionNumberAtEpoch": 21232,
        "bStar": 4.324E-5,
        "line1": "1 39161U 13021C   17110.89072807 +.00000221 +00000-0 +43240-4 0
9990",
        "line2": "2 39161 098.0275 197.6385 0008525 335.1429 024.9360
14.71866691212322"
    }
}
```

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Karel Liiv

1. Herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Contact Automation For The ESTCube-2 Mission Control System,

supervised by Umesh Anilchandra Bhat and Tõnis Eenmäe.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **11.05.2017**