Prototypical implementation of a load balancing and control system for low-voltage grids

Johannes Kruse
Student @ Technische Hochschule Köln
Cologne (Köln), Germany

Abstract— **As part of the PROGRESSUS research project, the TH Köln is researching various concepts for power flow optimization in the supply network. Various algorithms are to be tested on a demonstrator in the institute's laboratory. As a basis for the algorithms, an energy management system (EMS) is to be created that runs on a Raspberry Pi. The EMS will be developed under several framework conditions. For example, the programming language Python is to be used because Python is very popular with a large proportion of the staff. In the end, the functions should include data acquisition from measuring devices, control of loads (simulated with battery powered inverters), and cross-node data exchange. This document will focus primarily on the software so that at the end there is a good understanding of the architecture and operation so that others can continue the work.**

1 Introduction

Energy distribution in today's power grid is more complex than it was a century ago. Among other things, the steadily growing number of decentralised energy producers in the grid ensures that power no longer flows only from the large power plants to the consumers, but is instead generated locally and consumed over shorter distances. What at first seems to be less lossy and therefore good, unfortunately also brings with it some problems. Because decentralised measurement data are often not collected across the board, the grid operator can often only estimate the grid status in the low-voltage grid. Also, the lines far from the power plants are significantly smaller in size, which means there is a risk of overload. Research projects, such as PROGRESSUS, aim to develop concepts that prepare the grid for such problems. The so-called smart grid is a good alternative to otherwise expensive line expansion. In this process, the actors of the power grid are networked with each other, which enables an optimisation of the power flow so that limit values of voltages and loads can be observed.

During the practical project at the TH-Cologne, I realised a networking of the participants in the test environment in cooperation with the company Devolo. Measurement data from several network analysers can be collected and exchanged between different nodes. Based on the collected measurement data, the algorithms can calculate an optimised target state. Several battery-supported inverters, which are distributed over the test environment, are available to the control system as manipulated variables of the power flow. By charging and discharging the batteries, the power of the control can be adjusted bidirectionally. The test setup itself is modular and allows many different grid situations to be mapped. For example, the grid topology can be adapted via a plug-in panel. Likewise, different cable lengths can be used and interconnected there.

The software developed during the work includes monitoring, data storage, automatic data exchange between the nodes and offers interfaces for measured values and control.

The challenge of dealing with the existing problem, learning more about it and developing my programming skills motivated me throughout the work and taught me a lot.

2 Software-Anforderungen

As already mentioned, the development decisions are closely linked to the requirements and wishes of the PROGRESSUS staff at the TH Köln. This is of great importance to ensure that the software is subsequently used. The following is a list of the explicitly mentioned requirements.

1. python is to be used as the programming language.
2. the program should run on a Raspberry Pi and be as performant as possible, because the controls will also be integrated on the Raspberry Pi
3. data from measuring devices should be recorded at least once a minute.
4. the data should be stored historically so that later analysis is possible.
5. data should be exchanged between the Raspberry Pi across nodes so that each node can make decisions autonomously. A central server is not desired.

6. communication only takes place via the local network. Internet access via the existing TH network is not available to the Raspberry Pi.

To 1: Python is very popular among engineers. Many staff and students at the TH Köln are also familiar with Python, which is why it makes sense to use it. Python aims to be easy to read and promotes a concise programming style. For example, it is not necessary to place a semicolon after each statement. Also, statements within loops or if-conditions are only structured by indenting them into blocks instead of enclosing them with curly braces.

However, Python also has a few pitfalls that can lead to problems if ignored. For example, data is not type safe (unlike in C, for example). Python generally allows similar data types to be mixed up. Thus, a data type can quickly deviate from the expected data type if care is not taken.

To 2.: Care was taken to ensure that the performance of the software was as good as possible so that the control algorithms would not be restricted later. This was achieved by sleep instructions of the time module in each loop. This ensures a significantly lower utilization of the processor and gives enough time for many necessary calculations.

To 3.: New measured values are distributed to all nodes approximately once per second. With the inverters, the data acquisition is less than three seconds. This means that a time of significantly less than a full minute is achieved. This enables a very good analysis of the current grid status.

