DISCOS Simulators Documentation

Release 1.0

Giuseppe Carboni, Marco Buttu

Mar 17, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 About this project

***DISCOS Simulators*: a framework for writing hardware simulators for the *DISCOS* control software**

Giuseppe Carboni <giuseppe.carboni@inaf.it>

This document describes the ***DISCOS Simulators*** framework. This framework was designed with the purpose of providing a means to integrate several hardware simulators of the three Italian radio telescopes (especially the *Sardinia Radio Telescope*) under the same environment.

Writing a simulator helps the developers in writing good code for the actual control software of the radio telescope, the *DISCOS* control software. Being able to test the control software code without having to rely on the hardware represents a huge advantage in the development and maintenance processes, it provides a way to test each new addition or modification to the code, making sure that the control software keeps behaving as expected. Furthermore, the framework allows to test how the control software code reacts under expected error conditions, in fact, it provides an easy way to simulate unlikely scenarios that are very difficult or, in some cases, impossible to replicate by only using the hardware. This allows the developers to write more reliable and robust code, that is likely capable to allow the user to recover from an error condition without having to resort to a complete reboot of the system.

Having a fast way to write a simulator of a new incoming device, also allows to easily verify that the communication protocol on which the software developing team and the provider of the new device agreed upon, is working as expected. In case some of the tests yield different results between the simulator and the real hardware, it is easier to understand which one of the two parties committed an implementation error, whether it is a bug in the code of the control software or in the firmware of the device (or just in an update of one of the two).

In order to do so, it is necessary to have a simulator for each critical component of the radio telescope. Since the vast majority of the communications between the control software and the devices (or the simulator, in this case) are carried on via network, under similar circumstances, having a framework already capable of handling these kind of communications on its own, is priceless. It allows the developers to focus on the simulator code and communication protocol, without having to re-write anything related to the communication infrastructure, providing the same simple architecture for all the simulators.

Another important aim of this project is that it opens the possibility of performing automated tests of the control software code. A suite of simulators, capable of reproducing various different scenarios, can be exploited to write and execute a great variety of tests whenever a modification to the control software code gets pushed to the main online repository. This workflow is called *continuous integration*. [1]

Chapter two of this document describes the framework in detail, its structure, its layers and the classes that compose it. *Chapter three* explains how to install the package and run the simulators. *Chapter four* goes into detail in describing how to write a new simulator, test it, and integrate it into the framework. In *chapter five*, the reader will find the documentation of the *utils* library, which contains several useful functions to easily perform some recurrent tasks such as format conversions.

1.2 The *DISCOS* Control Software

The following paragraph describes the *DISCOS* control software. These lines were taken from the official *DISCOS* documentation. [2]

*"**DISCOS** (**D**evelopment of the **I**talian **S**ingle-dish **CO**ntrol **S**ystem) is the control software produced for the Italian radio telescopes. It is a distributed system based on **ACS** (**A**LMA **C**ommon **S**oftware) [3] commanding all the devices of the telescope and allowing the user to perform single-dish observations in the most common modes. As of today, the code specifically implemented for the telescopes (i.e. excluding the huge ACS framework) amounts to about 650000 lines. Even VLBI (or guest-backend) observations partly rely on DISCOS, as it must be used to perform the focus selection and the frontend setup."*

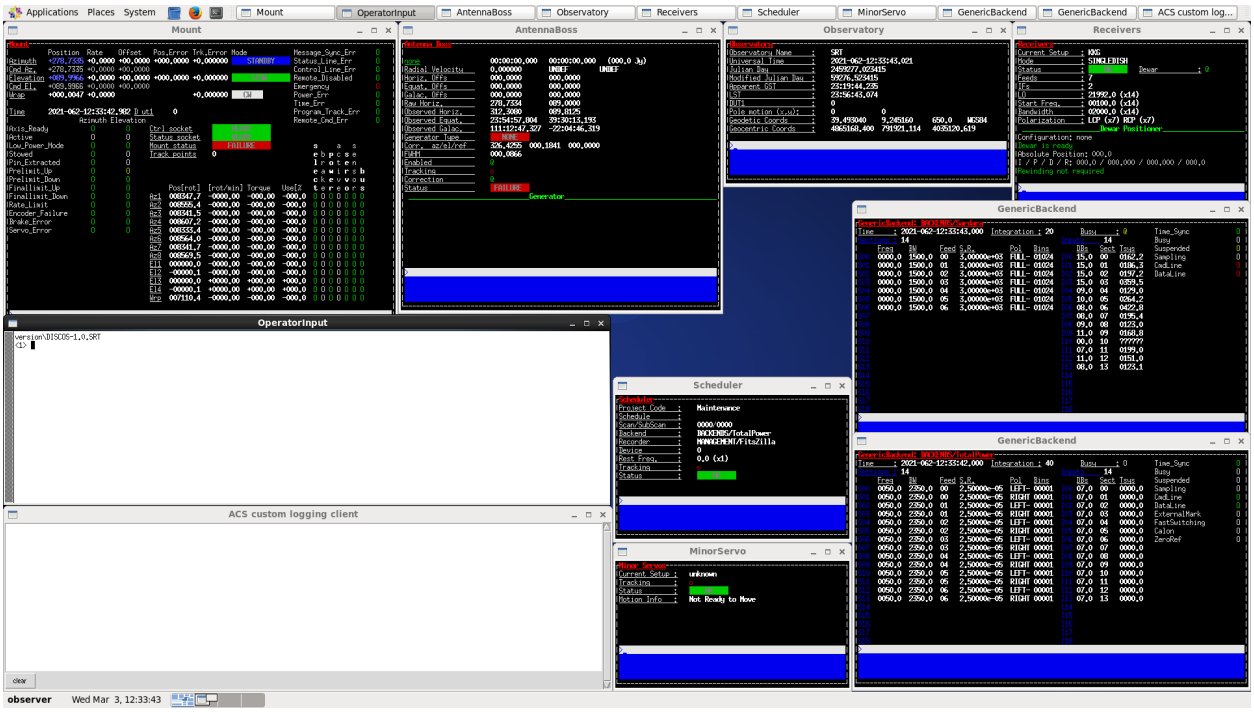

Fig. 1.1: The *DISCOS* console user interface.

