External Control of the Yebes Observatory Radiotelescopes for Man-made Satellites

Pablo Collado Soto
Francisco Javier Beltrán Martínez
Pablo de Vicente Abad

CDT Technical Report 2021-5

Contents

Listings

List of Figures

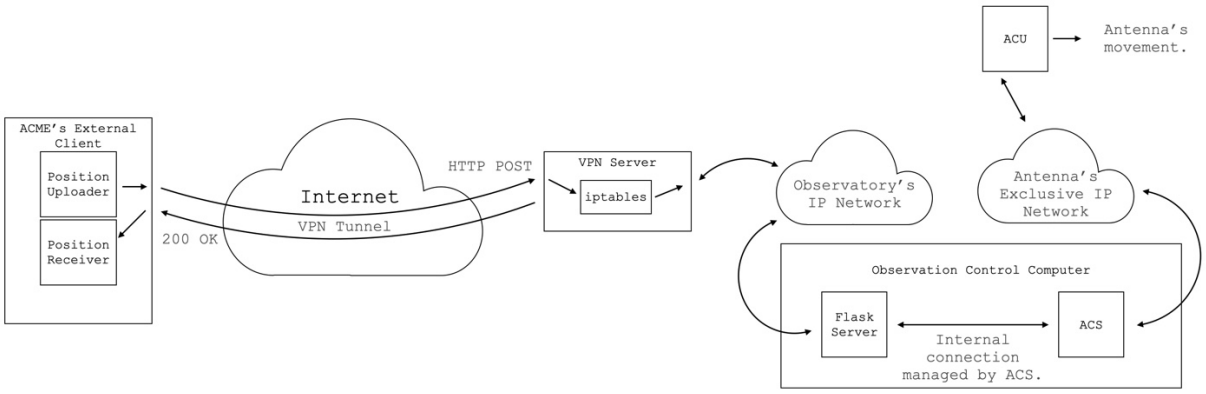

Figure 1: A "bird's eye view" of the designed system (check [1]).

1 Introduction

Radio telescopes are versatile instruments capable of carrying out a myriad of tasks. Even though the *Yebes Observatory* 40 m *Radio Telescope* (`40m RT` from here on) is mainly used for radioastronomy, it can also be leveraged to track man-made satellites and probes. Tradition has it that entities interested in using *Aries* for this latter purpose provide tables containing the positions the antenna should visit **before** the observation is to take place.

The goal of the work described in this report is to provide a system that allows controlling and monitoring the `40m RT` from off-site locations whilst maximizing safety and simplicity. The basic idea behind it is to allow the upload of tracking positions in almost real time. The original observation control system did not allow such possibility, but its modularity has allowed adding such functionality without being too invasive.

Providing this feature had us devise an `HTTP`-based server capable of controlling the antenna together with a way of providing secure access to it through the public `Internet`. On top of that, we have written several clients that upload positions to the antenna together with a program for querying the current antenna position.

Allowing users off-site to access the observatory's network was a critical part of the overall design. We decided to leverage the benefits of `VPN` technologies through the `WireGuard VPN` implementation. We added an extra layer of control on top of it so that external users could only access a subset of the observatory's machines. This limitation can be imposed on a per-user basis, granting almost surgical precision when it comes to access control.

This report is concerned with the explanation of how all these "pieces" work in consonance and of how to leverage this system from the perspective of a user. We have included an overall view of the system on figure 1. Even though it might seem a bit complex now we hope it's made clearer as the report progresses.

2 Getting the code

The different software components comprising the project can be queried and downloaded at git.oan.es. The configuration for the `VPN` is **not** publicly accessible as it contains rather sensitive

information. The repository itself contains a rather extense `README.md` file covering several aspects regarding the system we have developed. It's organized as a set of directories whose main components are described in the following subsection. Please note we'll mention several terms that might not make that much sense at the moment. We encourage the reader to work through the entire document and then revisit this section, as things will be much clearer then.

2.1 The repository's structure

The repository's root is composed of both a `README.md` file containing information on the system and a series of directories:

2.1.1 `clients/`

This directory contains the clients we have developed together with a `JSON` file containing test positions we make the antenna move to.

2.1.2 `server/`

This directory contains the `flask`-based server together with the `antenna_commander.py` file where we define the `antenna_commander` class managing the antenna itself. We can also find a script that automatically launches the server together with a `unit` file that allows the server to be run as a `systemd` daemon.

2.1.3 `reimplemented_acu_controller/`

This directory contains the code implementing the `ACS` *ACU* component. In order to accommodate the existing system to our needs we had to modify a part of the preexisting code which we decided to control with `git` as well.

2.1.4 `reimplemented_sockets/`

This directory contains the code handling sockets connection within `ACS`. Even though these changes didn't make it into the `ACS` installation currently running *Aries* we decided to leave our proposed changes in the repository should they come in handy one day.

2.1.5 `tests/`

This directory contains a single script we once used to test whether we were converting dates in different formats correctly. It's intended to be used as a "playground" for rough sketches and ideas as well, even though they don't always get uploaded to the repository.

3 The antenna control system: ACS

Aries is controlled by means of the ACS (ALMA Common Software) software system. It was originally developed for the *ALMA Interferometer* and has been adapted for the *Yebes Astronomical Observatory*.

ACS is only concerned with managing components and containers that in turn control real life equipment such as *backends* or the *ACU* (*A*ntenna *C*ontrol *U*nit), for instance. Due to the fact that ACS supports several programming languages, it imposes severe rules that programs should adhere in order to be a part of the ACS installation. This allows for an almost non-existent degree of flexibility that the developer must overcome.

Given the limited time we had to develop a solution it felt like integrating our code with ACS was bound to be a hard task. What's more, the existing codebase is huge and full of abstraction layers between the real-world equipment and the coded representations of them. Given we wanted to implement a new enhancement with far reaching implications it would have been truly awful to modify each and every layer of the preexisting work.

Lucky for us, ACS allows external programs control *preexisting* containers through a TCP/IP-based client. This is the approach we will follow: the software controlling the antenna will be implemented as a standalone HTTP server that controls the antenna through an ACS client.

3.1 What do we need to control?

We mustn't forget that the input to our system is a collection of positions the antenna is to move to. Institutions wanting to leverage *Aries* for tracking will usually provide their own *backends* capable of processing and/or storing the required data. Then, we only need to move the antenna based on the positions uploaded by the user.

Aries is quite a complex system where a lot fo components are deeply related and work in an interconnected fashion. One of the key pieces is the *ACU* we referred to before. The *ACU* is an independent computer controlling all the aspects related with the antenna's movement that offers a TCP/IP interface. That interface is employed by the ACS component in charge of commanding the antenna to different positions.

Then, we need to somehow control the *ACU* from our standalone server by opening an ACS client to the container controlling the *ACU* (i.e. OCAcu).

3.2 Overcoming existing limitations: the *ACU*'s tables

We would like to begin by stating that this report is not strictly concerned with the internals of the antenna's control. We should have been able to work with preexisting implementations of ACS but, given how they were initially designed, they posed several limitations. Let's analyze *how* we overcame these issues and *what* these issues were in the first place.

The *ACU* driving the antenna is implemented as a program whose input data comes from several tables it manages. In order to instruct the *ACU* to go to a specific position at a given time we need to insert time-tagged entries into these tables. The *ACU* will then continually read them and move the antenna to the commanded positions until they have all been visited. The control

server's task is then acting as an "intermediary" between the clients and the *ACU* itself as it'll receive the commanded positions that it'll then append to the existing *ACU*-managed tables. It is important to note that the *ACU* will **interpolate** the provided positions in an effort to move in a "smooth" fashion. In a later paragraph we discuss how new positions must be provided in batches of at least 5 elements. This condition is imposed on us by the fact that the *ACU* needs at least 5 points as the input to its interpolation process. This is nonetheless transparent to us: we *implicitly* control this interpolation through the `timestamps` associated to the commanded positions but we do not need to carry it out ourselves.