To 4: For data storage, a database is set up on the Raspberry Pi. On Devolo's recommendation, I decided to use an InfluxDB (database). It is fast and best suited for simple data types such as floats. It also offers the possibility of setting the data retention by means of retention policies. This means you can also set how long you want to store data and adapt this to the available storage space.

To 5: The cross-rack data exchange is realized with Zeroconf. Data packets are "held up" in the network without additional setup. Other participants can recognize this and access the data. The connection is established automatically. The scalability of this method (in terms of network load) has not been tested but will probably not be suitable for large networks. However, this method is well suited for the test setup at TH Köln.

To 6: Due to a lack of internet access, missing packages must be installed in a roundabout way. A temporary internet hotspot via laptop or mobile phone offers a remedy here. A problem not solved by this is the time synchronization of the Raspberry Pi system time. As soon as the Raspberry Pi is switched off, the time no longer runs and must be resynchronized. Usually this is done via the Internet connection. Devolo has provided an IoT gateway that is connected to the Internet via an LTE module. I set up the IoT gateway as a time server. Using the Network Time Protocol (NTP), the Raspberry Pi is now synchronized within the first 15 minutes of operation. Unfortunately, this leads to the first measured values being saved under the wrong time, which leads to inaccuracies in the data analysis.

3 Software Architecture

For the software, I was inspired by the open-source software OpenEMS. Because OpenEMS is written in Java and therefore does not meet the requirements of the TH, it was not implemented in the lab.

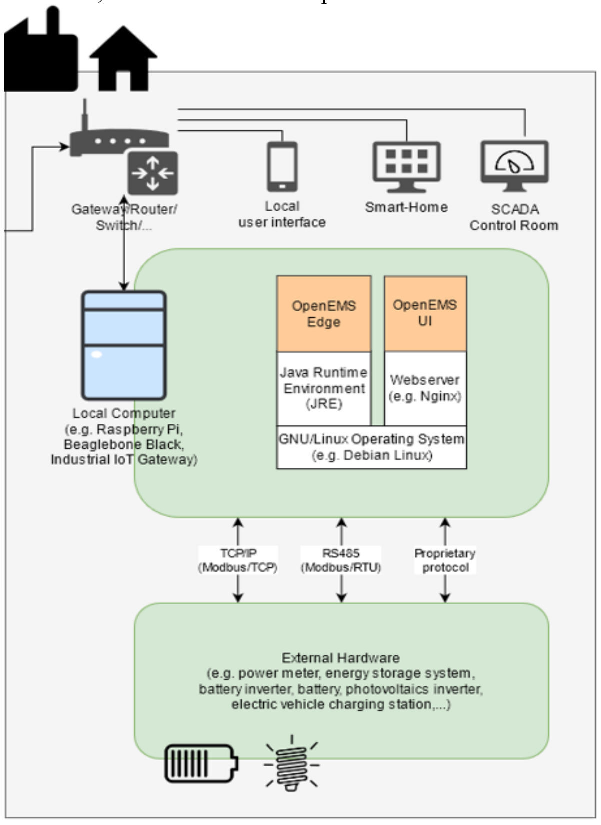
Gateway/Router/ Switch/...
Local user interface
Smart-Home
SCADA Control Room
OpenEMS Edge
OpenEMS UI
Java Runtime Environment (JRE)
Webserver (e.g. Nginx)
GNU/Linux Operating System (e.g. Debian Linux)
Local Computer (e.g. Raspberry Pi, Beaglebone Black, Industrial IoT Gateway)
TCP/IP (Modbus/TCP)
RS485 (Modbus/RTU)
Proprietary protocol
External Hardware (e.g. power meter, energy storage system, battery inverter, battery, photovoltaics inverter, electric vehicle charging station,...)