CHAPTER

TWO

THE FRAMEWORK

The framework is written in Python 2.7. This *Python* version was chosen to maintain compatibility with the *ACS* default *Python* version. By matching the two versions, all the simulators can be executed on the same machine where the *DISCOS* control software is running, no matter if it is a production, a development, a physical or a virtual machine.

The framework architecture is composed of two layers. The topmost layer is in charge of handling network communications, it behaves as a server, listening for incoming connections from clients and relaying every received byte to the other layer of the framework, the simulation layer. This layer is where received commands are parsed and executed, it can faithfully replicate the behavior of the hardware it is supposed to simulate, or it can simply answer back to clients with a simple response, whether the latter is the correct one or not, to simulate an error condition and thoroughly test the *DISCOS* control software.

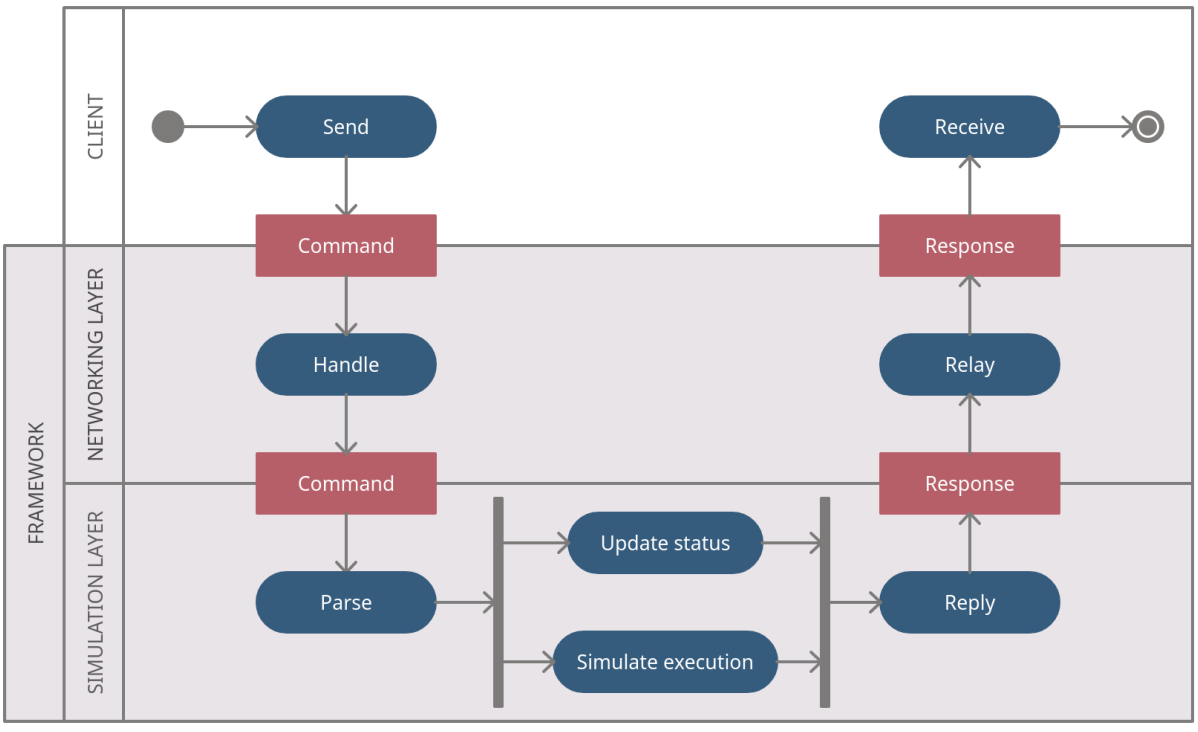

Fig. 2.1: Framework layers and communication behavior.

2.1 Networking layer

2.1.1 The *Simulator* class

The *Simulator* class represents the topmost layer of the framework. Each simulator system can have multiple instances or servers to be launched (i.e. the Active Surface simulator has 96 different instances), this class has the task to start a server process for each of the instances needed for the given simulator.

class simulators.server.**Simulator**(*system_module*, ***kwargs*)

This class represents the whole simulator, composed of one or more servers.

Parameters **system_module** (*module that implements the System class, string*) – the module that implements the System class.

start(*daemon=False*)

Starts a simulator by instancing the servers listed in the given module.

Parameters **daemon** (*bool*) – if true, the server processes are created as daemons, meaning that when this simulator object is destroyed, they get destroyed as well. Default value is false, meaning that the server processes will continue to run even if the simulator object gets destroyed. To stop these processes, method *stop* must be called.

stop()

Stops a simulator by sending the custom *$system_stop%* command to all servers of the given simulator.