Even though the tables themselves are implemented as a *ring buffer*, which means we can think of them as an "infinite" global table, each of them contains 50 entries. The implementation communicating with the *ACU* we found was **always** adding at least 50 entries to these tables when commanding the antenna. This doesn't pose a problem when we need to upload a precomputed position table with more than 50 positions but, when we have less points (which is our case) that does pose a problem. Those functions wanting to drive the antenna were just replicating the last position $50 - N$ times where N is the number of positions they did "really" want to visit. This approach might be good enough for certain use cases, but this was not our case. When being driven in real time, we expect clients to provide "small" position batches (with definitely less than 50 positions). We cannot then replicate entries and add them to the tables: we need to somehow circumvent this issue.

One might think that we could just delete unneeded entries and add new ones on demand, but the *ACU* itself isn't too flexible in that regard. We can only either `reset` the tables, create a `new` table or `add` entries to the current one (before it finishes). On top of that we need to specify a reference time at which the table will begin being processed and we need to tag every position with a `timestamp` relative to the initial time to let the antenna know when it should reach each position. This allows the antenna to interpolate between separated points so that it moves in a smooth fashion, something critical for tracking duties.

We then see how there was no getting around the main limitation: we needed to be able to add a variable number of entries to the main position table. Due to *ACU* limitations this number had to belong to the $[5,\ 50]$ interval nonetheless. In any case, the limitation was rooted in the fact that the `C++` program communicating with the *ACU* stored the positions on a 50-element `static array` and it iterated over it based on a `constant` value, not the number of positions we really had to upload to the *ACU* itself. This position array is then uploaded through a `TCP/IP` socket to the *ACU* so that the commanded positions become effective.

In order to meet our system's requirements we had to `dynamically allocate` the position `array`s through the `new` operator and to then free them with the `delete[]` operator. These are practically equivalent to the `malloc()` and `free() C` functions for those "classic programmers". Thus, we only had to allocate as many elements as positions we were provided which unlocked the *ACU*'s full potential.

These changes regarding memory management called for alterations in the `IDL` interfaces that were also enforcing the use of fixed 50-element arrays within `ACS`. We finally had to write our own `method` for uploading the `dynamic array`s to the *ACU* given the handling of `pointers` brought up a few subtleties that we didn't have to deal with before. This method had been originally implemented on the `reimplemented_sockets/` directory. We nonetheless decided to include it in the `acu.cpp` file instead thus leaving the source files regarding sockets

untouched. Even though these are **not** on the "live" antenna control system we have left them here should they come in handy in the future.

All these changes are what we have implemented in the source files found under the `reimplemented_acu_controller/` directory. These have been running on the antenna for some time now so we are moderately confident that they have been written in a clean and durable way that, up to now, hasn't caused any problems whatsoever.

We have included the `method` handling the upload of dynamic position entries to the *ACU* together with some `dynamic memory allocations` on listing 1. These lines are all extracted from the `acu.cpp` file.

```
/*
    Method capable of sending a command containing a dynamic list over a TCP socket.
*/
UINT32 Acu::writeDynamicCommand(const void* cmd, const int& hdr_size,
const void* dyn_data, const int& dyn_size,
const void* end_flag, const int& end_flag_size) {
    ACS_SHORT_LOG((LM_INFO, "Sending command header"));
    cmdACU->Write(cmd, hdr_size, 0);

    ACS_SHORT_LOG((LM_INFO, "Sending dynamic data"));
    cmdACU->Write(dyn_data, dyn_size, 0);

    ACS_SHORT_LOG((LM_INFO, "Sending end flag"));
    int status = cmdACU->Write(end_flag, end_flag_size, 0);

    ACS_SHORT_LOG((LM_INFO, "Command sent!"));

    return status;
}

/*
    A dynamic memory allocation and its respective deallocation.
*/
mdpt->az_el_sequence = (TimeAzEl*) new (nothrow) TimeAzEl[sequence_length];
delete[] mdpt->az_el_sequence;
```

Listing 1: The `WriteDynamic()` method and a dynamic allocation.

4 The control server

Given how pervasive the `IP` technology is today, carrying out tasks remotely is usually synonym with providing some sort of `IP` aware server. We have chosen to leverage the `HTTP` application protocol (sitting on top of `TCP` and `IP`) through the `flask` server implementation. An external client will supply time-tagged positions for the `40m RT` using `HTTP` requests to a `flask`-implemented server which will parse and process them. Then, the server acts as an "intermediary" which receives a request containing positions as arguments and passes these to an `ACS` component (through a `python`-written client provided by `ACS`'s own internal "machinery") that redirects them to the *ACU*.

As we have pointed out before, the `HTTP` server has been implemented using `flask`. `Flask` advertises itself as a *micro web framework* that enables a user to write web applications using the `python` language. We will be using it in a somewhat "unorthodox" way in the sense that we won't be providing a regular `HTTP` server offering web pages to users: we'll instead be implementing an `HTTP` interface through which we can upload positions to the antenna

and query its current positions for instance. The choice of `flask` over a more traditional `HTTP` server (such as `Apache2` or `Nginx`) has been motivated by the fact that it's light and that it offers *endpoints*. These *endpoints* are different `URL` endings that will trigger an associated function when an `HTTP` request is "aimed at them". This makes responding to these requests and triggering actions through them extremely easy.

4.1 A quick note on `flask`

We have developed a `flask` server offering **five** different `endpoints` to and end user. Each of these endpoints is associated with a function that'll be executed upon the reception of an `HTTP` request. The function has complete access to the request's data, which lets these requests contain *payloads* such as the positions we are to send the antenna to, for instance. Replies to those requests can of course carry additional data such as the antenna's current position in our case.

Now, if a client wants to make use of the server's functionality it only needs to send `HTTP` requests to the different endpoints containing the correct payloads (i.e. the contents of the `HTTP` request message). These requests can be made through programs such as `wget` and `curl` and from `python` libraries as well. The client capable of uploading antenna positions we have developed leverages the `requests` python module, for instance.

We'll look into the payload formats when discussing clients in the next section.

4.2 Installation on Debian platforms

Installing `flask` on a system-wide level can be accomplished with `sudo apt update && sudo apt -y install python-flask`. We have decided to instead run `flask` on a *python* virtual environment so that the `flask` installation is isolated from the rest of the system. We carry our the entire procedure automatically through the `launch_command_server.sh` script which we include on listing 2.

```
1  #!/bin/bash
2
3  # If the virtual environment doesn't exist create it
4      # and install flask right away. Otherwise
5      # source the existing virtual environment.
6  if [ ! -d antenv ]
7  then
8      # You may need to run 'sudo apt update && sudo apt -y install python-virtualenv'
       beforehand
9      echo "Creating the antenv virtual environment"
10     python -m virtualenv antenv
11
12     echo "Activating the antenv virtual environment"
13     source antenv/bin/activate
14
15     echo "Updating pip"
16     python -m pip --upgrade install pip
17
18     echo "Installing flask..."
19     python -m pip install flask
20 else
21     echo "Activating the antenv virtual environment"
22     source antenv/bin/activate
23 fi
```

```
24
25 # We tell flask we are running in a development environment
26     # so it doesn't complain!
27 echo "Running the command server"
28 FLASK_ENV=development python ./antenna_command_server.py
```

Listing 2: Automatic server activation.

