Praxisprojekt Elektrotechnik

Protyptische Implementierung eines Lastausgleichs- und Steuerungssystems für Niederspannungsnetze

vorgelegt von

Johannes Kruse

Matrikelnummer: 11132595

Verantwortlicher Betreuer

Prof. Dr. Eberhard Waffenschmidt

März 2023

Fakultät für
Informations-, Medien-
und Elektrotechnik

Technology
Arts Sciences
TH Köln

Kurzfassung

Die TH Köln forscht im Rahmen des Forschungsprojektes PROGRESSUS an verschiedenen Konzepten zur Leistungsflussoptimierung im Versorgungsnetz. Verschiedene Algorithmen sollen hierfür an einem Demonstrator im Labor des Instituts geprüft werden. Als Basis für die Algorithmen soll ein Energiemanagement System (EMS) erstellt werden, dass auf einem Raspberry Pi läuft. Das EMS wird unter einigen Rahmenbedingungen entwickelt. Zum Beispiel soll die Programmiersprache Python genutzt werden, weil Python sich bei einem großen Anteil der Mitarbeiter großer Beliebtheit erfreut. Die Funktionen des EMS sollen am Ende die Datenakquise von Messgeräten, die Ansteuerung von Lasten (simuliert durch batteriegestützte Wechselrichter), sowie einen Knoten übergreifenden Datenaustausch umfassen. In dieser Ausarbeitung wird primär auf die Software eingegangen, sodass am Schluss ein gutes Verständnis für die Architektur und die Bedienung vorhanden ist, sodass andere Personen die Arbeit vorführen können.

Abstract

As part of the PROGRESSUS research project, the TH Köln is researching various concepts for power flow optimisation 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 a number of 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.

Inhaltsverzeichnis

Kurzfassung ..... **II**
Abstract ..... **II**
Inhaltsverzeichnis ..... **III**
1 Einleitung und Motivation **1**
2 Software-Anforderungen **2**
3 Software-Architektur **3**
4 Module **9**
4.1 Main 9
4.2 Webserver 12
4.3 Database 17
4.4 Zeroconf Registration 21
4.5 Zeroconf Browser 24
4.6 Data Collector 26
4.7 Data Gatherer 30
4.8 Power Meter 31
4.9 Inverter 33
5 Anleitungen **36**
5.1 Inbetriebnahme des Testaufbaus 36
5.2 Remote arbeiten (LAN) 36
5.2.1 PuTTY (Terminalzugriff unter Windows) 37
5.2.2 WinSCP (Filemanager unter Windows) 38
5.3 Nützliche Terminal Befehle 41
5.4 Zeitsynchronisation mit NTP 42
5.5 Ansteuerung der Wechselrichter 44
5.5.1 Manuell im Browser 44
5.5.2 Automatisch mit Python Skript 45
5.6 Datenbank auslesen 46
5.7 Erstellen eines Installationsskripts 47
5.7.1 Schritt 1 – Daten auf Raspberry Pi kopieren 47
5.7.2 Schritt 2 – Erstellen einer Installationsdatei mit PyInstaller 48
5.7.3 Schritt 3 – „create_installer.sh“ ausführen 49
5.8 Software deployment 50

5.8.1 Schritt 1 - Raspberry Pi OS installieren: 50
5.8.2 Schritt 2 - Ausführen des bereits erstellten Installationsskripts 51
5.8.3 Schritt 3 - Software konfigurieren mittels Config-Datei 53
5.8.4 Schritt 4 - Raspberry neustarten 54
5.8.5 Schritt 5 - Ordnungsgemäßen Betrieb prüfen (Debugging) 54
5.9 Alternative Startmethode mit Desktopverknüpfung 55
6 Fazit 58
Abbildungsverzeichnis V
Abkürzungsverzeichnis VII

1 Einleitung und Motivation

Die Energieverteilung im Stromnetz von heute ist komplexer als noch vor einem Jahrhundert. Unter Anderem sorgt die stetig wachsende Anzahl an dezentralen Energieerzeugern im Netz dafür, dass die Leistung nicht mehr nur von den großen Kraftwerken zu den Verbrauchern fließen, sondern stattdessen lokal erzeugt und über kürzere Wege verbraucht wird. Was im ersten Moment verlustärmer und damit gut erscheint, bringt leider auch einige Probleme mit sich. Weil dezentral oft nicht flächendeckend Messdaten erfasst werden, kann ein Netzzustand vom Netzbetreiber oft nur im Niederspannungsnetz geschätzt werden. Auch sind die kraftwerksfernen Leitungen deutlich kleiner dimensioniert, wodurch die Gefahr von Überlastung besteht. Durch Forschungsprojekte, wie PROGRESSUS, sollen Konzepte entwickelt werden, die das Netz auf solche Probleme vorbereiten. Das sogenanntes Smart Grids ist eine gute Alternative zum sonst teuren Leitungsausbau. Dabei werden die Akteure des Stromnetzes miteinander vernetzt, wodurch eine Optimierung des Leistungsflusses ermöglicht wird, sodass Grenzwerte von Spannungen und Lasten eingehalten werden können.

Während des Praxisprojektes an der TH habe ich in Zusammenarbeit mit der Firma Devolo eine Vernetzung der Teilnehmer in der Testumgebung realisiert. Messdaten mehrerer Netzanalysatoren können erfasst und zwischen verschiedenen Knoten ausgetauscht werden. Auf Basis der erhobenen Messdaten können die Algorithmen ein optimierten Soll-Zustand berechnen. Als Stellgrößen des Leistungsflusses stehen der Regelung mehrere batteriegestützte Wechselrichter zur Verfügung, die auf der Testumgebung verteilt sind. Durch Be- und Entladen der Batterien kann die Leistung der Regelung bidirektional eingestellt werden. Der Testaufbau selbst ist modular aufgebaut und ermöglicht es viele verschiedene Netzsituationen abzubilden. So kann z.B. die Netztopologie über eine Stecktafel angepasst werden. Ebenfalls können dort verschiedene Leitungslängen genutzt und zwischengeschaltet werden.

Die während der Arbeit entstandene Software umfasst das Monitoring, die Datenspeicherung, den automatischen Datenaustausch zwischen den Knoten und bietet Schnittstellen für Messwerte und der Ansteuerung.

Die Herausforderung sich mit der vorhandenen Problemstellung auseinanderzusetzten, mehr darüber zu erfahren und meine Programmierfähigkeiten weiterzuentwickeln, haben mich während der gesamten Arbeit motiviert und mich viel gelehrt.

2 Software-Anforderungen

Wie bereits erwähnt sind die Entscheidungen der Entwicklung eng mit den Anforderungen und Wünschen der PROGRESSUS Mitarbeiter der TH Köln verknüpft. Dies hat einen hohen Stellenwert, damit die Software anschließend auch weiterverwendet wird. Im Folgenden sind die explizit genannten Anforderungen aufgelistet.

1. Python ist als Programmiersprache zu verwenden
2. Das Programm soll jeweils auf einem Raspberry Pi laufen und möglichst performant sein, weil die Regelungen ebenfalls auf den Raspberry Pi integriert werden
3. Daten von Messgeräten sollen mindestens einmal pro Minute erfasst werden
4. Die Daten sollen historisch gespeichert werden, sodass eine spätere Analyse möglich ist
5. Daten sollen knotenübergreifend zwischen den Raspberry Pi ausgetauscht werden, sodass jeder Knoten autark Entscheidungen treffen kann. Ein zentraler Server ist nicht erwünscht.
6. Kommunikation erfolgt nur über das lokale Netzwerk. Ein Internetzugang, über das bestehende TH-Netz, steht den Raspberry Pi nicht zur Verfügung

Zu 1.:

Python erfreut sich unter Ingenieuren großer Beliebtheit. Auch an der TH Köln sind viele Mitarbeiter und Studierende mit Python vertraut, weshalb es nahe liegt diese zu verwenden. Python hegt den Anspruch gut lesbar zu sein und fördert einen knappen Programmierstil. So ist es beispielsweise nicht nötig hinter jeder Anweisung ein Semikolon zu setzen. Auch werden Anweisung innerhalb Schleifen oder If-Bedingungen nur durch Einrücken in Blöcke strukturiert, anstatt mit geschweiften Klammern umschlossen zu werden.

Python hat jedoch auch ein paar Stolperfallen, die bei Nichtbeachtung zu Problemen führen können. So sind beispielsweise Daten (anders als beispielweise in C) nicht typesafe. Python lässt ein verrechnen von ähnlichen Datentypen grundsätzlich zu. So kann ein Datentyp schnell vom erwarteten Datentyp abweichen, wenn nicht darauf geachtet wurde.

Zu 2.:

Insbesondere wurde darauf geachtet, dass die Performance der Software möglichst gut ist, damit die Regelalgorithmen später möglichst nicht eingeschränkt werden. Dies wurde durch Sleep Anweisungen des Time Moduls in jeder Schleife erreicht. Dies sorgt für eine deutlich niedrigere Auslastung des Prozessors und gibt genügend Zeit für viele nötigen Berechnungen.

Zu 2.:

Ca. einmal pro Sekunde werden neue Messwerte an alle Knoten verteilt. Bei den Wechselrichtern liegt die Datenerfassung bei weniger als drei Sekunden. Es wird also eine Zeit deutlich kleiner einer vollen Minute erreicht. Dies ermöglicht eine sehr gute Analyse des vorliegenden Netzzustandes.

Zu 4.

Für die Datenspeicherung wird eine Datenbank auf den Raspberry Pi aufgesetzt. Ich habe mich auf Empfehlung von Devolo für eine InfluxDB (Datenbank) entschieden. Sie ist schnell und für einfache Datentypen wie floats bestens geeignet. Auch bietet sie mittels „retention policies“ die Möglichkeit die Datenhaltung einzustellen. Somit kann man auch einstellen, wie lange man Daten speichern möchte und dies auf den vorhandenen Speicherplatz anpassen.

Zu 5.

Der rackübergreifenden Datenaustausch wird mit Zeroconf realisiert. Datenpakete werden ohne zusätzlich nötiges Einrichten im Netzwerk „hochgehalten“. Andere Teilnehmer können dies erkennen und die Daten abgreifen. Der Verbindungsaufbau geschieht dabei automatisch. Die Skalierbarkeit dieser Methode (hinsichtlich Netzbelastung) wurde nicht getestet, wird aber voraussichtlich nicht für große Netzwerke geeignet sein. Für den Testaufbau der TH Köln ist diese Methode jedoch gut geeignet.

Zu 6.:

Aufgrund mangelnden Internetzugangs müssen fehlende Pakete über Umwege installiert werden. Ein temporärer Internet-Hotspot über Laptop oder Mobiltelefon bietet hier Abhilfe. Ein dadurch nicht gelöstes Problem ist die Zeitsynchronisation der Raspberry Pi Systemzeit. Sobald der Raspberry Pi ausgeschaltet wird, läuft die Zeit nicht mehr mit und muss neu synchronisiert werden. Üblicherweise geschieht dies über die Internetverbindung. Devolo hat ein IoT-Gateway zur Verfügung gestellt, dass mittels eines LTE-Moduls mit dem Internet verbunden ist. Das IoT-Gateway habe ich als Zeitserver eingerichtet. Über das Netzwerk Time Protocol (NTP) werden die Raspberry Pi nun innerhalb der ersten 15 Minuten Laufzeit synchronisiert. Dies führt leider dazu, dass die ersten Messwerte unter der falschen Zeitangabe gespeichert werden, was zu Ungenauigkeiten in der Datenanalyse führt.

3 Software-Architektur

Für die Software habe ich mich von der open Source Software *OpenEMS* inspirieren lassen. Weil OpenEMS in Java geschrieben ist und damit nicht die Anforderungen der TH erfüllt, wurde es nicht

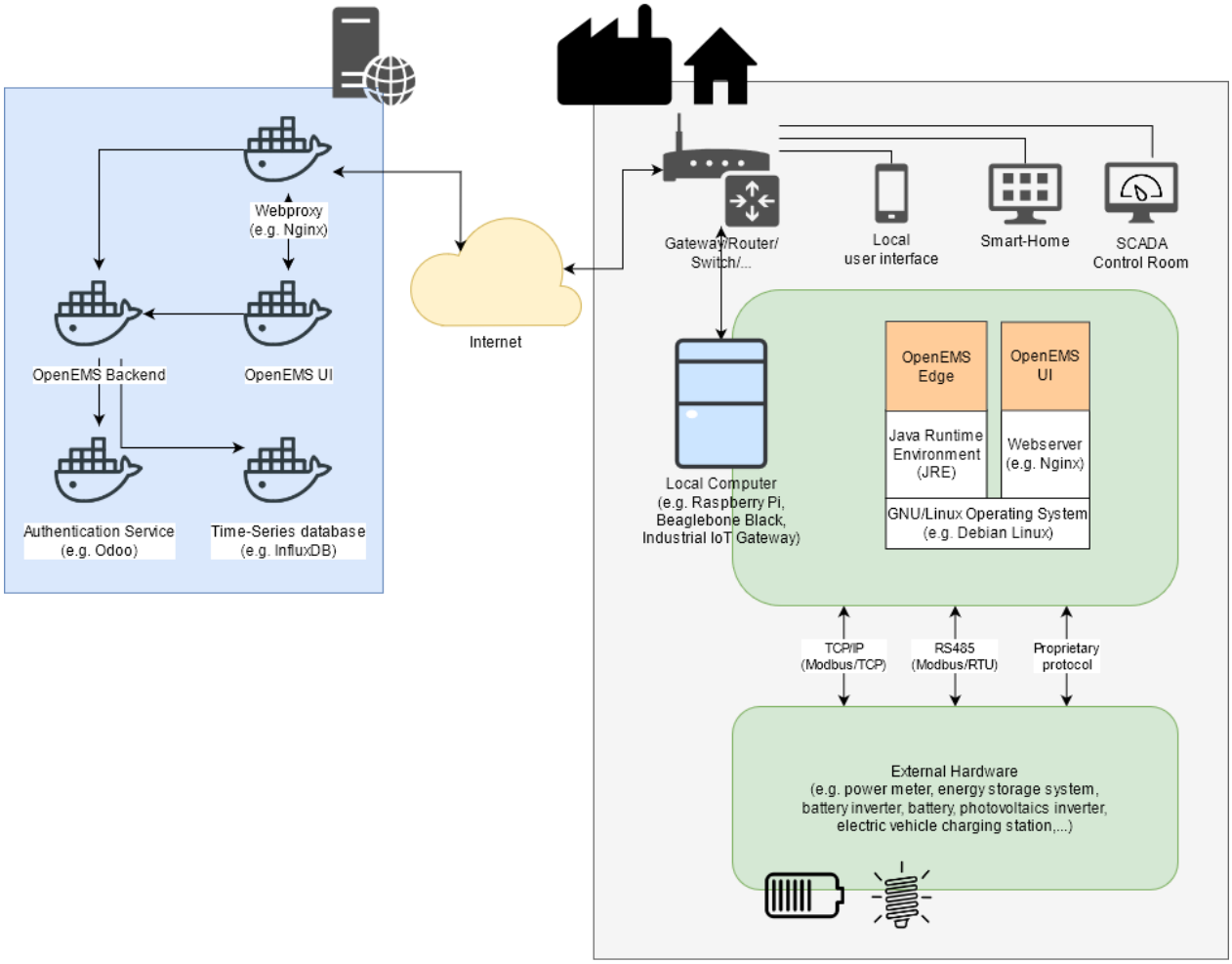


Abb. 1: OpenEMS System Architektur - **Quelle**: https://openems.github.io/openems.io/openems/latest/introduction.html (11.03.2023)

im Labor implementiert.

In Abb. 1 ist eine Übersicht von OpenEMS zu sehen. Da die Raspberry Pi nicht mit dem Internet verbunden sind, ist nur der Inhalt des rechten grauen Kastens von Relevanz. In meinem Python Programm nutze ich das Python Package Flask als Webserver. Über eine dort implementierte Representational State Transfer (REST) -Schnittstelle können Steuerbefehle, entweder über weitere Programme mittels POST-Requests oder manuell über ein Webinterface mittels eines Browser-Tabs, an die Wechselrichter gesendet werden. Außerdem ermöglicht das Webinterface das Einsehen von den aktuellen lokalen Messwerten. Externe Hardware, wie die Wechselrichter und die Netzanalysatoren, werden über das Modbus TCP Protokoll angesteuert. Um das Protokoll nicht selbst zu implementieren, habe ich hierfür das Python Package PyModbus verwendet. Für die Datenakquise wird eine Influx Datenbank auf den Raspberry Pi integriert. Diese ist für einfache Datentypen gut geeignet. Für den Datenaustausch zwischen den Racks nutze ich Zeroconf. Zeroconf oder auch Zero Configuration Networking ermöglicht, wie der Name vermuten lässt, eine

Netzwerkkommunikation ohne manuelles Eingreifen einer Person. Per Multicast werden, wie bei einem Netzwerkdrucker Erkennungsdienst (z.B. Bonjour oder Avahi), Geräte im Netzwerk erkennbar gemacht. Diese können dann von Geräten, die auf der gleichen Domain arbeiten, erkannt und bereitgestellt Daten empfangen werden.

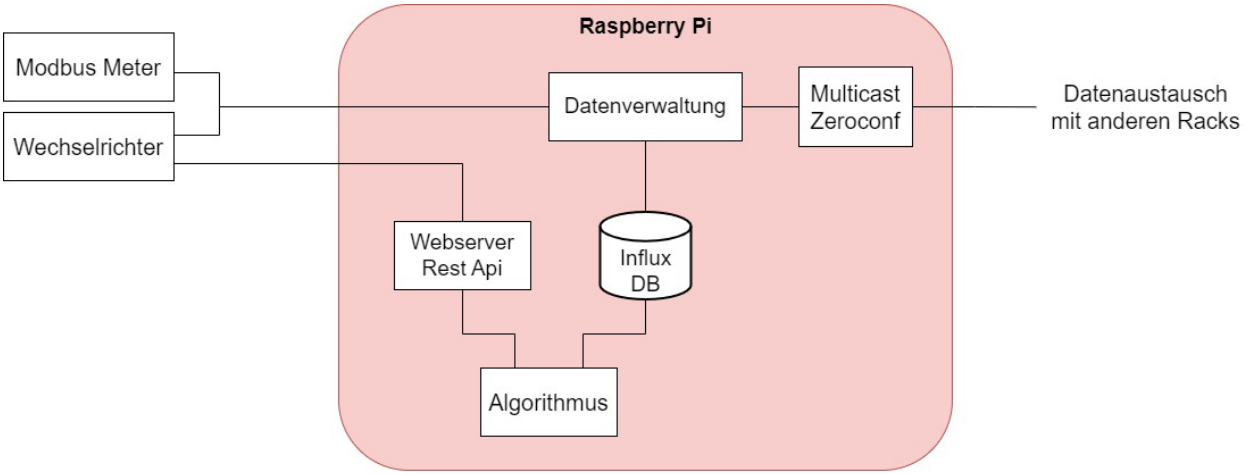


Abb. 2: schematischer Aufbau des EMS (Funktionsprinzip)

Abb. 2 veranschaulicht nochmal den zum Abschluss dieser Arbeit entwickelten schematischen Aufbau des Energiemanagementsystems der für die Regelungen genutzt werden kann.

Damit der Code übersichtlich bleibt wurden nach Möglichkeit die einzelnen Aufgaben in Module gekapselt. Entsprechend ist der Code in mehrere Python Dateien aufgeteilt. Dies ist insbesondere dann praktisch, wenn ein gezielter Aufgabenbereich angepasst werden soll.

Folgende Python Module sind zum aktuellen Zeitpunkt vorhanden:

```
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
```

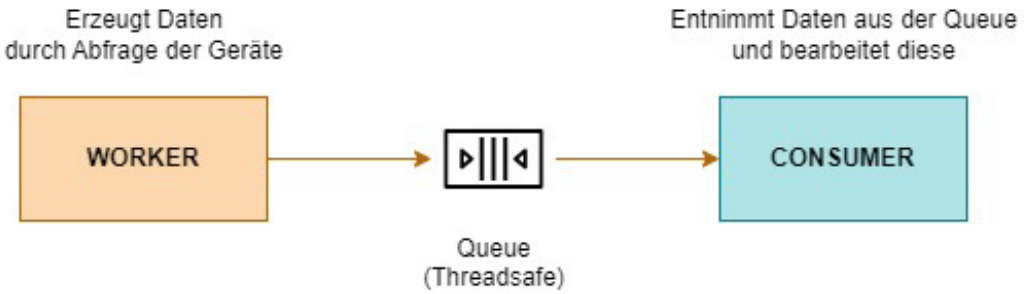


Abb. 3: Producer-Consumer-Pattern

In den späteren Unterpunkten werden die einzelnen Module erklärt.