2.1.2 The *Server* class

As previously mentioned, every simulator exposes one or multiple server instances via network connection. Each server instance is an object of the *Server* class, its purpose it to listen on a given port for incoming communications from any client. This class inherits from the *SocketServer.ThreadingMixIn* class and, depending on which type of socket it will use, it also inherits either from the *SocketServer.ThreadingTCPServer* class or from the *SocketServer.ThreadingUDPServer* class. The reader can find more information about the *system* parameter in the *The System class* section.

class simulators.server.**Server**(*system*, *server_type*, *kwargs*, *l_address=None*, *s_address=None*)

This class inherits from the ThreadingMixIn class. It can instance a server for the given address(es). The server can either be a TCP or a UDP server, depending on which *server_type* argument is provided. Also, the server could be a listening server, if param *l_address* is provided, or a sending server, if param *s_address* is provided. If both addresses are provided, the server acts as both as a listening server and a sending server. Be aware that if the server both listens and sends to its clients, *l_address* and *s_address* must not share the same endpoint (IP address and/or port should be different).

Parameters

- **system** (*System class that inherits from ListeningServer or/ and SendingServer*) – the desired simulator system module
- **server_type** (*ThreadingTCPServer or ThreadingUDPServer*) – the type of threading server to be used
- **kwargs** (*dict*) – the arguments to pass to the system instance constructor method
- **l_address** ((*ip, port*)) – the address of the server that exposes the *System.parse()* method
- **s_address** ((*ip, port*)) – the address of the server that exposes the *System.subscribe()* and *System.unsubscribe()* methods

serve_forever(*poll_interval=0.05*)
Overrides the base class *serve_forever* method. Before calling the base method, which would stay in a loop until the process is stopped, it starts the eventual child server as a daemon thread.

Parameters **poll_interval** (*float*) – the interval used by the class to check for incoming shutdown requests

start()
Starts a daemon thread which calls the *serve_forever* method. The server is therefore started as a daemon.

stop()
Stops the server and its eventual child.

The *Server* class is capable of behaving in three different ways. The first one is to act as a listening server, a server that awaits for incoming commands and sends back the answers to the client. The second possible behavior is to act as a sending server, sending its status message periodically to its clients after they start any communication. Finally, a server can act as a combined version of the aforementioned two. It is therefore able to behave as both listening and sending server on two different ports but relaying incoming commands to and sending the status of a single simulator object to its connected clients. Depending on how a server is configured to behave, it will use different handler classes to manage communications with its clients.

2.1.3 Handler classes of a *Server*

The handler class of a server object, is the endpoint class, in charge of handling any communication between the server object and its underlying simulator. Depending on the type of handler a server object has underneath, its behavior when receiving a message will be different.

The *BaseHandler* class

The main handler class in the framework is the *BaseHandler* class. It inherits from the *BaseRequestHandler* class.

class simulators.server.**BaseHandler**(*request*, *client_address*, *server*)
This is the base handler class from which *ListenHandler* and *SendHandler* classes are inherited. It only defines the custom header and tail for accepting some commands not related to the system protocol, and the *_execute_custom_command* method to parse the received custom command.

Example A $system_stop% command will stop the server, a $error% command will configure the system in order to respond with errors, etc.

custom_header = '$'

custom_tail = '%'

_execute_custom_command(*msg_body*)
This method accepts a custom command (without the custom header and tail) formatted as *command_name:par1,par2,...,parN*. It then parses the command and its parameters and tries to call the system's equivalent method, also handling unexpected exceptions.

Parameters **msg_body** (*string*) – the custom command message without the custom header and tail ($ and % respectively)

setup()
Method that gets called whenever a client connects to the server.

The *BaseHandler* class alone is not able to handle any incoming message, its only purpose in fact is to act as a base class for the following two classes, providing its children classes the *_execute_custom_command* method.

The *ListenHandler* class

The *ListenHandler* class, as its name suggests, listens for incoming messages from any client, and relays these messages to the underlying simulator. If the simulator answers with a response message, the message is then relayed back to the client.

class simulators.server.**ListenHandler**(*request*, *client_address*, *server*)

handle()
Method that gets called right after the *setup* method ends its execution. It handles incoming messages, whether they are received via a TCP or a UDP socket. It passes down the *System* class the received messages one byte at a time in order for the *System.parse()* method to work properly. It then returns the *System* response when received from the *System* class. It also constantly listens for custom commands that do not belong to a specific *System* class, but are useful additions to the framework with the purpose of reproducing a specific scenario (i.e. some error condition).

The *SendHandler* class

The *SendHandler* class, as soon as a client opens a communication channel, starts retrieving and sending periodically the status message of its underlying simulator class to its connected clients.

class simulators.server.**SendHandler**(*request*, *client_address*, *server*)

handle()
Method that gets called right after the *setup* method ends its execution. It handles messages that the server has to periodically send to its connected client(s). It also constantly listens for custom commands that do not belong to a specific *System* class, but are useful additions to the framework with the purpose of reproducing a specific scenario (i.e. some error condition).

2.2 Simulation layer

2.2.1 The *System* class

The *System* class is the main class of any hardware simulator. It is the class in charge of parsing any incoming command received from the *Handler* object of the *Server*, and/or periodically provide the status of the simulator it is supposed to be mimicking.

The *BaseSystem* class

The *BaseSystem* class is simply bare implementation of a full *System* class. This class is the right place where to implement any custom method that can be helpful to handle some behavior that is common to all simulators. As it can be seen from the API below, it is the case of *system_greet()* and *system_stop()* methods, which have to be defined for every simulator. They can be overridden in case a *System* object has to behave differently than the default implementation.

class simulators.common.**BaseSystem**

System class from which every other *System* class is inherited. If a custom command that can be useful for every kind of simulator has to be implemented, this class is the right place.

static system_greet()

Override this method to define a greeting message to send to the clients as soon as they connect.

Returns the greeting message to sent to connected clients.

static system_stop()

Sends back to the server the message *$server_shutdown%* ordering it to stop accepting requests, to close its socket and to shut down.

