Fachochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit 5.Semester

Einbindung der devolo WiFi Geräte in ein Remote Network Monitoring System

Autor:
Yannick Gasper, 3277289

Betreuer:
Prof. Dr. Alexander Voß
M. Sc. Thomas Bong

Aachen, 15. Dezember 2022

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Seminararbeit mit dem Thema

Einbindung der devolo WiFi Geräte in ein Remote Network Monitoring System

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Yannick Gasper

Aachen, 15. Dezember 2022

Zusammenfassung

Diese Arbeit befasst sich mit der Einbindung der devolo WiFi Geräte in ein Remote Network Monitoring System. Dem Nutzer werden die Daten von, den in einem Netzwerk befindlichen, devolo Geräten in einer Visualisierungsapplikation dargestellt. Die devolo Geräte ermöglichen eine Datenübertragung über die hausinternen Strom-, Telefon- und Koaxialleitungen und somit einen einfachen Aufbau eines Netzwerks, ohne zusätzliche Leitungen verlegen zu müssen.

Im Rahmen dieser Arbeit wird das bestehende Remote Network Monitoring System erweitert. Auf der Visualisierungsapplikation sollen dafür die Daten von den WiFi Geräten in Form von verschiedenen Graphen übersichtlich angezeigt werden.

Inhaltsverzeichnis

1 Einleitung

Die devolo AG stellt Produkte zur Datenübertragung über die hausüblichen Strom-, Telefon- und Koaxialkabel her. Die Geräte der neusten Generationen basieren auf dem G.hn Standard der International Telecommunication Union (ITU) und bieten einen einfachen Netzwerkaufbau. Dabei besitzen manche devolo Produkte die Funktion ein eigenes WiFi Netzwerk aufzubauen oder die Signalstärke eines bestehenden Netzwerkes zu verstärken.

Mithilfe des Remote Network Monitoring Systems sollen die Daten von den devolo Geräten über Zeit visualisiert werden und dadurch die Möglichkeit geben, sie zu einem späteren Zeitpunkt analysieren zu können. Dadurch können beispielsweise Softwarefehler wie Memory Leaks oder unüblich hohe CPU Auslastung erkannt und behoben werden.

1.1 Ziel

Aktuell können Geräte schon über das Layer 2-Konfigurationsverwaltungsprotokoll LCMP, welches durch den Standard ITU-T G.9961 zur Kommunikation mit G.hn Geräten definiert ist, aus dem Remote Network Monitoring System heraus, angesprochen werden. Das heißt die grundlegende Kommunikation zwischen Powerline Communication (PLC) Gerät und dem System, dem System und der Datenbank, sowie zwischen Datenbank und der Visualisierungssoftware ist im Remote Network Monitoring System integriert.

Ziel dieser Arbeit ist es, die devolo WiFi Geräte in das bestehende Remote Network Monitoring System zu integrieren. Dies bedeutet, dass von allen WiFi Geräten Daten gesammelt werden können, diese dann in die entsprechende Datenbank geschrieben werden und dann in Form von Graphen, oder anderen graphischen Formen, visualisiert werden.

1.2 Gliederung

Die Arbeit ist in folgende Kapitel unterteilt:

Kapitel 2 Grundlagen In diesem Kapitel werden die technischen Grundlagen, sowie die genutzten Anwendungen und der aktuelle Stand des Remote Network Monitoring Systems vorgestellt.

Kapitel 3 Anforderungsanalyse Hier wird darauf eingegangen, welche Anforderungen an die Integration der WiFi Geräte gesetzt werden.

Kapitel 4 Implementierung Innerhalb dieses Kapitels werden die relevanten Funktionen und, die für diese Arbeit neu erstellten, Klassen anhand von Codeausschnitten beschrieben.

Kapitel 5 Zusammenfassung und Ausblick Zum Schluss werden die durchgeführten Arbeiten zusammengefasst und es werden Punkte zur Weiterentwicklung des Systems genannt.

2 Grundlagen

Im Folgenden werden die, für die Arbeit wichtigen, Grundlagen vorgestellt, die im weiteren Verlauf noch einmal aufgegriffen werden. Es wird vor allen Dingen ein Fokus auf die Technologien gesetzt, die in den devolo WiFi Geräten zum Einsatz kommen, auf die für das Paket relevanten Systeme und auf den jetzigen Stand des Remote Network Monitoring Systems.

2.1 HTTP

Um mit Webservern kommunizieren zu können, kommt das zustandslose, verbindungslose und medienunabhängige Hypertext Transfer Protocol (HTTP) zum Einsatz. Es bildet die Schnittstelle zwischen Anwendungen wie dem Webbrowser und Webservern und arbeitet somit die Anfragen von Anwendungen und die Antworten von Webserver ab. Beispiele für Anwendungen, die HTTP nutzen sind: Browser, RESTful APIs oder Javascript Object Notation Remote Procedure Call (JSON-RPC).

