4.2 Codes Explanations

After an overview of the field in which this project has been developed, now an accurate analysis of the implementation of the RIAPS apps and the codes built is needed. Therefore in this paragraph first take place a brief description of RIAPS implementation and then some critical aspects of the codes are underlined.

4.2.1 RIAPS Implementation

After all the theoretical insight of the chapter 3, now it is shown a practical demonstration of the RIAPS power, also describing how to launch applications in this extraordinary Smart Grid platform.
A RIAPS application is made up of several elements such as run-time libraries, configuration file, and all the codes to apply on the *Target Nodes*. As already explained, one of the strengths of the middleware mentioned above is the capability to systematically pack and deploy all those files on the wanted *RIAPS Target Nodes* which are automatically added in the *RIAPS CTRL* app thanks the use of the *RIAPS Deploy* service contained on each node. All this is possible by using the *RIAPS CTRL* app, which is manually launched in the *Control Node*, and those nodes that receive the algorithms are now called *Actor Nodes*, each one with the predetermined role.
Figure 4.7 show the *RIAPS CTRL* app, the command window accessible for only the *Control Nodes* of the system. It has the fundamental task to manage all the *Target Nodes* attached to the platform; in particular, it allows us to start, stop, or remove the file packages on the *Target Nodes*. That brilliant app is also very useful to have an overview of the controlled system providing the IP address of the nodes and the type of algorithm deployed on it. Moreover, it provides also the possibility to obtain a graphics view of the just built interaction network, see Appendix A in which an example of Linux dot file shows the created *RIAPS Target Nodes* interaction (for this example is possible to see only three *Control Nodes* instead of the nine of the real project for order and clarity reason).
Another important capability of *RIAPS CTRL* is the possibility to monitor all the systems thanks to the user interface in which it is possible to visualize the eventual failure or bugs in the network.

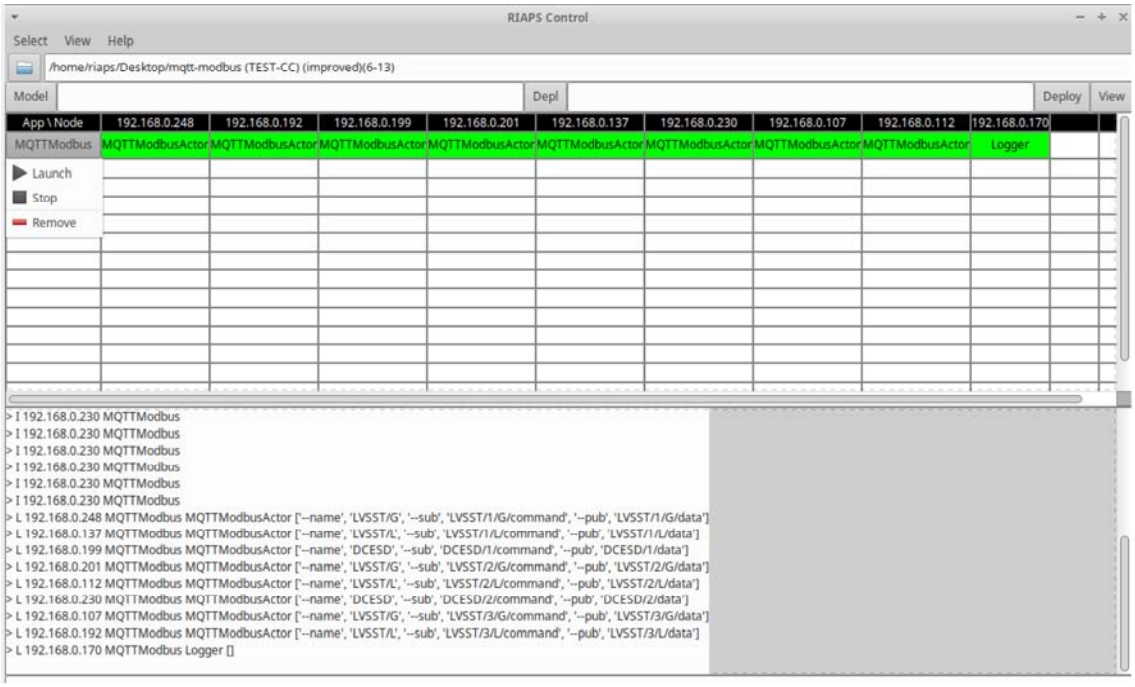

Figure 4.7: *RIAPS CTRL* app

4.2.2 Codes developed

The *RIAPS CTRL* app so far explained, it is underlined the ability to download the developed codes directly inside the wanted elements. Now it necessary to talk about those piece of codes that are deployed, describing their functions.
Beside the paho-MQTT library, Modbus library, and the RIAPS configuration files, the other elements packed and downloaded into the nodes are the developed codes. They are divided into six different files, each one with a distinct fundamental role:

- *MQTTModbus.riaps*
- *MQTTModbus.deplo*
- *MQTT.py*
- *ComputationalComponent.py*
- *ModbusUartReqRepDevice.py*

- *Logger.py*

In Figure 4.7 is it possible to see two blank boxes, "Model" and "Depl," should be filled respectively with the *MQTTModbus.riaps* file and *MQTTModbus.deplo* file.

RIAPS Model

The .riaps file written in C++, present in Appendix B.1, took the role of the main file in which all the piece of codes are handled. In particular, it contains all the file and function calls, the *RIAPS Actor* which dynamically loads and launches *RIAPS Components*, and the *RIAPS Device* that is a particular variant of *RIAPS Actor* which has only *Device Component*. The *RIAPS Actor* of the *MQTTModbus.riaps* are the *MQTTModbusActor* and the *Logger*. The first holds all the files that provide the communications between DSP and BBB and between BBB and MQTT client, but also contain the algorithms to deploy on the BBBs (in particular eight BBBs obtain this Actor as shown in Fig 4.7).

ComputationalComponent

Included in the *MQTTModbusActor*, there is the *RIAPS Component* that play the fundamental roles of algorithms container for the various device and above all to acts as a bridge between the DSP and the MQTT Client, see Figure 4.8 for an overview of the codes interactions in which the ComputationalComponent play the central role.
As it is possible to see in Appendix B.4, the *ComputationalComponent.py* file contains the initialization of the input registers (the registers those contain the data retrieved from the actual Hardware) and the holding register (those that contain the command value) of the DPS, more detail about the command and data send and received are provided in chapter 5.1. Instead to in Appendix C is possible to see the MQTT.fx windows for publish the commands (written inside a python dictionary) and for subscribe and receive the messages from the distributed smart devices.
The *ComputationalComponent.py* file also includes the "clock" method that takes care of the Modbus status request and to the Modbus request for receiving data sent to the *ModbusUartReqRepDevice.py* every eight seconds (the time clock is set inside the RIAPS model). Another fundamental method is the "RecCommand" to handle the messages received (commands) from