Returns a message telling the server to proceed with its shutdown.

In order for a system object to be able to either parse commands or send its status to any connected client, writing a *System* class that inherits from *BaseSystem* is not enough. The *System* class of a simulator in fact has to inherit from one (or both) of the two classes described below. It the *System* class inherits from both the classes, it will have to implement all the required methods and define the required attributes.

The *ListeningSystem* class and the *parse* method

In order for any *System* class to be able to parse any command received by the server, a *parse* method has to be defined. This method takes one byte (string of one character, in Python 2.7) as argument and returns:

- *False* when the byte is not recognized as a message header and the system is still waiting for the correct header character
- *True* when the system has already received the header and it is waiting to receive the rest of the message
- a response to the client, a non empty string, built according to the protocol definition. The syntax of the response thus is different between different simulators.

If the system has nothing to send to the client, as in the case of broadcast requests, *System.parse()* must return *True*. When the simulator is brought to behave unexpectedly, a *ValueError* has to be raised, it will be captured and logged by the parent server process.

class simulators.common.**ListeningSystem**

Implements a server that waits for its client(s) to send a command, it can then answer back when required.

parse(*byte*)

Receives and parses the command to be sent to the System. Additional information here: https://github.com/discos/simulators/issues/1

Parameters **byte** (*byte*) – the received message byte.

Returns False when the given byte is not the header, but the header is expected. True when the given byte is the header or a following expected byte. The response (the string to be sent back to the client) when the message is completed.

Return type boolean, string

Raises **ValueError** – when the declared length of the message exceeds the maximum expected length, when the sent message carries a wrong checksum or when the client asks to execute an unknown command.

The *SendingSystem* class and the *subscribe* and *unsubscribe* methods

If the *System* class inherits from *common.SendingSystem*, it has to define and implement the *System.subscribe()* and *System.unsubscribe()* methods, along with the *sampling_time* attribute.

Both the *System.subscribe()* and *System.unsubscribe()* methods interfaces are described in issue #175 on GitHub.

The *subscribe* method takes a queue object as argument and adds it to the list of the connected clients. For each client in this list the system will then be able to send the required message by putting it into each of the clients queues.

The *unsubscribe* method receives once again the same queue object received by the *subscribe* method, letting the system know that that queue object, relative to a disconnecting client, has to be removed from the clients queues.

The *sampling_time* attribute defines the time period (in milliseconds) that elapses between two consecutive messages that the system have to send to its clients. It is internally defined in the *SendingSystem* base class, and its default value is equal to 10ms. If a different sampling time is needed, it is sufficient to override this variable in the inheriting *System* class.

class simulators.common.**SendingSystem**

Implements a server that periodically sends some information data regarding the status of the system to every connected client. The time period is the one defined as *sampling_time* variable, which defaults to 10ms and can be overridden. The class also accepts simulator-related custom commands, but no regular commands are accepted (they are ignored and immediately discarded).

```
sampling_time = 0.01
```

subscribe(*q*)

Passes a queue object to the System instance in order for it to add it to its clients list. The System will therefore put any new status message into this queue, along with the queue of other clients, as soon as the status message is updated. Additional information here: https://github.com/discos/simulators/issues/175

Parameters **q** (*Queue*) – the queue object in which the System will put the last status message to be sent to the client.

unsubscribe(*q*)

Passes a queue object to the System instance in order for it to be removed from the clients list. The System will therefore release the handle to the queue object in order for the garbage collector to destroy it when the client has finally disconnected. Additional information here: https://github.com/discos/simulators/issues/175

Parameters **q** (*Queue*) – the queue object that contains the last status message to send to the connected client.

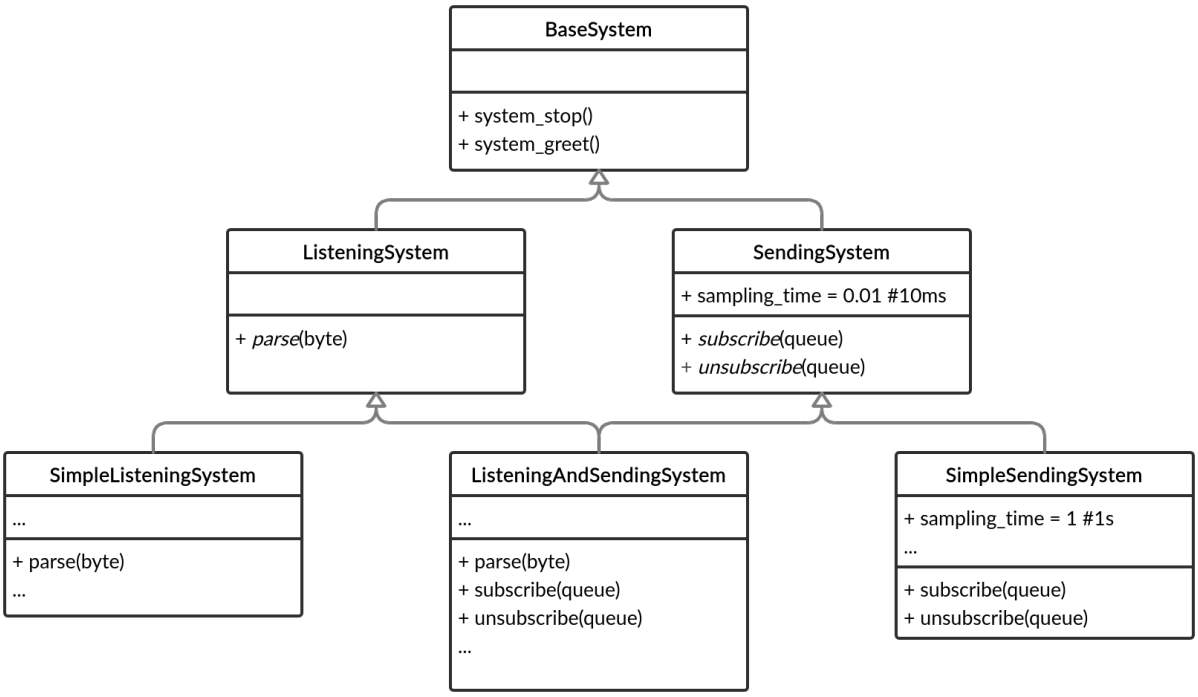

Fig. 2.2: System class inheritance.