Figure 1: OpenEMS System Architecture

Fig. 1 shows an overview of OpenEMS. In my Python programme, I use the Python package Flask as a web server. Via a Representational State Transfer (REST) interface implemented there, control commands can be sent to the inverters, either via other programs using POST requests or manually via a web interface using a browser tab. The web interface also allows the current local measured values to be viewed. External hardware, such as the inverters and the grid analysers, are controlled via the Modbus TCP protocol. To avoid implementing the protocol myself, I used the Python package PyModbus. For data acquisition, an Influx database is integrated on the Raspberry Pi. This is well suited for simple data types. For the data exchange between the racks I use Zeroconf. Zeroconf or Zero Configuration Networking enables, as the name suggests, network communication without manual intervention by a person. Via multicast, devices in the network are made recognisable, just like a network printer discovery service (e.g. Bonjour or Avahi). These can then be recognised by devices operating on the same domain and receive data provided.

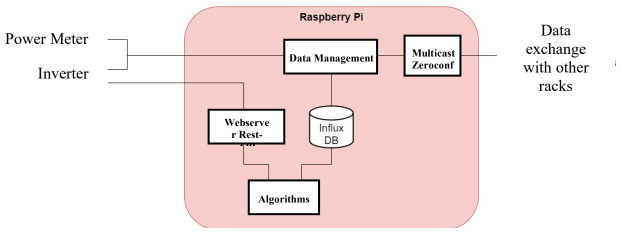

Figure 2: EMS System

Fig. 2 again illustrates the schematic structure of the energy management system developed at the end of this work, which can be used for the regulations.

To keep the code clear, the individual tasks were encapsulated in modules where possible. Accordingly, the code is divided into several Python files. This is particularly practical if a specific task area is to be adapted.

The following Python modules are currently available:

1. cmd_list.py
2. database.py
3. DataCollector.py
4. DataGatherer.py
5. InverterModule.py
6. main.py
7. PowerMeterModule.py
8. webserver.py
9. zeroconf_browser.py
10. zeroconf_registration.py

The individual modules are explained in the later subsections.

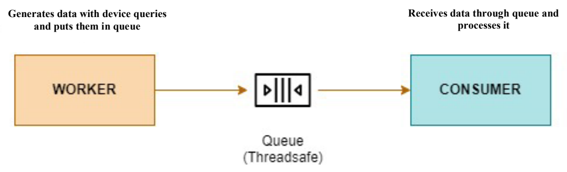
Generates data with device queries and puts them in queue
Receives data through queue and processes it
WORKER
CONSUMER
Queue (Threadsafe)

Figure 4: Producer-Constumer-Pattern

A producer-consumer pattern is used so that a large number of devices can be queried as quickly and promptly as possible. An example of the functional principle can be seen in Fig. 3. I use the standard Python package threading for this. Worker threads and consumer threads are thread safe connected to each other via queues. Errors due to race conditions, which can be caused by several threads accessing the same variables/memory, are thus avoided. The worker threads generate the necessary data, which is then taken from the queues of consumer threads and processed. It should be noted that the Python threading module only works on one core. Although several tasks are added to the thread pool, this does not provide true parallelism. Thus, measured values from different measuring devices cannot be retrieved perfectly simultaneously. This was also not a goal and not a requirement. Nevertheless, the measuring devices are queried at high frequency (in the range of seconds) so that the network status can be traced. The queues are also included as standard in current Python versions. Internally, the queues work with locks that block access to the content until the processing thread releases them again.

The following figure shows the schematic structure and clarifies which tasks the modules have and how they communicate with each other.

