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)

May 19, 2020

rijksuniversiteit
groningen

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. 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 and power demand determined by the user. The digital replica of the battery has to be implemented as a server which uses the Modbus protocol for communication with the client-side implementation of the user. Therefore, all data produced by the battery is stored in the Modbus registers and is available for the user to read and/or update. As different battery providers have different registers mappings it should be possible for the user to configure this mapping. This mapping, information about the battery and the configuration of the simulation's flow are stored in `env.py` module are easily customisable for the user.

The software can be split up into following parts responsible for certain concerns:

2.1 Configuration & User interaction

The way the user interacts with the software is by working in `env` module. There is no GUI provided, as client and potential users have programming experience and prefer working directly with software files. Essentially this module is the Python dictionary and acts as a flexible 'constructor' where user can configure various things:

- General information about device such as address, id, battery capacity
- Simulation flow parameters like speed of the simulation, maximum number of iterations, enabling/disabling graphing and writing to database
- Modbus mapping and parameters of the fields: it a dictionary where the key is the name of the field and the value which is another dictionary, where register type, address, encoding and optionally initial value can be defined by the user. Here, user can add his own custom fields as well. If some field should be taken and updated from the `.csv` file with historical data, user can provide a pair of field name and corresponding `.csv` name

To make environment configuration more user-friendly, some frequently used parameter names are defined in `var_names` module, therefore the user can use variables instead of strings to avoid possible errors or typos.

2.2 Server

Since the main task was to develop a server-side of the software which would represent the battery, the decision was made to implement a `Battery` class as a wrapper for the modbus server. More specifically, an asynchronous TCP server was used.
All data for parameters and state of the battery is stored in modbus registers. Mostly, the available methods are dealing with registers manipulations like `set_value(field, value)` and `get_value(field)`.

Modbus registers define themselves how to store floats, different battery providers implement different methods. In general, there are 2 ways:

1. Storing floats by scaling: each float is multiplied by a specified scaling factor and rounded up to an integer. When retrieving the value it is divided by the scaling factor again to obtain the original value
2. Storing floats in combination of two registers: the float is split up into an integer part and a decimal part and those are stored in two different registers.

To provide this functionality the helper class `PayloadHandler` was implemented which encodes and decodes values according to user-specified parameters.

2.3 Simulation

Since one of the main requirements of the client was **modularity** of the software, the best solution to provide maximum flexibility while working with the software was using inheritance. `SimulationSuper` class was implemented such that it updates the registers and provides default mathematical relations for basic battery fields.
However, this class does not deal with the essential user input: methods for power calculations are not defined. Therefore, they should be implemented in `Simulation` subclass. This class is meant to be customised by the user according to his needs: besides of defining `active_power_out`, `reactive_power_out`, `active_power_in` and `reactive_power_in` any other method can be overridden. This can be done in a several ways:

- providing a different formula
- providing a `.csv` file with historical data
- combination of two above (e.g: retrieving historical data and and scaling it)

Moreover, the inheritance allows user to define methods to calculate new custom fields. Therefore, the whole software is not bounded by hardcoded set of the battery parameters, but can be expanded.
The superclass is also responsible for sharing information with `Graph` and `Database`.

