NKUA Weather Station Network Project

Weather Station Network Simulator

Version 2

Ronaldo Tsela

(August 2023)

Contents

List of Figures

1. Purpose

This document aims to introduce the UOA Weather Station Network Simulator, which was employed during the development of the UOA Weather Station Network System. The simulator serves the purpose of testing and validating various functionalities. Detailed descriptions of both the graphical user interface and the program logic are provided, allowing interested users to comprehend, modify, and effectively utilize this software.

2. Weather Station Simulator Logic

2.1 The Class

The UOA Weather Station Network Simulator was developed to replicate the authentic behavior of a sequence of physical weather monitoring devices within a network. At the heart of this simulator lies the weather station controller, constructed in a manner closely resembling the operational principles of the physical station drivers. Each station is represented as an instance of the *Weather_Station* class (wsn_sim.py, line 27).

Within this class, an set of member functions emulates the functioning of real weather station units and incorporates a finite state machine responsible for the overall operations, effectively establishing a digital twin, mirroring the behavior of the physical systems under study.

Weather_Station.__init__(station_id) : This constructor function is responsible for generating, initializing, and managing a designated station, identified by its unique station ID (wsn_sim.py, line 28). The FSM logic is also implemented in that function (wsn_sim.py, line 71).

Sensor data functions: A set of virtual data generators are utilized for simulating the data recording provided by the sensors (wsn_sim.py, line 183). The functions are responsible for generating random numbers that represent the temperature, humidity, pressure, wind speed, wind direction and rainfall rate. These functions can be replaced with similar functions that instead of randomly generating data values they can read values provided by a user-input file with the respective data values.

Telemetry data functions: Similarly, to the sensor data functions, a set of telemetry related functions are utilized for providing virtual telemetry data for the specific station (wsn_sim.py, line 203). The also randomly generated data represent the internal temperature, the bus voltage and current and the solar array voltage. Also, a version for these functions capable of reading a file may be another possible approach to this design.

Weather_Station.transmit_sensorData(data_packet): This function serves the purpose of transmitting a collection of data values stored within the *data_packet* list through an HTTP POST request to the corresponding URL (wsn_sim.py, line 218). The composition and formation of the data packet values is presented in wsn_sim.py, line 226. Upon execution, the function provides the response received from the backend data handler.

Weather_Station.transmit_sensorData(data_packet): This function similarly to the previous one is utilized in order to transmit the telemetry data packet though an HTTP request to the corresponding URL and reading back the appropriate response from the backend telemetry handler (wsn_sim.py, line 232). In wsn_sim.py, line 240 the composition and formation of the data packet entry is presented.

Weather_Station.read_command(): This function is employed to retrieve telecommands from the associated backend telecommand transmitter, specific to the respective weather station (wsn_sim.py, line 246). These telecommands play a role in updating the mode and state of the weather station controller (wsn_sim.py, line 279). A telecommand is formatted as *$command id, command argument$*.

Thus, this function conducts an HTTP GET request to the backend command transmitter (wsn_sim.py, line 250) to locate available commands adhering to the format mentioned before.

Upon identifying a command (wsn_sim.py, line 251 and line 252), the fields *command id* and *command argument* are extracted (wsn_sim.py, line 255, line 257, and line 258). Subsequently, if these fields correspond to a valid command, the mode/ state is updated accordingly, otherwise, no changes occur (wsn_sim.py, line 260). To confirm receipt by the station, the command is substituted with an empty string by performing a POST request to the backend command receiver (wsn_sim.py, line 276).

2.2 FSM Logic

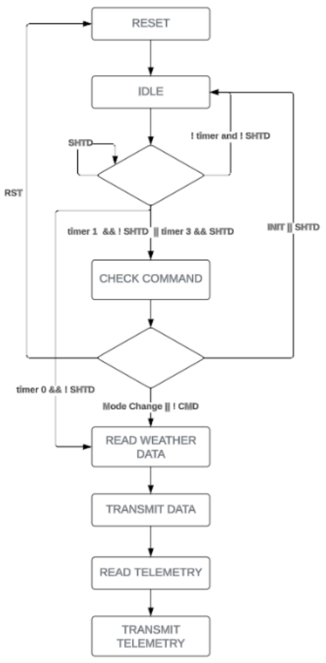


Figure 1 *– The weather station FSM controller logic*

The FSM logic is depicted in Figure 1.