4.3 Running the server as a daemon

We have also written a `unit file` letting the `flask` server run as a *daemon* controlled by `systemd`. We are including it on listing 3. It currently runs the provisioning server within a virtual environment, but could easily be adapted to run it together with a system-wide installation. Installing the file is just a matter of running `sudo cp <server_unit_file> /etc/systemd/system` on the machine acting as the provisioning server. After that, one should be capable of seeing the service through `systemctl status <unit_file_name>`. As usual, we can enable it at boot with `sudo systemctl enable <unit_file_name>` and start it with `sudo systemctl start <unit_file_name>`. This is by no means a guide on using `systemd`: take these commands with a grain of salt and adapt them to your system.

```
1 [Unit]
2 Description=Flask-based Antenna Control Server
3 After=network.target
4
5 [Service]
6 User=server-username
7 WorkingDirectory=/path/to/antenna_command_server_source/
8 ExecStart=/bin/bash /path/to/antenna_command_server_source/launch_command_server.sh
9 Restart=always
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 3: Antenna command server unit file.

4.4 The server code

The functionality offered by the command server is decoupled in two different files. The server itself is contained in `antenna_command_server.py` whose contents are displayed on listing 4. The server implements the following *endpoints*:

1. **`/`:** Allows a client to check whether the control server is listening. This is considered a *safe* endpoint.

2. **`/get_position_test`:** Allows testing a client for retrieving positions that's being developed. This endpoint will just return a couple of random numbers. This is considered a *safe* endpoint.

3. **`/get_position`:** Allows a client to query the antenna's current position. This is considered a *safe* endpoint.

4. **`/upload_position`:** Allows a client to upload a batch of positions to the antenna. This is considered a *risky* endpoint.

5. **`/halt_antenna`:** Allows a client to stop the antenna's movement. This is considered a *risky* endpoint.

We would like to draw the reader's attention to how we have deemed each of the previous endpoints as either *safe* or *risky*. This distinction is related to whether the associated functions will try to actively command the antenna. The first group will just query `ACS` for information or just return some information (like **`/`** does). The latter will instead try to either move or stop the antenna, which could have unintended side effects. That's why these will only be available if whoever brings up the server explicitly enables them by passing the `enrisky` argument when doing so.

Note that the code contained in listing 4 is **not** concerned with the interaction with `ACS` at any time. We have tried to decouple the logic driving the antenna from the `HTTP` server itself so that these two parts are as independent as possible. This in turn makes the system modular and easier to upgrade in the future, should new functionalities be required.

```
# Import the required modules
import flask
import random
import sys
import antenna_commander

# Flag controlling whether "risky" endpoints are enabled or not
enable_risky_endpoints = False

# We'll control the definition of risky endpoints based on a parameter
    # passed through the command line. We'll also print an informative
    # message regarding the state that's being configured.
if __name__ == '__main__':
    if len(sys.argv) > 1:
        #if sys.argv[1] == "enrisky":
        if "enrisky" in sys.argv:
            print("INFO: Risky endpoints are enabled!")
            enable_risky_endpoints = True
        else:
            print("INFO: Risky endpoints are disabled!")
    else:
        print("INFO: Risky endpoints are disabled!")

# Instantiate a Flask app
app = flask.Flask(__name__)

# Instantiate the antenna commander
ant = antenna_commander.antenna_commander()

# Note the following 3 endpoints are considered "safe" and will always be
    # available when the server is up and running.

# Define a default endpoint for checking whether the server is listening as it should.
@app.route('/')
def check_server():
    return flask.jsonify("You see me! :P" if enable_risky_endpoints else "You see a reduced
    version of me! :P"), 200

# Define an endpoint allowing the test of clients that are intended to query the antenna's
    # position
@app.route('/get_position_test')
def get_position_test():
```

```
42      return flask.jsonify({'azimuth': round(random.random() * 360, 2), 'elevation': round(
        random.random() * 90, 2)})
43
44  # Query the current antenna position
45  @app.route('/get_position')
46  def get_position():
47      return flask.jsonify(ant.currPosition())
48
49  # The following endpoints are considered "risky" and will only be enabled if
50      # an option is explicitly given on the command line.
51  if enable_risky_endpoints:
52      # Upload a given position batch the antenna is to follow
53      @app.route('/upload_position', methods = ["POST"])
54      def upload_position():
55          print("Got a request from {} with content: {}".format(flask.request.remote_addr, flask
        .request.json))
56          pos_data = flask.request.json
57          if ant.gotoHorizontalPosition(pos_data['batch']) == 0:
58              return flask.jsonify("Position sent correctly!"), 200
59          else:
60              return flask.jsonify("There was trouble sending data to the antenna..."), 500
61
62      # Halt the antenna
63      @app.route('/halt_antenna')
64      def halt_antenna():
65          if ant.haltAntenna() == 0:
66              return flask.jsonify("Antenna halted correctly!"), 200
67          else:
68              return flask.jsonify("There was trouble halting the antenna..."), 500
69
70  # Run the flask application
71  if __name__ == '__main__':
72      try:
73          # Run the flask application
74          app.run(host = '0.0.0.0', port = 8080, debug = False)
75      except KeyboardInterrupt:
76          print("Ctrl + C pressed, exiting...")
77      finally:
78          # Free up any connections made to ACS components
79          ant.release()
80
81          # Exit with a "clean" exit code
82          sys.exit(0)
```

Listing 4: The antenna command server (`antenna_command_server.py`).

4.5 The antenna commander

As pointed out in the previous section, we have chosen to implement the logic driving the antenna itself in a separate class in an effort to increase the system's elegance and modularity. This is what came to be the `antenna_commander` class (defined on the `antenna_commander.py` file) that we include on listing 5. The code itself is documented, so we believe it to be a bit pointless to explain things twice. We just want to point out how the server is just concerned with calling `antenna_commander`'s methods through an instance of the class (i.e. an `object`). This allows for a simplified server and a cleaner design, just what we sought to achieve.

```
1  # ACS-related stuff
2  from Acspy.Clients.SimpleClient import PySimpleClient
3  from Acspy.Nc.Consumer import Consumer
4  import oanAcu40m, oanAntenna, scanUtils
```

```
5
6  # General purpose modules
7  from datetime import datetime
8  from astropy.time import Time
9  import time, copy
10
11 class antenna_commander():
12     # Open the necessary connections to ACS components
13     def __init__(self, acu_component = "ARIES21"):
14         try:
15             self.sc = PySimpleClient()
16             self._a40m = self.sc.getComponent(acu_component)
17             self.componentName = acu_component
18             self.curr_pos = None
19             self.pos_consumer = Consumer(oanAntenna.MAINAXISCHANNEL)
20             self.pos_consumer.addSubscription(oanAntenna.mainAxisAntennaNotifyBlock, self.
    _positionHandler)
21             self.pos_consumer.consumerReady()
22         except:
23             print("Couldn't open the ACU Component through Name: {}".format(acu_component))
24             exit(-1)
25
26         # Initialize the required attributes
27         cdb_element = self.sc.getCDBElement("alma/ANTENNA_40M", "Antenna")[0]
28         self._movementLimits = {
29             k: float(v) for k, v in [(name, cdb_element[name]) for name in ["azUpperLimit", "
    azLowerLimit",
30                                                                             "elUpperLimit", "
    elLowerLimit",
31                                                                             "vmaxAz", "vmaxEl"
    ]]
32         }
33
34         self._acu_table_limits = {'minLines': 5, 'maxLines': 50}
35
36         self._mis_parameters = {'acuSocketTime': 0.1, 'beganMoving': False, 'refTime': None, '
    pastPositions': [], 'lastTime': None}
37
38     # This method will be called every time a new position notification arrives. It'll
39         # just update the antenna's current position.
40     def _positionHandler(self, antStr):
41         self.curr_pos = antStr
42
43     # Return the antenna's current position
44     def currPosition(self):
45         # Avoid performing the check on a position and then working
46             # with a different one if the handler was called in between!
47         antInfo = self.curr_pos
48         if antInfo is None:
49             return  {
50                         'dt': datetime(2000,1,1),
51                         'azimuth': 0.0,
52                         'elevation': 0.0,
53                         'azimuth_error': 0.0,
54                         'elevation_error': 0.0
55                     }
56         else:
57             return {
58                         'dt': Time(antInfo.jdacu, format = 'jd').datetime.strftime("%Y/%m/%d %
    H:%M:%S.%f"),
59                         'azimuth': antInfo.azimuth,
60                         'elevation': antInfo.elevation,
61                         'azimuth_error': antInfo.azError,
62                         'elevation_error': antInfo.elError
63                     }
64
65     # Check the passed coordinates are within the required bounds.
```