Der Uniform Resource Locator (URL) spezifiert die Ressource, die per HTTP angefragt wird. In dem Request Header wird die HTTP Methode, die HTTP Version und noch andere Parameter mitgeteilt, wie beispielsweise der User-Agent, der mitteilt über welche Applikation / Anwendung der Aufruf geschickt wird. Nachdem der Webserver die Anfrage erhalten und interpretiert hat, schickt er den Status des Ergebnisses der Anfrage und die eventuell angefragten Daten mit einem HTTP Response zurück an den Browser [1]. Die häufigsten Status Codes bei einer Response sind 200 (OK), 301 (Moved Permanently), 400 (Bad Request), 401 (Unauthorized) und 404 (Not Found) [2].

2.1.1 HTTP Methoden

Jeder Request Header, der an einen Webserver geschickt wird, beinhaltet eine HTTP Methode. Diese legt fest, wie man mit dem Server interagieren möchte. Im Folgenden werden die meist verwendeten Methoden aufgeführt und erklärt:

GET Über die GET Methode kann ein Client eine Ressource von einem Webserver anfragen. Diese wird über die URL spezifiziert. Dabei wird die angeforderte Ressource in dem Format zurückgeschickt, welche man in der HTTP Anfrage festgelegt hat. Die häufigsten HTTP Status Codes, die man beim GET erhält sind 200 (OK), 404 (Not Found) oder 400 (Bad Request).[2][3]

POST Mithilfe der POST Methode können Ressourcen, die in dem HTTP Body mitgeschickt werden, erstellt oder aktualisiert werden. Bei dieser Methode sind die am meisten genutzten Status Codes die 200 (OK) oder die 400 (Bad Request).[2][3]

2.2 OpenWrt

OpenWrt ist ein Linux-Betriebssystem für eingebettete Geräte, wie Router, welches ein beschreibbares Dateisystem und einen eigenen Paketmanager bereitstellt. OpenWrt liefert bei der Installation alle notwendigen Anwendungen mit, um den Betrieb im Netzwerk zu ermöglichen. Diese können jedoch durch den Paketmanager individuell erweitert werden. Außerdem können selbst geschriebene Programme für die OpenWrt Umgebung erstellt und eingebunden werden.[4] Die Firmware der devolo WiFi Geräte basiert auf OpenWrt.

2.2.1 Ubus

Ubus ist für die Interprozesskommunikation zwischen Daemons und verschiedenen Applikationen auf dem Gerät verantwortlich. Es stellt den *ubusd* daemon bereit, welches als Interface für andere Daemons dient, bei dem sie sich unter einem Namespace registrieren können. Jeder dieser Namespaces kann verschieden viele Funktionen mit unterschiedlich vielen Argumenten haben und diese können auch eine Nachricht zurück schicken.

'network.interface.lan'
'up': { }
'down': { }
'status': { }
'add_device': { 'name': 'String' }
'remove_device': { 'name': 'String' }

Die obige Grafik ist ein Beispiel für einen Service mit dem Namespace 'network.interface.lan'. Dieser hat die Funktionen up, down, status, add_device und remove_device. Dabei benötigen add_device und remove_device jeweils ein Argument als String.[5]

2.2.2 JSON-RPC

Um die von Ubus bereitgestellten Funktionen von den registrierten Services aus einem anderen Adressraum aufzurufen oder die Antwort auf eine Anfrage zurück zu schicken wird JSON-RPC genutzt. Dies ist ein Protokoll, welches den Aufruf von entfernten Methoden in anderen Systemen ermöglicht. Dabei ist es transportunabhängig und kann beispielsweise über Sockets und HTTP verwendet werden. Als Format wird Javascript Object Notation (JSON) genutzt, welches vier primitive Datentypen repräsentieren kann: Strings, Numbers, Booleans und Null. Außerdem unterstützt es zwei Skrukturtypen: Objekte und Arrays. JSON-RPC unterstützt die asynchrone Kommunikation, da alle Anfragen und Antworten eine ID enthalten, die eine Zuordnung vereinfachen. Es wird ein ein Client-Server Modell genutzt, welches die Kommunikation in beide Richtungen ermöglicht. [6]

Anfrage Die Anfrage, oder auch JSON-RPC-Call besteht aus einem JSON-Objekt, welches vom Client zum Server geschickt wird. Die Bestandteile einer Anfrage sind:
jsonrpc: Enthält den Namen der JSON-RPC-Version
method: Die Funktion die aufgerufen werden soll
params: Mögliche Parameter als Objekt oder Array
id: Eindeutiger Identifikator für die Anfrage [6]