2.2.2 The *MultiTypeSystem* class

Some simulators might have multiple different implementations, having therefore multiple *System* classes that behave differently from one another. For example, among the already developed simulators, there are two different implementations of the *IF Distributor* simulator. In order to keep multiple different *System* classes under the same simulator name, another class called *MultiTypeSystem* was written, it acts as a *class factory*. This means that it works by receiving the name of the configuration of the system we want to launch as *system_type* keyword argument.

class simulators.common.**MultiTypeSystem**

This class acts as a 'class factory', it means that given the attributes *system_type* and *systems* (that must be defined in child classes), creating an instance of *MultiTypeSystem* (or some other class that inherits from this one) will actually create an object of *system_type* type if it's defined in the *systems* list. This class is meant to be used in systems that have multiple simulator types or configuration in order for the user to be able to choose the desired type when launching the simulator.

static __new__(*cls*, ***kwargs*)

Checks if the desired configuration is available and returns its correspondent class type.

Returns the System class correspoding to the one selected via command line interface, or the default one.

The main *System* class, just like a regular *System* class, should be defined in the *__init__.py* file, inside the module main directory. It must inherit from the *MultiTypeSystem* class and override the *__new__* method as shown below:

```
systems = get_multitype_systems(__file__)

class System(MultiTypeSystem):

```

(continues on next page)

(continued from previous page)

```
    def __new__(cls, **kwargs):
        cls.systems = systems
        cls.system_type = kwargs.pop('system_type')
        return MultiTypeSystem.__new__(cls, **kwargs)
```

As you can see from the code above, before defining the class, it is necessary to retrieve the list of the available configurations for the given simulator. This can be done by calling the *get_multitype_systems* function, defined in the *The simulators.utils library* library. The said function will recursively search for any *System* class in the given path. Generally speaking, the passed *__file__* value will ensure that only the *System* classes defined in the module's directory and sub-directories will end up inside the *systems* list. For more information, take a look at the *function* in the *The simulators.utils library* section. The default system configuration can be defined as *system_type* inside the *kwargs* dictionary.

To know how to launch a simulator of this kind, please, take a look at *this paragraph*.

CHAPTER

THREE

USER GUIDE

3.1 Package setup

The package installation, along with its requirements, requires no longer than a few minutes. First of all, after downloading the package, it is necessary to run the following command to install its Python dependencies:

```
$ pip install -r requirements.txt
```

This command will automatically install all the required dependencies. Currently, the only required Python packages are Numpy and Scipy.

After the requirements have been installed, it is time to install the package itself. In order to do so, execute the following command inside the repository main directory:

```
$ python setup.py install
```

This command will install all the simulator libraries and scripts into the current Python environment, and they will be immediately available for execution.

3.2 Run the simulators

All the simulators can be run at once, by simply executing the following command:

```
$ discos-simulator start
```

To stop all the simulators at once:

```
$ discos-simulator stop
```

By adding the `--system` or `--s` flag to the command, it is possible to run a single simulator:

```
$ discos-simulator start --system active_surface
$ discos-simulator start -s acu
```

To stop the desired simulator:

```
$ discos-simulator stop -s active_surface
```

To run a specific configuration for a simulators, add the `--type` flag, followed by the desired configuration:

```
$ discos-simulator --system if_distributor --type IFD start
```

Not all simulators have multiple configurations. Providing an unknown configuration will prevent the system from starting and the command will fail.

To know the currently available simulators, execute the command using the the `list` action:

```
$ discos-simulator list
Available simulators: 'active_surface', 'acu', 'backend', 'calmux', 'if_distributor',
↪'lo', 'mscu', 'weather_station'.
```

CHAPTER

FOUR

DEVELOPERS DOCUMENTATION

4.1 How to implement a simulator

To implement a simulator, it is necessary to create a module that defines both a *servers* list and a *System* class.

4.1.1 The *servers* list

The *servers* list defines all the instances that should be running when the given simulator starts. Each element of the *servers* list is a tuple, composed of the following items:

- the server listening address, *l_address*
- the server sending address, *s_address*
- the type of threading server from the *SocketServer* package to use to run the simulator
- a dictionary of optional keyword arguments, *kwargs*, eventually used by *System.__init__()*

The *l_address* item is the address on which the server will listen for incoming commands to pass down to the *System.parse()* method. The *s_address* item is the address from which the server will periodically send its data to its connected clients. The type of threading server from the *SocketServer* argument can either be *ThreadingTCPServer* or *ThreadingUDPServer*, depending on the type of socket the server has to use. These Python object types have to be imported as follows:

```
from SocketServer import ThreadingTCPServer
```

or:

```
from SocketServer import ThreadingUDPServer
```

Some examples

Suppose the reader wants to simulate a system that has 2 listening TCP servers and no sending servers, the first one with address `('192.168.100.10', 5000)` and the second one with address `('192.168.100.10', 5001)`. In this case we have to define the *servers* list as follows:

```
servers = [
    (('192.168.100.10', 5000), (), ThreadingTCPServer, {}),
    (('192.168.100.10', 5001), (), ThreadingTCPServer, {}),
]
```