RESET State (wsn_sim.py, line 79): This state involves initializing all variables to zero, initiating timers, and reconfiguring the station to its default mode. Once the reset is completed, the station transitions to the IDLE State (wsn_sim.py, line 81). The RESET state is activated every time an instance is created (wsn_sim.py, line 43).

IDLE State (or SHUTDOWN state) – ST0 (wsn_sim.py, line 109): In this state, internal control processes occur. The station remains in the IDLE state when it is not engaged in other operational states (wsn_sim.py, line 119) or when it's in SHUT DOWN mode (wsn_sim.py, line 115). While in the IDLE state, the station remains there until the sampling counter, ticks. If the station isn't in SHUT DOWN mode, it transitions to the READ WEATHER DATA State once the sampling counter ticks. However, if the station is in SHUT DOWN mode, it continues to stay in the IDLE state (wsn_sim.py, line 109).

Furthermore, once the heartbeat counter ticks (wsn_sim.py, line 123), the station updates its heartbeat (wsn_sim.py, line 128). If the station is not in SHUT DOWN mode, it seeks a new potential command. In the case of SHUT DOWN mode, the station waits for the shutdown timer to tick before attempting to read a new command (wsn_sim.py, line 131).

CHECK COMMAND State: Although not established as an independent state, it is embedded within the IDLE state. In that state the function *read_command()* presented previously is employed (wsn_sim.py, line 132).

READ WEATHER DATA State – ST1 (wsn_sim.py, line 134): In this state, the station reads the data produced by the sensor simulators. Then the station proceeds to the next state which is the TRANSMIT DATA state (wsn_sim.py, line 135).

TRANSMIT DATA State – ST2 (wsn_sim.py, line 147): In this state, the station generates an HTTP POST request to store the data by accessing the designated backend handler. The transmission utilizes the *transmit_sensorData()* (wsn_sim.py, line 154) function described earlier. The station then proceeds to the READ TELEMETRY state (wsn_sim.py, line148).

READ TELEMETRY State – ST2 (wsn_sim.py, line 158): In this state, the station retrieves data from the virtual monitors utilizing the telemetry related functions described earlier. The station then proceeds to the next state which is the TRANSMIT TELEMETRY (wsn_sim.py, line 159).

TRANSMIT TELEMETRY State – ST2 (wsn_sim.py, line 169): In this state, the station constructs an HTTP POST request to transmit the telemetry data to the appropriate backend handler (wsn_sim.py, line 176). The station once this state is completed redirects its state controller to the IDLE state (wsn_sim.py, line 170).

2.3 Program Parameters

2.3.1 URL's

Backend Data Handler: [redacted]
Description: This URL redirects to the backend data handler. The content transmitted (data packet) is issued in state ST2.

Backend Telemetry Handler: [redacted]
Description: This URL redirects to the backend telemetry handler. The content transmitted (telemetry packet) is issued in state ST4.

Command Transmitter Handler: [redacted]
Description: This URL redirects to the command transmitter handler.

Command Receiver Handler: [redacted]
Description: This URL redirects to the command receiver handler.

2.3.2 Operation Modes

FAST: Conduct sample recording at a 3-minute interval (wsn_sim.py, line 8).

NORMAL: Conduct sample recording at a 5-minute interval (wsn_sim.py, line 9).

SLOW: Conduct sample recording at a 10-minute interval (wsn_sim.py, line 10).

POWER SAVING: Conduct sample recording at a 30-minute interval (wsn_sim.py, line 11).

SHUTDOWN: Do not collect samples. Change the internal-control mode to restricted. This operation mode refers to an internal FSM state.