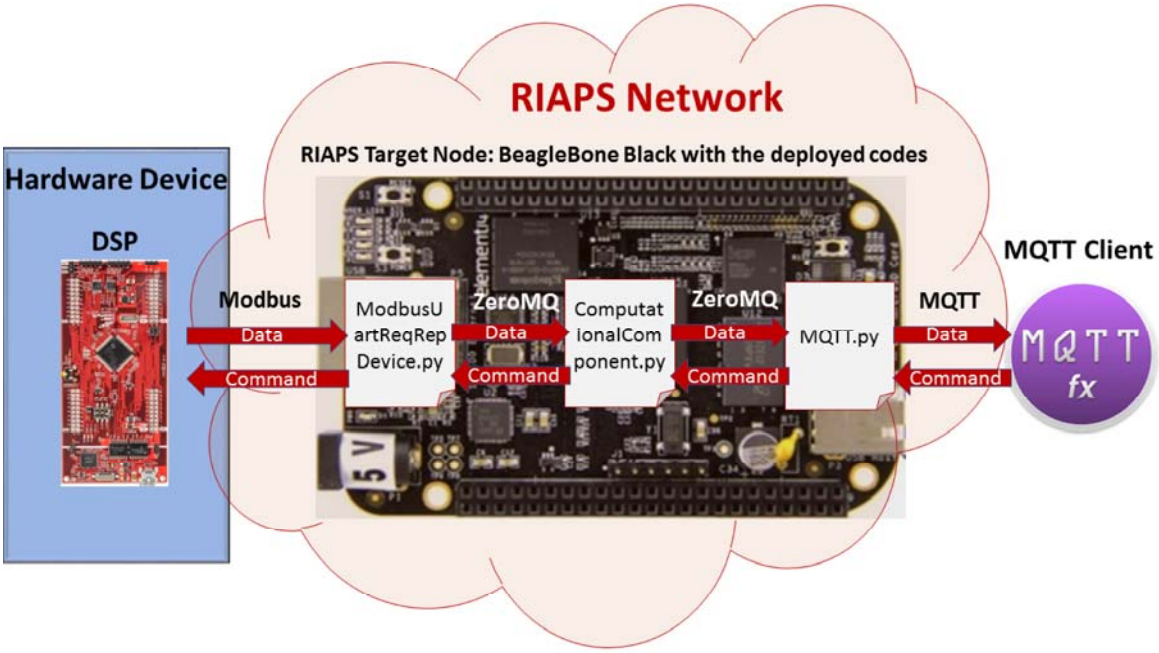

Figure 4.8: Interaction overview of a single *RIAPS Target Node*

MQTT.py, it is first converted from analogue to digital and than in base on the type of command a Modbus request is sent through ZeroMQ to change the value of the wanted holding registers of the DSP. The "Conversions" method instead converts the digital data coming from the input register of the DSP to analog human-readable data and packed to a Python dictionary. After the conversion phase, the dictionary goes to the "modbusCommandReqPort" method in which it is sent to the *MQTT.py* through ZeroMQ. It has been chosen as a dictionary to simplify the MQTT message and allow a JSON conversion.

Logger

Instead, the *Logger* actor has the role in taking a register of all the event and message exchanged during the run phase of the software, so it is a handy tool to monitor the execution of the codes and also to provide a simple way for debugging. The *Logger.py* file contained in the *Logger* Actor, showed in Appendix B.6, is downloaded in only one *Target Node*. Usually, it is the same control machine that in this case, is both Control and Target Node.

MQTT Device and Code

Regarding the *RIAPS Devices*, contained inside the *MQTTModbus.riaps*, one of these is *MQTT Device* which contain initialization of the used communication ports, timer clock of the loop functions, references and fundamental parameters for its Python code which needs the address of the chosen broker the QoS and the topic for which publish and subscribe. Now focusing on the Python code also included in the *MQTTModbusActor*, *MQTT.py*, which contain all the standard functions for an MQTT communication that can be seen in Appendix B.3.
The Python method "pubPort" has the important role in handling the message that comes from the DSP, containing the data of the devices, and after storing them in a variable has the important task to publish them in MQTT style (by topic).
Instead, the method "incoming" manage the commands that comes from the MQTT client, subscribing for them and once received, send them through ZeroMQ to the DSP passing from the *ComputationalComponent.py* file. For a better overview of the communication architecture, see Fig. 4.8 and Appendix A in which are shown all the interaction between nodes.

ModbusUartReqRepDevice Device and Code

The other *RIAPS Devices* present in the RIAPS model is the *ModbusReqRepDevice* that represents the Modbus-UART communication lines. In fact, inside this RIAPS function, there is the initialization of the two port utilized; one used for test the communication between the two devices and to notice the status ("modbusStatusRepPort"); the second is the real port utilized for the exchange of data ("modbusCommandRepPort"). Moreover, it contains also all the parameters useful for the *ModbusUartReqRepDevice.py* file, in particular, the slave address and baud rate (Modbus slave parameters) and the BBB UART port used, in this case, the first (UART1).
Once received those parameters the Python code (see Appendix B.5) which include the Modbus Library needed for the communication, answer to all the request received by the *ComputationalComponent.py*, such as status information command execution and need of data from DSP.

RIAPS Deplo

The .deplo is a proper RIAPS file written in C++. As we can see from Appendix B.2, It has the critical role to route the various *RIAPS Actors*

in the wanted devices attached in the network, and represented by their a priori fixed IP address. It is also possible to choose to work only with certain node instead of all together. Moreover, it provides the possibility to the developer to change and choose three important parameters for the *MQTTModbusActor*:

- The name of the actual device for which it will work, very useful for applying the right conversations and algorithm inside the *ComputationalComponent.py*.
- The topic for which the *MQTT.py* has to subscribe, in this case, the commands that come from the MQTT Client.
- The topic for which the *MQTT.py* has to publish, in this case, the data that comes from the DSP.