If the *System* class accepts some extra arguments, two integers, for instance, it is possible to pass them via the *kwargs* dictionary:

```
servers = [
    (('192.168.100.10', 5000), (), ThreadingTCPServer, {'arg1': 10, 'arg2': 20}),
    (('192.168.100.10', 5001), (), ThreadingTCPServer, {'arg1': 4, 'arg2': 5}),
]
```

If the *System* to simulate has instead a single listening UDP server, the *servers* list will be defined as follows:

```
servers = [
    (('192.168.100.10', 5000), (), ThreadingUDPServer, {}),
]
```

A *System* with 3 sending TCP servers and no listening servers will have the *servers* list defined in the following way:

```
servers = [
    ((), ('192.168.100.10', 5002), ThreadingTCPServer, {}),
    ((), ('192.168.100.10', 5003), ThreadingTCPServer, {}),
    ((), ('192.168.100.11', 5000), ThreadingTCPServer, {}),
]
```

Finally, a system instance can act as both listening and sending server. In this case, each server list entry must be defined as follows:

```
servers = [
    (('192.168.100.10', 5003), ('192.168.100.10', 5004), ThreadingTCPServer, {}),
    (('192.168.100.10', 6000), ('192.168.100.10', 6001), ThreadingTCPServer, {}),
]
```

Be aware that multiple lines in the *servers* list will cause the simulator to spawn a *System* object per line. Every one of the spawned *System* objects is independent from the others and they will all act as different simulators.

4.1.2 Custom commands

Custom commands are useful for several use cases. For instance, suppose we want the simulator to reproduce some error conditions by changing the *System* state. We just need to define a method that starts with *system_* inside the *System* class. I.e:

```
class System(ListeningSystem):

    def system_generate_error_x(self):
        # Change the state of the System
        ...
```

After implementing this method, the clients are able to call it by sending the custom command *$system_generate_error_x%*. It is also possible to define a method that accepts some parameters. In this case the custom command will have the form *$system_commandname:par1,par2,par3%*. Since every *Server* object is not limited to only a single connection, custom commands can be also sent by a different client that the main one. This allow the reproduction of error scenarios even when the *DISCOS* control software is already connected to some simulator.

In order to avoid name clashing for custom methods, it is sufficient to not use the *system_* prefix for any other *System* method, so make sure to only use this convention for custom commands.

4.1.3 Useful functions

In order to make it faster to write and implement new simulator's methods, which sometimes require converting data from a format to another, a library of useful functions called *simulators.utils* has been written and comes within the

simulators package. Its API is described in the *The simulators.utils library* section.

4.2 Testing environment

In the *continuous integration* workflow, the tests are executed more than once. During the development process, tests will be executed locally, and after pushing the code to Github, they will be executed on Travis-CI.

4.2.1 Dependencies

To *Run the unit tests* there is no need to install any additional depencency. That is possible thanks to the *unittest* framework, included in the Python standard library. But we do not want to only run the unit tests: we want to set up an environment that allows us to check for suspicious code, test the code and the documentation, evaluate the testing coverage, and replicate the Travis-CI build locally. To accomplish this goal we need to install some additional dependencies:

```
$ pip install coverage           # testing coverage tool
$ pip install codecov            # testing coverage tool
$ pip install coveralls          # testing coverage tool
$ pip install prospector         # Python linter
$ pip install sphinx             # documentation generator
$ pip install sphinx_rtd_theme   # HTML doc theme
$ pip install tox                # testing tool
$ sudo apt install ruby          # apt, yum, ...
$ sudo gem install wwtd          # run travis-ci locally
```

4.2.2 Run all tests at once

All tests can be run at once by executing this single command:

```
$ wwtd
```

The *wwtd* program (*What Would Travis Do*) reads the *.travis.yml* file and executes the tests accordingly. The tests can also be run manually, one by one, as described in the following sections.

4.2.3 Run the linter

To run the linter move to the project's root directory and execute the following command:

```
$ prospector
```

4.2.4 Run the unit tests

Move to the project's root directory and execute the following command:

```
$ python -m unittest discover -b tests
```

4.2.5 Check the testing coverage

To check the percentage of code covered by test, run the unit tests using Coverage.py:

```
$ coverage run -m unittest discover -b tests
```

Now generate an HTML report:

```
$ coverage combine && coverage report && coverage html
```

To see the HTML report open the generated *htmlcov/index.html* file with your browser.

4.2.6 Test the documentation

Several things have to be tested:

- the docstring examples
- the documentation (*doc* directory) examples
- the links inside the documentation must point correctly to the target
- the HTML must be generated properly

To test the docstring examples, we use the Python standard library *doctest* module. Simply move to the root directory of the project and execute the following command:

```
$ python -m doctest simulators/*.py
```

To test the examples in the *doc* directory:

```
$ cd doc
$ make doctest
```

To check if there are broken URLs in the documentation:

```
$ make linkcheck  # From the doc directory
```

To generate the HTML:

```
$ make html  # From the doc directory
```

CHAPTER

FIVE

THE *SIMULATORS.UTILS* LIBRARY

This part of the documentation describes all useful functions that were implemented in the *utils* library in order to speed up the development of new simulator modules.

5.1 The Utils library

simulators.utils.**checksum**(*msg*)

Computes the checksum of a string message.

Parameters **msg** (*str*) – the message of which the checksum will be calculated and returned

Returns the checksum of the given string message

Return type chr

```
>>> checksum('fooo')
'L'
```

simulators.utils.**binary_complement**(*bin_string*, *mask=''*)

Returns the binary complement of bin_string, with bits masked by mask.

