University of Groningen

Software Engineering

Architecture Document

GreenerSimulation

Team:
Chris Worthington
Sjoerd Hilhorst
Victor-Cristian Florea
Duncan Saunders
Mariya Shumska

Client:
Jim van Oosten
Jasper Clarijs
(Greener Power Solutions)

April 6, 2020

1 Introduction

Greener Power Solutions provides sustainable energy for the temporary energy market, by delivering large-scale batteries, energy forecasting and monitoring software.
All control and monitoring happens as a client-server interaction via Modbus protocol. Naturally, the client side (both hardware and software) is represented as GreenerEye and the server as a battery. New software for the GreenerEye must always be tested, as errors can lead to costly, and even dangerous situations.
The goal of the project is to make testing easier by providing a software which simulates the battery as a server. Therefore, there is no need for physical batteries, power sources, and electrical loads.

2 Architectural Overview

The GreenerSimulation is the software designed to mimic the physical battery behaviour with varying energy input (determined by the client) and power demand (determined by a random number generator, historic data or a simulation). The digital replica of the battery has to be implemented as a server which uses the Modbus protocol for communication with client. Therefore, all data produced by the battery is stored in the Modbus registers and is available for client to read. Such registers have some limitations like fixed sizes (1 or 16 bits). Since most of the battery fields are floats, some implications for the design caused by that fact have to be considered. Moreover, as different battery providers have different registers mappings it should be possible for user to configure this mapping. On the back-end the addresses configuration is stored in a JSON file which can be loaded to the program providing the mapping for the battery and other customisable values.

The software consists of several modules:

2.1 Battery

This module holds the class of the `Battery` itself. This class could be considered as a very specific wrapper for the Modbus server. All data for parameters and state is stored in Modbus registers. Mostly, the available methods are dealing with registers manipulations like `set_value(address, value)` and `get_value(address)`, or mimicking the real battery behaviour. For instance, `connect_power(power)` simulates some physical actions and their impact on the state.
Just like the real battery, its digital replica has to update all its fields with respect to the energy input. Therefore, the method `update()` is meant to set up new values to the battery parameters according to the mathematical relations between the fields.
Since the updates have to happen quite frequently in a loop, a blocking on server might occur preventing the client from reading the registers. This problem was solved by running the server on the separate thread. Therefore, when the method `run()` is called on the battery, the server is available for client to respond while updating its registers.

2.2 Simulations

This directory contains all the logic for the simulations. The simulations connect to the battery and when the battery updates it gets the new simulated values. These simulations are done in 3 ways: by generating random values, by feeding the program historic data or by performing a real

simulation. This directory contains following modules:

- `simulation`
  This class is the superclass of all the other classes. It contains an initializer which defines the minimum required fields, and a getter which first calls `update()` and retrieves all previously mentioned fields. The `update()` function should be implemented by the subclasses, as all simulations update differently

- `random_simulation`
  This simulates the energy input and output based on random values in a given range. The `update()` function sets the fields to a random values between the set intervals

- `historic_simulation`
  The functionality of this class is not yet implemented and will be part of a future sprint. This class simulates the energy input and output based on the historical data. The `update()` function retrieves the next line of data from the historical file and updates the fields to these values

- `simulation_model`
  The functionality of this class is also not yet implemented and will be part of a future sprint. This class simulates the energy input and output based on a set of initial values and a set of rules for the simulation. The update function takes the current fields and applies these rules to generate the new energy I/O values.

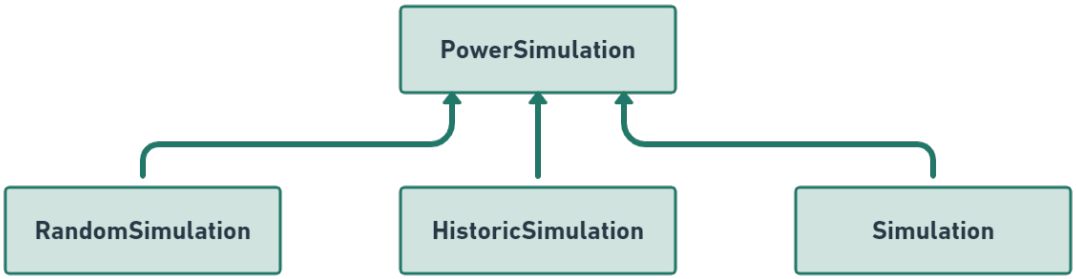

Figure 1: Class Hierarchy Diagram

2.3 math_engine

Currently this package contains `MathEngine` class which is meant to perform all required calculations with respect to mathematical relations between battery's fields provided by the client. The decision to put all this functionality to a separate class from `Battery` was made in order to improve code clarity and avoiding too long classes. Moreover, in future once more calculations will be needed for power simulation a new class for that might be created and put into this module.