Damit eine Vielzahl an Geräten möglichst schnell und zeitnah abgefragt werden kann, wird ein Producer-Consumer-Pattern angewandt. Ein Beispiel für das Funktionsprinzip ist in Abb. 3 zu sehen. Hierfür verwende ich das Standard Python Package *threading*. Über Queues werden Worker-Threads und Consumer-Threads threadsafe miteinander verbunden. Fehler durch Race Conditions, die durch den Zugriff von mehreren Threads auf die gleichen Variablen/Speicher hervorgerufen werden können, werden damit vermieden. Die Worker-Threads erzeugen nötige Daten, die anschließend aus den Queues von Consumer Threads entnommen und verarbeitet werden. Es ist anzumerken, dass das Python threading Modul nur auf einem Kern arbeitet. Es werden zwar mehrere Tasks zum Threadpool hinzugefügt, aber echte Parallelität ist damit nicht gegeben. Somit können Messwerte verschiedener Messgeräte nicht perfekt zeitgleich abgerufen werden. Dies wurde auch nicht weiter angestrebt und war keine Anforderung. Dennoch werden die Messgeräte hochfrequent (im Sekundenbereich) abgefragt, sodass der Netzzustand nachvollziehbar ist. Die Queues sind ebenfalls standardmäßig in aktuellen Python Versionen enthalten. Intern arbeiten die Queues mit Locks, die den Zugriff auf den Inhalt sperren, bis der bearbeitende Thread sie wieder frei gibt.

Hinweis Unerwartetes Verhalten kann mit Queues auftreten, wenn mehr produziert als verbraucht wird

Die folgende Abbildung zeigt den schematischen Aufbau und verdeutlicht, welche Aufgaben die

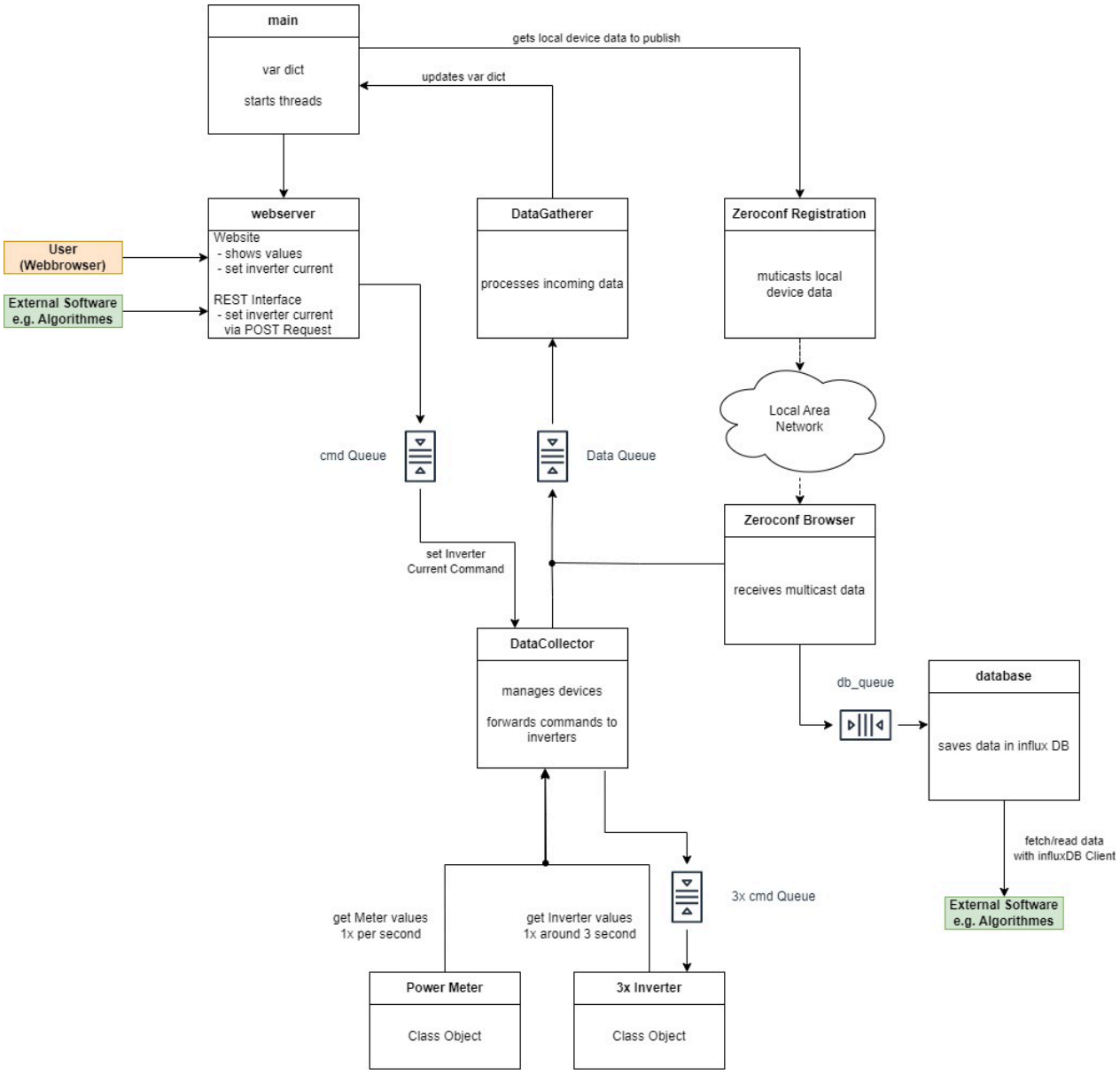


Abb. 4: schematischer Aufbau des EMS (Modulübersicht)

Module haben und wie sie miteinander kommunizieren.

Die rechteckigen Blöcke geben die verwendeten Module wieder. Dabei steht oben immer der Modulname und unten eine Kurzbeschreibung der primären Aufgaben. Ich empfehle sich auf diese Abbildung zu beziehen, wenn die Funktionen der einzelnen Module erklärt werden, um den gesamten Zusammenhang nachzuvollziehen.

In Grün und Orange sind externe Zugriffe über die vorhandenen Schnittstellen dargestellt. Anwender können manuell über ein Browserfenster mittels IP und Port (z.B. 192.168.11.43:5000 für Rack 3) auf das Webinterface zugreifen. Dort kann manuell in einem Eingabefester der Strom der lokalen Wechselrichter gesetzt werden. Außerdem können die aktuellen Messwerte eingesehen werden.

Einzig das *var dict* aus der main.py sei hier noch kurz beschrieben. Es handelt sich dabei um ein Dictionary in dem die aktuellen Messwerte zwischengespeichert sind. Es folgt einer fixen Datenstruktur und wird vom Daten Gatherer, dem Webserver und dem Zeroconf Registration Modul verwendet. Alle nutzen manuell gesetzte Locks, um einen threadsafen Prozess zu gewährleisten. Man könnte das Dictionary durch weitere Queues ersetzen. Es ist durch die zeitliche Entwicklung des Programms übergeblieben und war sehr früh entstanden. Um Arbeit in mehreren Modulen zu sparen, wurde dies nicht mehr verändert. Ein Vorteil davon ist, dass der Webserver nicht sämtliche lokalen Messwerten in einem weiteren Speicher vorhalten muss.

Das Modul cmd_list.py wird hier nicht dargestellt. Im Modul cmd_list befindet sich nur eine Klasse mit einer Enumeration für verschiedene Befehle. Das erspart ein wiederholtes Definieren von Befehlen in Modulen, die mit Befehls-Queues (cmd_queues) arbeiten.

Für externe Programme sind ebenfalls zwei Schnittstellen definiert. Die REST-Schnittstelle des Flask Webservers erlaubt es, die Wechselrichterströme mit wenigen Zeilen Code zu setzen. Aktuelle und bereits vergangene Messwerte lassen sich über einen Python InfluxDB Client, welcher Funktionen für die InfluxDB bereitstellt, abrufen.

Hinweis Weitere Informationen zu den Modulen folgen in den entsprechenden Modulkapiteln.

4 Module

In diesem Kapitel werden die verschiedenen Module detaillierter beschrieben, sodass ein besseres

> **Hinweis** Die Python Skripte sind mit zahlreichen Kommentaren versehen und Vieles lässt sich direkt im Code nachvollziehen

Verständnis der Software-Arbeitsweise vorhanden ist.

4.1 Main

Das main,py Modul ist der Einstiegspunkt des Programms. Hier werden die Startparameter gelesen, die Einstellungen der config.json Datei gelesen, der Logger vorbereitet und die weiteren Modul-Threads gestartet.

Struktogramm:

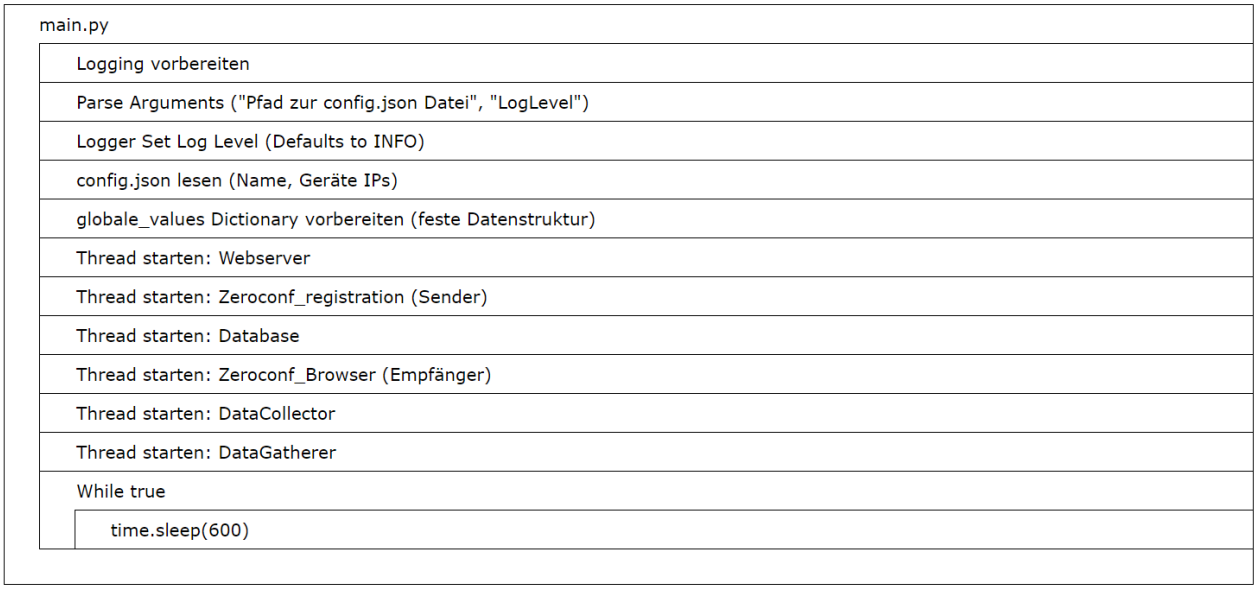


Abb. 5: Struktogramm - main.py

Oberhalb ist ein Struktogramm zu sehen aus dem der grobe Arbeitsablauf ersichtlich ist. Für Einzelheiten muss der Quellcode betrachtet werden. Es sei nochmal erwähnt, dass im Code viele Kommentare platziert sind, die zum Verständnis beitragen.

Zwei Startargumente sind im Code implementiert.

- - c oder –config. Gibt den Pfad zur config.json Datei an.
  Default ist /etc/config.json

- -d oder –debug. Gibt das LogLevel als String Parameter vor.
  Auf Groß- & Kleinschreibung muss **nicht** geachtet werden. Möglich ist: DEBUG, INFO, WARNING, CRITICAL oder ERROR.
  Default ist INFO

Beispiel für ein Terminalbefehl mit non-default Startparametern mit dem der Code ausgeführt werden kann:

```
python main.py -c config.json -d "WARNING"
```

In diesem Fall wird im aktuellen Verzeichnis nach der config.json Datei gesucht und nur Logeinträge mit einem Log Level von mindestens WARNING geloggt.

> **Hinweis** Verschiedene Verfahren zum Programmstart werden in den Anleitungskapiteln beschrieben

Mit Hilfe einer Config Datei werden einige Parameter beim Programmstart mitgegeben, die nicht automatisch ermittelt werden. Die config.json Datei (üblicherweise in /etc/config.json abgelegt) hat folgenden Inhalt im JSON Datenformat:

```
{
    "Name": "rack_2",
    "IP_Power_Meters": [
        "192.168.11.62"
    ],
    "IP_Inverters": [
        "192.168.11.12",
        "192.168.11.22",
        "192.168.11.32"
    ]
}
```

Der Name (im Bsp. „rack_2“) wird an mehreren Stellen im Programm verwendet. Im Webserver gibt er den Namen des Browser Tabs an. Für Zeroconf dient er als Gerätenamen, mit der die Daten unter der Zeroconf-Domain veröffentlicht werden. Ich habe setzts die übliche Rack Nummer als Namen angegeben, also rack_1 bis rack_6.

> **Achtung** Es ist zwingend darauf zu achten, dass der Name einzigartig ist! Ist dies nicht der Fall, stürzen die Zeroconf Threads mit entsprechendem Fehler ab.

> **Hinweis** Die Phasenzuordnung der Wechselrichter erfolgt in der Config Datei! Es ist deshalb wichtig die IPs in der Reihenfolge L1, L2 und dann L3 anzugeben.

Die Listen mit den IPs geben die entsprechenden Geräteadressen an. Mit den Powermetern sind die Netzwerkanalysatoren von PQ Plus gemeint. Inverters steht für die Wechselrichter. Das Programm ist so eingestellt, dass dynamisch die Anzahl der Geräte entsprechend der Anzahl der vorgegeben IPs angepasst wird. Es lassen sich allerdings nur die ersten drei Wechselrichter ansteuern. Sollten mehr Wechselrichter in einem Rack gewünscht sein, dann müssen entsprechende Endpunkte im Webserver manuell hinzugefügt werden. Während meiner Zeit an der TH war die Anzahl der Wechselrichter und Powermeter an einem Rack statisch und nie größer drei, deshalb wurde die Umwandlung in eine vollständige dynamische Implementation nicht weiterverfolgt und finalisiert. Grundsätzlich ist dies aber möglich.

Das globale_values Dictionary wird genutzt, um die aktuellen Messwerte aller Messgeräte zwischenzuspeichern. Der Webserver, DataGatherer und der Zeroconf Browser arbeiten mit dem Dictionary. Für ein threadsafes Lesen und Schreiben wurde manuell ein Lock in den Threads implementiert. Auf eine Queue wurde abgesehen, auch wenn dies theoretisch möglich wäre. Dann müsste die main jedoch regelmäßig Daten an den Webserver und an den Zeroconf Browser senden, sowie die Daten vom DataGatherer entgegennehmen. Außerdem müssten alle lokalen Messwerte im Webservermodul zwischengespeichert werden, wobei die Datenmenge relativ gering ist.

Beim Start der Threads werden benötigte Objektreferenzen mit übergeben. Anschließend wechselt die main main in einen idle State ohne weitere Funktionalität.

Imports:

```
# standard Module
import logging
import sys
from threading import Lock
import queue
import time
import json
import argparse

# eigene Module:
from cmd_list import CmdEnum
import DataCollector
import DataGatherer
import zeroconf_browser
import zeroconf_registration
import webserver
import database

# extra:
import requests    # wird genutzt, um den Webserver herunterzufahren
```

4.2 Webserver

Der Webserver dient als Schnittstelle zwischen Benutzern und externen Anwendungen und der Software. Er ermöglicht das Setzen von Wechselrichter Strömen und ermöglicht eine manuelle Einsicht von aktuellen Messwerten der lokal angeschlossenen Geräte. Für den Webserver wurde das Python Modul *Flask* verwendet. Flask ist abhängig von der Jinja Vorlage und dem Werkzeug Python Web Server Gateway Interface (WSGI) Toolkit. Alle Anforderungen an den Webserver werden von Flask abgedeckt und sind mit relativ geringem Aufwand implementierbar.

Struktogramm:

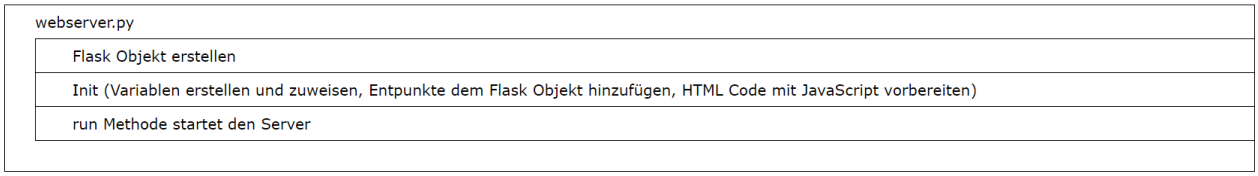


Abb. 6: Struktogramm - webserver.py

Aus der main übergeben wird der Rack Name als String Parameter, ein Command Queue Objekt das auch zum Data Collector übergeben wird, eine „Referenz“ zum global_values Dictionary der main und das Lock Objekt zum Sperren des global_values Dictionary, damit keine Fehler durch gleichzeitigen Zugriff entstehen. Wie bei allen weiteren im Thread gestarteten Klassen, erbt die Webserver Klasse vom threading Modul und bekommt eine überlagerte run Methode für den Start des Threads. Die Ausführungsreihenfolge ist folgende:

- Klassen Objekt wird erstellt (Aus Main Modul initiiert)
    - Flask Objekt wird erstellt
    - Init Methode wird ausgeführt
- run Methode wird ausgeführt (aus Main Modul initiiert)

Weil das Flask Objekt ebenfalls eine run Methode besitzt und der Thread gehalten wird bis der Webserver beendet wird und zusätzlich keine Exit Aufgaben erfüllt werden müssen, muss die run Methode nicht in eine Dauerschleife (Idle State) übergehen.

Der Webserver wird mit der localhost IP ausgeführt. Zum aktuellen Stand sind die IPs für die Raspberry Pi „192.168.11.41“ bis „192.168.11.46“, wobei die letzte Ziffer die Racknummer angibt. Die Webserver laufen über den Flask Standard-Port „5000“.

Die Adresse für Rack 1 ist somit beispielsweise „192.168.11.41:5000“.

Der HTML-Code mit JavaScript ist als mehrzeiliger String (“““) im Code hinterlegt und **nicht** in einer externen Datei ausgelagert. Dies hielt ich für nicht nötig, weil relativ wenige Funktionen vorhanden sind und die meisten davon nur Kopien sind. Außerdem lässt sich so der gesamte Webserver in einer Datei bearbeiten. Der HTML String wird in der Init Methode angepasst, sodass der Rack Name, welcher ursprünglich aus der *config.json* Datei stammt, übernommen wird.