```
66      def _validatePositions(self, positions):
67          for position in positions:
68              az = position['azimuth']
69              if az < self._movementLimits['azLowerLimit'] or az > self._movementLimits['
    azUpperLimit']:
70                  print("Provided azimuth ({} deg) coordinate is out of range!".format(az))
71                  return -1
72              el = position['elevation']
73              if el < self._movementLimits['elLowerLimit'] or el > self._movementLimits['
    elUpperLimit']:
74                  print("Provided elevation ({} deg) cordinate is out of range!".format(el))
75                  return -1
76          return 0
77
78      # Move the antenna to the commanded positions
79      def gotoHorizontalPosition(self, newPositions):
80          if self._validatePositions(newPositions) == -1:
81              return
82
83          interpolationMode = oanAcu40m.Acu40m.Newton
84          trackingMode = oanAcu40m.Acu40m.azel
85          minLines = self._acu_table_limits['minLines']
86
87          if not self._mis_parameters['beganMoving']:
88              loadMode = oanAcu40m.Acu40m.newL
89              self._mis_parameters['refTime'] = datetime.strptime(newPositions[0]['startTime'],
    "%Y/%m/%d %H:%M:%S.%f")
90              self._mis_parameters['refTimeStr'] = newPositions[0]['startTime']
91              self._mis_parameters['beganMoving'] = True
92          else:
93              loadMode = oanAcu40m.Acu40m.addL
94
95          alphaDeltaSequence = [
96                  oanAcu40m.Acu40m.mainDriveStruct(
97                      (datetime.strptime(pos['startTime'], "%Y/%m/%d %H:%M:%S.%f") - self.
    _mis_parameters['refTime']).seconds * 1000,
98                      pos['azimuth'],
99                      pos['elevation']
100                 ) for pos in newPositions
101         ]
102
103         tInt = (datetime.strptime(newPositions[-1]['startTime'], "%Y/%m/%d %H:%M:%S.%f") -
    self._mis_parameters['refTime']).seconds/49.0 * 1000
104
105         tInt_steps = [tInt * i for i in range(50)]
106
107         alphaDeltaSequence_offsets = [
108                 oanAcu40m.Acu40m.mainDriveStruct(
109                     int(value),
110                     0.0,
111                     0.0
112                 ) for value in tInt_steps
113         ]
114
115         [alphaDeltaSequence.append(copy.copy(alphaDeltaSequence[-1])) for i in range(minLines
    - len(newPositions))]
116
117         year, doY, msoD = self._toNaturalDate(self._mis_parameters['refTimeStr'])
118
119         print("Start time decomposition: Y = {}\tDoY = {}\t msoD = {}".format(year, doY, msoD)
    )
120
121         print("Commanded position:\n\tST -> {}\n\tLoad mode -> {}\n\tSeq Length -> {}\n\
    tGenerated alpha-delta sequence ->\n{}".format(
122                 self._mis_parameters['refTime'], loadMode, len(alphaDeltaSequence),
    alphaDeltaSequence
123         ))
```

```
124
125         try:
126             self._checkAck(
127                 self._a40m.mainDriveProgramTrack(interpolationMode, trackingMode, loadMode,
    len(alphaDeltaSequence), year, doY, msoD,
128                     self._movementLimits['vmaxAz'], self._movementLimits['vmaxEl'],
    alphaDeltaSequence)
129             )
130         except Exception as e:
131             print("Exception when calling mainDriveProgramTrack() when driving the antenna!:
    %s" % str(e))
132             return -1
133         return 0
134
135     # Bootstrap the process stopping the antenna
136     def haltAntenna(self):
137         res = self._stopSpatialPattern()
138         if res == -1:
139             return -1
140         ret_val = self._stopAntenna()
141         return ret_val
142
143     # Stop the antenna's motors
144     def _stopAntenna(self):
145         self._clearMainDriveTable()
146
147         time.sleep(self._mis_parameters['acuSocketTime'])
148
149         try:
150             self._checkAck(self._a40m.mainDriveStop(oanAcu40m.Acu40m.bothS))
151         except:
152             print("Exception when calling mainDriveStop() when stopping the antenna!")
153             return -1
154         return 0
155
156     # Reset the positions currently loaded on the antenna
157     def _clearMainDriveTable(self):
158         az = self._a40m.actPosAz()
159         el = self._a40m.actPosEl()
160
161         nowEph = scanUtils.nowEfem()
162         nowEph = scanUtils.addMSecondstoEfemDateTime(nowEph, 500)
163         year, doY, msoD = scanUtils.yearDayAndMsOfDay(nowEph)
164
165         interpolationMode = oanAcu40m.Acu40m.Newton
166         trackMode = oanAcu40m.Acu40m.azel
167         loadMode = oanAcu40m.Acu40m.resetL
168
169         nlines = self._acu_table_limits['minLines']
170
171         alphaDeltaSequence = []
172         for i in range (0, nlines):
173             ms = int(i * 1000 / (nlines - 1))
174             alphaDeltaSequence.append(oanAcu40m.Acu40m.mainDriveStruct(ms, az, el))
175         for i in range (nlines, self._acu_table_limits['maxLines']):
176             alphaDeltaSequence.append(oanAcu40m.Acu40m.mainDriveStruct(ms, az, el))
177
178         try:
179             self._checkAck(
180                 self._a40m.mainDriveProgramTrack(interpolationMode, trackMode, loadMode,
    nlines, year, doY, msoD,
181                     self._movementLimits['vmaxAz'], self._movementLimits['vmaxEl'],
    alphaDeltaSequence)
182             )
183         except:
184             print("Exception when calling mainDriveProgramTrack() when stopping the antenna!")
185
```

```
186     # Clear the auxiliary position tables
187     def _stopSpatialPattern(self):
188         nowDateTime = scanUtils.nowEfem()
189         year, doY, msoD = scanUtils.yearDayAndMsOfDay(nowDateTime)
190 
191         try:
192             self._checkAck(self._a40m.mainDriveProgramOffsetControl(oanAcu40m.Acu40m.stopC,
    year, doY, msoD))
193         except:
194             print("Exceptionw when calling mainDriveProgramOffsetControl() when stopping the
    antenna!")
195             return -1
196         return 0
197 
198     # Log out an informative message based on the received ACK from the ACU
199     def _checkAck(self, ackCmd):
200         if ackCmd == oanAcu40m.Acu40m.cmdACCEPTED:
201             return 0
202         else:
203             if ackCmd == oanAcu40m.Acu40m.cmdUNDEF:
204                 print("Wrong ACU acknowledgement: Undefined")
205             elif ackCmd == oanAcu40m.Acu40m.cmdPARAMERR:
206                 print("Wrong ACU acknowledgement: Parameter error")
207             elif ackCmd == oanAcu40m.Acu40m.cmdLENERR:
208                 print("Wrong ACU acknowledgement: message length error")
209             elif ackCmd == oanAcu40m.Acu40m.cmdWRONGMODE:
210                 print("Wrong ACU acknowledgement: Wrong mode")
211             else:
212                 print("Wrong ACU acknowledgement: Unknown acknowledgment!")
213             return -1
214 
215     # Extract the year, day of the year and milliseconds of the day from a text-based date
216     def _toNaturalDate(self, date, date_format = "%Y/%m/%d %H:%M:%S.%f"):
217         try:
218             if type(date) == unicode:
219                 t = datetime.strptime(date, date_format)
220             elif type(date) == float or type(date) == int:
221                 t = Time(date, format = 'mjd').to_datetime()
222             return t.year, t.timetuple().tm_yday, int((t.hour * 3600 + t.minute * 60 + t.
    second) * 1e3 + t.microsecond / 1e3)
223         except:
224             print("The provided date ({}) has some kind of format error!".format(date))
225             return -1, -1, -1
226 
227     # Release the ACS-related clients and connections to allow for a clean exit
228     def release(self):
229         self.pos_consumer.disconnect()
230         self.haltAntenna()
231         self.sc.releaseComponent(self.componentName)
232         self.sc.disconnect()
```

Listing 5: The antenna commander (`antenna_commander.py`).

5 The clients

We have written two simple clients taking care of uploading positions and querying the current antenna location, respectively. These are just examples and are **enforced** upon the user. As long as other clients stick to the *endpoints* defined in our server and the data formats they handle one can use whatever he or she chooses. That's one of the main advantages of leveraging `IP`-based infrastructure: the clients could very well be written in any other programming language!

Let's analyze how the positions are to be provided to the system and the format they should adhere to.

5.1 Specifying the positions

Our system expects to be provided *azimuth* and *elevation* displacements in degrees (°). The clients we have developed supports reading these from a `JSON` formatted file and sending them right away. This is **not compulsory**: it's dictated by the client the end user decides to use. Should an institution code their own, they are the ones who should decide the format they feed their own script with. What they must ensure however is that this content is **indeed** added to the `HTTP` request content field as a `JSON` object so that the server can handle it! The opposite is true: the current position of the antenna is provided as a `JSON` object when queried. It's up to the client to process that information as it pleases.

5.2 A client for uploading positions

We have developed a small `CLI` client written in `python` that's capable of reading positions from a `JSON` file and uploading them to the command server. The code is contained in the `sample_client.py` file and it's also provided on listing 6.