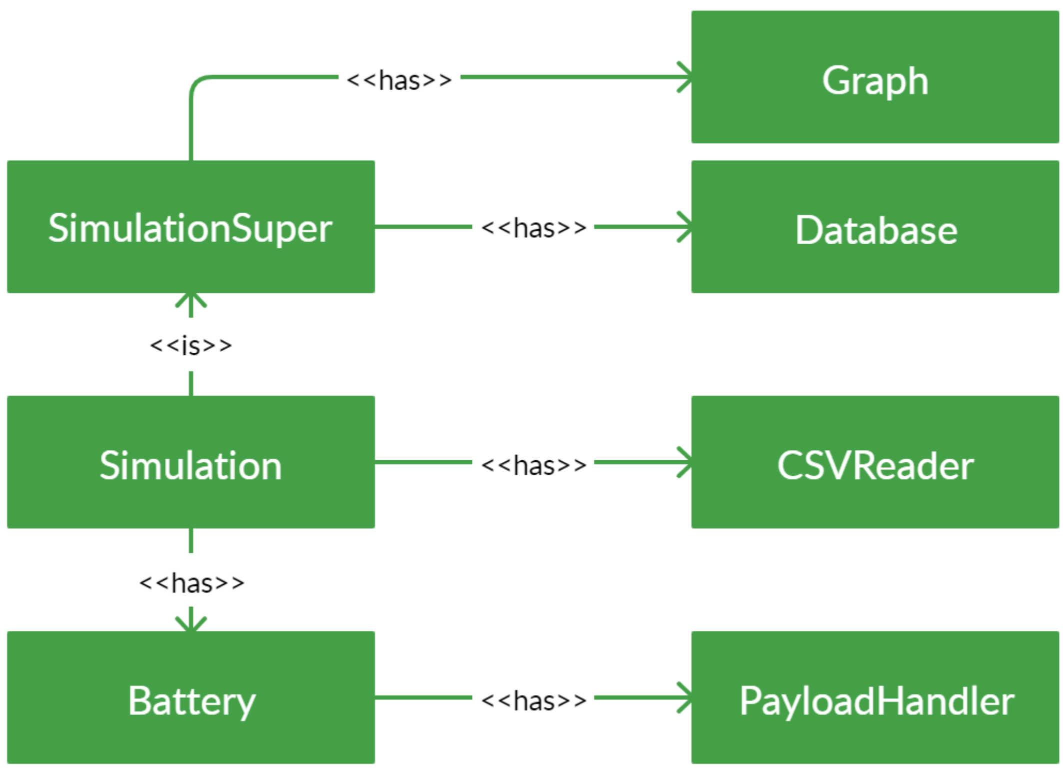
2.4 Output manipulation

The software suggests two possible manipulations of the output data:

1. Storing the output in the database
   The `Database` class is responsible for storing the value of every field for every iteration along with a timestamp. It is implemented with the use of SQLite

2. Real-time plotting
   The `Graph` class provides a real-time visual representation of the field values against the time. The graph runs in a separate thread from the simulation, and updates every 100 milliseconds when it displays the new data obtained form the `Simulation` class. The fields to be shown are defined in `env` module. The obtained graph can be saved.

2.5 Class diagram overview



3 Technology Stack

The following technologies and languages are used (for more information about libraries used please see the appendix):

- 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
- Matplotlib
  - Library for graphing data
  - First PyQtGraph was used but this library was suggested by the client for the built in functionality of saving the graphs
- Pandas
  - Used for retrieving historical data from CSV files and feeding them into the simulation
- SQLite
  - Easy and lightweight Database Management system used for storing the simulation data.
  - Suggested by the client

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"
29.04.2020	Mariya: "Architectural Overview" (2) structure is refactored, (2.1) completed
13.05.2020	Mariya: "Architectural Overview" (2) structure is refactored, (2.2), (2.3) added
14.05.2020	Mariya: section (2.1) added
16.05.2020	Sjoerd: Expanded technology stack
16.05.2020	Mariya: Update of (1), (2) according to Sjoerd's feedback
17.05.2020	Mariya: section (2.4) added
18.05.2020	Mariya: section (2.5) and appendix added
1.-05.2020	Sjoerd: section (2.4) small expansion

A Appendix: requirements.txt

```
attrs==19.3.0
Automat==20.2.0
constantly==15.1.0
cycler==0.10.0
hyperlink==19.0.0
idna==2.9
incremental==17.5.0
kiwisolver==1.2.0
matplotlib==3.2.1
numpy==1.18.4
pandas==1.0.3
pkg-resources==0.0.0
PyHamcrest==2.0.2
pymodbus==2.3.0
pyparsing==2.4.7
pyserial==3.4
python-dateutil==2.8.1
pytz==2020.1
six==1.14.0
Twisted==20.3.0
zope.interface==5.1.0
```