Auf das wesentliche heruntergebrochen sieht das JavaScript im String wie folgt aus:

```
1  <script>
2  async function meter() {
3   let options = { method: 'GET'}
4   var txt = "";
5   const response = await fetch('/data');
6   const dJson = await response.json();
7   myJson = dJson;
8     txt += "<table width=\x22100%\x22>";
9     txt += "<tr><th align=\x22left\x22> Timestamp </th> <td
    align=\x22left\x22>"+myJson["timestamp_unix"]+" </td></tr>";
10    txt += "<tr><th align=\x22left\x22> GRID_VOLTAGE </th><td
    align=\x22left\x22>"+myJson["GRID_VOLTAGE"]+" V</td></tr>";
11    txt += "<tr><th align=\x22left\x22> GRID_CURRENT </th> <td
    align=\x22left\x22>"+myJson["GRID_CURRENT"]+" A</td></tr>";
12    txt += "<tr><th align=\x22left\x22> BATTERY_VOLTAGE </th> <td
    align=\x22left\x22>"+myJson["BATTERY_VOLTAGE"]+" V</td></tr>";
13    txt += "<tr><th align=\x22left\x22> BATTERY_CURRENT </th> <td
    align=\x22left\x22>"+myJson["BATTERY_CURRENT"]+" A</td></tr>";
14    txt += "<tr><th align=\x22left\x22> TEMPERATURE </th> <td
    align=\x22left\x22>"+myJson["TEMPERATURE"]+" °C</td></tr>";
15    txt += "</table>";
16 document.getElementById("demo").innerHTML = txt;
17 }
18 setInterval(meter, 3000);
19 </script>
```

Zeile 1 und 19 zeigen den Start und das Ende des JavaScripts im HTML Code. Die asynchrone Funktion meter() ermittelt wird genutzt, um die angezeigten Werte des Webinterfaces zu aktualisieren. In der Variable *txt* werden wird hierfür ein HTML Text generiert. In Zeile 5 wird der Endpunkt /data aufgerufen, welcher die Methode get_data der Webserverklasse aufruft und ein JSON String zurück gibt. Die Zeile 6 ist eventuell überflüssig, aber stellt nochmals sicher, dass es sich um das JSON- Format handelt. Die Zeilen 8 bis 15 erzeugen den HTML Code für die Tabelle mit den entsprechenden zugehörigen Werten. In Zeile 16 wird dieser Text dann aktualisiert, sodass der Benutzer das Ergebnis sieht. Im tatsächlichen Code wurden Zeile 4 bis 16 kopiert und die Variablennamen entsprechend indiziert, sodass alle vier Geräte (3x Wechselrichter, 1x Netzanalysator) abgerufen werden. Mit der setInterval Methode in Zeile 18 am Ende des JavaScripts lässt sich die Aktualisierungsrate des Browser Tabs einstellen. Diese wurde auf 3 Sekunden

Hinweis Die Belastung des JavaScripts tritt nur bei aufgerufenem Browser Tab auf. Ansonsten wird keine zusätzliche Leistung benötigt.

eingestellt, damit der Webserver den Prozessor wenig belastet.

Imports:

```
# standard Module
import logging
import threading
import datetime

# eigene Module
from cmd_list import CmdEnum

# extra:
from flask import Flask, jsonify, request
```

Methodenübersicht:

```
1   __init__(self, rack_name, cmd_queue, global_values, mutex)
2   get_page(self)
3   get_data1(self)             # Wechselrichter
4   get_data2(self)             #     ...
5   get_data3(self)             #     ...
6   get_data4(self)             # Netzanalysator
7   set_power1(self)
8   set_power2(self)
9   set_power3(self)
10  run(self)
11  shutdown_server(self)
```

Die *__init__* Methode setzt die entsprechenden Parameter und Variablen. Mutex bezeichnet das Lock Objekt des Threading Moduls, womit der Zugriff auf das global_values Dictionary des main Moduls temporär für andere Threads gesperrt wird.

Die *get_page* Methode wird beim Aufruf der Webserver Adresse ausgeführt und gibt den HTML-Code zurück, sodass die Seite erfolgreich angezeigt wird.

Die Methoden *get_data1* bis *get_data4* werden durch das JavaScript aufgerufen und geben die aktuellen Messgerätewerte im JSON Format zurück. Dieser dann wie bereits erwähnt durch das JavaScript auf der angezeigten Webseite im drei Sekunden Turnus aktualisiert. Der Ablauf innerhalb der Methode ist wie folgt:

- Das globale_values Dictionary nach dem Gerät entsprechenden Gerät durchsuchen
- Wenn gefunden, sonst „NaN“ Werte zurückgeben:
    - Aktuellen Datensatz kopieren und währenddessen das Dictionary sperren
    - Aus der Kopie die Werte auf zwei Stellen nach dem Komma Runden (Besser für die Darstellung)
    - Den Unix Zeitstempel in ein lesbares Format konvertieren
    - Das Daten Dictionary in JSON konvertieren und zurückgeben

Die Methoden *set_power1* bis *set_power3* sind als POST Endpunkte definiert und können genutzt werden, um die Wechselrichter Ströme für Phase L1 bis L3 zu setzten.

Hinweis Die Reihenfolge in der Config.json Datei ist hierfür relevant! Die IPs sollten deshalb möglichst in der Phasenreihenfolge 1 bis 3 angegeben werden.

Die *set_power* Methoden haben folgenden Ablauf:

- Request wird als ASCII String decodiert.
- Der Text wird an „=“ Zeichen getrennt.
- Das zweite Teil, also der Teil hinter dem ersten gleich Zeichen als float Datentyp gecastet und wird als Payload in einem SET_POWER_WR Command Element eingesetzt und an die

> **Hinweis** Eine Überprüfung von Grenzwerten findet erst im Inverter Modul statt. Grundsätzlich ist als float castbare Wert hier erfolgreich.

Befehls Queue zum Data Collector geschickt.

Das Command (CMD) Element als folgendes Dictionary mit zwei Keys strukturiert:

```
cmd_element = {'c_enum': CmdEnum.SET_POWER_WR1, 'payload': 0}
```

Wobei der Key “c_enum“ ein Wert aus dem cmd_list Modul bekommt. In “payload“ wird der zu setzende Strom-Sollwert übergeben.

> **Hinweis** Der Stromwert wird AC-seitig angegeben. Die resultierenden Stromwerte sind jedoch nicht 100 % genau und haben eine geringe Abweichung.

Die *run* Methode ist eine überlagerte Methode des vererbten threading Moduls. Sie führt wiederum die run Methode des Flask Objektes aus. Der *debug* Parameter kann wahlweise auf False oder True gesetzt werden. Ich habe den Wert auf True belassen, sodass Kommunikation mit dem Webserver später geloggt wird und ersichtlich ist. Auf False werden Return Werte und Requests nicht mit geloggt. Beim Start des Webservers wird ebenfalls eine Warnung geloggt, bei der erklärt wird, dass es sich um einen Development Server und nicht dieser nicht für die Produktion geeignet ist. Falls dies stört muss sich jemand mit dem Flask Webserver tiefer auseinander setzen oder einen anderen Log Handler an das Flask Objekt übergeben. Letzteres entfernt jedoch doch Möglichkeit des Debugging.

Zuletzt gibt es noch die *shutdown_server* Methode. Diese kann benutzt werden, um den Server Thread „sanft“ zu beenden. Dabei wird werden die zuletzt eingegangen Requests noch abgearbeitet und anschließend der der Server beendet. Dies wird nur bei manueller Nutzung des Programms in einem Terminal relevant, weil die Funktion nur ausgeführt wird, wenn ein Keyboard interrupt Fehler im Main Modul eingeht. Üblicherweise wird dem Raspberry Pi der Strom weggeschaltet und dieser abrupt heruntergefahren. Ein sicheres Herunterfahren ist somit nicht möglich.

4.3 Database

Das database Modul nimmt hat als Input eine Datenqueue. Die Daten werden dann vordefiniert strukturiert (je nach Gerät) und in die Datenbank geschrieben. Wie alle anderen Module, wird die Datenbank als Thread gestartet und erbt vom threading-Modul.

Imports:

```
# standard Module:
import logging
import time
from datetime import datetime
import threading

# extra:
from influxdb import InfluxDBClient
```

InfluxDBClient ist ein Python Wrapper für die meisten Influx DB Funktionen. Er umfasst alle benötigten Interaktionen mit der Datenbank, wie dem Erstellen, das Abrufen und dem Schreiben in die Datenbank.

> **Hinweis** Das später beschriebene Installationsskript installiert das benötigte Paket systemweit mit, sodass andere Programme nur das Modul importieren müssen

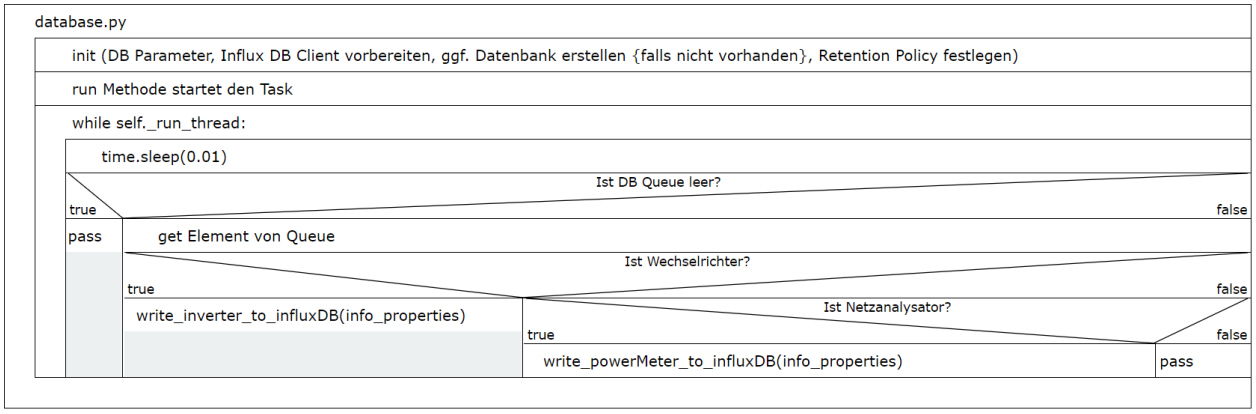


Abb. 7: Struktogramm - database.py

Struktogramm:

Das Klassenobjekt wird wieder über die main instanziiert und über die run Methode gestartet.

Beim Erzeugen wird die Init Methode ausgeführt und einige DB-Parameter und der Client eingerichtet. Dabei wird die Datenbank Adresse festgelegt, sowie Benutzer, Passwort und der

> **Hinweis** Datenbank spezifische Parameter müssen hier eingestellt werden.
> Ggf. können ausgewählte Parameter in die Config.json Datei übernommen werden

Datenbankname. Auch wird die Retention Policy eingestellt. Ich habe sie auf 52 Wochen (ein Jahr) festgelegt.

Mit der run Methode wird der Thread gestartet und ggf. Datensätze verarbeitet, die über eine Queue dem Thread übergeben werden.

Methodenübersicht:

```
1  __init__(self, db_queue)
2  run(self)
3  stop(self)
4  write_inverter_to_influxDB(self, info_properties)
5  write_powerMeter_to_influxDB(self, info_properties)
```

In der *__Init__* Methode werden nötige Parameter in Attributen des Klassenobjektes gespeichert und dort definiert. Dazu gehören folgende Wertepaare:

```
self._run_thread = True
self._db_queue = db_queue
self._host = 'localhost'
self._port = 8086
self._user = 'root'
self._password = 'root'
self._dbname = 'progressus'
```

Die boolesche Variable _run_thread dient als Exit Bedingung für die Dauerschleife des run Threads. Damit kann der Thread aus der main beendet werden. Wie bereits erwähnt wird der Thread üblicherweise durch das Entfernen der Stromzufuhr beendet, weshalb dies selten zum Einsatz kommt.

> **Hinweis** Man hätte hierfür auch den Quit Command des cmd_list Moduls nutzen können.

Über die _db_queue werden Gerätedaten in Form eines Dictionary an das Modul gesendet. Die Variablen *_host* und *_port* definieren die Adresse der Datenbank. Da jedes Modul die Werte im eigenem System speichern soll ist hier die *localhost* als IP angegeben. Als Port wurde der Standard von Influx DB *8086* genommen. Die Variablen *_user* und *_password* sind als root angegeben. Der Client verlangt nach den Parametern, aber beim manuellen Erstellen einer Datenbank über das mitinstallierte Command Line Interface (CLI) wird nicht danach gefragt. Es kann also sein, dass diese Parameter irrelevant sind. Die Variable *_dbname* hingegen ist wichtig. Sie gibt den Datenbanknamen an. Ich habe stets den selben Namen *progressus* verwendet, Somit werden alle Werte in eine lokale Datenbank geschrieben und nicht in mehrere Teile aufgetrennt.

Der Client wird dann wie folgt instanziiert:

```
_client = InfluxDBClient(_host, _port, _user, _password, _dbname)
```

Anschließend wird mit folgender Methode eine Datenbank erstellt:

Hinweis Sollte bereits eine Datenbank mit dem Namen existiert, dann ignoriert der Client den Befehl

```
_client.create_database(_dbname)
```

Zuletzt wird noch eine Retention Policy eingestellt mit dem Befehl:

```
_client.create_retention_policy(
        'monitoring',
        '52w',
        '1',
        'progressus',
        True,
        '0s'
)
```

Der erste Parameter *monitoring* legt den Namen für die Retention Policy fest.

Der zweite Parameter gibt die Dauer der Speicherung an. Unterstützte Parameter sind h (Stunden), m (Minuten), d (Tage), w (Wochen).

Hinweis Das Retention Policy Minimum beträgt 1h.
Das Maximum ist unendlich und kann mit ‘INF‘ (infinite) angegeben werden

Achtung Da der Speicherplatz der SD-Karte endlich ist, kann eine lange Retention Policy zu einem Fehler führen.

Der dritte Parameter *‘1‘* steht für die Anzahl an Replications. Er legt fest wie viele unabhängige Kopien von jedem Datenpunkt innerhalb eines Clusters gespeichert werden. Ich habe ihn mit 1 festgelegt, da wir nicht mit weiteren Instanzen, wie bspw. einer Cloud arbeiten.

Der vierte Parameter *‘progressus‘* gibt an für welche Datenbank die Retention Policy erstellt werden soll. In unserem Fall ist das *progressus*.

Der fünfte Parameter legt fest, ob die Retention Policy als Standard festgelegt werden soll. Damit die Retention Policy übernommen wird habe ich *True* gewählt.

Der letzte Parameter ‘0s‘ stellt die shard_duration ein. Die gleichen Zeitformate wie beim zweiten Parameter werden unterstützt. 0s wird von der Datenbank als Standarddauer interpretiert und wurde von mir gewählt. Shard Groups hat Einfluss auf die interne Speicherung der Datenbank. Die komprimierten Daten werden in Shard gespeichert. Und Shard Groups fassen mehrere dieser Shard zusammen.

Die *stop* Methode kann genutzt werden um die _run_thread Flag auf False zu setzten und damit den Thread nach aktuellen Schleifendurchlauf zu beenden.

Die Methode write_inverter_to_influxDB und write_powerMeter_to_influxDB strukturieren die Daten entsprechend dem Gerät um und schreiben sie mittels Client in die Datenbank.

Im Folgenden ist die Funktion für den Wechselrichter beispielhaft (aufs wesentliche reduziert) dargestellt:

```
def write_inverter_to_influxDB(info_properties):
    json_payload = [] # Liste in der Alle Werte zwischengespeichert werden
    data = {          # strukturiertes Dictionary mit aktuellen Messwerten
        "measurement": "monitoring",
        "tags": {
            "host": info_properties['rack_name'],
            "device": "INVERTER",
            "IP": info_properties['port']
        },
        "time": str(info_properties['timestamp_unix']),
        "fields": {
            "AC_VOLTAGE": float(info_properties['GRID_VOLTAGE']),
            "DC_VOLTAGE": float(info_properties['BATTERY_VOLTAGE']),
            "AC_CURRENT": float(info_properties['GRID_CURRENT']),
            "DC_CURRENT": float(info_properties['BATTERY_CURRENT']),
            "TEMPERATURE": float(info_properties['TEMPERATURE'])
        }
    }
    json_payload.append(data)
    _client.write_points(json_payload) # send all data to InfluxDB
```

In *info_properties* befinden sich ein Dictionary mit allen Messwerten des Gerätes. Diese werden in das data Dictionary mit DB passender Struktur geschrieben.

Das Dictionary besteht ausfolgenden Keys:

- measurement
- tags
- time
- fields

Der Key *measurement* gibt den Namen für den Messwert an. Ich habe alle Messwerte immer unter dem Namen *monitoring* gespeichert. Ggf. hätte man zwischen den Messgeräten (Inverter & PowerMeter) unterscheiden können.

Stattdessen habe ich für die Unterscheidung *tags* verwendet. Tags sind Schlüsselwörter, die für eine spätere Filterung genutzt werden können. Eine beliebige Anzahl vorhandener Tags kann im Suchstring angegeben werden und alle übereinstimmenden Ergebnisse werden zurückgegeben. Die genutzten Tags sind host, device und IP.

In *time* wird die Zeit angegeben, unter der die Datenbank den Wert aufnimmt. Sollte bereits ein Wert mit selbem Zeitstempel existieren wird dieser durch den neuen Datensatz ersetzt.

Unter *fields* werden die Messwerte als Key – Value Dictionary mitgegeben.

Zuletzt wird die neu entstandene Datenvariable in eine Liste gepackt, weil die Datenbank nur eine Funktion für mehrere Messpunkte enthält. Die Liste enthält dann nur ein Element, welches dann in die Datenbank geschrieben wird.

4.4 Zeroconf Registration

Zeroconf ist in zwei Module aufgeteilt (Registration & Browser). Der Registration Teil veröffentlicht neue Datenpakete mit aktuellen Geräteinformationen im Netzwerk und kann als Transmitter verstanden werden. Dabei ist keine zusätzliche Einstellung eines Admins nötig. Zeroconf ist ein Erkennungsdient der ohne manuelle Konfiguration auskommt. Auf dem gleichen Prinzip arbeiten auch Netzwerkdrucker, die sich im Netz ankündigen müssen. Es ist sogar so, dass einige Druckerdienste wie bspw. Apples Bonjour mit Zeroconf kompatibel sind.

Imports:

```
# standard Module
import logging
import time
import threading
import socket
import pickle

# extra:
from zeroconf import IPVersion, ServiceInfo, Zeroconf
```

Das Python Modul pickle wird nur in den Zeroconf Modulen verwendet. Es ermöglicht das Komprimieren von Python Objekten in Form von Byte Streams. Dies ist nötig, da die maximal

> **Hinweis** Sollte das Datenlimit doch irgendwann erreicht werden, dann ist eine mögliche Lösung, die Datenpakete in Teile aufzuteilen und zu versenden. Dabei wird eine Reduktion der Austauschfrequenz in Kauf genommen.

Datengröße von Zeroconf 256 Bytes nicht überschreiten darf.

Struktogramm:

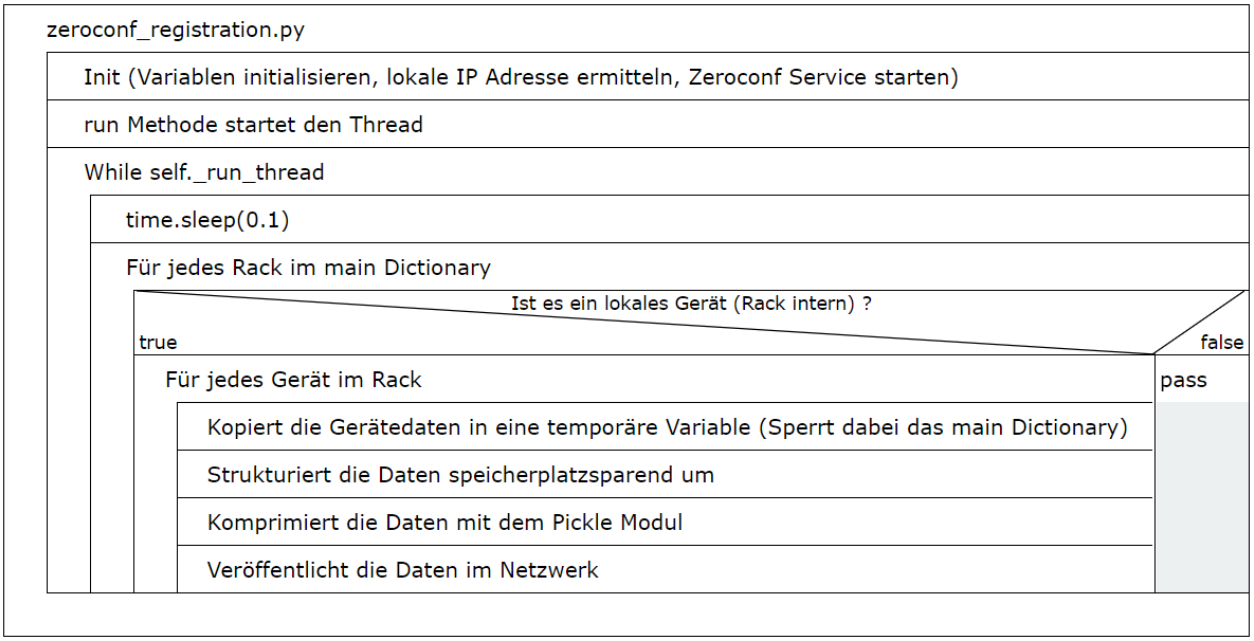


Abb. 8: Struktogramm - zeroconf_registration.py

Beim Erstellen des Objektes wird die Init Methode aufgerufen. Dabei werden nötige Variablen erstellt und die lokale IP-Adresse mittels Socket bestimmt. Der anschließend gestartete Zeroconf Service benötigt die genaue IP-Adresse. Die Angabe von localhost reicht nicht aus.