Antwort Eine Antwort wird in Form eines JSON-Objekts zurück geschickt, falls es sich um eine Anfrage handelt. Die Antwort vom Server enthält:
jsonrpc: Enthält den Namen der JSON-RPC-Version
result: Das Rückgabeobjekt, wenn kein Fehler aufgetreten ist
error: Das Fehlerobjekt, beim Auftritt eines Fehlers
id: Gleicher Identifikator zu der geschickten Anfrage [6]

> Beispiel für einen Ubus Call mithilfe von curl: E.1

2.3 Kommunikationsformen

In Kommunikationsnetzwerken gibt es verschiedene Möglichkeiten ein Paket von einem Sender zu einem Empfänger zu senden. Drei dieser Kommunikationsarten werden im Folgenden näher erläutert:

2.3.1 Unicast

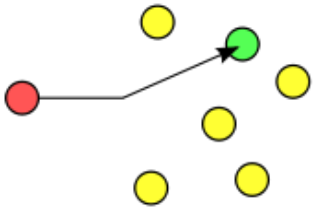

Abbildung 2.1: Unicast [7]

Unicast bezeichnet in der Informatik eine Verbindung zwischen zwei Teilnehmern. Dabei ist es unwichtig, ob diese bidirektional oder unidirektional abläuft.
In dem obigen Bild ist so eine Unicast Verbindung abgebildet. Hier geht der Sender (roter Punkt) mit dem Empfänger (grünen Punkt) eine unidirektionale Verbindung (Pfeil geht nur zu einer Seite). Da dieser Informationsfluss nur zwei Teilnehmer beinhaltet, liegt hier eine Unicast Verbindung vor. Ein Beispiel für eine Unicast Verbindung ist die Verbindung von einem Client zu einem Webserver.[8]

2.3.2 Broadcast

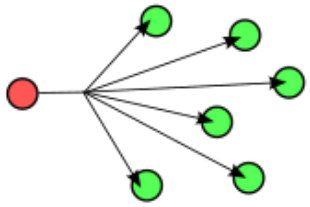

Abbildung 2.2: Broadcast [7]

Anders als beim Unicast gibt es bei einer Broadcast Verbindung mehr als zwei Teilnehmer. Hierbei werden alle Teilnehmer eines Netzwerks von dem Sender angesprochen. Die Empfänger können dabei selbst entscheiden, ob sie das empfangene Paket verarbeiten oder es verwerfen. Um alle Teilnehmer ansprechen zu können wird die in jedem Netzwerk reservierte Broadcast Adresse genutzt.
In dem gezeigten Bild erkennt man eine solche Verbindung. Hier schickt der Sender (roter Punkt) ein Datenpaket an alle Teilnehmer des Netzwerkes (grüne Punkte). Ein Beispiel für Broadcast ist Dynamic Host Configuration Protocol (DHCP).[9]

2.3.3 Multicast

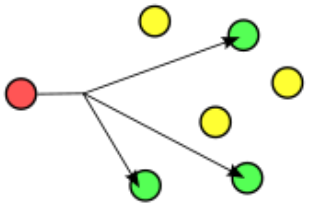

Abbildung 2.3: Multicast [7]

Multicast ist eine Punkt-zu-Mehrpunkt-Verbindung, da der Sender hierbei mehrere Empfänger anspricht. Der Unterschied zu Broadcast ist hierbei, dass ein Empfänger vorher mitgeteilt haben muss, Pakete an eine oder mehrere bestimmte Multicast-Adressen empfangen zu wollen. Dafür müssen sie Teil der Multicast-Gruppe sein, damit sie über die zugehörige Multicast Adresse angesprochen werden können. Für den Beitritt zu IPv4-Multicastgruppen nutzen Clients das Protokoll Internet Group Management Protocol (IGMP). Für IPv6 wurde die Funktionalität von IGMP in das Protokoll Multicast Listener Discovery (MLD) integriert.[10]

2.4 mDNS

Multicast Domain Name System (mDNS) ergänzt den Standard Domain Name System (DNS) um eine Multicastabfrage innerhalb des lokalen Netzwerks. Beim DNS werden Hostnamen (z.B. www.google.de) in IP Adressen aufgelöst (z.B. 142.250.179.163), indem ein DNS-Server angefragt wird, der die Zuordnung von Name und Adresse zur Verfügung

stellt. DNS vereinfacht die Nutzung des Internets für den Nutzer, da er sich die potenziell verändernden IP Addressen nicht merken muss, sondern nur einen leicht zu merkenden Namen, wie beispielsweise www.google.de.