```
# Import the required modules
import requests, sys, json, time
from datetime import datetime, timedelta

def main(url, delay):
    # Overwrite the start time for the positions and make the batch start now
    ref_t = datetime.now()

    # Configure the delay between positions...
    pos_delta_t = timedelta(seconds = delay)

    # And the delay between batches. Note a batch is made up of 5 positions
    batch_delta_t = timedelta(seconds = 5 * delay)

    # Iterate over the batches contained in the `tesst_displacements.json` file and send them
    to the antenna
    try:
        for i, batch in enumerate(json.load(open("test_displacements.json"))['movement_batches
    ']):
            for j, position in enumerate(batch):
                position['startTime'] = (ref_t + i * batch_delta_t + j * pos_delta_t).strftime
    ("%Y/%m/%d %H:%M:%S.%f")
            print("Sending position batch:\n\t{}".format(batch)
            print("POST request status code: {}\n".format(
                requests.post(url, json = {'batch': batch}).status_code
            )
            time.sleep(5 * delay / 2.0)
    except KeyboardInterrupt:
        # Note we can quit the client gracefully by pressing CTRL + C at any moment.
        exit(0)

if __name__ == "__main__":
    # If no arguments are provided connect to a local server and send positions
        # every minute
    if len(sys.argv) == 1:
        main("http://localhost:8080/upload_position", 60)

    # If a single argument is provided assume it's the URL identifying the server
```

```
36          # and send positions every minute as well.
37      elif len(sys.argv) == 2:
38          if sys.argv[1].startswith('http://'):
39              main(sys.argv[1], 60)
40          else:
41              print("Usage: {} [<server_url>] [<delay_secs>]".format(sys.argv[0]))
42              exit(-1)
43
44      # If two arguments are provided they are both the URL and the delay
45          # between positions in seconds.
46      elif len(sys.argv) == 3:
47          if sys.argv[1].startswith('http://'):
48              try:
49                  main(sys.argv[1], float(sys.argv[2]))
50              except ValueError:
51                  print("Usage: {} [<server_url>] [<delay>]".format(sys.argv[0]))
52                  exit(-1)
53
54          # Otherwise tell the user how to use the client properly...
55          else:
56              print("Usage: {} [<server_url>] [<delay>]".format(sys.argv[0]))
57              exit(-1)
```

Listing 6: A client for uploading positions (`sample_client.py`).

This client will read positions from the `test_displacements.json` file which we provide on listing 7. It contains a single 5-position batch. We would also like to mention that our client is overwriting the `startTime` field in each position so that we speed up the testing procedures. Either our client or a brand new one can be made aware of this data field (that's what we expect clients to do actually) to just pass it transparently to the commanding server. This would simplify the client's logic as it would only need to read the positions and pass them "as is" to the server. As we stated before, we have **not** used these positions willingly so that tests could be carried out faster without the need to modify these dates each time we launched the client.

```
{
    "movement_batches": [
        [
            {
                "azimuth": 41,
                "elevation": 87,
                "startTime": "2021/03/23 10:00:00.000"
            },
            {
                "azimuth": 42,
                "elevation": 87,
                "startTime": "2021/03/23 10:01:00.000"
            },
            {
                "azimuth": 43,
                "elevation": 87,
                "startTime": "2021/03/24 10:05:00.500"
            },
            {
                "azimuth": 44,
                "elevation": 87,
                "startTime": "2021/03/24 10:05:00.500"
            },
            {
                "azimuth": 45,
                "elevation": 87,
                "startTime": "2021/03/24 10:05:00.500"
            }
```

```
        ]
    ]
}
```

Listing 7: The test displacemens we have worked with (`test_displacements.json`).

5.3 A client querying the current antenna location

We have also developed a very simple `CLI` based `python` client that leverages the `get_position_test` endpoint to showcase how a client could query the current antenna position. The code is contained in the `position_client.py` file and it's also provided for convenience on listing 8.

```
# Import the needed modules
import requests, time, sys

def main(url, refresh_rate = 2.5):
    try:
        # Just query positions indefinitely until stopped with CTRL + C
        while True:
            # Get the current position
            curr_pos = requests.get(url + "/get_position").json()

            # And print the information
            print("Antenna position at {}:\n\tAzimuth:\t{:.4f} deg\n\tElevation:\t{:.4f} deg\n\tAzim. Error:\t{:.2f} arcsec\n\tElev. Error:\t{:.2f} arcsec\n".format(
                curr_pos['dt'], curr_pos['azimuth'], curr_pos['elevation'], curr_pos['azimuth_error'], curr_pos['elevation_error']
            ))

            # Wait a bit before requesting the next position
            time.sleep(refresh_rate)
    except KeyboardInterrupt:
        exit(0)

if __name__ == "__main__":
    # If we are run with no arguments connect to a local server
    if len(sys.argv) == 1:
        main("http://localhost:8080")

    # If we are provided a single argument assume it's the server's URL
    elif len(sys.argv) == 2:
        main("http://" + sys.argv[1])