Wird der Thread mit der run Methode gestartet, dann geht der Thread in eine Dauerschleife über. Darin werden zyklisch alle lokalen Geräte, des in der main befindlichen Dictionary, durchlaufen und die aktuellen Werte im Netzwerk veröffentlicht. Davor werden die Daten komprimiert, damit das

Hinweis Ein Python Dictionary „reserviert“ einen definierten Speicherplatz, welcher erst bei der Überschreitung erweitert wird. Dadurch kann es sein, dass die Speichernutzung unverändert erscheint, sobald ein neuer Wert hinzugefügt wurde

Hinweis Die Funktion *sys.getsizeof( )* ermittelt die Größe in Bytes

Datenlimit von 256 Bytes nicht erreicht wird.

Methodenübersicht:

```
1  __init__(self, global_values, name, mutex)
2  __register_service__(self)
3  update(self, local_measurements)
4  run(self)
5  stop(self)
```

```
6   __del__(self, info)
```

Wie bereits erwähnt, werden beim Erstellen des Objekts über die *__Init__* Methode mehrere Variablen gesetzt. Dazu gehören:

```
self._run_thread = True
self._name = name
self._global_values = global_values
self._mutex = mutex
```

Ebenfalls wird die IP-Adresse in der *__init__* ermittelt. Diese wird für den Zeroconf Service benötigt. Genutzt wurde dafür das *Socket* Modul:

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 80))
self._local_ip = s.getsockname()[0]
```

Anschließend wird die Methode *__register_service__( )* ausgeführt und der Zeroconf Service wie folgt gestartet werden:

```
register_service__(self):
      self._info = ServiceInfo(
          "_monitoring._tcp.local.",
          self._name + "._monitoring._tcp.local.",
          addresses=[socket.inet_aton(self._local_ip)],
          port=80,
          properties={},
          server="progressus.local.",
      )
      self._zeroconf_service_ = Zeroconf(ip_version=self._ip_version)
      self._zeroconf_service_.register_service(self._info)
```

Die Funktion *ServiceInfo* des *zeroconf* Moduls bereitet den zu registrierenden Service vor. Der erste Parameter gibt den Typ des Zeroconf Services vor. Ich habe *_monitoring._tcp.local.* gewählt. Die Trennung mit Punkten ist nicht zufällig gewählt, sondern vorausgesetzt. Abweichungen resultieren in einer entsprechende Fehlerausgabe. Der zweite Parameter ist der Name des Services innerhalb der

Achtung Der Name muss einzigartig sein, damit kein Fehler erzeugt wird.
Der Rack Name bietet sich an, weil er jeweils unterschiedlich ist

Typ Domain. Hierfür habe ich den Rack Namen verwendet.

Weitere Parameter können den offiziellen Docs von python-zeroconf entnommen werden. Quelle: https://python-zeroconf.readthedocs.io/en/latest/api.html (Stand: 14.03.2023)

Nachdem der Thread initialisiert wurde, kann er mit der run Methode gestartet werden. In einer While-Schleife werden zyklisch neue Messwerte mit der update Methode veröffentlicht. In der

Methode werden die Messwerte komprimiert und mittels der _update_service( ) Funktion des Zeroconf Services im Netzwerk bereitgehalten.

Die *stop* Methode kann wieder zum Stoppen des Threads genutzt werden.

Der Destruktor *__del__* wurde überladen und sorgt für das Abmelden und Schließen des Services.

4.5 Zeroconf Browser

Der Zeroconf Browser ist der zweite Teil von Zeroconf und stellt den Empfänger dar. Der Browser reagiert, sobald ein registrierter Service seinen Status verändert. Die Daten (auch Properties genannt) können dann abgerufen werden. Anschließend werden die Daten für das global_values Dinctionary passend strukturiert. Zum Schluss werden die Daten in über eine Queue an den Data Gatherer geschickt. Sollte es sich um lokale Geräte handeln, dann werden die Daten über eine weitere Queue auch an das Database Modul weitergeleitet.

Imports:

```
# standard Module:
import logging
from time import sleep
import threading
import pickle

# extra:
from zeroconf import IPVersion, ServiceBrowser, ServiceStateChange,
                                    Zeroconf, ZeroconfServiceTypes
```

Struktogramm:

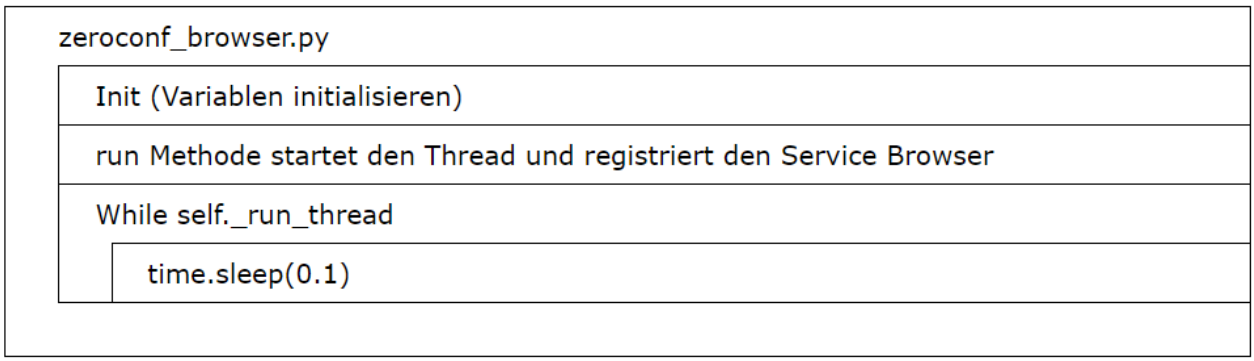


Abb. 9: Struktogramm - zeroconf_browser.py

Der Zeroconf Browser arbeitet ähnlich dem Registrator. Sobald das Modulobjekt erstellt ist und der Thread gestartet wurde, wird ein Service Browser Objekt bzw. Client erstellt. Dabei wird als Handler eine Methode übergeben, die automatisch aufgerufen wird, sobald ein Zeroconf Event erkannt wird. Nachdem der Zeroconf Browser gestartet wurde, geht der Zeroconf Browser in eine Dauerschleife

Hinweis Die Sleep Zeit wurde geringer als im Main Modul gewählt, weil die Main die Möglichkeit haben soll, den Thread ohne langes Warten zu beenden

mit time.sleep Funktion über (Idle State).

Methodenübersicht:

```
1   __init__(self, data_queue, db_queue, rack_name)
2   on_service_state_change(
       self,
       zeroconf: Zeroconf,
       service_type: str,
       name: str,
       state_change: ServiceStateChange
    ) -> None
3   run(self)
4   stop(self)
```

In der ***__init__*** Methode werden folgende Variablen festgelegt:

```
self._run_thread = True
self._data_queue = data_queue
self._db_queue = db_queue
self._info_name = rack_name + "._monitoring._tcp.local."
```

Die ersten drei Variablen werden wie zuvor verwendet. ***_info_name*** wird in der Handler Methode genutzt, um lokale Geräte zu erkennen. Der String ist dafür identisch wie der registrierte Service Name im Zeroconf Registrator Modul.

Innerhalb der *run* Methode wird der Service Browser wie folgt aktiviert.

```
ip_version = IPVersion.V4Only
zeroconf = Zeroconf(ip_version=ip_version)
services = ["_monitoring._tcp.local.", "_hap._tcp.local."]
browser = ServiceBrowser(
            zeroconf,
            services,
            handlers=[self.on_service_state_change]
)
```

Der Parameter *zeroconf* der Funktion *ServiceBrowser* ist eine Zeroconf Instanz. Der zweite Parameter *services* gibt wieder einen qualifizierten Service Typ Namen an. Dieser ist identisch mit dem des Registrators, sodass die Werte gefunden werden können. Mit den *handlers* werden aufrufbare Funktionen übergeben, die *ServiceStateChange* Events bearbeiten können.

Sobald der *ServiceBrowser* instanziiert ist, wartet das Browser Modul auf Events, die zum Aufruf der *ServiceStateChange* Methode führen.

Mit stop Methode kann die Dauerschleife abgebrochen werden.

Der Großteil der gewünschten Funktionalität wird in der on_service_state_change *Methode* umgesetzt. Der Datenerfassung ist komplexer, weil viele If-Abfragen existieren. Für eine besser

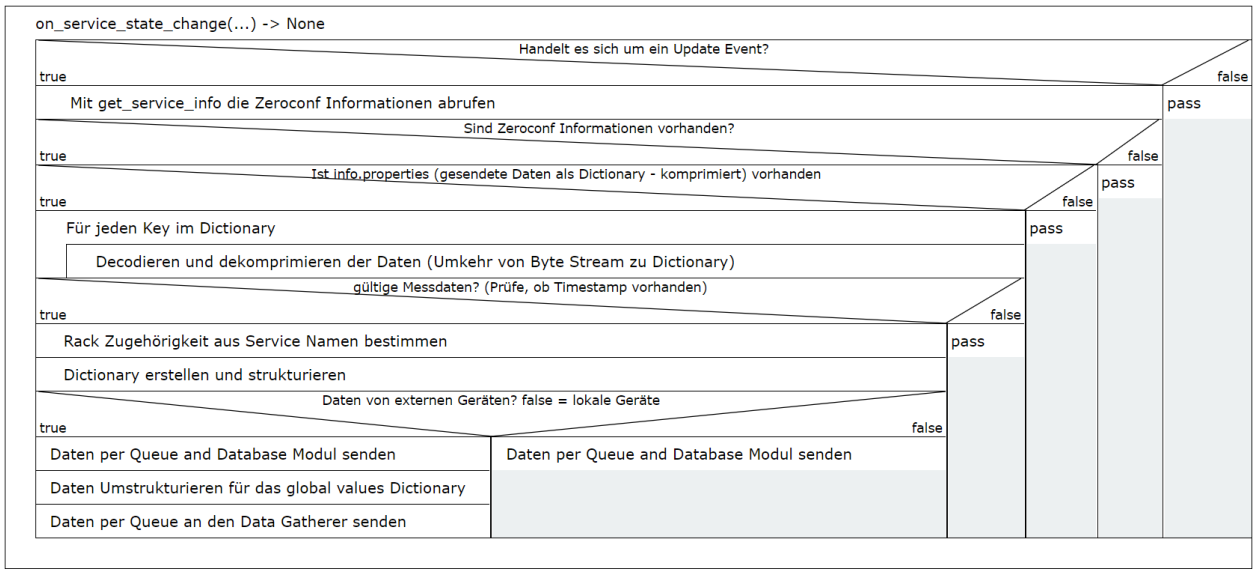


Abb. 10: Struktogramm - on_service_state_change

Übersicht folgt ein weiteres Struktogramm, dass die Methode beschreibt:

Das Struktogramm beschreibt die Arbeitsweise recht genau. Es sei kruz erwähnt, dass nur externe Geräte an den Data Gatherer weitergeleitet werden, weils die Internen Geräte bereits vom Data

Hinweis Sowohl Rack interne als auch externe Geräte werden über den Browser empfangen und erst dann an die Datenbank gesendet. Dies sorgt dafür, dass alle Datenbanken identische Werte sehen können.

Hinweis Durch den Zeitstempel behalten die Messwerte ihre Genauigkeit

Achtung Verpasst ein Raspberry Pi veröffentlichte Werte im Netz, dann können ungleiche Datenbanken entstehen. Ein Datenbankabgleich existiert nicht.

Collector kommen und somit schneller zur Verfügung stehen.

4.6 Data Collector

Das Data Collector Modul sorgt für die Kommunikation mit den Wechselrichtern und den Netzanalysatoren (Power Meter). Im Data Collector Modul wird für jedes Gerät ein eigener Thread gestartet. Die habe ich jedoch auf einem etwas anderen Weg gestartet. Die “run“ Funktionen (*_inverter_thread* und *_power_meter_thread*) sind nicht im entsprechenden Gerätemodul, sondern

im Data Collector definiert. So können Beide ggf. an einem Ort angepasst werden. Außerdem müssen die Module für Wechselrichter und Netzanalysatoren nicht zusätzlich für threading angepasst werden.

Imports:

```
# standard Module:
import logging
import time
import threading
import queue

# eigene Module:
from cmd_list import CmdEnum
import PowerMeterModule
import InverterModule
```

Struktogramm:

Weil ein großes Struktogramm nicht lesbar ist, werden die einzelnen Methoden des Data Collectors

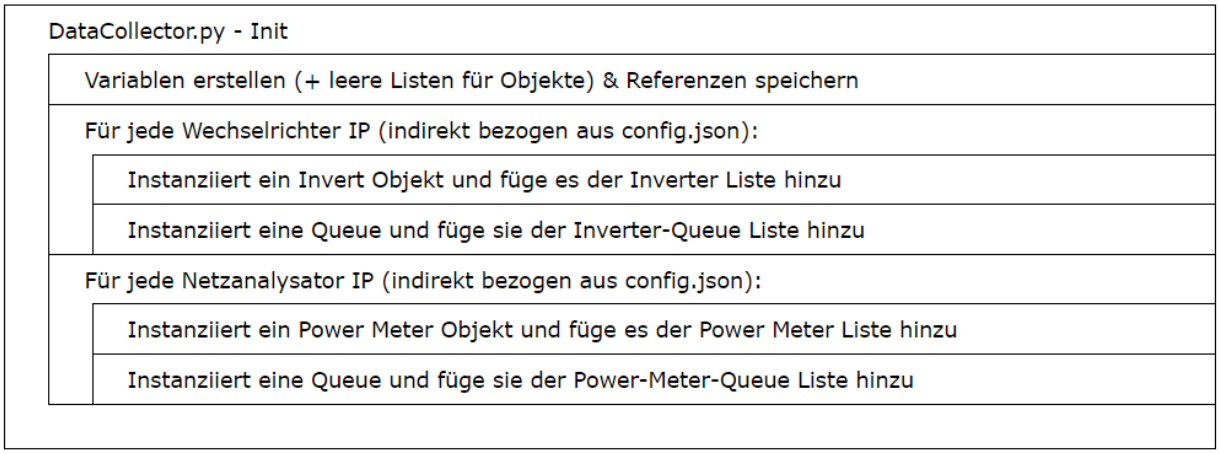


Abb. 11: Struktogramm – DataCollector.py - Init

in eigene Struktogramme unterteilt.

Bei Objektinitzialisierung werden zu zusätzlich auch leere Listen erstellt, um erstellte Objekte aufzunehmen, sodass diese iterativ angesprochen werden können. Objekte von Wechselrichter und Netzanalysatoren werden für jede gefundene IP Adresse erstellt. Zusätzlich wird eine Command Queue für jedes Objekt erstellt, welche es dem Collector ermöglichen während der Laufzeit mit dem Gerätethreads zu interargieren.

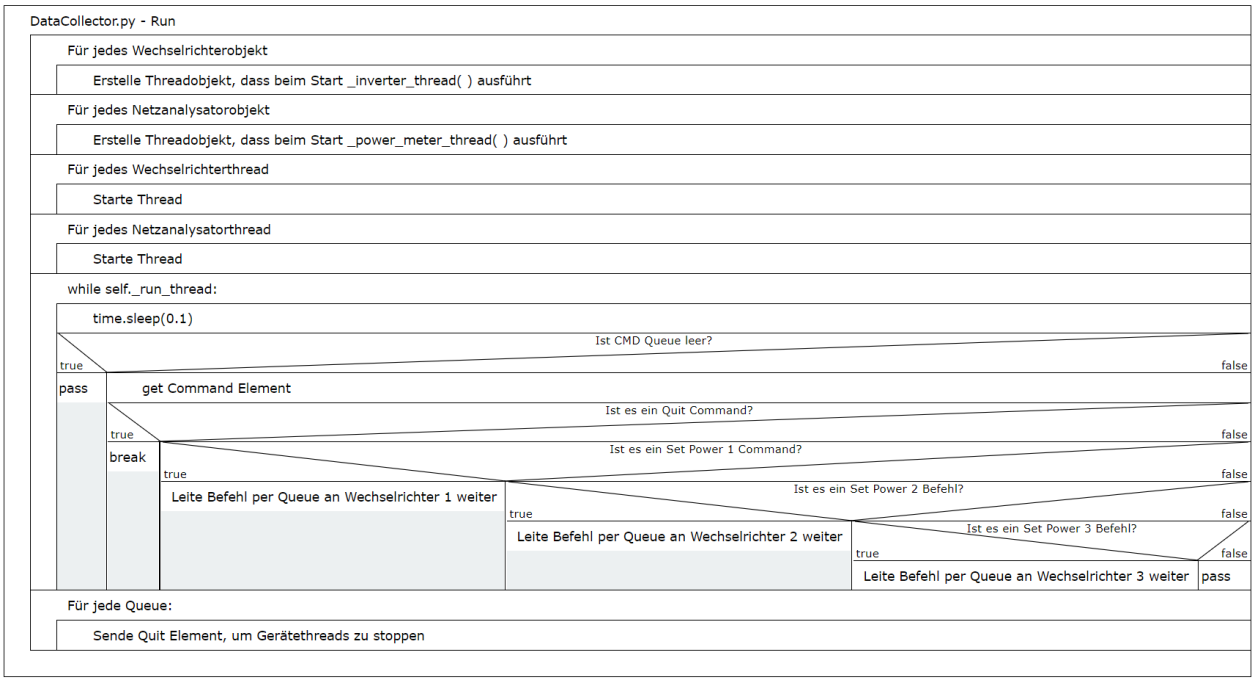


Abb. 13: Struktogramm - DataCollector.py - run

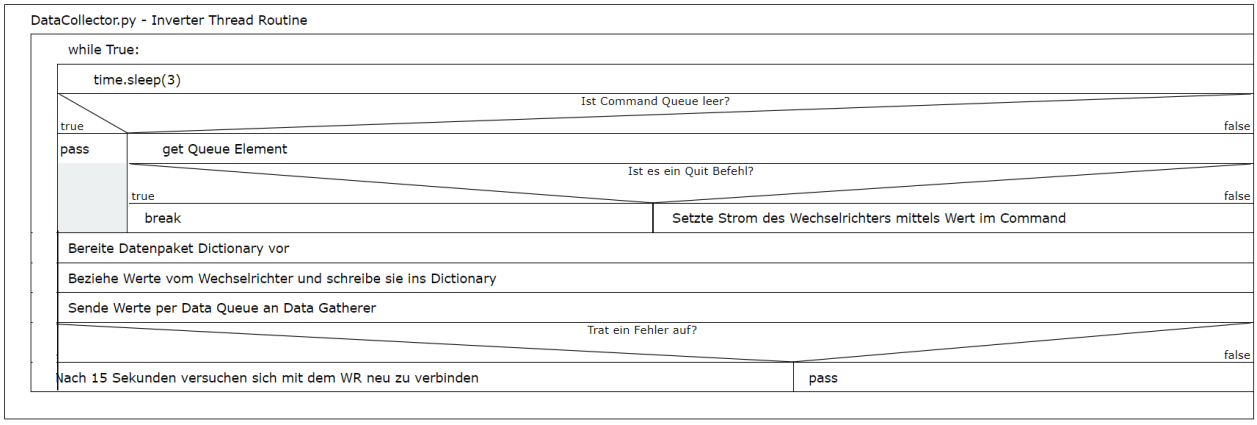


Abb. 12: Struktogramm - DataCollector.py - Inverter Thread

Die Collector run Funktion erstellt und startet die Gerätethreads. Anschließend geht sie in eine While-Schleife über und überwacht eine Command Queue, welche mit dem Webserver verbunden ist. Der Webserver sende hierüber Steuerbefehle für die Wechselrichter. Dieser werden vom Data Collector an den entsprechenden Wechselrichter weitergeleitet.

Die run Routine des Wechselrichters fragt zyklisch Messwerte ab und sendet diese mittels Queue an das Data Gatherer Modul. Vor jeder Messwertabfrage wird die Command Queue auf Befehle geprüft. Sollte ein Quit oder „Strom setzen“ Befehl vorliegen, wird der Befehl verarbeitet bevor neue

Hinweis Der Logger ist so konfiguriert, dass nur der Verbindungsverlust und die erfolgreiche Neuverbindung geloggt werden

Messwerte abgefragt werden. Am Ende des Struktogramms ist eine If-Abfrage gezeit, bei der ein auf Fehler geprüft wid. Im Programm wird dies mit einem Try- und Except Block realisiert. Das Struktogrammprogramm hatte hierfür keine Bausteine. Sollte also bei der Werteabfrage ein Fehler

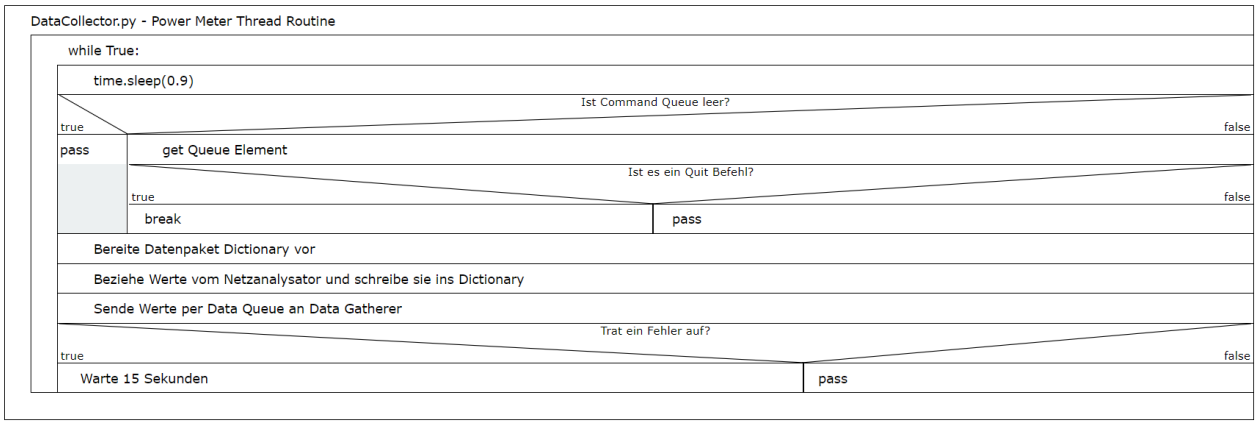


Abb. 14: Struktogramm -DataCollector - Power Meter Thread

entstehen, wird eine Reconnect Funktion des Wechselrichters aufgerufen. Dies sorgt dafür, dass der Thread sich neu mit dem Gerät verbinden kann, sobald der Fehler beseitigt wurde.

Die run Routine der Netzanalysatoren ähnelt den Wechselrichter Routinen. Nur ein Fehler wird anders verabeitet. Anstelle eine Reconnect Funktion des Power Meter Moduls zu starten, kann beim Netzanalysator einfach der Thread pausiert werden. Der Modbus TCP Client ist robuster implementiert, als die Socket Verbindung mit den Wechselrichtern.

Methodenübersicht:

```
1   __init__(
        self,
        cmd_queue,
        data_queue,
        local_inverter_ports,
        local_powermeter_ip,
        rack_name
    )
2   _invert_thread(self, InverterObj, cmd_queue)
3   _power_meter_thread(self, PowerMeterObj, cmd_queue)
4   stop(self)
5   run(self)
```

In diesem Modulabschnitt werden die Funktionen nicht weiter im Detail erklärt. Der Code und die Beschreibung unter den Struktogrammen sollten ausführlich genug sein. Einzig die *stop* Methode wurde nicht beschrieben, funktioniert aber genau wie in den Modulen zuvor.

4.7 Data Gatherer

Der Data Gatherer empfängt Daten über eine Queue und speichert diese im global_values Dictionary der main ab. Dafür wird während des Schreibvorgangs das Dictionary mittels eines Thread Locks gesperrt, sodass keine Schreibfehler entstehen können. Die Messdaten können dabei von lokalen Geräten (vom Data Collector) oder von externen Geräten (Zeroconf Browser) stammen. Der Gatherer wird verwendet, um die Datenverwaltung nicht in mehreren Modulen zu bewerkstelligen. Dies trägt zu einem übersichtlicheren Programmierstil bei.

Imports:

```
# standard Module
import logging
import time
import threading

# eigene Module
from cmd_list import CmdEnum
```

Für den Data Gatherer werden nur wenig Module benötigt.

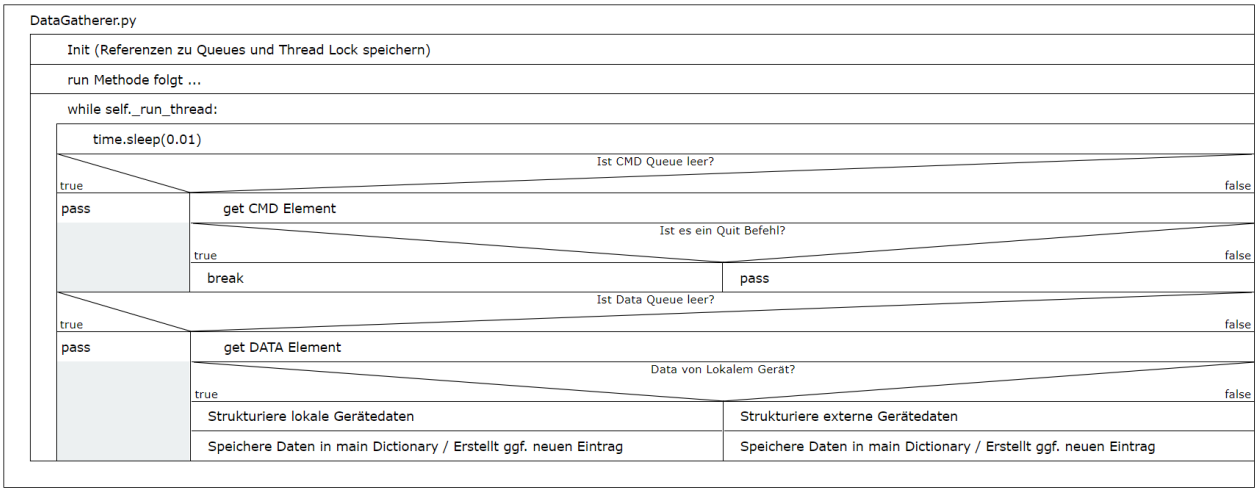


Abb. 15: Struktogramm - DataGatherer.py

Struktogramm:

Bei Erstellung des Gatherer Objekts werden die nötigen Referenzen in Variablen gespeichert. Wenn der Thread gestartet wird, geht der Thread in eine While-Schleife über. Auch in dieser Schleife befindet sich eine Sleep Funktion für bessere Performance. Die Command und die Data Queue werden im Wechsel nach vorhandenen Elementen durchsucht. Für die Command Queue existiert nur der Quit Befehl, der die Schleife verlassen lässt. Die Data Queue erhält die Datenpakete des

Data Collectors und des Zeroconf Browsers. Die Daten werden für das main Dictionary vorbereitet. Anschließend wird zum nächst möglichen Zeitpunkt das main Dictionary andere Threads gesperrt und die Daten abgelegt. Ist für ein Gerät noch kein Datensatz vorhanden, wird ein neuer Eintrag angelegt. Andernfalls werden die alten Werte mit aktuellen überschrieben.

Methodenübersicht:

```
1   __init__(self, cmd_queue, data_queue, global_values, mutex)
2   run(self)
3   stop(self)
```

Die *__init__* Funktion speichert Objekt Referenzen in Variablen. Das Thread Lock Objekt wird wieder mutex genannt.

Mit dem Start des Threads wird die Ablaufroutine des Gatheres gestartet, welche mit der *stop* Methode beendet werden kann.

Der Code im Data Gatherer ist gut nachzuvollziehen, weshalb die Methoden hier nicht detaillierter beschrieben werden.

4.8 Power Meter

Mit dem Power Meter Modul wird der Netzanalysator UMD 98 der Firma PQ-Plus integriert. Das Modul kann jedoch auch für weitere Geräte, die dem Modbus-Standard folgen, genutzt werden. Insbesondere dann, wenn die Registeradressen 19xxx verwendet werden. Diese sind genormt und für identische Messwerte verwendet. Im Power Meter Modul werden nur benötigte Funktionen implementiert. Weitere müssen bei Bedarf ergänzt werden.

Imports:

```
# standard Module:
import time
import sys

# extra:
from pymodbus.compat import IS_PYTHON3, PYTHON_VERSION
from pymodbus.client.sync import ModbusTcpClient as ModbusClient
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.constants import Endian
```

Die Kommunikation mit dem Netzanalysator erfolgt über das Modbus Protokoll. Das Python Paket *PyModbus* kann auf viele Jahre Erfahrung zurückgreifen und wird von mir verwendet, um die Umsetzung eines Standards zu umgehen.

Struktogramme werden in der Methodenübersicht verwendet. Anders als vorherige Module wird das Power Meter Objekt aus einem externen Modul (Data Collector) gesteuert und besitzt keine eigene Handlung, sondern dient als Interface.

Methodenübersicht:

```
1   __init__(self, IP_Address, port, unit)
2   _default_values(self)
3   __byteSkip_decode_float(self, decoderObj, nBytes)
4   __update__(self)
5   get_ip_address(self)
6   getValuesDict(self)
7   stop(self)
```

In der *__Init__* Methode werden einige Parameter in Variablen gespeichert. Außerdem wird die *_default_values* Methode aufgerufen. Diese erstellt ein Dictionary mit allen abzurufenden Werten, wobei diese noch mit None beschrieben werden. Zusätzlich in beim Initialisieren der Modbus Client mit folgendem Konstruktor erstellt:

```
ModbusClient(self._ip_address_, self._port_)
```

get_ip_address gibt als Rückgabewert die Geräte IP zurück.

Die stop Methode setzt ein boolesches Attribut des Objekts, welches für zum Thread Stop verwendet werden kann.

getValuesDict ruft die Update Methode auf und gibt eine Kopie des Dictionary an zurück.

In der __update__ Methode werden die Register des Netzanalysators gelesen und in das Dictionary geschrieben. Es hat sich gezeigt, dass das Auslesen von mehreren Registern, die aneinander liegende Adressen haben, schneller ist als Einzelaufrufe. Deshalb wird in __update__ der gesamte benötigte Adressbereich ausgelesen und anschließend die nötigen Register decodiert. Dafür wird die __byteSkip_decode_float__ Methode verwendet. Mit ihr können Bytes des Registers entfernt werden, sodass beim das gewünschte Register erreicht wird. Im folgenden wird die Update Funktion gezeigt:

```
1       def __update__(self):
2           timestamp_unix = time.time()
3           meter_rr = self._client_.read_input_registers(19000, 122, unit=0x01)
4           decoderObj = BinaryPayloadDecoder.fromRegisters(meter_rr.registers,
    byteorder=Endian.Big)
5           self._valuesDict_ = {
6               'timestamp_unix': timestamp_unix,
7               'VOLTAGE_L1': self.__byteSkip_decode_float__(decoderObj, 0),  # Address: 19000
8               'VOLTAGE_L2': self.__byteSkip_decode_float__(decoderObj, 0),           # 19002
9               'VOLTAGE_L3': self.__byteSkip_decode_float__(decoderObj, 0),           # 19004
10              'CURRENT_L1': self.__byteSkip_decode_float__(decoderObj, 2 * 6),       # 19012
11              'CURRENT_L2': self.__byteSkip_decode_float__(decoderObj, 0),           # 19014
12              'CURRENT_L3': self.__byteSkip_decode_float__(decoderObj, 0),           # 19016
13              'ACTIVE_POWER_L1': self.__byteSkip_decode_float__(decoderObj, 2 * 2),  # 19020
14              'ACTIVE_POWER_L2': self.__byteSkip_decode_float__(decoderObj, 0),      # 19022
15              'ACTIVE_POWER_L3': self.__byteSkip_decode_float__(decoderObj, 0),      # 19024
16              'COSPHI_L1': self.__byteSkip_decode_float__(decoderObj, 2 * 18),       # 19044
```

```
17              'COSPHI_L2': self.__byteSkip_decode_float__(decoderObj, 0),          # 19046
18              'COSPHI_L3': self.__byteSkip_decode_float__(decoderObj, 0),          # 19048
19              'FREQUENCY': self.__byteSkip_decode_float__(decoderObj, 0),          # 19050
20              'THD_U1': self.__byteSkip_decode_float__(decoderObj, 2 * 58),        # 19110
21              'THD_U2': self.__byteSkip_decode_float__(decoderObj, 0),             # 19112
22              'THD_U3': self.__byteSkip_decode_float__(decoderObj, 0),             # 19114
23              'THD_I1': self.__byteSkip_decode_float__(decoderObj, 0),             # 19116
24              'THD_I2': self.__byteSkip_decode_float__(decoderObj, 0),             # 19118
25              'THD_I3': self.__byteSkip_decode_float__(decoderObj, 0)              # 19120
26      }
```

Zuerst wird der Zeitstempel im Dictionary aktualisiert. Mit der read_input_registers Funktion werden mehrere Register von dem Messgerät ausgelesen. Begonnen wird dabei bei Adresse 19000 und anschließend 122 weitere Adressen (aufsteigend) ausgelesen. Nachfolgend wird ein Decoder Objekt erstellt und die Bitreihenfolge festgelegt. Mit der byteSkip Methode wird dann alle gewünschten Adressen decodiert. Dafür wird jeweils das Decoder Objekt mit übergeben, als auch die Anzahl an

> **Hinweis** Eine Registeradresswert besteht aus zwei Bytes. Eine 2 bedeutet damit, dass eine Adresse übersprungen wird. Zur besseren Lesbarkeit wurden größere Sprünge mit 2 * n angegeben. Wobei n die Anzahl der zu überspringenden Adressen ist.

Bytes die zu überspringen sind. Bei einer 0 wird keine Adresse übersprungen.

4.9 Inverter

Das Inverter Modul dient als Interface für die Kommunikation mit dem Wechselrichter. Funktionen zum Abrufen der Messwerte, sowie dem Setzen des Stromes, sind implementiert. Viel Arbeit verbirgt sich darin, die Kommunikation stabil zu gestalten.

Imports:

```
# standard Module
import logging
import struct
import serial.urlhandler.protocol_socket
import time
```

Auch der Inverter wird extern durch den Data Collector gesteuert und besitzt keine eigene Ablaufroutine. Deshalb werden Struktogramme nur (wenn sinnvoll) in der Methodenübersicht verwendet.

Methodenübersicht:

```
1  __init__(self, uart_port)
2  _default_values(self)
3  __open_serial__(self, port_name)
4  __sync__(self)
5  __send_check__(self, input, check, bytesToRead)
6  __read_serial_buffer__(self, expectedBytes)
7  __get_allValues__(self)
8  retry_decorator(func)
9  get_ip_address(self)
10 get_grid_voltage(self)
11 get_grid_current(self)
12 get_temp(self)
13 get_battery_voltage(self)
14 get_battery_current(self)
15 set_IBat(self, setCurrent)
16 get_valuesDict(self)
17 reconnect(self, duration)
18 stop(self)
```

Die Methoden werden im Detail mittels Kommentare im Code beschrieben. Auf Grund der Menge werden die Funktionen nur beschreiben und nicht auf das Detail heruntergebrochen.

Die *__init__* Methode speichert einige Parameter in Variablen und ruft die Methoden *_default_values* und *__open_serial__* der Reihenfolge nach auf. In *_default_values* werden viele nötige Parameter gesetzt. Außerdem wird das Werte Dictionary für die Messwerte vorbereitet. Die open Serial Methode stellt eine Verbindung mit dem Wechselrichter wie folgt her:

```
serial.serial_for_url('socket://' + port_name + ':5000/logging=debug')
```

Das zurückgegebene Objekt wird dann von anderen Methoden für die Kommunikation genutzt.

Die *reconnect* Methode wird vom Data Collector genutzt, um die Verbindung auf gleichem Wege zu erneuern. Vorher werden jedoch mögliche vorhandene Verbindungen geschlossen. Die Verbindung

Hinweis Ich habe versucht die Verbindung vor jedem Senden zu schließen, sodass eventuell andere Programme mit dem Wechselrichter abwechselnd kommunizieren können, jedoch ohne Erfolg. Nach ca. 10 Neuverbindungen folgte „Error 111: Connection refused“

mit dem Wechselrichter beleibt bis zu einem Fehler bestehen.

Die __sync__ Methode wird vor jeder Kommunikation mit dem Wechselrichter aufgerufen. Die State Machine des Wechselrichter wird hiermit in einen Standard Zustand versetzt in dem er für neue Befehle bereit ist. Damit wird sichergestellt, dass nichts unvorhersehbare geschieht.

Mit *__read_serial_buffer__* können Rückgabewerte des Wechselrichters ausgelesen werden.

Mit *__send_check__* werden mitgegebene Bytes gesendet und die Rückgabewerte auf erwartete Werte geprüft. Raises *ValueError*, im Falle eines unerwarteten Rückgabewertes.

Der *retry_decorator* ist eine Decorator Funktion die eine Funktion als Parameter übernimmt. Die Arbeit mit dem Wechselrichter hatte anfangs einige Schwierigkeit mit sich gebracht. Oft wurden Befehle z.B. einfach ignoriert. Der retry_decorator sorgt in einem solchen Fall dafür, dass die Funktion weitere Male ausgeführt wird, bis ein Maximum an Versuchen durchgeführt wurde. Dies hat

> **Hinweis** Inzwischen wurden einige Entstörungsversuche unternommen. Es kann sein, dass der Decorator inzwischen nicht mehr nötig ist. Für die Stabilität bleibt es implementiert

die Problematik erfolgreich umgangen.

Über die *get* Methoden lassen sich die Sensorwerte der Wechselrichter auslösen. Dabei können Werte für Temperatur, AC- & DC- Spannung, AC- & DC- Strom ausgelesen werden. Zusätzlich kann die IP des Gerätes angefragt werden.

Mit *set_IBat* lässt sich ein Steuerbefehl an den Wechselrichter senden. Dabei wird ein AC Soll-Strom vorgegeben. Die Funktion prüft den übergebenen Wert auf Fehler und bricht ggf. ab, bevor der Befehl an den Wechselrichter weitergeleitet wurde. Ein entsprechender Fehler würde geloggt werden.

Die *get_valuesDict* Methode ist vereint die Abfrage aller Messwerte. Sie ruft dafür die Methode *__get_allValues__* auf und gibt eine Kopie des Werte Dictionary zurück.

5 Anleitungen

In diesem Kapitel folgen einige Anleitungen zur Bedienung der Software und dem Arbeiten mit dem Raspberry Pi. Die Anleitungen zeigen die Arbeit per Remote Zugriff, eine Methode zum Software deployment, wie der Raspberry eingerichtet wird und noch einige weitere Abläufe.

5.1 Inbetriebnahme des Testaufbaus

Die Inbetriebnahme ist größtenteils selbsterklärend, aber es gibt dennoch eine Stolperfalle, die programmiertechnisch nicht lösbar war. Nachdem die Spannungszuvor der Racks erfolgt und die Racks über die Motorschutzschalter eingeschaltet werden, muss sichergestellt sein, dass die Fritzbox in aktueller Konfiguration bereits gestartet ist. Läuft der DHCP-Server der Fritzbox **nicht** und die Wechselrichter werden eingeschaltet, bezieht der Wechselrichter (bzw. das Ethernet Modul) nicht die richtige IP und bekommt diese auch nicht mehr.

Achtung Eine Kommunikation mit dem Wechselrichter ist dann nicht möglich!

Die einzige Lösung ist ein Neustart der Wechselrichter, sobald die Fritzbox läuft oder bereits vorher mit dem Einschalten zu warten. Wurde dies beachtet funktioniert auch die reconnect Funktion der Software und die Wechselrichter verbinden sich automatisch. Dies ist der einzige Fehler, der noch existiert und nicht gelöst wurde.

Eventuell gibt es aber auch hierfür eine Lösung. Eine Abhilfe könnte die manuelle IP-Zuweisung im Netzwerk sein. Die Netzanalysatoren, Wechselrichter Ethernet-Adapter und Raspberry Pi lassen sich alle mit festen IPs konfigurieren. So wäre sichergestellt, dass die IP des Wechselrichters stimmt und das Verbindungsproblem gelöst.

Nachteil wäre, dass neu hinzugefügte Geräte ebenfalls mit statischer IP eingerichtet werden müssen und nicht automatisch im Netzwerk aufgenommen werden, weil kein DHCP-Server mehr läuft.

5.2 Remote arbeiten (LAN)

Dieses Kapitel beschreibt, wie die Raspberry Pi bedient werden können, ohne einen Bildschirm, Maus und Tastatur anschließen zu müssen, sofern sich der eigene Rechner im selben lokalen

> **Hinweis** SSH kann bereits bei der Installation des Betriebssystems des Raspberry Pi aktiviert werden. Eine Anleitung ist in Kapitel 5.8.1 zu finden.

Netzwerk befindet. In vielen Fällen lässt sich damit Zeit sparen. Ein Nachteil des Fernzugriffs ist, dass die Anweisung an den Raspberry Pi über das Linux übliche Terminal erfolgen müssen. Dies kann anfangs ungewohnt sein. Die Verbindung wird dabei mittels Secure Socket Shell (SSH) durchgeführt, welches zuvor bei den Raspberry Pi aktiviert sein muss. Unter Windows habe ich für die SSH-Verbindung zwei Programme genutzt, PuTTY (Terminal) und WinSCP (Filemanager). Eine Anleitung für beide Programme folgt in den nachfolgenden Kapiteln.

5.2.1 PuTTY (Terminalzugriff unter Windows)

Unter Windows Betriebssystemen bietet die Software PuTTY die Möglichkeit sich mit einem Benutzer beim Raspberry Pi anmelden und eine Verbindung zum Terminal herstellen. Die Arbeit mit dem Terminal ist dann identisch wie auf dem Raspberry Pi. Zuerst muss dafür PuTTY installiert werden.

PuTTY Download: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html 

Nach der Installation kann das Programm gestartet werden und zeigt folgenden Startfenster:

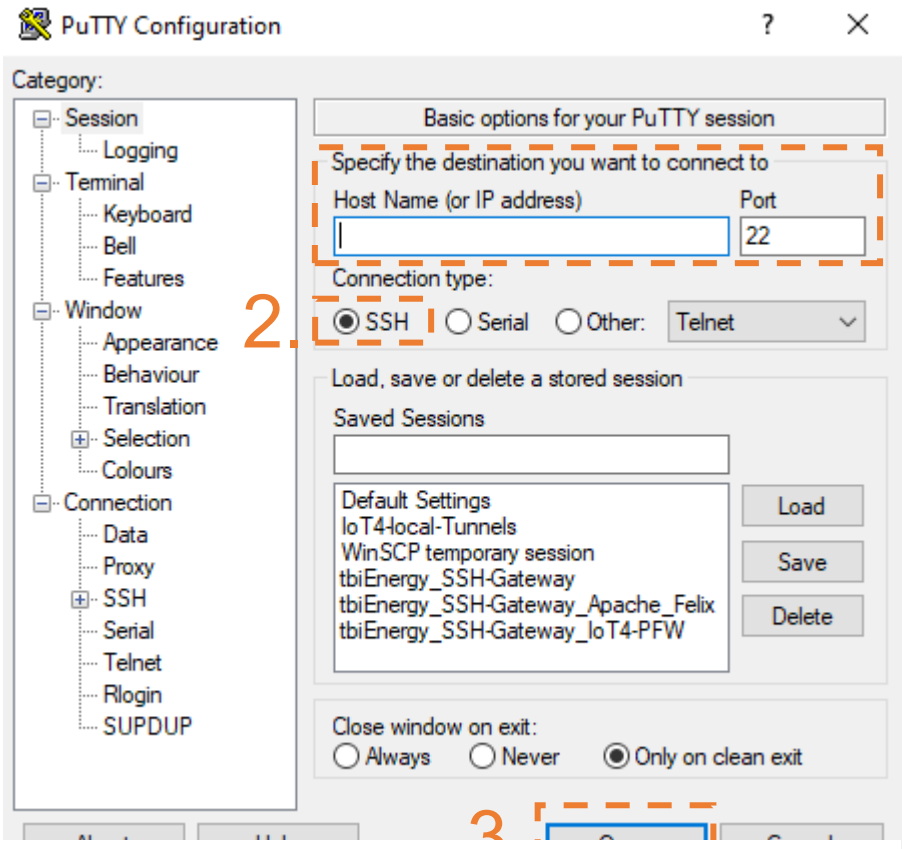


Abb. 16: PuTTY Startfenster

In **1.** wird muss die Adresse eingegeben werden von dem Raspberry mit dem sich verbunden werden soll. Für Rack 1 wäre das bspw. **192.168.11.41**. Bei Port wird die 22 stehen gelassen. Es ist möglich mehrere PuTTY Fenster zu öffnen und somit ein Terminal für jeden Raspberry gleichzeitig zu öffnen.

Bei **2.** Muss sichergestellt werden, dass SSH ausgewählt ist. Danach muss bei **3.** auf **Open** geklickt werden.

Es öffnet sich ein Terminal und was auffordert nacheinander den Benutzernamen und das Passwort einzugeben. Der Benutzer muss auf dem Raspberry vorhanden sein. Die Raspberry Pi im Labor sind alle mit einem Benutzer ***pi*** mit Passwort ***progressus*** eingerichtet. Nach erfolgreicher Eingabe ist das Terminal zu sehen. In Bild unten wurde sich an Rack 3 mit dem Benutzer progressus und Passwort progressus angemeldet. Dieser Nutzer ist nur in Rack 3 vorhanden! Ein Benutzer pi wurde hier

Hinweis Bei erstmaliger Anmeldung an einem Gerät muss eine SSH übliche Sicherheitswarnung mit Accept bestätigt werden.

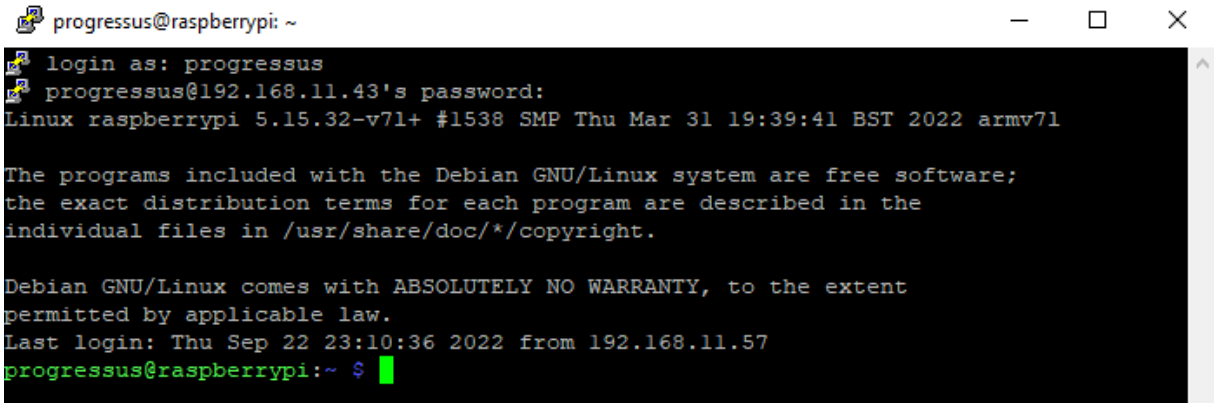


Abb. 17: PuTTY Terminalsession nach erfolgreicher Anmeldung

nachträglich eingerichtet.

Fertig. Das Terminal kann nun für Zwecke, wie „Installieren fehlender Pakete“, „Starten von Programmen“ uvm., genutzt werden. Mehr dazu in Kapitel 5.3.

5.2.2 WinSCP (Filemanager unter Windows)

Die Software WinSCP ist ein Filemanager, der sich mittels SSH mit dem Raspberry Pi verbinden kann. Damit können Dateien per Drag & Drop vom eigenen Windowsrechner zum Raspberry kopiert werden und umgekehrt. Zuerst muss WinSCP heruntergeladen und installiert werden.

WinSCP Download: https://winscp.net/eng/download.php

Nach der Installation kann das Programm gestartet werden und begrüßt einen mit folgendem Fenster:

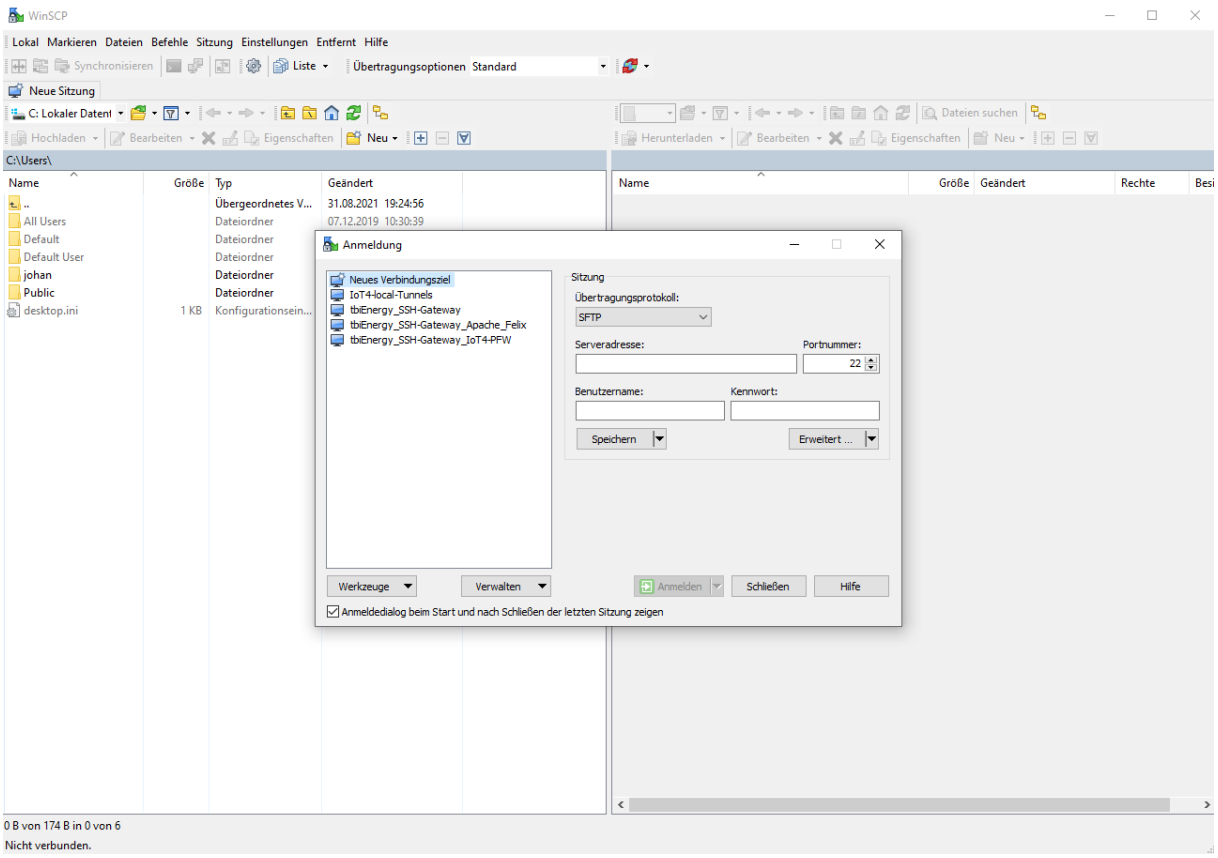


Abb. 18: WinSCP Startfenster

Im mittleren Fenster wird wieder die Adresse des Verbindungsziels angegeben. Benutzernamen und Kennwort können hier bereits angegeben werden.

Serveradresse: 192.168.11.41 bis 46 (Raspberry Pi IPs)

Port: 22

Benutzername: pi

Kennwort: progressus

Genau wie bei PuTTY erfolgt erfolgt eine erstmalige Sicherheitswarnung pro Gerät, die mit „*Aktualisieren“* bestätigt werden muss.

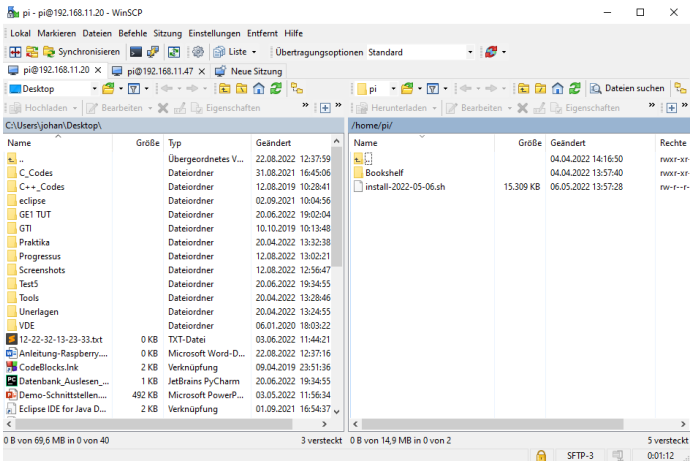


Abb. 19: WinSCP Hauptfenster

Das Hauptfenster (s. oben) zeigt nun auf der linken Hälfte einen Fileexplorer für den eigenen Windowsrechner an. Die rechte Hälfte zeigt die Dateien des Raspberry Pi. Ich empfehle die linke Hälfte gar nicht zu verwenden sondern nur die rechte Hälfte. Dateien können dann per Drag & Drop vom eigenen System einfach in das rechte Fenster gezogen werden und damit in das offene Verzeichnis kopiert werden bzw. auch anders herum. Oberhalb des zwei Hälften befindet sich ein Tab für jede offene Sitzung. Mit Klick auf **Neue Sitzung** können weitere parallel laufende Verbindung geöffnet und über die Tabs gewechselt werden.

Fertig. Es können nun bequem und zentral Daten verschoben werden (auch zwischen zwei Geräten).

5.3 Nützliche Terminal Befehle

In diesem Kapitel folgen einige Befehle die ich häufiger verwendet habe mit einer kurzen Beschreibung. In den weiteren Kapiteln wird auf dieses Kapitel referenziert, sodass die weiteren Abschnitte wniger im Kontext hin und her wechseln. Sollte für eine Aufgabe ein Befehl fehlen, dann hilft eine kurze Google Suche schnell weiter.

cd (Change Directory genannt) ändert das aktuelle Verzeichnis im Terminal. Befehle im Terminal wirken sich oft auf das akutelle Verzeichnis aus, wenn keine Pfadangabe gemacht wird. Dabei decken drei cd Anweisungen alle erforderlichen Verzeichniswechsel ab.

- **cd ..** geht in der Verzeichnisstruktur eines nach oben z.B. von */home/pi/* nach */home/*
- **cd /home/pi/** Absolute Pfadangaben werden durch ein führendes / signalisiert.
- **cd pi/** Relative Pfadangaben werden ohne führendes / geschrieben. In diesem Beispiel müsste man also schon im Verzeichnis home gewesen sein, um nach pi zu kommen.

Eine weitere sehr hilfreiche Funktion ist die Tabulator Taste bei Nutzung des Terminals. Während der Eingabe von Pfaden zu Ordnern oder Dateien ist das Drücken von Tab eine große Hilfe. Durch doppeltes Drücken von Tab werden alle noch möglichen Angaben, die zur aktuellen Eingabe passen, angezeigt. Einfaches Drücken ergänzt den Text sofern nur ein möglicher Treffer existiert. Dies beschleunigt die Eingabe meist deutlich.

ls (l = kleines L, ls = list) listet alle sichtbaren Verzeichnisse und Dateien im aktuellen Verzeichnis farbcodiert auf. Die Eingabe von ls kann also dafür genutzt werden sich einen Überblick des aktuellen Ordners zu verschaffen. In Blau werden Ordern dargestellt. Grau sind Dateien ohne Ausführrechte und Grün sind Dateien die ausführbar sind.

sudo Die Eingabe von sudo vor einem Befehl führ den Befehl nicht durch den angemeldeten Benutzer (standard) sondern mit Rechten des root Benutzer aus. Dies ermöglicht das Ausführen einiger Befehle, welche auf Dateien zugreifen wollen, die nicht von z.B. dem Benutzer pi genutzt werden können.

nano Bei diesem Befehl handelt es sich um einen Terminal basierten Texteditior. Bei kleinen Änderungen an Code Dateien oder Ähnlichem ist nano eine gute Wahl. Hinter dem Befehl nano muss ein Pfad zu einer Datei angegeben werden. Bsp.: “sudo nano /etc/config.json“. Mit Strg + O gefolgt von Enter kann die Datei dann gespeichert werden. Strg + X schließt den Editor wieder.

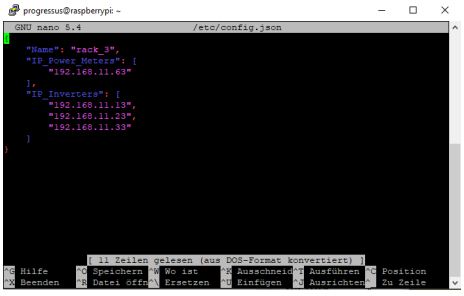


Abb. 20: nano Terminalbefehl Beispiel

Hinweis Die Navigation erfolgt ausschließlich über die Pfeiltasten! Per Mausklick kann im Terminal nichts ausgewählt werden.

chmod +x *Dateipfad*	Dieser Befehl ändert die Rechte einer Datei. +x fügt die Rechte für das Ausführen einer Datei dem Benutzer hinzu. Beim Installationsskript kann dies erforderlich sein.
./*Dateipfad*	Der Punkt gefolgt von einem Slash führt eine Datei aus, sofern nötige Rechte vorhanden sind. Damit kann z.B. das Installationsskript ausgeführt werden.
systemctl enable rack_monitor.service	Fügt einen Service zum Autostart per Systemd hinzu.
systemctl disable rack_monitor.service	Entfernt einen Service vom Autostart mittels Systemd
systemctl start rack_monitor.service	Startet einen Service manuell (ohne Neustart)
systemctl stop rack_monitor.service	Stoppt einen Service manuell
journalctl	Kann genutzt werden, um Log-Einträge Systemd Servicen einzusehen. Anwendungsbeispiele sind in Kapitel 5.8.5 zu finden.

5.4 Zeitsynchronisation mit NTP

Die Raspberry Pi haben keinen Quarz im System, der die Systemzeit mitlaufen lässt, wenn das System ausgeschaltet wird. Normalerweise beziehen die Raspberry Pi ihre Zeit über das Internet, was in der TH jedoch nicht zur Verfügung steht. Ich habe eine Alternative mittels Network Time Protocol (NTP) umgesetzt. Dies erlaubt es den Raspberry Pi eine Systemzeit von einem anderen Gerät im lokalen Netzwerk zu beziehen. Hierfür bediene ich mich beim IoT Gateway von Devolo, welches in Rack 2 verbaut ist. Dieses hat Zugriff aufs LTE-Netz und damit eine Internetverbindung. Das Betriebssystem des IoT Gateways ist auch ein Debian System und ähnelt stark dem eines Raspberry Pi.

Hinweis Das später genannte Installationsskript richtet NTP auf dem Raspberry Pi für das Labor mit ein und erfordert keine weiteren Schritte.

Im Folgenden wird die Einrichtung einer Zeitsynchronisation mittels NTP erklärt.

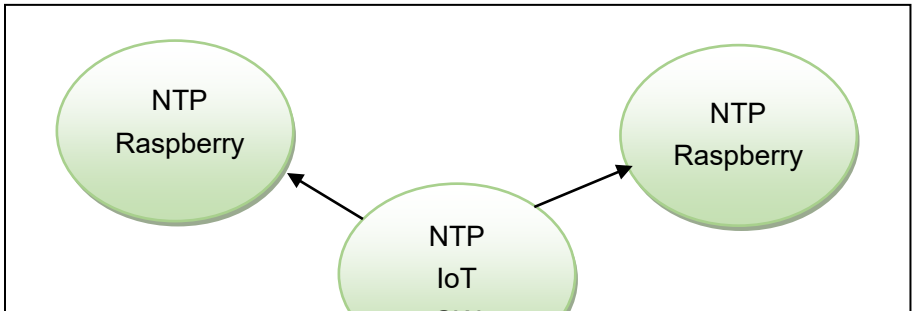


Abb. 21: NTP-Kommunikationsübersicht

Das Bild oberhalb zeigt das Kommunikationsschema von NTP. Das IoT GW wird als Server eingerichtet von dem andere Geräte die Zeit abrufen können. Dieser Vorgang kann bei den

Achtung Dies sorgt dafür, dass die ersten Messwerte ggf. mit falschem Zeitstempel gespeichert werden.

Raspberry Pi nach dem Systemstart bis zu 15 Minuten in Anspruch nehmen.

Auf dem IoT GW und den Raspberrys muss zuerst der NTP Dienst installiert werden. Die Installation erfolgt mit folgendem Befehl:

```
apt-get install ntp
```

Nachdem NTP installiert wurde befindet sich im Verzeichnis */etc/* die Datei *ntp.conf*, welche auf allen Systemen angepasst werden muss.

Auf dem IoT GW (Server) müssen folgende Einträge wie folgt angepasst werden (IPs anpassen und einige Zeilen auskommentieren):