2.4.1 DNS Records

DNS definiert verschiedene sogenannte Resource Records, die unterschiedliche Informationen enthalten, wie bspw. die IP Adresse oder dem Hostnamen. Im folgenden werden die für mDNS wichtigen Recordtypen vorgestellt:

A-Record Der Address Record wird meistens zur Zuordnung eines Hostname zu einer IPv4 Adresse genutzt. Demnach gibt dieser Record die IPv4-Adresse des angefragten Hostnamen zurück.[11]

AAAA-Record Dieser Record ist wie der A-Record, enthält aber die IPv6-Adresse des angefragten Hostnamen.[11]

PTR-Record Dieser Record gibt die eindeutige Bezeichnung, die benötigt wird, um den SRV und TXT Record aufzulösen. [11]

TXT-Record Hier können Informationen für den Nutzer in Form von einem Text oder maschinenlesbare Informationen stehen.[11]

SRV-Record In diesem Service locater Record steht zum Einen der Port des angefragten Service und die Information, die gebraucht wird, um zu wissen welche A und AAAA Record angefragt werden müssen.[11]

2.4.2 mDNS Records

Anders als bei DNS werden bei mDNS keine Anfragen an zentrale DNS-Server geschickt. Es wird über Mulitcast eine Gruppe von Teilnehmern eines Netzwerkes angefragt, ob der Hostname zu einem Netzteilnehmer passt. Befindet sich der gesuchte Teilnehmer in dem Netzwerk, antwortet dieser Netzwerkteilnehmer auch wieder über Multicast, dass es sich bei diesem Namen um ihn handelt. Dadurch wissen alle Netzwerkteilnehmer, die Teil der Multicast-Gruppe sind, dass es sich bei dem angefragten Hostnamen um genau diesen einen

Teilnehmer handelt. Somit nimmt jeder Benutzer der Gruppe die erhaltenen Informationen in ihren mDNS-Cache auf. Wichtig bei mDNS ist, dass nur Hostnamen aufgelöst werden die auf '.local' enden. Das heißt mDNS ist auf das lokale Netzwerk begrenzt und Anfragen die eine Top-Level-Domain beinhalten (z.B. ".de", ".com") werden von mDNS nicht verarbeitet.[12]

mDNS nutzt dieselben Records wie DNS. Um den Zusammenhang zwischen den verschiedenen Records besser verstehen zu können, wird dies im Folgenden anhand eines Beispiels erläutert.

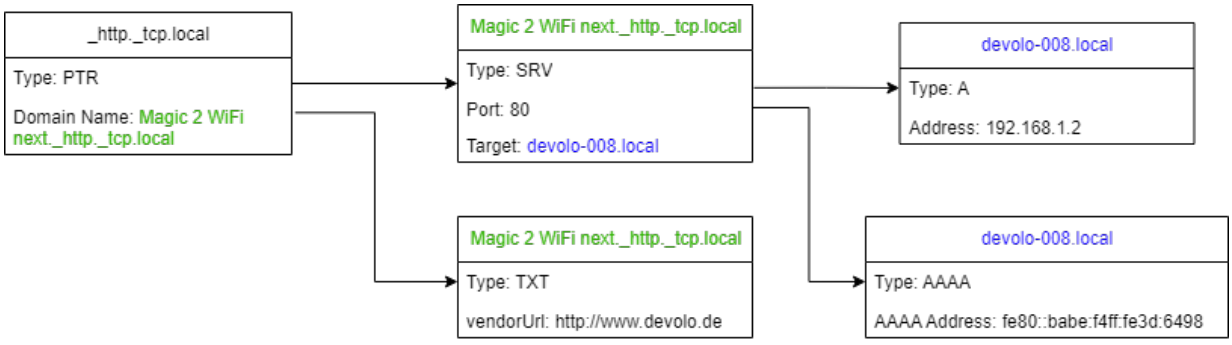

Abbildung 2.4: mDNS Beispiel

In dem obigen Beispiel wird in einem lokalen Netzwerk nach dem Service *_http._tcp.local* gesucht. Alle Teilnehmer des Netzwerks, die den zugehörigen PTR Record besitzen schicken ihren zurück. In diesem PTR Record steht der Domain Name des Teilnehmers, womit der SRV und TXT Record aufgelöst werden kann. Der TXT Record enthält in dem Beispiel die URL zu dem Hersteller des Gerätes. Allgemein können dort weitere Informationen stehen, jedoch ist dies abhängig von der jeweiligen Konfiguration des Netzwerkteilnehmers. In dem SRV Record steht der Port und das Target, womit die beiden nächsten Records angefragt werden können. Durch den Target kann zum Einen der A Record mit der IPv4 Adresse des Teilnehmers abgefragt werden, sowie der AAAA Record, der die IPv6 Adresse enthält.