2.3.3 Timers

Timer 0: Utilized for overseeing the sample recording process. Utilized with reference the operation modes for sample recording.

Timer 1: Employed to manage the heartbeat update process and command check mode. Every heartbeat update is conducted with a 5 minute interval (wsn_sim.py, line 13).

Timer 2: Applied to regulate the command check mode when the shutdown mode is activated.

2.3.4 Posting Logic

The data is transmitted to the server handlers through HTTP POST requests using the provided URLs. This functionality is enabled using the *request* Python module. Each request generates a corresponding response that includes both the request state and the data. To extract the data, the *text* method is employed to convert the response into a string format.

3. Weather Station Network Simulator GUI

The Graphical User Interface (GUI) comprises of four widgets. The interface is designed using the *Tkinter* Python library, known for creating simple and fast GUIs. As illustrated in Figure 2, the GUI encompasses an input field to specify the number of stations utilized in the network, one button to initiate the simulation, and another button to terminate it.

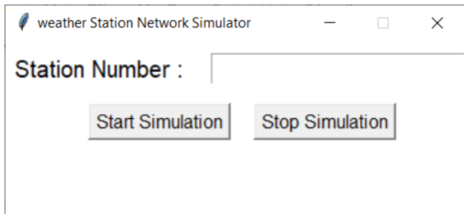


Figure 2 *main UI window*

Upon inputting a number ranging from 1 to 99 into the application's entry field and subsequently clicking the "Start Simulation" button, a corresponding number of threads is generated. Each thread signifies a distinct station entity. The label "Simulation Started" is displayed to signify the simulation is in progress. When the "Stop Simulation" button is pressed, the label changes to "Simulation Terminated," and all threads terminate, effectively concluding the simulation. Figure 2 and Figure 3 illustrate the states of running and terminating the simulation.

Data are displayed also in the command prompt as depicted in Figure 4, indicating the different states that each station undergoes and the activities that manages, such as data transmission and command reception.

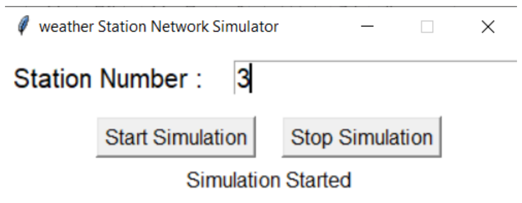


Figure 3 *Simulation is running*

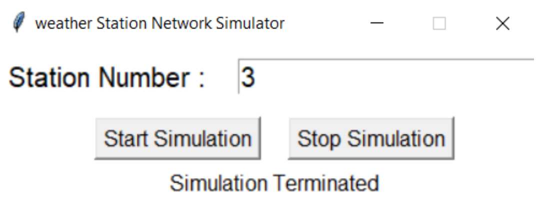
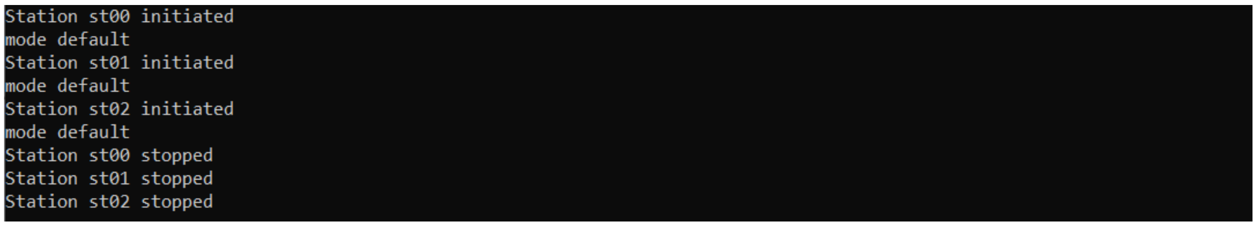


Figure 4 *Simulation is terminated*



```
Station st00 initiated
mode default
Station st01 initiated
mode default
Station st02 initiated
mode default
Station st00 stopped
Station st01 stopped
Station st02 stopped
```

Figure 5 *Command prompt respective messages*