```
# pool.ntp.org maps to about 1000 low-stratum NTP servers.  Your server
# will pick a different set every time it starts up.  Please consider
# joining the pool: <http://www.pool.ntp.org/join.html>
pool 0.debian.pool.ntp.org iburst
pool 1.debian.pool.ntp.org iburst
pool 2.debian.pool.ntp.org iburst
pool 3.debian.pool.ntp.org iburst

# Clients from this (example!) subnet have unlimited access, but only if
# cryptographically authenticated.
restrict 192.168.11.0 mask 255.255.255.0

# If you want to provide time to your local subnet, change the next
# line. (Again, the address is an example only.)
broadcast 192.168.11.255
```

Auf den Raspberrys (Empfänger) sind es die folgenden Zeilen:

```
# You do need to talk to an NTP server or two (or three).
```

```
server 192.168.11.100

# If you want to listen to time broadcasts on your local subnet, de-
# comment the next lines.  Please do this only if you trust everybody on
# the network!
disable auth
broadcastclient
```

Hinweis Die IP-Adressen sind auf das Labor angepasst. Ggf. müssen sie auf die Netzumgebung angepasst werden.

Fertig. Nach einem Neustart aller Geräte sollte nun die Systemzeit automatisch bezogen werden, sofern das IoT GW Internet Zugang hat.

5.5 Ansteuerung der Wechselrichter

5.5.1 Manuell im Browser

Für die manuelle Ansteuerung kann ein gewöhnlicher Internetbrowser genutzt werden. Wenn die Software läuft, dann kann die Adresse des Raspberry Pi mit Port 5000 geöffnet werden.

URL-Beispiel für Raspberry Pi in Rack 3:

`192.168.11.43:5000`

Im Browser URL Bereich eingegeben öffnet sich folgender Tab:

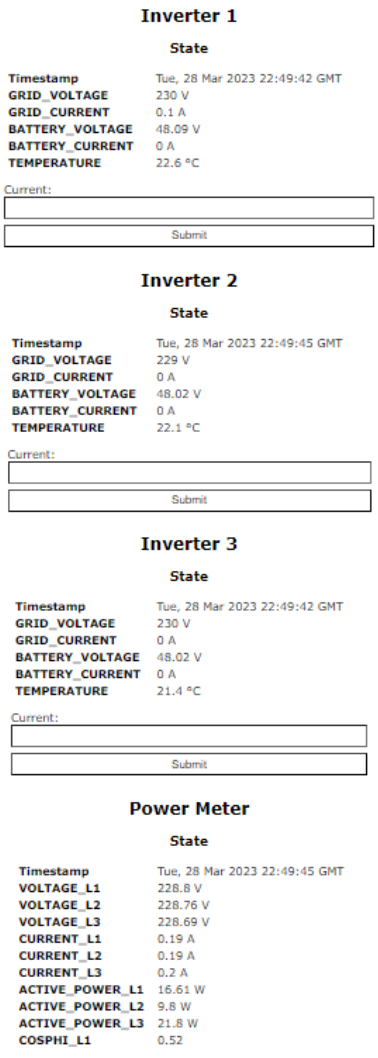


Abb. 22: Webserver im Browsertab

Alle Werte lokaler Geräte werden dort angezeigt.

Der Browser Tab Name (hier nicht zu sehen) gibt den Namen des Racks an z.B.: “rack_3 – Status“. So ist es einfach die Messwerte zuzuordnen, wenn für jedes Rack ein Tab geöffnet ist.

Über die Current Eingabefelder kann ein beliebiger Zahlenwert eingetragen und mit Klick auf Submit an den entsprechenden Wechselrichter gesendet werden. Die Zuordnung erfolgt dabei in der config.json Datei. Inverter 1 sollte dann Phase 1 entsprechen usw.

Alle Messwerte aktualisieren sich automatisch alle 3 Sekunden. Der Timestamp zeigt die Zeit für den aktuellen Messwert an. Bitte beachten, dass die Zeit nicht synchronisiert sein kann, weil der NTP-Server keine Internetverbindung hat oder die Raspberry Pi noch ein paar Minuten für die Synchronisation brauchen können. Ändert sich der Zeitstempel, aber die Zeit stimmt nicht, dann wird es sich um aktuelle Messwerte nur mit falschem Zeitstempel handeln.

5.5.2 Automatisch mit Python Skript

Für die zu implementierenden Regelungen bietet sich das Ansteuern der Wechselrichter direkt im Python Skript an. Dabei kann die REST-Schnittstelle des Webservers genutzt werden, um die Wechselrichter zu steuern. Dafür sind nur sehr wenige Zeilen Code erforderlich. Ich habe das Python Modul Reqeuests dafür verwendet. Andere Module mit der Möglichkeit POST Requests zu senden, sind ebenfalls nutzbar. Zuerst muss Requests für Python installiert werden. Dies geht mit dem Befehl:

```
pip install requests
```

Anschließend kann requests im eigenen Code implementiert werden

```
import requests

requests.post('http://localhost:5000/set_power1', data='current=-5')
requests.post('http://localhost:5000/set_power2', data='current=-5')
requests.post('http://localhost:5000/set_power3', data='current=-5')
```

Der gezeigte Code setzt alle drei Wechselrichterströme des Raspberry Pi, auf dem der Code ausgeführt wird, auf -5 A AC-seitig. Die Abweichung kann dabei bis ca. 2 % betragen. Die Reaktionszeit sollte dabei nicht länger als 3 Sekunden betragen. Sollen aus einem Code mehrere Wechselrichter aus verschiedenen Racks angesteuert werden, dann kann localhost gegen die entsprechende Raspberry IP ausgetauscht werden. Von einem beliebigen Gerät im Netzwerk sieht dies für Wechselrichter an Phase 1 in Rack 1 wie folgt aus:

```
requests.post('http://192.168.11.41:5000/set_power1', data='current=-5')
```

Hinweis Negative Stromwerte entladen die Batterien. Positive beladen sie und ein Wert von 0 ist gleichbedeutend mit einem Standby Zustand

Fertig. Eine Zeile Code und ein Import genügt einen Wechselrichter zu steuern.

Hinweis Pro Wechselrichter wird nur der letzte gefundene Befehl vom Wechselrichter bearbeitet. Alle Vorherigen werden verworfen.

5.6 Datenbank auslesen

Zum Auslesen von Werten der Datenbank sind ebenfalls wenige Schritte notwendig. Das Beispiel zeigt den Weg mittels eines Python Skripts und dem installierten Python Influx DB Client.

Folgende Code wird benötigt:

```
from influxdb import InfluxDBClient

client = InfluxDBClient('localhost', 8086, "root", "root", "progressus")

# Für Wechselrichterwerte
result = client.query("select time, AC_VOLTAGE, DC_VOLTAGE, AC_CURRENT,
DC_CURRENT, TEMPERATURE from monitoring where device = 'INVERTER' and
host = 'rack_3' and IP = '192.168.11.21' order by time desc limit 1")



# Für Netzanalysatorwerte
result = client.query("select time, VOLTAGE_L1, VOLTAGE_L2, VOLTAGE_L3,
CURRENT_L1, CURRENT_L2, CURRENT_L3, ACTIVE_POWER_L1, ACTIVE_POWER_L2,
ACTIVE_POWER_L3, COSPHI_L1, COSPHI_L2, COSPHI_L3, FREQUENCY from
monitoring where device = 'POWER_METER' and host = 'rack_3' order by
time desc limit 1")

result_list = list(result.get_points(measurement='monitoring'))

# Die Ergebnisse können dann verarbeitet werden
for dict in result_list:
        print(dict)
```

Das Beispiel zeigt die Abfrage von Netzanalysator in Rack 3 und Wechselrichter an L2 in Rack 3 an. Der oben genannte Code würde nur die Power Meter Werte anzeigen, weil die Variable result durch beide Abfragen überschrieben wird. Entweder nur eine Suchfunktion verwenden oder in getrennten Variablen speichern. Alternativ kann das Dict in der Resultliste des ersten Queries der zweiten Liste hinzugefügt werden.

Viele weitere Filter sind zusätzlich möglich. Ich empfehle dazu eine Recherche nach Influx Query Language. Der Datenpunkt Aufbau mit allen Tags und Messwerten kann der Codebeschreibung in Kapitel 4.3 entnommen werden.

5.7 Erstellen eines Installationsskripts

Um die spätere Installation zu vereinfachen, wird ein Installationsskript erstellt. Folgende Aufgaben übernimmt das Skript:

- Installiert noch benötigte Python Pakete (Internetzugang erforderlich)
- Kopiert nötige Dateien an erforderliche Stellen
- Konfiguriert den NTP-Server für die Zeitsynchronisation mit IoT-Gateway
- Fügt das Programm dem Systemstart hinzu (systemd)

Das Installationsskript kombiniert viele Schritte zu wenigen und wurde einzig aus diesem Grund erstellt.

Sollten Änderungen an der Software gewünscht sein, dann kann der Quellcode angepasst werden und ein neues Installationsskript daraus erstellt werden. Der Installationsvorgang bleibt der Gleiche, wie in Kapitel 5.8.2 beschrieben. Alle älteren Dateien werden dann auf dem Raspberry Pi ersetzt. Das bedeutet allerdings auch, dass die Config Datei wieder angepasst werden muss! Die nötigen Schritte hierfür werden in Kapitel 5.8.3 beschrieben.

5.7.1 Schritt 1 – Daten auf Raspberry Pi kopieren

Als Erstes müssen die benötigten Dateien auf dem Raspberry Pi abgelegt werden (s. unten „Zwingende Dateistruktur“). Dies ist nötig, weil PyInstaller vorhandene Abhängigkeiten nur aufzulösen kann, wenn es auf den Zielsystem ausgeführt wird. Eine Möglichkeit zum Datentransfer ist in Kapitel 5.2.2 zu finden. Mit Abhängigkeiten sind zusätzliche Pakete gemeint, die PyInstaller mit in die erstellte Binary-Datei packt, sodass diese im Zielsystem nicht zusätzlich installiert werden

> **Hinweis** Die Pakete für die Influx DB und den NTP-Dienst (Zeitsynchronisation) ließen sich nicht mit PyInstaller hinzufügen. Deshalb wird eine Internetverbindung bei der Installation benötigt

müssen.

Gewünschte Änderungen werden an den Python-Dateien (Enden auf .py) vorgenommen. Die Software ist in mehrere Module gekapselt. Sind z.B. nur Änderungen an der Datenbank erforderlich, dann muss die Datei *Database.py* angepasst werden.

Ich empfehle die Dateien einmal auf einem Raspberry abzulegen und anschließend immer dort die veränderten Dateien zu ersetzen oder sogar direkt dort zu verändern und anschließend das erstellte Installationsskript auf das/die Wunschgerät(e) zu kopieren.

Zwingende Dateistruktur:

```
*beliebiges Verzeichnis*                          anzupassende Dateien = grün
    ↳ | installfiles                              nicht zu verändernde Dateien = rot
      |     ↳ | root                                                  Ordner = blau
      |       |   ↳ | etc                                              ↳ = In Ordner hinein
      |       |     | ↳ |config.json                                   | = Orientierungshilfe
      |       |     |   |ntp.conf
      |       |     | lib                                             Selbstverständliche dürfen alle Dateien
      |       |     | ↳ systemd                                       angepasst werden, vorausgesetzt man weiß
      |       |     |       ↳ system                                  was man tut!!! Ich empfehle ein Backup.
      |       |     |             ↳ rack_monitor.service
      |       |     | usr
      |       |     | ↳ |bin
      |       |     |   | ↳ |rack_monitor
      |       |     |   |   |start_rack_monitor
      |       |     |   | lib
      |       |     |   | ↳ ntp
      |       |     |   |       ↳ ntp-systemd-wrapper
      |       | install.sh
      | env
      |   ↳ (alle Dateien unverändert)
      | archiv.tar.bz
      | cmd_list.py              (Enum verwendeter Commands)
      | create_installer.sh
      | database.py              (Influx DB)
      | DataCollector.py         (zyklische Werteabfrage,)
      | DataGatherer.py          (verwaltet Messdaten)
      | InverterModule.py        (Wechselrichter)
      | main.py                  (Startet die Threads und ließt Parameter aus config.json)
      | PowerMeterModule.py      (UMD-Messgeräte)
      | webserver.py             (REST-Schnittstelle zum Ansteuern der Wechselrichter)
      | zeroconf_browser.py      (Empfänger des Multicasts)
      | zeroconf_registration.py (Sender des Multicasts)
```

Gezeigt werden nur alle notwendigen Dateien. Zusätzliche Dateien in der Struktur bereiten **keine** Probleme. Die Dateien sind in der Sciebo Cloud zu finden.

(***Sciebo*** *-> Progressus -> Demonstrator -> JohannesKruse -> Programm_Dateien*)

5.7.2 Schritt 2 – Erstellen einer Installationsdatei mit PyInstaller

Sind alle gewünschten Änderungen an den Python Dateien vorgenommen kann das Erstellen des Installationsskripts fortgesetzt werden.

Hierfür benötigen wir zuerst PyInstaller, um eine einzelne Installationsdatei zu erstellen, an die wir später unsere Zusatzdateien (Konfigurations-, Systemstart und Zeitserver relevante Dateien) anhängen. Dieser Schritt war leider nicht automatisierbar, weil PyInstaller asynchron arbeitet und leider nicht auf

die Fertigstellung wartet. Falls noch nicht geschehen, kann PyInstaller mit dem Befehl „sudo pip install pyinstaller“ installiert werden.

Ist Pyinstaller installiert navigieren wir mit „cd“ zu unserem Verzeichnis mit obiger Datenstruktur. Im kommenden Beispiel heißt der Ordner „SourceCode“.

In SourceCode angekommen muss im Terminal das virtual Environment aktiviert werden, weil PyInstaller sonst die Abhängigkeiten nicht auflösen kann und nötige Pakete später fehlen würden. Das virtual Environment kann mit dem folgendem Befehl aktiviert werden:

source env/bin/activate

Die erfolgreiche Eingabe kann daran erkannt werden, dass im Terminal in der Eingabezeile ganz links (env) steht.

Nun geben wir den Befehl „pyinstaller --onefile main.py“ ein. PyInstaller erstellt nun eine einzelne Installationsdatei, in der die meisten Abhängigkeiten aufgelöst werden. Das heißt, dass alle benötigten Module und einige der Pakete mit eingebunden sind. Der Vorgang kann beim ersten Mal ca. 3 Minuten dauern. Nach erfolgreichem Abschluss befinden sich neue Dateien im aktuellen Verzeichnis („main.spec“ und die Ordner „dist“ und „build“). Im Ordner „dist“ ist unsere unvollständige Installationsdatei.

Abb. 23: PyInstaller - Konsolenbefehl zum Erstellen der Installationsdatei

5.7.3 Schritt 3 – „create_installer.sh“ ausführen

Jetzt fehlen nur noch die zusätzlichen Dateien aus dem Ordner „installfiles“ z.B. die Config.json Datei (IP-Adressen der Geräte), Dateien für den Systemd (Automatischer Programmstart) und den NTP-Zeitserver (Zeitsynchronisation mit dem IoT-Gateway). Hierfür existiert in unserem Verzeichnis die Datei „create_installer.sh“, die nur ausgeführt werden muss.

Zuerst benötigen wir allerdings die Rechte zum Ausführen der Datei.
Konsolenbefehl: „chmod +x create_installer.sh”.

Abb. 24: Terminal – Konsolenbefehl, um das Skript ausführbar zu machen

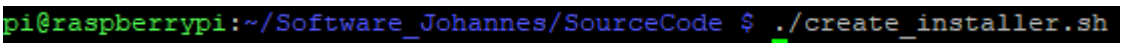


Abb. 25: Konsolenbefehl zum Ausführen des Skriptes

Der Befehl „./create_installer.sh“ (führenden Punkt beachten!) führt die Datei aus.

Nach kurzem Warten ist das finale Installationsskript fertig und kann im Ordner „installfiles“ gefunden werden. Der Dateiname ist „install-*Jahr*-*Monat*-*Tag*.sh“. Sollte am selben Tag ein weiteres Skript erstellt werden, dann wird diese Datei ersetzt!

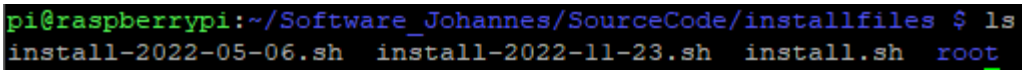


Abb. 26: Zu sehen sind zwei erstellte Installationsskripte vom 06.05.22 und 23.11.22

Fertig.

Eine Zusammenfassung für den Fall, das der gleiche Raspberry Pi erneut verwendet wird:

- Quellcode im “SourceCode“ Ordner nach Wünschen anpassen
- mit “cd“ zu dem Verzeichnis mit den Quellcode Dateien navigieren
- Mit “source env/bin/activate” das virtual Environment aktivieren
- ”pyinstaller --oncefile main.py“ ausführen
- “./create_installer.sh” ausführen

Die nötigen Schritte zum Installieren werden im Kapitel 5.8 beschrieben.

5.8 Software deployment

Nach dem erfolgreichen Erstellen einer Installationsskripts muss das Endgerät (Raspberry Pi) eingerichtet und konfiguriert werden. Die Anleitung beschreibt den Durchführungsablauf über eine SSH-Verbindung, die keinen Anschluss eines Monitors, einer Tastatur und einer Maus am Raspberry Pi erfordert.

Kurzanleitung / Schritte:

- Raspberry Pi OS installieren (SSH aktiveren, Benutzer anlegen, Internetzugang einrichten)
- Ausführen des bereits erstellten Installationsskripts auf dem Raspberry Pi
- Software konfigurieren mittels Config-Datei (Pfad: *„/etc/config.json“)*
- Raspberry neustarten
- Ordnungsgemäßen Betrieb prüfen

5.8.1 Schritt 1 - Raspberry Pi OS installieren:

Vorbereitung: SD-Karte am PC anschließen ggf. mit einem passenden Adapter.

Für das Installieren des Operating System (OS) den Raspberry Pi Imager herunterladen.
Link: Raspberry Pi Imager (Stand: 28.10.2022)

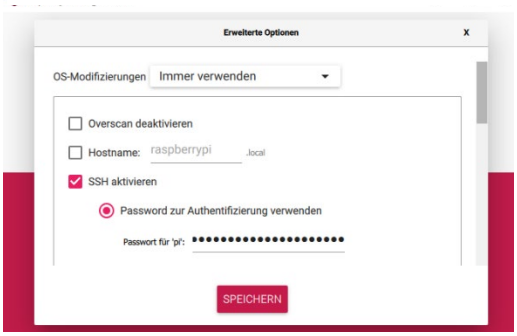
Abb. 27: Raspberry Pi Imager

Pi Imager ausführen. Als Betriebssystem “Raspberry Pi OS (32-bit)” auswählen und ggf. herunterladen.

Mit der Tastenkombination „*Strg + Shift + x*“ und Raspberry Pi Imager im Fokus, die erweiterten Optionen öffnen und folgende Einstellungen vornehmen:

- SSH aktivieren.

- Benutzer und Passwort festlegen (Ich habe zum Schluss immer „pi“ und „progressus“ verwendet. Raspberrys, die schon länger immer Umlauf sind,



können noch „progressus“ und „progressus“ als Kombination verwenden.

Abb. 28: SSH aktiviert und Passwort eingegeben

- Wifi mittels SSID und Passwort einrichten (z.B. mobiler Hotspot vom Handy). Internetzugang wird für die Installation von zusätzlichen Paketen benötigt, die nicht von PyInstaller aufgelöst werden konnten. (s. Erstellen eines Installationsskripts).

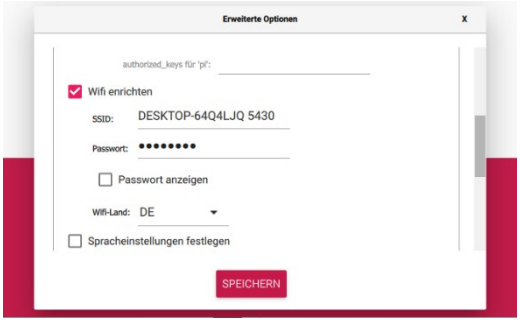


Abb. 29: WLAN Hotspot konfiguriert

- Klicken Sie nun auf „Speichern“ und anschließend auf „Schreiben“. Es folgt eine Warnung, dass sämtliche Daten auf der SD-Karte gelöscht werden. Akzeptieren und warten bis der Schreibvorgang abgeschlossen ist.

- Nach einer kurzen Wartezeit ist die SD-Karte betriebsbereit und kann in den Raspberry Pi eingeschoben werden.

5.8.2 Schritt 2 - Ausführen des bereits erstellten Installationsskripts

Vorbereitung: Raspberry Pi und PC/Laptop im LAN des Demonstrators anschließen und Stromzufuhr gewährleisten.

Als Erstes benötigen wir die IP des Rasperry Pi, welche automatisch von der Fritzbox vergeben wurde. Dafür öffnen wir einen neuen Tab eines beliebigen Webbrowsers und geben „fritz.box“ als URL ein.

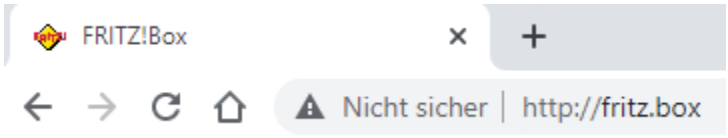


Abb. 30: URL im Browser "fritz.box"

Es öffnet sich die Bedienoberfläche der Fritzbox. Ein Linksklick auf Heimnetz (linke Seite) zeigt eine Liste aller bekannten Geräte an. Hier können und sind einigen Geräten feste IP-Adressen zugewiesen. Wenn erwünscht, dann weisen Sie jetzt dem neuen Rasperry Pi eine feste und

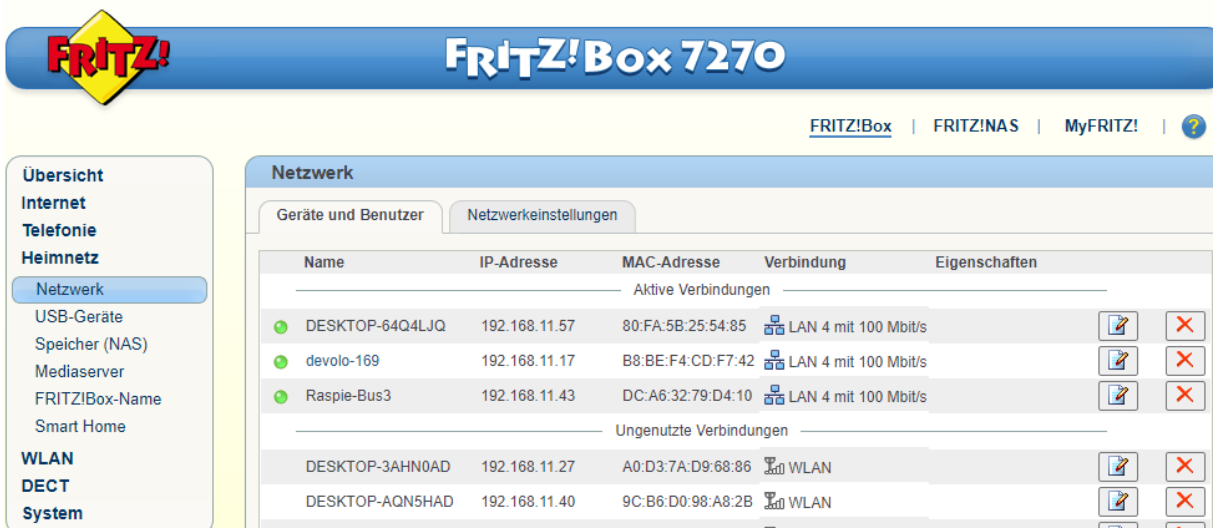


Abb. 31: Raspberry Pi mit zugewiesener IP und Namen Raspie-Bus3

ungenutzte IP zu. Angeschlossene und aktive Geräte sind mit einem grünen Kreis markiert. In dem Bild oberhalb ist bereits eine IP und ein Netzwerkname dem Raspberry Pi zugewiesen.

Der Spalte *IP-Adresse* entnehmen wir die IP des neu angeschlossenen Raspberry Pi.

Nun muss das Installationsskript auf den Raspberry kopiert werden. (z.B. mit WinSCP)

Ziel ist es weiterhin das Installationsskript auszuführen. Für die Ausführung ist allerdings ein Internetzugang erforderlich. Diesen haben wir in Schritt 1 bereits eingerichtet. Die Fritzbox, ohne eigene Internetverbindung, bringt leider einige Umstände mit sich, da diese stets versucht anderen Geräten Internet bereitzustellen. Das führt in vielen Fällen zu einem Timeout. Es folgen einige Schritte, welche das Problem umgehen.

Der Raspberry Pi sollte sich automatisch mit dem eingerichteten Hotspot verbinden, sobald es ihn findet. Prüfen können wir das z.B. mit dem Befehl „ifconfig“. Dort schauen wir, ob unter der Schnittstelle „wlan0“ eine IP zugewiesen wurde. Auch ist es möglich am Gerät, welches den Hotspot zur Verfügung stellt, eine Liste aller verbundenen Geräte einzusehen. Ist ein neues Gerät in dieser Liste aufgetaucht (üblicher Name raspberry), dann ist davon auszugehen, dass das Gerät erfolgreich mit dem Hotspot verbunden ist. Damit der Raspberry nicht versucht über die Fritzbox ins Internet zu gelangen geben wir folgenden Befehle ein:

„sudo route del default“

Anschließend geben wir „route“ ein und warten ca. 10 Sekunden.