2.5 Time series databases

Eine Time series database (TSDB) ist eine Datenbankart, die darauf optimiert ist Daten mit einem Zeitstempel zu speichern. Diese time series data sind Datensammlungen, die von einer Quelle in einem bestimmten Zeitintervall abgefragt werden. Das ermöglicht das Erfassen von Veränderungen der Datenpunkte über Zeit. Dabei kann der Zeitraum der Veränderung variabel sein (Sekunden, Tage oder Jahre). Wichtig ist, das die Datenpunkte,

die neu gespeichert werden, immer als neuer Eintrag gewertet werden und nicht einen schon existierenden Datenpunkt überschreiben. Ein Beispiel für den Einsatz von TSDB ist, die Speicherung von Sensordaten über Zeit, um zu schauen, ob es zeitliche Korrelationen gibt.[13]

2.5.1 InfluxDB

InfluxDB ist eine Open Source TSDB, welche ein bestimmtes Format zum Speichern von Daten nutzt: '<measurement-name> <tag-set> <field-set> <timestamp>'. Der measurement-name ist ein String, welcher festlegt, zu welcher Messung dieser neuer Eintrag gehört. Der 'tag-set' ist eine Reihe von Schlüssel-Wert Paaren aus Strings, die den Datensatz spezifizieren. Der 'field-set' ist eine Reihe von Schlüssel-Wert Paaren aus Integern, Floats, Booleans und Strings, die den Wert enthalten. Der 'timestamp' kann in Sekunden, Millisekunden, Mikrosekunden oder Nanosekunden angegeben werden.

> cpu,host=serverA,region=uswest idle=23,user=42,system=12 1464623548s

Auf der Festplatte werden die Daten in einem spaltenförmigen Format organisiert, in dem zusammenhängende Zeitblöcke für die Messung, die Tags und das Feld festgelegt werden.[13]

2.6 Grafana

Grafana ist eine Open-Source Anwendung, die plattformunabhängig Daten aus verschiedenen Datenquellen in Graphen oder andere Visualisierungsmöglichkeiten graphisch darstellen kann. Diese können in Dashboards zusammengelegt werden, sodass auf einem Blick mehrere Informationen angezeigt werden können. Grafana stellt durch zusätzliche Plug-ins die Unterstützung von vielen verschiedenen Anzeigemöglichkeiten und Datenquellen zur Verfügung. Dabei unterstützt Grafana Metriken, Logging und Tracing.
Grafana kommt meistens für Monitoring-Aufgaben und die Visualisierung von Messdaten zum Einsatz und arbeitet dabei mit vielen Zeitreihen-Datenbanken (2.5) wie InfluxDB (2.5.1) oder auch mit relationalen Datenbanken. Entwickelt wurde es von Torkel Ödegaard im Jahr 2014 und wird von Grafana Labs als Open-Source Anwendung weiterentwickelt.[14]

> Beispiel für ein Grafana Dashboard: E.1

3 Anforderungsanalyse

Eine Anforderungsanalyse ist ein wesentlicher Bestandteil der agilen Softwareentwicklung und wird meistens in Zusammenarbeit mit dem Kunden erstellt. Hierbei wird in einem sogenannten Lastenheft festgehalten, was das zu erstellende System zu leisten haben soll, damit für jeden Teilhabenden ersichtlich ist worauf man hinarbeitet. So eine Analyse ist für viele Softwareprojekte von essenzieller Wichtigkeit, da sich in dem laufenden Entwicklungsprozess immer wieder Sachen ändern können oder auch der Kunde noch etwas zu seinen Wünschen an das System ergänzen möchte.

Bei einer Anforderungsanalyse unterscheidet man im Wesentlichen zwischen zwei Anforderungen: Die funktionalen Anforderungen und die nicht-funktionalen Anforderungen.

funktionale Anforderungen Diese Anforderungen beschäftigen sich mit der Frage, was das System leisten soll. Dabei wird ein besonderes Augenmerk auf die zu benötigten Eingaben, die Verarbeitungsschritte, die das System braucht um eine Ausgabe geben zu können, und die eigentliche Ausgabe gelegt. Außerdem soll darauf geachtet werden, wie sich das System in speziellen Situationen verhält und welche Schritte es auf jeden Fall nicht machen soll.[15]

nicht-funktionale Anforderungen Nicht-funktionale Anforderungen spezifizieren die Art und Weise, wie das System zu arbeiten hat. Hierbei wird vor allen Dingen auf die Themen Zuverlässigkeit, Performance und Benutzerfreundlichkeit geachtet.[15]

3.1 Funktionale Anforderungen

Im Folgenden wird erörtert, welche Anforderungen an die Ziele dieser Arbeit gestellt wurden. Diese wurden in Diskussionen mit dem Auftraggeber festgelegt.