    # Otherwise tell the user how to use us!
    else:
        print("Usage: {} [server-root-url]".format(sys.argv[0]))
```

Listing 8: A client querying positions (`position_client.py`).

5.4 A manual client

We have also developed a very simple client for uploading positions in a manual fashion. The client itself is implemented on file `manual_client.py` and we are **not** providing it here as we believe it adds little value to an "end user". If one really wants to see how it's coded it can be accessed on the repository. We just wanted to mention its existence to show evidence on how one can implement virtually any imaginable client as long as he or she sticks to the

interface provided by the server. The idea behind this otherwise "strange" client was making the development and testing of the server easier by giving us a tighter control over the timing with which positions were uploaded.

6 Providing a secure connection over the Internet: WireGuard

VPNs come in many shapes and sizes. We need to differentiate between what a VPN **is** and how it's **implemented**. VPNs *emulate* the services characterizing private networks such as the observatory's LAN whilst relying on public infrastructure such as the Internet. A common technology supporting them is MPLS (MultiProtocol Label Switching) which is label or connection oriented. This technology is gaining momentum in the backbone of Internet service providers (ISPs) for instance. Some other options exist, such as IPSec and OpenVPN but we have decided to leverage a rather new technology.

One of the aspects setting WireGuard aside from its competitors is that it's implemented in kernel-space. This mind boggling term is just a "fancy" way of saying that it's deeply interwoven in the operating system itself which, in turn, translates to a faster operation and a myriad of fantastic and powerful options. What we are the most concerned with is the fact that this technology is implemented as a standard network interface in the Linux kernel. This interface is **not** a physical one but the kernel will handle it as a regular point-to-point interface.

One might wonder why we would go to such lengths when the use of a VPN is not "strictly" required in our case: we could just expose a port to the public Internet to make the antenna control server visible to the outside world. Opening ports up on public IP addresses is bound to send chills down everyone's spine. In doing so we are enabling anybody to send information to the process bound to the specified port and, given the sophistication of today's attacks, we are just providing a recipe for disaster. We then need a way of limiting the freedom of external users whilst still providing access to whoever needs it. This feat is easily accomplishable by a VPN when working in coordination with a firewall. This is why we feel the use of a full-fledged VPN is justified.

We would like to note that, as we later discuss in more detail on section 7, this VPN implementation offers a restricted access to the observatory's internal network. Thanks to a firewall implemented through iptables, we only allow external users to communicate with specific ports on the Observation Control Computer seen on figure 1. This implies that, even though external peers are physically connected to the entirety of the internal network, they can only access a single machine: they are effectively isolated so as to guarantee the observatory's network security.

6.1 Certificate generation and double layer cryptography

In order for a user to access the Observatory's network through the VPN he or she needs a *certificate*. This is nothing more than a text document containing the encryption keys to be used between said client and the VPN server itself when exchanging data. Then, a *certificate* offers both authentication and encryption services. These certificates are to be generated by

the Observatory's personnel to then be provided to external users who are to access the internal network.

As we stated before, data exchanged over the public Internet between clients and the server is encoded so that it cannot be captured by anybody in between. WireGuard can leverage both symmetric and asymmetric key encryption for this task.

As its name implies, asymmetric key encryption uses two different keys, a public and a private one. The catch is that the public key is generated from the private key and, more importantly, one cannot recover the private key from the public key. This way of doing things is based on "one-way" functions that make it too computationally expensive to compute in the "opposite" direction. This scheme is what SSH uses, as it is based on RSA keys after all.

Even though asymmetric key cryptography solves the ever pressing problem of key distribution, it is a bit slower than symmetric key encryption. This latter option leverages a single, pre-shared key to encrypt data. It is indeed faster but, how can we share the key over an insecure medium? That's the problem asymmetric key cryptography solves very elegantly. When we have access to both schemes we can do the following:

1. Use asymmetric key cryptography to share a symmetric key. The key itself is rather small, so it's not a cumbersome, time consuming process.

2. Use the symmetric key we just shared to exchange data in a secure fashion faster than we would normally be able to with asymmetric keys.

What's more, the symmetric key adds a layer that's "quantum resistant". The term "quantum" tends to be jaw dropping in the field of science, or a synonym for a ton of money in the consumer realm. Given already developed quantum algorithms like Shor's, factoring large numbers is bound to get feasible in terms of time (the matter regarding the feasibility of useful quantum computers is another topic). Some asymmetric key cryptography algorithms such as RSA rely on the impossibility of factoring large numbers in a reasonable time to remain secure. That means that, should a quantum computer appear, these algorithms would stop being secure right away. This is not the case with symmetric algorithms as they do not rely on factoring numbers to generate keys. That's why they are considered "quantum secure" or "quantum resilient".

Asymmetric keys will need to be generated for each new peer and we can **optionally** generate symmetric keys for each new of these new peers as well. Just like any other peer, the server has got its own asymmetric keys. The next subsection showcases the commands used to generate new pairs of keys.

6.1.1 Generating asymmetric and symmetric keys

Once WireGuard is installed on a linux-based machine we have the wg utility. We can leverage it to generate private and public keys as well as symmetric ones. This can be done with the wg genkey, wg pubkey and wg genpsk utilities, respectively. Some examples are provided on 9. Note that private keys **must remain private**. One can assure this by setting a 600 permission to the generated files.

```
# Generate a private key, saving it to 'priv.key' and it's public counterpart
    # saving it to 'pub.key'.
wg genkey | tee priv.key | wg pubkey > pub.key

# Generate a symmetric key and save it to 'sym.key'
wg genpsk > sym.key
```

Listing 9: Generating WireGuard keys.

6.2 Installing WireGuard on a Debian 10 (Buster) server

Even though WireGuard leverages the peer-to-peer architecture, we can adapt it to our needs by exploiting the server-client paradigm. We'll then have N peers where one of them will behave as a server whilst the $N-1$ others act as clients. We can configure each peer in such a way that, even though WireGuard treats them in a similar way, we are indeed making each behave as either a client or a server. This model is easily scalable to manage arbitrarily large corporate networks and that's one of the reasons it's slowly making its way into the observatory's network as the *de facto* solution enabling staff to work remotely. We can then generate a certificate for any enterprise that might be willing to use the antenna control system we have devised.

Please note some of the commands in the following sections will need to be run with elevated privileges. This can be achieved through sudo or by running the commands as root him/herself. The latter can be attained by running su -, for instance.

6.2.1 Enabling backports

On Debian releases older than Bullseye (Debian 11), we need to enable the backports repository in order to get a hold of the needed packages. Anybody is free to compile WireGuard from source. It's not a monumental task as seen here, but it does make it harder to keep up with the new updated versions. On the other hand, we can rest assured that we have the latest version if we compile the sources ourselves... The source git repository for the project can be found here.

In any case, one only needs to add the line found on listing 10 to /etc/apt/sources.list. This can be done either automatically (this is what's shown on the aforementioned listing) or through a text editor such as vim or nano. We would like to point out, as a curiosity, why we haven't included a repository starting with deb-src. These are only needed if one wants to explicitly download the sources for a given package instead of the compiled binary version. We won't be needing it, so we didn't add it on our sources.list.

```
echo 'deb http://deb.debian.org/debian buster-backports main' | sudo tee -a /etc/apt/sources.
    list
```

Listing 10: The backports repository

6.2.2 Installing the packages

Once the backports are added to the system one can just run the command found on listing 11 to install the required packages. Note the wireguard package is a *metapackage* in

the sense that it contains nothing. It just depends on the `wireguard-tools` and either the `wireguard-dkms` or `wireguard-modules` packages so that they are both installed. This is quite a common practice with package managers.

```
apt update && apt install wireguard
```

Listing 11: Installing `WireGuard`

It's also curious to note how we will install either the `wireguard-dkms` or `wireguard-modules` package. As `WireGuard` runs as a kernel module (i.e. it doesn't live in the user space) it needs to install itself in a particular way in order to run. Starting with the `5.x.x Linux` kernels, the `WireGuard` module is installed in a certain way, whilst with previous versions one needs to leverage the *DKMS* "system" to circumvent some kernel limitations. Even though this doesn't really affect the user, it's a nice thing to be aware of.

6.2.3 Installing `WireGuard` on certain kernels

We had to install `WireGuard` on a machine running the `4.19.0-6-amd64 Linux` kernel. There's an error related to a security vulnerability as described here that was solved by changing the name of a struct member. Whilst this doesn't affect the `4.19.0-13-amd64` or `4.19.0-16-amd64` kernels (which we also employ at the observatory), it did prevent the `WireGuard` DKMS module from compiling correctly on the aforementioned release.

In order to fix the issue we had to manually edit the `/usr/src/wireguard-1.0.20210219/compat/compat.h` file to fix the naming of a `struct member`. We altered line 94 by removing the `!` logical operator preceding a `defined(ISDEBIAN)` check from the line shown on listing 12. A correct version is included in listing 13. The change will force the line on listing 14 to be preprocessed, thus solving the issue. We will then have to run `dpkg -configure -a` to rebuild the package and the installation should now be successful.