Parameters

- **bin_string** (*str*) – the binary_string of which the binary complement will be calculated
- **mask** (*str*) – a binary string that will act as mask allowing to complement only bin_string's digits corresponding to mask's ones, leaving the bin_string's digits corresponding to mask's zeros as they are

Returns the binary complement of the given bin_string

Return type str

```
>>> binary_complement('11010110')
'00101001'
```

```
>>> binary_complement('10110', '10111')
'00001'
```

simulators.utils.**twos_to_int**(*binary_string*)

Returns the two's complement of binary_string.

Parameters **binary_string** (*str*) – the string containing only zeros and ones. It is mandatory to pad this value to the desired bits length before passing it to the method in order to avoid representation errors.

Returns the two's complement of the given binary_string, as an integer

Return type int

```
>>> twos_to_int('11111011')
-5
```

```
>>> binary_string = '111'
>>> twos_to_int(binary_string)
-1
>>> binary_string = binary_string.zfill(8)
>>> binary_string
'00000111'
>>> twos_to_int(binary_string)
7
```

simulators.utils.**int_to_twos**(*val*, *n_bytes=4*)

Returns the two's complement of the given integer as a string of zeroes and ones with len = 8*n_bytes.

Parameters

- **val** (*int*) – the signed integer to be returned as binary string in two's complement
- **n_bytes** (*int*) – the number of total bytes to use for the given signed integer conversion to two's complement

Returns the two's complement of val, as a binary string

Return type str

```
>>> int_to_twos(5)
'00000000000000000000000000000101'
```

```
>>> int_to_twos(5, 2)
'0000000000000101'
```

```
>>> int_to_twos(-1625)
'11111111111111111111100110100111'
```

```
>>> int_to_twos(-1625, 2)
'1111100110100111'
```

```
>>> int_to_twos(4294967295)
Traceback (most recent call last):
    ...
ValueError: 4294967295 out of range (-2147483648, 2147483647).
```

simulators.utils.**binary_to_bytes**(*binary_string*, *little_endian=True*)

Converts a binary string in a string of bytes.

Parameters

- **binary_string** (*str*) – the original binary string that has to be converted to a byte string
- **little_endian** (*bool*) – boolean indicating whether the returned string should be formatted with little endian or big endian notation

Returns the bytestring representation of binary_string

Return type str

```
>>> binary_to_bytes('0110100001100101011011000110110001101111', False)
'hello'
```

simulators.utils.**bytes_to_int**(*byte_string*, *little_endian=True*)

Converts a string of bytes to an integer (like C atoi function).

Parameters

- **byte_string** (*str*) – the signed integer represented as bytes
- **little_endian** (*bool*) – boolean indicating whether the byte_string param was received with little endian or big endian notation

Returns the value of byte_string, converted to signed integer

Return type int

```
>>> bytes_to_int(b'hello', False)
448378203247
```

simulators.utils.**bytes_to_binary**(*byte_string*, *little_endian=True*)

Converts a string of bytes to a binary string.

Parameters

- **byte_string** (*str*) – the byte string to be converted to binary
- **little_endian** (*bool*) – boolean indicating whether the byte_string param was received with little endian or big endian notation

Returns the binary representation of byte_string

Return type str

```
>>> bytes_to_binary(b'hi', little_endian=False)
'0110100001101001'
```

simulators.utils.**bytes_to_uint**(*byte_string*, *little_endian=True*)

Converts a string of bytes to an unsigned integer.

Parameters

- **byte_string** (*str*) – the unsigned integer represented as bytes
- **little_endian** – boolean indicating whether the byte_string param was received with little endian or big endian notation

Little_endian bool

Returns the value of byte_string, converted to unsigned integer

Return type int

```
>>> bytes_to_uint(b'hi', little_endian=False)
26729
```

simulators.utils.**real_to_binary**(*num*, *precision=1*)

Returns the binary representation of a floating-point number (IEEE 754 standard). A single-precision format description can be found here: https://en.wikipedia.org/wiki/Single-precision_floating-point_format A double-precision format description can be found here: https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

Parameters

- **num** (*float*) – the floating-point number to be converted
- **precision** (*int*) – integer indicating whether the floating-point precision to be adopted should be single (1) or double (2)

Returns the binary string of the given floating-point value

Return type str

```
>>> real_to_binary(619.34000405413)
'01000100000110101101010111000011'
```

```
>>> real_to_binary(619.34000405413, 2)
'0100000010000011010110101011100001010100000010111010011111110111'
```

```
>>> real_to_binary(0.56734, 1)
'00111111000100010011110100110010'
```

simulators.utils.**real_to_bytes**(*num*, *precision=1*, *little_endian=True*)

Returns the bytestring representation of a floating-point number (IEEE 754 standard).

Parameters

- **num** (*float*) – the floating-point number to be converted
- **precision** (*int*) – integer indicating whether the floating-point precision to be adopted should be single (1) or double (2)
- **little_endian** (*bool*) – boolean indicating whether the byte string should be returned with little endian or big endian notation

Returns the bytes string of the given floating-point value

Return type str

```
>>> [hex(ord(x)) for x in real_to_bytes(436.56, 1, False)]
['0x43', '0xda', '0x47', '0xae']
```

```
>>> [hex(ord(x)) for x in real_to_bytes(436.56, 2, False)]
['0x40', '0x7b', '0x48', '0xf5', '0xc2', '0x8f', '0x5c', '0x29']
```

simulators.utils.**bytes_to_real**(*bytes_real*, *precision=1*, *little_endian=True*)

Returns the floating-point representation (IEEE 754 standard) of bytestring number.