Gerätesuche Sobald das System an ein Netzwerk von devolo WiFi Geräten angeschlossen ist, sollen diese Geräte auch direkt im Netzwerk gesucht und gefunden werden, damit dem Nutzer die Daten aller seiner Geräte angezeigt werden können.

Datenabfrage vom Gerät Ist das System an ein devolo Netzwerk von WiFi Geräten angeschlossen, sollen die zu abbildenden Daten abgefragt werden. Dafür sollen die Daten des Gerätes in einem bestimmtem Intervall unabhängig vom Nutzer abgefragt werden.

Datenspeicherung Möchte der Benutzer nicht nur die aktuellen Daten analysieren, sondern auch die Daten die in der Vergangenheit abgefragt wurden, müssen diese Informationen auch gespeichert werden, damit sie abrufbar bleiben. Dafür sollen die abgefragten Daten in einer Datenbank gespeichert werden.

Visualisierung der Daten Es sollen wesentliche Geräteparameter visualisiert werden, die Aufschluss über den Gerätezustand geben können. Darunter fallen unter anderem die CPU und Memory Auslastung, die zeitlich aufgelöst eine gute Übersicht bieten Softwarefehler wie Speicherlecks zu zeigen.

3.2 Nicht-funktionale Anforderungen

Erscheinungsbild Das Erscheinungsbild des Dashboards sollte sauber und sortiert sein, damit man auf einem Blick die Daten die zusammenhängen auch sieht. Dabei sollte auch eine gewisse Einheitlichkeit zwischen den Graphen herrschen, wie z.B. bei der Farbwahl.

Vollständigkeit Es sollten keine Daten, die wichtig sind, außen vorgelassen werden. Das heißt es sollen alle für das Gerät relevanten Daten abgefragt und auch visualisiert werden.

4 Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung, der einzelnen Komponenten, in das Remote Network Monitoring System. Dabei wird zu Anfang auf die einzelnen Komponenten selber eingegangen und am Ende wird die Zusammenarbeit dieser erläutert. In der gesamten Implementierung wurde mit der Programmiersprache Python gearbeitet.

4.1 Device Discovery

Durch die Einbindung der WiFi Geräte musste eine Art des Gerätesuchens in das Paket eingebaut werden, die vorher noch nicht vorhanden war. Da die einzubindenden Geräte einen WiFi Chip beinhalten und dieser für die Abhandlung der Prozesse zuständig ist, kann man auch nur über diesen die IPv4 Adresse des Gerätes herausfinden. Um diesen Prozess der 'Device Discovery' zu realisieren wurde die Klasse **mDNS** erstellt.

4.1.1 mDNS Klasse

Wie schon in Kapitel 2.4 erwähnt, ist mDNS ein Protokoll, der eine Gruppe von Teilnehmern in einem Netzwerk über Multicast anspricht. Um die devolo WiFi Geräte zu detektieren, die die device API anbieten, müssen Geräte gefunden werden, die per mDNS mitteilen, dass sie diesen Service anbieten. Dies geschieht mittels des PTR-Records **_dvl-deviceapi._tcp.local.**. Um dies umzusetzten, wurde auf das schon bestehende Paket *zeroconf* zurückgegriffen.[16] Dieses stellt eine Bibliothek zur Verfügung, um über mDNS in einem Netzwerk nach Geräten zu suchen, welche die device API anbieten.

zeroconf

Für die Verwendung von *zeroconf* muss eine eigene Listener Klasse erstellt werden, die von *ServiceListener* erbt. Diese ist verantwortlich für die Speicherung der gefundenen Geräte, die die device API anbieten. Für die Feststellung und Speicherung der Geräte, besitzt die *MyListener* Klasse ein Attribut *services*. Dieses wird mit den gefundenen Hosts gefüllt, falls die *add_service* Methode aufgerufen wird und es werden Hosts aus dem Attribut gelöscht, falls *remove_service* aufgerufen wird.

```
1 from zeroconf import ServiceBrowser, ServiceListener, Zeroconf
2
3 class MyListener(ServiceListener):
4
5     def __init__(self):
6         self.services = {}
7
8     def remove_service(self, name):
9         print(f"Service {name} removed")
10        self.services[name] = None
11
12    def update_service(self, zc: Zeroconf, type_: str, name: str):
13        print(f"Service {name} updated")
14
15    def add_service(self, zc: Zeroconf, type_: str, name: str):
16        info = zc.get_service_info(type_, name)
17        self.services[name] = info
18
19 def _get_mdns_services(service_type: str):
20    zeroconf = Zeroconf()
21    listener = MyListener()
22    browser = ServiceBrowser(zeroconf, service_type, listener)
23    time.sleep(5)
24    zeroconf.close()
25    return(listener.services)
```