```
#elif (LINUX_VERSION_CODE < KERNEL_VERSION(5, 4, 5) && LINUX_VERSION_CODE >= KERNEL_VERSION(5,
     4, 0)) || (LINUX_VERSION_CODE < KERNEL_VERSION(5, 3, 18) && LINUX_VERSION_CODE >=
    KERNEL_VERSION(4, 20, 0) && !defined(ISUBUNTU1904)) || (!defined(ISRHEL8) && !defined(
    ISDEBIAN) && !defined(ISUBUNTU1804) && LINUX_VERSION_CODE < KERNEL_VERSION(4, 19, 119) &&
    LINUX_VERSION_CODE >= KERNEL_VERSION(4, 15, 0)) || (LINUX_VERSION_CODE < KERNEL_VERSION(4,
     14, 181) && LINUX_VERSION_CODE >= KERNEL_VERSION(4, 10, 0)) || (LINUX_VERSION_CODE <
    KERNEL_VERSION(4, 9, 224) && LINUX_VERSION_CODE >= KERNEL_VERSION(4, 5, 0)) || (
    LINUX_VERSION_CODE < KERNEL_VERSION(4, 4, 224) && !defined(ISUBUNTU1604) && !defined(
    ISRHEL7))
```

Listing 12: Wrong line from `WireGuard`'s source code.

```
#elif (LINUX_VERSION_CODE < KERNEL_VERSION(5, 4, 5) && LINUX_VERSION_CODE >= KERNEL_VERSION(5,
     4, 0)) || (LINUX_VERSION_CODE < KERNEL_VERSION(5, 3, 18) && LINUX_VERSION_CODE >=
    KERNEL_VERSION(4, 20, 0) && !defined(ISUBUNTU1904)) || (!defined(ISRHEL8) && defined(
    ISDEBIAN) && !defined(ISUBUNTU1804) && LINUX_VERSION_CODE < KERNEL_VERSION(4, 19, 119) &&
    LINUX_VERSION_CODE >= KERNEL_VERSION(4, 15, 0)) || (LINUX_VERSION_CODE < KERNEL_VERSION(4,
     14, 181) && LINUX_VERSION_CODE >= KERNEL_VERSION(4, 10, 0)) || (LINUX_VERSION_CODE <
    KERNEL_VERSION(4, 9, 224) && LINUX_VERSION_CODE >= KERNEL_VERSION(4, 5, 0)) || (
    LINUX_VERSION_CODE < KERNEL_VERSION(4, 4, 224) && !defined(ISUBUNTU1604) && !defined(
    ISRHEL7)
```

Listing 13: Correct line from `WireGuard`'s source code.

```
#define ipv6_dst_lookup_flow(a, b, c, d) ipv6_dst_lookup(a, b, &dst, c) + (void *)0 ?: dst
```

Listing 14: Line defining the `ipv6_dst_lookup_flow` memeber.

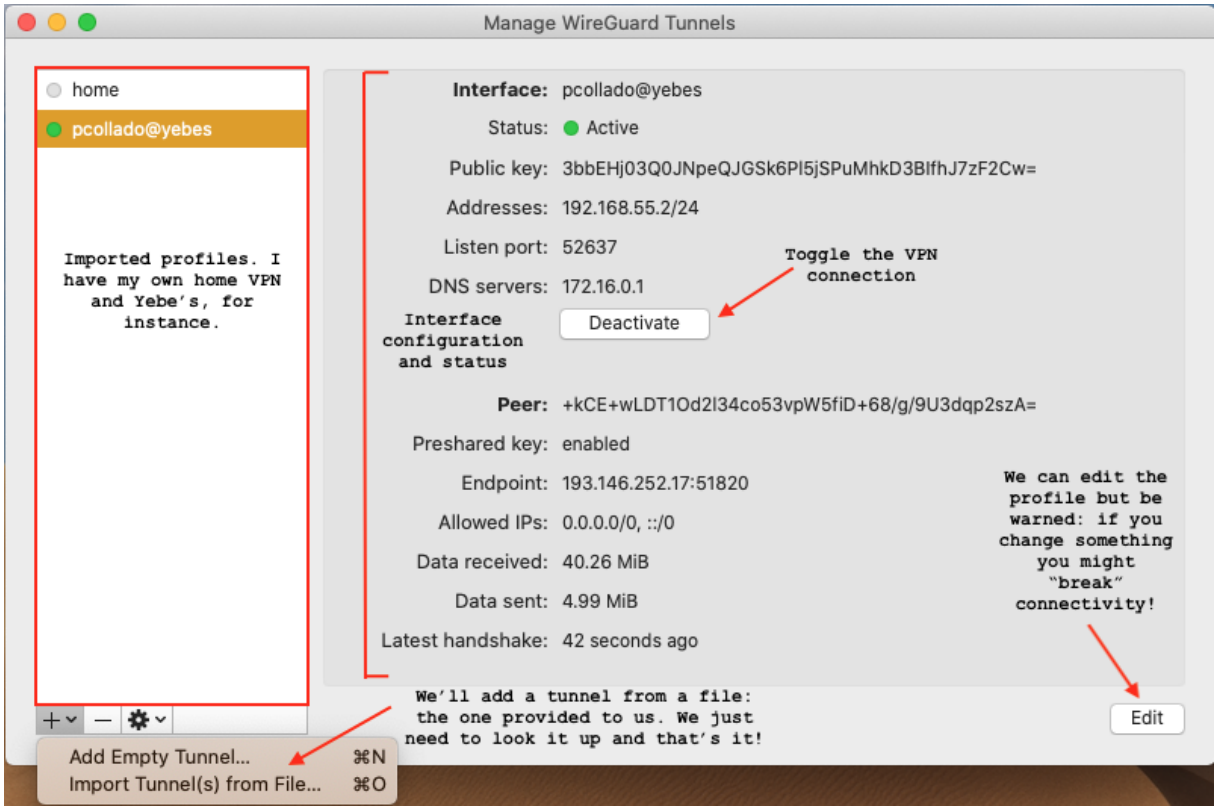

Figure 2: The `WireGuard GUI`.

6.3 Installing the `WireGuard` client

A `VPN` client only needs to install the `WireGuard Client` and import the `VPN certificate` that's provided to him or her. Both `Windows` and `macOS` have graphical clients available. When it comes to `Linux`-based distributions we can make use of the `wg-quick` utility. We'll get into GUI options for `linux` in due time. As a bonus, it's nice to know we have clients available for the two largest mobile operating systems (namely `Android` and `iOS`).

6.3.1 `Windows` systems

We can download the client installer from here. Once the installer finishes we can launch the application (which will probably require administrative privileges based on the user's computer configuration) and we'll be presented with a window similar to the one found on figure 2. In order to get up and running one just needs to add the provided profile by clicking on the + icon on the bottom-left of the window. After that one can activate the connection by clicking on the appropriate button.

6.3.2 macOS systems

The procedure in this case is very simple too. We just need to install the WireGuard client from the App Store and proceed like a Windows user would. Figure 2 was actually taken on a machine running macOS, so it can be used as starting point regarding the use of the client.

6.3.3 Linux-based systems

Given WireGuard's peer-to-peer architecture, managing WireGuard is practically the same no matter if you are running it as a "server" or as a "client". A user need only store the provided certificate (usually a file with a *.conf suffix) on /etc/wireguard. This will likely require elevated privileges that can be obtained as described above. Once that's one, a user can use the wg-quick up <profile_name> command to activate the VPN and wg-quick down <profile_name> to tear it down. Note that the parameter to pass to wg-quick is determined by the profile's name. If a user is given the foobar.conf profile the parameter should then be foobar. In any case, one can invoke wg from a shell to query the WireGuard interface's status at any time.

One can also leverage a systemd service with a name following wg-quick@<profile_name>, where <profile_name> would be foobar for the example above. One could then run sudo systemctl start wg-quick@foobar and sudo systemctl stop wg-quick@foobar to perform the same as the above. The only "advantage" over the use of wg-quick directly is that the output of systemctl status wg-quick@foobar shows whether the interface is up and running or not.

Given WireGuard runs as a kernel module, it's deeply intertwined with the standard utilities. We **have not tried this ourselves**, but one might try to use the builtin graphical connection manager provided on distros such as Debian to manage the VPN connection. We personally believe it's a bit pointless given the GUI won't provide any more information than the wg command, and depending on a GUI might prove to be cumbersome if it keeps changing with time or if it tends to crash: nothing is more stable that a regular shell after all... In any case we do encourage users going down this road to be sure of what they are doing as they might unintentionally "fight" WireGuard's attempts to configure the client in a particular way.

7 Filtering the incoming traffic: iptables

Once we are receiving traffic through the VPN it's time to limit what processes we grant access to. We mustn't forget we can address any process in the local network with an IP:PORT tuple. We can accomplish our goal with the help of the very well known iptables which is an administration tool letting us filter traffic alongside some other nifty functions. Whenever we provide access to external users through the WireGuard server we'll also allocate a particular IP address. We can then filter incoming traffic based on this IP so that we can indeed control what an external user can and can't do.

We can install a rule restricting access to a single machine with IP 172.16.0.10 for a client with allocated IP 192.168.55.100 by running the rule found on listing 15. This rule can be read as "**drop** any packets coming in through the wg0 interface whose source

IP is 192.168.55.100 and the destination is **not** 172.16.0.10". We are assuming the VPN server is implemented on the wg0 WireGuard interface and that the default policy is to ACCEPT packets that are to be forwarded (this should be the case on a machine running a VPN server). We can also restrict the ports the clients can reach but, given the existing requirements, this might not be necessary. Were this to become compulsory, one could leverage the -p and -destination-ports iptables options to make the desired behaviour effective. Now, this rule is to be applied to the FORWARD *chain*. Packets traversing a machine will be processed according to a rule belonging to one of three chains. The FORWARD chain is the one concerned with packets *traversing* the VPN server, so it's the one we have to apply the rule on. Iptables is a rather complex tool and we haven't even scratched the surface (we actually have several *tables*, each with several *chains*). We encourage curious readers to read through iptable's *manpage* which is accessible via the man iptables command on linux-based systems or through this website.