Parameters

- **bytes_real** (*str*) – the floating-point number represented as bytes
- **precision** (*int*) – integer indicating whether the floating-point precision to be adopted should be single (1) or double (2)
- **little_endian** (*bool*) – boolean indicating whether the bytes_real param was received with little endian or big endian notation

Returns the floating-point value of the given bytes string

Return type float

```
>>> round(bytes_to_real('Dw,1', 1, False), 2)
988.69
```

```
>>> round(bytes_to_real('@z%}.hQ]', 2, False), 2)
418.34
```

simulators.utils.**int_to_bytes**(*val*, *n_bytes=4*, *little_endian=True*)

Returns the bytestring representation of a given signed integer.

Parameters

- **val** (*int*) – the signed integer to be converted
- **n_bytes** (*int*) – the number of bytes to fit the given unsigned integer to
- **little_endian** (*bool*) – boolean indicating whether the byte string should be returned with little endian or big endian notation

Returns the bytes string value of the given signed integer

Return type str

```
>>> [hex(ord(x)) for x in int_to_bytes(354, little_endian=False)]
['0x0', '0x0', '0x1', '0x62']
```

simulators.utils.**uint_to_bytes**(*val*, *n_bytes=4*, *little_endian=True*)

Returns the bytestring representation of a given unsigned integer.

Parameters

- **val** (*int*) – the unsigned integer to be converted
- **n_bytes** (*int*) – the number of bytes to fit the given unsigned integer to
- **little_endian** (*bool*) – boolean indicating whether the byte string should be returned with little endian or big endian notation

Returns the bytes string value of the given unsigned integer

Return type str

```
>>> [hex(ord(x)) for x in uint_to_bytes(657, little_endian=False)]
['0x0', '0x0', '0x2', '0x91']
```

simulators.utils.**sign**(*number*)

Returns the sign (-1, 0, 1) of a given number (int or float) as an int.

Parameters **number** (*int*) – the number from which the sign will be extracted

Returns the sign multiplier of the given number

Return type int

```
>>> sign(5632)
1
```

```
>>> sign(0)
0
```

```
>>> sign(-264)
-1
```

simulators.utils.**mjd**(*date=None*)

Returns the modified julian date (MJD) of a given datetime object. If no datetime object is given, it returns

the current MJD. For more informations about modified julian date check the following link: https://core2.gsfc.nasa.gov/time/

Parameters **date** (*datetime*) – the object to calculate the equivalent modified julian date. If None, the current time is used.

Returns the modified julian date of the given date value

Return type float

```
>>> d = datetime(2018, 1, 20, 10, 30, 45, 100000)
>>> mjd(d)
58138.43802199074
```

simulators.utils.**mjd_to_date**(*original_mjd_date*)

Returns the UTC date representation of a modified julian date one.

Parameters **original_mjd_date** (*float*) – a floating point number representing the modified julian date to be converted to a datetime object.

Returns the datetime object of the given modified julian date

Return type datetime

```
>>> mjd_to_date(58138.43802199074)
datetime.datetime(2018, 1, 20, 10, 30, 45, 100000)
```

simulators.utils.**day_microseconds**(*date=None*)

Returns the microseconds elapsed since last midnight UTC.

Parameters **date** (*datetime*) – the object to calculate the total day amount of microseconds. If None, the current time is used.

Returns the number of microseconds elapsed since last midnight previous to given date

Return type int

simulators.utils.**day_milliseconds**(*date=None*)

Returns the milliseconds elapsed since last midnight UTC.

Parameters **date** (*datetime*) – the object to calculate the total day amount of milliseconds. If None, the current time is used.

Returns the number of milliseconds elapsed since last midnight previous to given date

Return type int

simulators.utils.**day_percentage**(*date=None*)

Returns the day percentage. 00:00 = 0.0, 23:59:999999 = 1.0

Parameters **date** (*datetime or timedelta*) – the datetime or timedelta object of which will be calculated the equivalent percentage. If None, the current datetime is used.

Returns the percentage of day elapsed since last midnight previous to given date

Return type float

simulators.utils.**get_multitype_systems**(*path*)

Returns a list of *.py* packages containing a *System* class. The path in which this method looks is the same path of the module that calls this very method. It is meant to be called by a module containing a *MultiTypeSystem* class.

Parameters **path** (*str*) – the path in which the function is going to recursively look for *System* classes

Returns a list of packages containing a *System* class

Return type list

simulators.utils.**list_simulators**(*path='/home/giuseppe/Documents/OAC/discos/simulators/simulators'*)
Returns the list of all available simulators in the package.

Parameters **path** (*str*) – the path in which the function will recursively look for simulators *System* classes

Returns the list of available simulators

Return type list

BIBLIOGRAPHY

[1] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] A. Orlati, M. Bartolini, S. Righini, A. Fara, C. Migoni, S. Poppi, M. Buttu, and G. Carboni. The discos project. Technical Report 522, IRA, 2018. URL: https://doi.org/10.20371/inaf/pub/2019_00001, doi:10.20371/INAF/PUB/2019_00001.

[3] Gianluca Chiozzi, Bogdan Jeram, Heiko Sommer, Alessandro Caproni, Mark Plesko, Matej Sekoranja, Klemen Zagar, David W Fugate, Paolo Di Marcantonio, and Roberto Cirami. The alma common software: a developer-friendly corba-based framework. In *Advanced Software, Control, and Communication Systems for Astronomy*, volume 5496, 205–218. International Society for Optics and Photonics, 2004.

INDEX

Symbols

B

C

D

G

H

I

L

M

P

R

S

T

U