Listing 4.1: mDNS Klasse

_get_mdns_services Diese Funktion ist die Schnittstelle über die die Geräte in dem lokalen Netzwerk über eine Serviceanfrage gesucht werden. Diese bekommt einen service_type in Form eines Strings, beispielsweise *_dvl-plcnetapi._tcp.local.* übergeben. Mit diesem übergebenen *service_type* wird dann in dem lokalen Netzwerk angefragt, wer diesen Service besitzt. Um danach suchen zu können, wird eine Instanz von der *Zeroconf* Klasse, der eigenen Listener Klasse *MyListener* und der *ServiceBrowser* Klasse erstellt. Dabei wird dem ServiceBrowser Objekt das Zeroconf Objekt, den gesuchten Service und das Listener Objekt übergeben. Durch das Erstellen des ServiceBrowsers startet die Suche nach Hosts. Falls sich ein Host meldet, dem dieser Service bekannt ist, wird die *add_service* Methode des übergebenen Listeners aufgerufen und der Host landet in dem Dictionary *services*. Nach

5 Sekunden wird die Suche beendet, da davon ausgegangen wird, dass in diesem Zeitraum sich jedes Gerät gemeldet hat, und es werden alle gefundenen Hosts zurückgegeben.

4.2 Geräterepresentation

Um die angeschlossenen Geräte nach der Device Discovery auch individuell ansprechen zu können und eine leichte Erweiterbarkeit gewährleisten zu können, wurde die Klasse **wifiDevice** erstellt, die ein WiFi Gerät als Klasse repräsentieren soll.

__init__ Der Konstruktor der Klasse erhält eine Vielzahl an Argumenten, welche die Eigenschaften des Gerätes beinhalten, beispielsweise die Seriennummer des Gerätes (*sn*), die aktuelle Firmware Version (*firmware_version*) oder die *path* Variablen für beide APIs, welche ein für jedes Gerät zufälliger Präfix ist, der bei einem zurückstellen auf Werkseinstellungen neu gesetzt wird. Die Variable *_ubus_session* wird nicht übergeben und wird bei der Erstellung eines Objektes der Klasse auf None gesetzt. Diese Variable steht in Verbindung mit der Methode *ubus*.

ubus Sie dient dazu, dass dem WiFiDevice Objekt nur ein Objekt der Ubus Klasse zugeordnet werden kann. Falls das WiFiDevice Objekt noch kein Ubus Objekt enthält wird eines mit den Standard Kennungen und der jeweiligen IPv4 Adresse erstellt. Es gewährleistet, dass jedes Gerät ein Ubus Objekt besitzt, damit auch bei den Ubus Calls die richtige IPv4 Adresse genutzt werden kann und dass die erhaltenen Daten auch einem Gerät zugeordnet werden können. Dem zu erstellenden Ubus Objekt werden die Argumente *username* und *password* übergeben. Diese sind standardmäßig auf auf 'root' und einen leeren String gesetzt, da davon ausgegangen wird, dass die Firmware, die auf dem WiFi Gerät läuft eine Entwickler Firmware ist und dort diese Standardwerte genutzt werden.

```
1 class wifiDevice:
2     def __init__(self,
3                  plcmac: str,
4                  ip: str,
5                  port: int,
6                  mt: int,
7                  sn: int,
8                  firmware_version: str,
```

```
9                 firmware_date: str,
10                product: str,
11                version: str,
12                pathdeviceapi: str,
13                pathplcnetapi : str,
14                api: List[str]):
15        self.plcmac = plcmac
16        self.ip = ip
17        self.port = port
18        self.mt = mt
19        self.sn = sn
20        self.wifi0 = None # 2.4 or 5 as number
21        self.wifi1 = None # 2.4 or 5 as number
22        self.firmware_version = firmware_version
23        self.firmware_date = firmware_date
24        self.product = product
25        self.version = version
26        self.pathdeviceapi = pathdeviceapi
27        self.pathplcnetapi = pathplcnetapi
28        self.api = api
29        self._ubus_session = None
30
31        self._differentitate_wifi0_wifi1()
32
33    def ubus(self):
34        if self._ubus_session:
35            return self._ubus_session
36        else:
37            self._ubus_session = Ubus(username="root", password="",
    address=self.ip)
38            return self._ubus_session
```

Listing 4.2: wifiDevice Klasse

4.3 Geräteschnittstelle

Um nun von den gefundenen Geräten Daten über Ubus abfragen zu können wurde eine **Ubus** Klasse erstellt, die dafür sorgt, dass die Anfragen über HTTP geschickt werden und die Antwort, die die angefragten Daten enthalten, auch zurück gegeben werden. Im weiteren

Verlauf wird nun auf die genannte Klasse mit ihren Methoden eingegangen.