```
progressus@raspberrypi:~ $ route
Kernel-IP-Routentabelle
Ziel            Router          Genmask         Flags Metric Ref    Use Iface
default         fritz.box       0.0.0.0         UG    202    0        0 eth0
192.168.11.0    0.0.0.0         255.255.255.0   U     202    0        0 eth0
```

Abb. 32: Internet route Übersicht

In dem Bild oberhalb wurde der Befehl „sudo route del default“ noch nicht eingegeben. Es zeigt also den Zustand, in welchem die Installation **nicht** möglich ist. Wir beachten, dass in einer Zeile „default“ als Ziel und bei Router „fritz.box“ steht.

Ist dies nicht der Fall können wir mit dem Ausführen der Installationsdatei fortfahren.

In unserem Fall befinden wir uns bereits in dem Verzeichnis, wo die Datei abgelegt wurde. Sollte die Datei abweichend von der Beschreibung abgelegt worden sein, ist es sinnvoll dorthin zu navigieren.

Hierfür nutzen wir den Befehl „cd“ (s. Kapitel 5.3)

„cd /home/pi“

Sind die Rechte erteilt und der Text grün kann das Skript mit “sudo ./*Name-des-Installskripts*“ ausgeführt werden.

5.8.3 Schritt 3 - Software konfigurieren mittels Config-Datei

Ein paar Parameter müssen der Software manuell mitgeteilt werden. Hierfür existiert nach der Installation eine Config-Datei im Dateipfad “*/etc/config.json*“. Auch dieser Schritt lässt sich mit dem Terminal ausführen. Hierfür nutzen wir den Konsolenbefehl “sudo nano /etc/config.json“

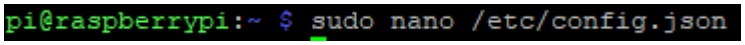


Abb. 33: PuTTY – Texteditor im Terminal per “nano” Konsolenbefehl

Die Datei öffnet sich im Terminal. Die lila gefärbten Parameter müssen angepasst werden. Dabei sind ein paar Dinge zu beachten.

1. Das Numpad funktioniert im Terminal nicht, also müssen alle Zahlen mit den Tasten oberhalb der Buchstaben eingegeben werden.
2. Die Navigation des grünen Markers geschieht über die Pfeiltasten. Eine Navigation mit dem Mauszeiger ist nicht möglich.
3. Die blauen Werte und die Syntax dürfen nicht verändert werden, da sonst das Importieren in die Software als Json-Datei fehlschlägt.
4. Der Name **muss** einzigartig sein und kann von keinem anderen Gerät verwendet werden. Bisher sind alle Namen mit „rack_x“ angegeben, wobei x die Nr. des Knotens ist. Der Name wird intern genutzt für den Datenaustausch mittels Zeroconf.

Abb. 34: PuTTY nano - Anzupassende Parameter in lila

Nachdem alle Werte angepasst wurden, kann die Datei mit „Strg + o“ gefolgt von „Enter“ gespeichert werden. Mit „Strg + x“ wird der Texteditor geschlossen.

5.8.4 Schritt 4 - Raspberry neustarten

Damit die Werte angenommen werden muss der Raspberry neugestartet werden.

- „sudo reboot“.

Alternativ ist auch ein Neustart der Software möglich.

- „sudo systemctl stop rack_monitor.service” beendet die Software.
- “sudo systemctl start rack_monitor.service” startet sie wieder.

5.8.5 Schritt 5 - Ordnungsgemäßen Betrieb prüfen (Debugging)

Die Software wird nun im Hintergrund laufen. Während des Betriebs werden einige Daten geloggt. Dies ermöglichet eventuell auftretende Probleme zu diagnostizieren und den Ordnungsgemäßen Betrieb festzustellen. Die Logger von Systemd gestarteten Anwendungen loggen in das Journald. Journald arbeitet eng mit dem Systemd zusammen und verwaltet alle Logdaten automatisch. So wird z.B. die Größe des Logspeicherbedarfs auf 10 % des maximalen Systemspeicherplatzes beschränkt. Wird das Limit erreicht, dann werden die ältesten Einträge gelöscht. Standardmäßig werden Logfiles permanent auf den Speicher gesichert. Es gibt aber auch die Möglichkeit die Logdaten nur im Arbeitsspeicher zu hinterlegen, sodass ein Neustart alle Daten entfernt. Für weitere

Konfigurationsmöglichkeiten des Journald empfehle ich eine eigene Recherche. Auf den Raspberry Pi wurden keine Änderung am Journald vorgenommen. Um die Logdaten einzusehen, muss im Terminal der folgende Befehl mit Parameterbeispielen ausgeführt werden:

sudo journalctl-b -u rack_monitor.service

Einige optionale Parameter werden noch kurz erklärt.

-u rack_monitor.service zeigt nur Einträge von der UNIT rack_monitor.service an

-b zeigt nur Einträge, die während des aktuellen Boots eingegangen sind

-f (follow) hooked sich ans Journald und zeigt nur alle kommenden Einträge an. Ältere Einträge sind nicht sichtbar!

Weitere Filteroptionen sind z.B. auf folgender Seite zu finden:

https://www.digitalocean.com/community/tutorials/how-to-use-journalctl-to-view-and-manipulate-systemd-logs-de

Im folgenden Bild ist noch ein Beispiel gezeigt, wie Logging Daten aussehen können.

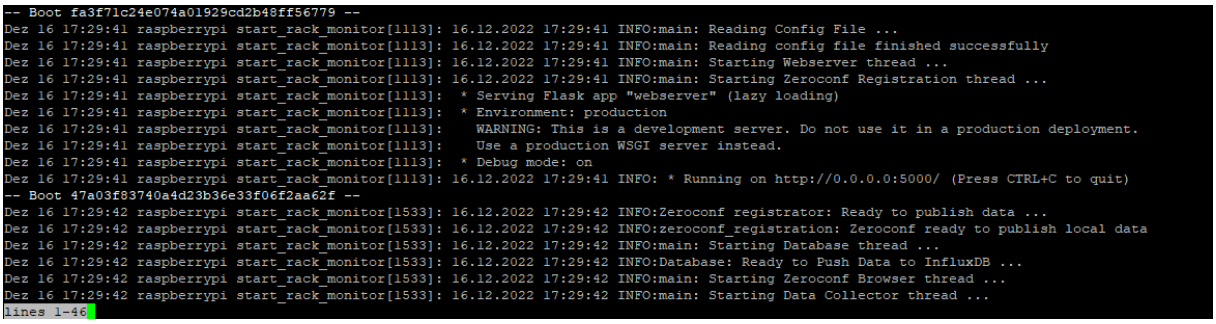


Abb. 35: Journald Log

Hinweis Mit den Pfeiltasten kann durch die Einträge gescrollt werden.

5.9 Alternative Startmethode mit Desktopverknüpfung

In diesem Abschnitt wird eine Alternative zum Systemd Start vorgestellt. Diese ermöglicht den manuellen Start der Software per Touchdisplay. Dies wurde sich gewünscht, weil auch per LabVIEW auf mit den Wechselrichtern kommuniziert werden soll und technisch ist nur eine Verbindung gleichzeitig möglich.

Hierfür wurde wird zuerst die Anwendung vom systemd gestoppt und vom Autostart entfernt. Die Terminalbefehle dafür sind die folgenden:

```
sudo systemctl stop rack_monitor.service
sudo systemctl disable rack_monitor.service
```

Nun wird auf dem Desktop eine Verknüpfung zu den Source Datei erstellt. Es wird also nicht mehr der Systemd Service gestartet! Dies bedeutet aber auch, dass die vorher von PyInstaller mit aufgelösten Abhängigkeiten (also Module) nachinstalliert werden müssen. Hierfür muss bei den Raspberry Pi Internet Zugriff eingerichtet werden und anschließend alle extra Module installiert werden. Die Installationsbefehle werden hier aufgelistet:

```
apt update
apt install influxdb
apt install influxdb-client
pip install requests
pip install -U pymodbus
pip install -U Flask
pip install zeroconf
```

Nun sind alle Pakete systemweit installiert und die main.py kann mit Python gestartet werden.

Hierfür im Terminal zu dem Source Code Verzeichnis navigieren. Anschließen kann zum Testen die Software mit folgendem Befehl im Terminal gestartet werden.

```
python main.py -d warning
```

Ich empfehle das Starten mit gezeigtem Parameter, weil hier nur wirklich relevante Logging Daten im Terminal erscheinen. Wie Verbindungsabbrüche oder Neuverbindungen, etc.

Läuft alles ohne Probleme kann die Desktopverknüpfung erstellt werden. Diese habe ich auch in Sciebo abgelegt, kann aber auch leicht selbst erstellt werden. Dafür auf dem Desktop eine Datei mit der Endung .desktop erstellen (z.B. “EMS.desktop“). Dann die Datei mit Texteditor öffnen und folgenden Inhalt hineinschreiben/kopieren:

```
[Desktop Entry]
Exec=lxterminal -e python *absoluter Pfad zu den Source Code Dateien*/main.py -d
"WARNING"
Name=EMS
Icon=/usr/share/icons/HighContrast/32x32/apps/preferences-system-network.png
Type=Application
StartupNotify=true
Terminal=true
```

Anschließend die per Doppelklick testen. Es muss danach noch auf Ausführen geklickt werden. Nun sollte ein Terminalfenster mit den normalen Loginformationen zu sehen sein. Wird das Terminal geschlossen beendet sich auch das Programm.

Fertig. Die Datei startet nun den Code mit Terminal manuell mittels einer Desktopverknüpfung.

6 Fazit

In meinem Fachpraktikum, meinem Praxisprojekt und meiner Bachelorarbeit konnte ich viel Neues lernen, was auch über den eigentlichen Modulplan des Studiums hinausging. Dies war eine leerreiche Zeit und ich bedanke mich insbesondere bei meinem Professor Herr Eberhard Waffenschmidt sowie auch meinem Ansprechpartner von Devolo Christop July. Beide konnte standen mit für Fragen stets zur Verfügung und haben mir Hilfestellungen gegeben.

Die Arbeit selber war durch einige Anfangsschwierigkeiten, ganz besonders den Kommunikationsschwierigkeiten im Netzwerk mit den Wechselrichtern geprägt. Doch zum Ende der Arbeit konnte ein stabiler Zustand hergestellt werden, was mich sehr gefreut hat.

Die Entwicklung der Software würde ich als erfolgreich bezeichnen. Alle Anforderungen der TH konnten befriedigt werden. Einzig die nicht parallele Kommunikation von mehreren Geräten gleichzeitig mit dem Wechselrichter ist schade. Dies ist aber auf das System des Wechselrichters zurückzuführen und ohne gänzlichen Austausch nicht realisierbar.

Die Skalierbarkeit lag nie im Fokus wird sich aber auf kleinere Systeme beschränken. Zeroconf kann nur in lokalen Netzwerken betrieben werden und nicht über Firewalls hinaus arbeiten. Zusätzlich steigt der Netzwerktraffic mit jedem Zeroconf Service an, wodurch eventuelle Überlastungen entstehen können. Bei den geringen Mengen im Labor ist dies jedoch kein Problem, sodass alle Algorithmen gut getestet werden können.

Hätte ich dieselbe Arbeit nochmal gemacht würde ich aus einigen Fehlern lernen und ein paar Sachen anders angehen. So würde ich den Data Gatherer vermutlich ganz entfernen und einen einheitlichen Aufbau nur mit Queues anstreben. Auch würde ich jetzt direkt mit dem Logging beginnen. Ich habe dies erst spät für mich entdeckt und vorher nur mit Print befehlen gearbeitet. Auch das war sehr lehrreich für mich, da gutes Error Management weiteres Arbeiten merklich verbessert. Nichts aussagende Fehler sollten stets vermieden werden.

Die Arbeit mit dem Raspberry Pi per Remote Zugriff fand ich äußerst nützlich und werde ich in zukünftige Projekte mit Debian Betriebssystemen übernehmen. Eventuell sogar auf weitere OS sollte dies möglich sein. Es war für mich sehr angenehm nicht alles umbauen zu müssen, besonders auf Grund meiner Rheuma Erkrankung.

Eine Sache, die ich gerne noch ausprobiert hätte, wäre die Nutzung der „Inspirationssoftware“ OpenEMS auf Java Basis. Ich finde Java spannend und hätte gerne auch weitere Eindrücke in diese Programmiersprache erhalten. Dies werde ich bei Gelegenheit nachholen.

Ich hoffe das die Ausarbeitung allen nachfolgenden Personen, die mit der Software arbeiten, ein erfolgreiches Arbeiten ermöglicht.

Viel Erfolg

Abbildungsverzeichnis

Abb. 1: OpenEMS System Architektur - **Quelle**: https://openems.github.io/openems.io/openems/latest/introduction.html (11.03.2023) ..4

Abb. 2: schematischer Aufbau des EMS (Funktionsprinzip).....5

Abb. 3: Producer-Consumer-Pattern.....6

Abb. 4: schematischer Aufbau des EMS (Modulübersicht) 7

Abb. 5: Struktogramm - main.py 9

Abb. 6: Struktogramm - webserver.py.....12

Abb. 7: Struktogramm - database.py 17

Abb. 8: Struktogramm - zeroconf_registration.py 22

Abb. 9: Struktogramm - zeroconf_browser.py 24

Abb. 10: Struktogramm - on_service_state_change.....26

Abb. 11: Struktogramm – DataCollector.py - Init 27

Abb. 12: Struktogramm - DataCollector.py - Inverter Thread 28

Abb. 13: Struktogramm - DataCollector.py - run.....28

Abb. 14: Struktogramm -DataCollector - Power Meter Thread.....29

Abb. 15: Struktogramm - DataGatherer.py 30

Abb. 16: PuTTY Startfenster 37

Abb. 17: PuTTY Terminalsession nach erfolgreicher Anmeldung.....38

Abb. 18: WinSCP Startfenster.....39

Abb. 19: WinSCP Hauptfenster.....40

Abb. 20: nano Terminalbefehl Beispiel 41

Abb. 21: NTP-Kommunikationsübersicht.....43

Abb. 22: Webserver im Browsertab 44

Abb. 23: PyInstaller - Konsolenbefehl zum Erstellen der Installationsdatei 49

Abb. 24: Terminal – Konsolenbefehl, um das Skript ausführbar zu machen 49

Abb. 25: Konsolenbefehl zum Ausführen des Skriptes.....49

Abb. 26: Zu sehen sind zwei erstellte Installationsskripte vom 06.05.22 und 23.11.22 49
Abb. 27: Raspberry Pi Imager 50
Abb. 28: SSH aktiviert und Passwort eingegeben 51
Abb. 29: WLAN Hotspot konfiguriert 51
Abb. 30: URL im Browser "fritz.box" 51
Abb. 31: Raspberry Pi mit zugewiesener IP und Namen Raspie-Bus3 52
Abb. 32: Internet route Übersicht 52
Abb. 33: PuTTY – Texteditor im Terminal per “nano” Konsolenbefehl 53
Abb. 34: PuTTY nano - Anzupassende Parameter in lila 54
Abb. 35: Journald Log 55

Abkürzungsverzeichnis

CLI ..... Command Line Interface
CMD ..... Command
DB ..... Datenbank
Dict ..... Dictionary
EMS ..... Energiemanagement System
GW ..... Gateway
IoT ..... Internet of Things
LAN ..... Local Area Network
NTP ..... Network Time Protocol
OS ..... Operating System
REST ..... Representational State Transfer
SSH ..... Secure Socket Shell
URL ..... Uniform Resource Locator