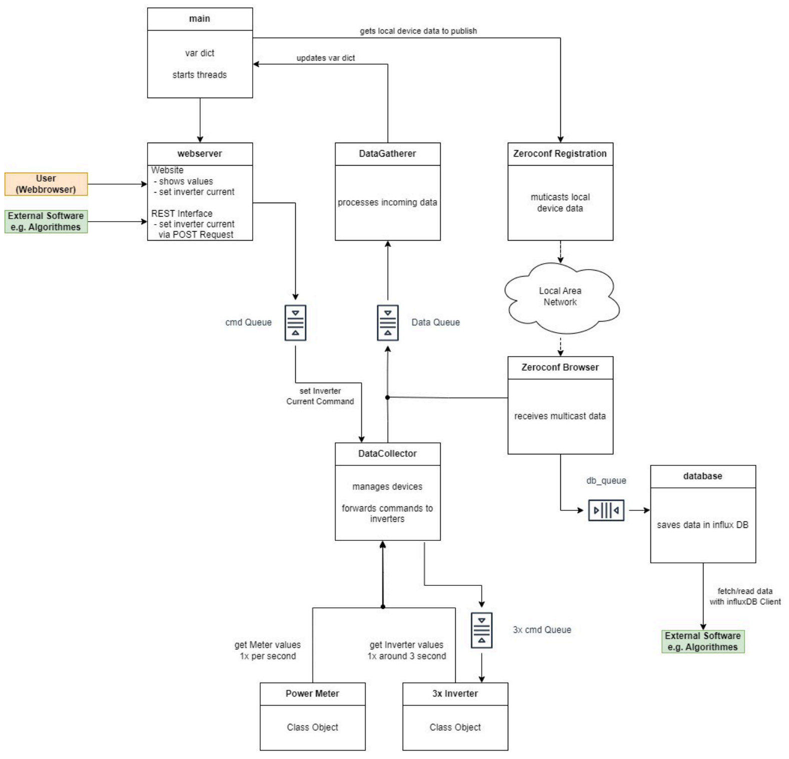

Figure 3: Module overview

The rectangular blocks show the modules used. The module name is always at the top and a brief description of the primary tasks is at the bottom. I recommend referring to this illustration when explaining the functions of the individual modules in order to understand the entire context.

External access via the existing interfaces is shown in green and orange. Users can manually access the web interface via a browser window using IP and port (e.g. 192.168.11.43:5000 for rack 3). There, the current of the local inverters can be set manually in an input window. In addition, the current measured values can be viewed.

Only the var dict from the main.py is described briefly here. It is a dictionary in which the current measured values are temporarily stored. It follows a fixed data structure and is used by the data gatherer, the web server and the Zeroconf registration module. All use manually set locks to ensure a thread-safe process. One could replace the dictionary with more queues. It was left over due to the development of the programme over time and was created very early. In order to save work in several modules, this was no longer changed. One advantage of this is that the web server does not have to keep all local readings in another memory.

The module cmd_list.py is not shown here. In the module cmd_list there is only one class with an enumeration for different commands. This saves having to define commands repeatedly in modules that work with command queues (cmd_queues).

Two interfaces are also defined for external programmes. The REST interface of the Flask web server allows inverter currents to be set with just a few lines of code. Current and past measured values can be retrieved via a Python InfluxDB client, which provides functions for the InfluxDB.

4 Conclusion

I was able to learn a lot of new things in my internship, my practical project and my bachelor’s thesis, which also went beyond the actual module plan of the degree program. This was a very rewarding time, and I would like to thank my professor, Mr. Eberhard Waffenschmidt, and my contact person at Devolo, Christop July. Both were always available to answer my questions and provide me with assistance.

The work itself was characterized by some initial difficulties, especially the communication difficulties in the network with the inverters. But by the end of the work, a stable state was achieved.

I would describe the development of the software as successful. All the requirements of the TH could be satisfied. Only the non-parallel communication of several devices simultaneously with the inverter is a pity. However, this is due to the inverter's system and cannot be realized without a complete replacement.

Scalability was never a focus but the software will be limited to smaller systems. Zeroconf can only be operated in local networks and cannot go beyond firewalls. In addition, network traffic increases with each Zeroconf service, which may cause congestion. However, with the low volumes in the lab, this is not a problem, so all algorithms can be tested well.

If I had to do the same work again, I would learn from some mistakes and approach a few things differently. For example, I would probably remove the data gatherer altogether and aim for a uniform structure with only queues. I would also start logging right away. I discovered this late in the project and previously only worked with print commands. This was also very instructive for me, as good error management noticeably improves further work. Errors that don't mean anything should always be avoided.

I found working with the Raspberry Pi remotely very useful and will adopt it in future projects with Debian operating systems. Possibly even to other OS should this be possible. It was very pleasant for me not to have to rebuild everything, especially because of my rheumatism.

One thing I would have liked to try out is the use of the "inspiration software" OpenEMS based on Java. I find Java exciting and would have liked to get more impressions of this programming language. I will make up for this when I get the chance.

I hope that the documentation uploaded to the Sciebo Cloud Service will enable all subsequent persons who work with the software to work successfully.