2.4 util

This package will include all util classes. Currently, there is a `FloatHandler` one, which is responsible for encoding and decoding float values to the registers. This is because Modbus registers give the developer the option on how to store floats, as there are multiple ways to do it. Our software provides two options for floats storage: by scaling them (`"SCALE"`, i.e a signed 32-bit int) or combining two registers (`"COMB"`, i.e 32-bit float). The float is stored in a IEEE-754 float format, which is a standardized way of storing floating point values. For both options `SCALE` and `COMB` the user can define the endianness of both the word and byte order.

2.5 json_config

This package just contains two functions for interacting with the config JSON files. Firstly `get_custom_json()` which takes a string as an input will load the JSON file of that name from the `config` directory into a dictionary variable which is returned form the function. The next function `get_data()` also has a string input and does the same as the previous function but only gives the Modbus address mapping rather than any initial data or simulation settings.

There is a `config` directory where any JSON configuration files will be stored. At this point it contains `env.json` and `default.json`. The JSON files keep all changeable information about the battery simulation which when changed will alter how the program runs without altering the python code.

Data stored in JSON files:

- The server address and port
- The server (battery) id
- The simulation type (`random/historical/simulation`)
- The mode of storing float values (`SCALE/COMB`)
- The word and byte orders (big/little endianness)
- The scale factor for the scaling mode of storing floats
- The number of data variables and number of state variables
- The value used to separate address from register type using div and mod
- The list of initial input field values for the simulation
- The Modbus address mapping for all fields in the battery

2.6 Main

In this module the `main()` function of the program takes place. First, the environment (`env`) is defined from the .json file and the battery is instantiated. Next, depending on the values from `env` the energy input `power_sim` is determined and fed to the battery. After all input-depended battery fields are filled in by `battery.update()`, `battery.run()` is called, meaning launching of the server and continuous update of the registers.

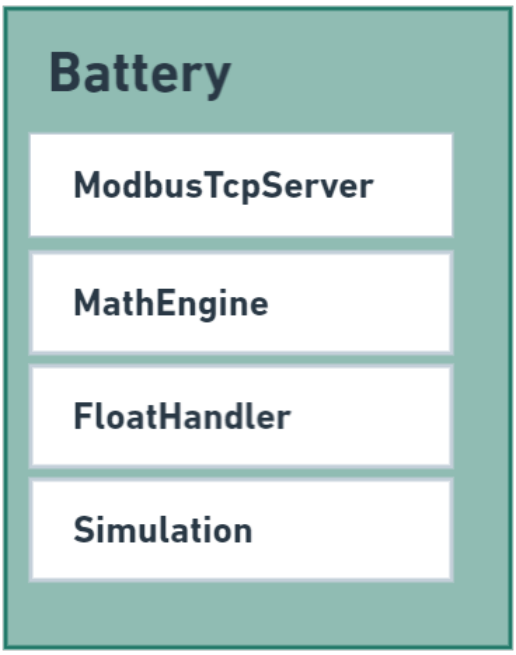

Figure 2: Architecture and dependencies of the Battery instance

2.7 Client

This module contains `GreenerEye` class, which was set for testing purposes and is not the part of the required deliverables. Essentially it is a simple Modbus client side which can read values from the battery.

3 Technology Stack

The following technologies and languages are used:

- Modbus
  - Used as the communication protocol with TCP frame format
  - Was the requirement by a client since the actual hardware uses Modbus protocol
- Python 3.5
  - Was suggested by a client for better compatibility with existing software
  - Suitable for simulations

  - PyModbus package is available as a full Modbus protocol implementation which can be used without any third party dependencies

- JSON
  - Was suggested by the client as easily interacted with in python.
  - Can be error prone for non-technical users, however will only be used by technical users that understand the JSON format

4 Team Organization

Sjoerd and Mariya were responsible for the general design of the software and implemented most of the code. Both of them actively contributed to the documents as well.
Duncan suggested and justified the use of JSON for configuration storage and implemented `json_config` module. Was a secretary for first two meetings and wrote minutes which could be found under "Meeting Log" section in Requirements document.
Victor did some research about physical processes and worked on putting code into SonarQube
Chris contributed to the documentation and managed Trello tasks
All major decisions were discussed within the team, the contribution to the presentation was equal from all members: each member prepared a section and presented it.

5 Change Log

Date	**Comment**
05.03.2020	Mariya: Set up arcitecture document and sections
05.03.2020	Mariya: wrote technology stack
25.03.2020	Chris: wrote Introduction section
25.03.2020	Chris: wrote start to Architectural overview
25.03.2020	Chris: wrote Code Structure
31.03.2020	Mariya: Architectural overview version + elaborating on (2.1)
31.03.2020	Mariya: elaborating on sections (2.3), (2.4), (2.7), (2.8)
04.04.2020	Chris: updated Code Structure (3)
04.04.2020	Sjoerd: elaborating on util (2.4) and simulations (2.2)
05.04.2020	Mariya: some formatting; sections `json_config` and `config` merged into one; "Technology Stack" (3) update; "Team Organization" (4) draft
06.04.2020	Mariya: updated "Introduction"