```
# Note this command will require elevated privileges
iptables -A FORWARD -i wg0 -s 192.168.55.100 ! -d 172.16.0.10 -j DROP
```

Listing 15: Iptables rules limiting traffic from select clients.

8 The user's perspective: using the system

The above discussion can be a tiny bit daunting. Our background is rooted in the world of networking where we are used to working with a lot of abstractions. We tend to look at things from the "real" and "logical" domains when it comes to a system's structure and functionality. Even though this can seem far fetched, it is exactly the same way of thinking employed by RF engineers who "study" signals in the "time" and "frequency" domains. We do understand network engineers are not as common, so it can be harsh to see things the way we do.

In an effort to make leveraging this system easier, we have prepared this section to walk new users through the steps needed to externally command the 40m RT. The provided explanations are intentionally shallow: the intricacies behind it all are detailed in their respective sections so we do encourage the reader to at least devote some time to them, either before or after reading this. One of the advantages of this section is that it very accurately portrays how an end user need not be concerned with many of previous sections. Even though it is a good practice to have a grasp of how the overall system works, it is true that the end user is not required to know most of the implementation aspects we have described throughout the report.

8.1 Yebes' end

Given this system allows externally controlling the antenna it **will not** be running in a continuous fashion. We intend to bring the control server up only when it's needed. That's why before attempting to command the antenna or querying its position one needs to make sure the server he or she has to talk to is effectively up. The Observatory's staff will also need to provide the IP address and port where the antenna control server can be contacted. They will also have to initiate the ARIES shell that's required for connections to ACS components to work. This subtlety is nonetheless transparent to the end user.

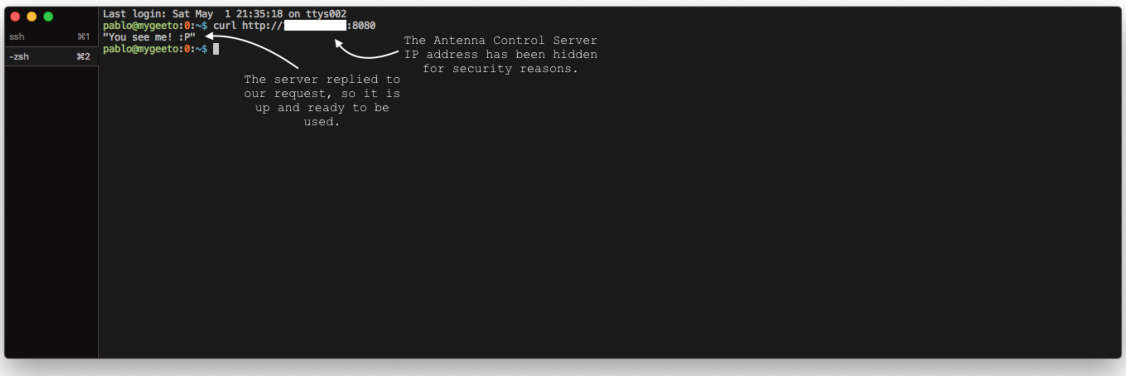

Figure 3: Checking to see whether the server is up and running with `curl`.

Figure 4: Checking to see whether the server is up and running with a browser.

As discussed on section 6, one needs to connect to the Yebes Astronomical Observatory through a `WireGuard VPN`. In order to to so, the end user needs to be provided a *certificate* by the Observatory's personnel.

8.2 Setting up the user's end

At this point the end user needs to:

1. Be sure the antenna control server is up and running.
2. Know the `IP` and `port` where the antenna control server is listening.
3. Have an active `VPN` connection to the Yebes Astronomical Observatory. This implies:
   (a) Having a `VPN` certificate in his or her possession.
   (b) Having a `WireGuard` client installed as explained on section 6.3.

The end user should now be capable of making sure everything is set up correctly by running the command shown on listing 16. We have also included a screenshot of what an end user should see on figure 3. This test can also be carried out through a common browser as seen on figure 4: we just need to navigate to `http://<antenna_control_server_IP>:<antenna_control_server_port>`.

```
1 curl http://<antenna_control_server_IP>:<antenna_control_server_port>
```

Listing 16: Checking we can communicate with the control server.

Figure 5: Simulating the querying of antenna positions.

What both `curl` and the browser are doing is sending an `HTTP` request to the `flask` server's *root* (`/`) endpoint. This endpoint's associated function will just reply with the `You see me! :P` message which we can see on both approaches.

8.3 Using the system

Once the user has verified he or she can indeed "see" the antenna control server they are ready to launch whichever client they desire. They can either use the ones we have provided on section 5 or come up with their own. As long as they respect the format the `HTTP` payloads must adhere to they can work in the way they desire. This is one of the "beauties" of `HTTP` interfaces: they impose no restrictions on the client's implementation.

As an example we are adding figure 5, which leverages the `/get_position_test` endpoint to show what a client querying said endpoint would have to work with. This figure contains a terminal window with two different sessions. One of them is using the client querying the positions we have written in `python` whilst the other is just using `curl` to the same effect. The main advantage of the `python` client is that it can programmatically process the data and respond to the values it gets while the `curl` counterpart will just display them to the user. Just like before, one can open the same `URL` on a browser and he or she will get the antenna's position on screen in a way similar to what we saw on figure 4. We do believe that this approach is rather useless beyond its use for demonstrative purposes.

Like we stated before, the user is free to use any of the provided endpoints. The clients we have included in section 5 have been proven to work and can be used by anybody wanting to do so.

9 Closing thoughts

This report covers a lot of aspects regarding the system we have devised for providing external users the possibility of remotely controlling the antenna in a secure fashion even over public networks. We hope to have been clear enough so that all the intricacies and subtleties have been explained before doubt. Thanks you for your attention.

References

[1] P. de Vicente, R. Bolaño, and L. Barbas, "Primera prueba con el sistema de control del radiotelescopio de 40m. Observaciones," *CDT Technical Report 2007-8*, 2007. Report available here.