__init__ Diese Methode ist der Konstruktor der Ubus Klasse. Er erhält einen Username, ein Passwort und eine Adresse als String. Bei dem Usernamen und dem Passwort handelt es sich um die Argumente, die man bei einem Ubus Call mitschicken muss, um zu überprüfen, ob der Benutzer, der die Anfrage stellt, Zugriff auf die aufgerufene Funktion hat. Bei der Adresse handelt es sich um die IPv4 Adresse des devolo WiFi Gerätes, an den die Anfragen geschickt werden sollen. Die Ubus Klasse besitzt das Attribut *_rpc_session* welches bei der Initialisierung des Objekts auf None gesetzt wird. In diesem wird später die aktuelle Ubus Sitzung, welche eine eindeutige Sitzungs ID beinhaltet, zwischengespeichert, sodass nicht bei jedem neuen Ubus Call eine neue Sitzung eröffnet werden muss, sondern die noch bestehende wieder verwendet werden kann.

```
1 import json
2 import requests
3
4 class Ubus:
5     def __init__(self, username: str = None, password: str = None,
      address: str = None, port: str = "80") -> None:
6         self.username = username
7         self.password = password
8         self.address = address
9         self.port = port
10        self._rpc_session = None
```

Listing 4.3: Ubus Konstruktor

_get_ubus_rpc_session Bei dieser Methode handelt es sich um eine einfache Get Methode, die die abgespeicherte Sitzung zurückgibt oder bei einer nicht vorhandenen Sitzung eine neue erstellt.

_get_new_rpc_session Diese Methode wird aufgerufen, falls eine neue Ubus Sitzung erstellt werden muss. Dabei werden, die in dem Objekt gespeicherten Attribute *username* und *password* verwendet. Nach der neu erstellten Sitzung, wird in der HTTP Antwort nach der Sitzungsnummer gefiltert und diese wird in *_rpc_session* abgespeichert. Dieser

HTTP Aufruf ist von OpenWrt so definiert und kann in der Dokumentation nachgeschaut werden.

```
1 def _get_ubus_rpc_session(self):
2         if not self._rpc_session:
3             self._rpc_session = self._get_new_rpc_session()
4         return self._rpc_session
5 
6     def _get_new_rpc_session(self):
7         params = {
8             "username": self.username,
9             "password": self.password,
10        }
11        res = self._call(section="session", rpc="login", ubus_rpc_session
    ="00000000000000000000000000000000", params=params)
12        return res.get('ubus_rpc_session')
```

Listing 4.4: Methoden zur Verwaltung der Ubus Sitzung

_call Hier wird die eigentliche HTTP Anfrage an das jeweilige Gerät abgeschickt. Dafür benötigt die Methode die Variablen:

- *section*, welcher als String übergeben wird und aussagt, welcher Service auf dem Gerät angesprochen wird.
- *rpc*, welcher die aufzurufende Funktion als String beinhaltet.
- *ubus_rpc_session*, die die aktuelle Ubus Sitzung beinhaltet und standardmäßig auf None gestellt ist, da in der Methode selber nocheinmal die akutelle Sitzung über _get_ubus_rpc_session (4.3) abgefragt wird.
- *params*, welche die Parameter für die aufzurufende Funktion in einem Dictionary beinhalten.

Mit diesen Variablen kann dann eine HTTP Anfrage gebildet werden und an die gespeicherte Adresse mit den vorher festgelegten Daten geschickt werden. Die Methode gibt das Ergebnis dieser Anfrage im JSON Format zurück. Diese kann entweder die angefragten Daten zurückliefern oder eine 'Permission Denied' Statusmeldung.[5]

```
1 def _call(self, section: str, rpc, ubus_rpc_session: str = None, params:
    dict = {}, timeout: int = 900):
2         params["timeout"] = timeout
3         req = {
4             "jsonrpc": "2.0",
5             "method": "call",
6             "params": [
7                 ubus_rpc_session if ubus_rpc_session else self.
    _get_ubus_rpc_session(),
8                 section,
9                 rpc,
10                params
11            ]
12        }
13        res = requests.post(url=f"https://{self.address}/ubus", data=json
    .dumps(req), verify=False)
14        return json.loads(res.content.decode('UTF-8')).get("result")[1]
```

Listing 4.5: Ubus Call Methode

4.4 Datenspeicherung

Um die angefragten Daten auch abspeichern und später nutzen zu können, werden sie in die aufgesetzte InfluxDB geschrieben. Dafür wird das Python Paket influxdb genutzt. Durch die Nutzung diese Paketes ist es möglich unter Angabe von Host, Port, Benutzer, Passwort und Datenbankname Einträge zu hinterlegen.

write_to_influx_db Diese Funktion dient der Abspeicherung der zuvor gesammelten Daten und mit dieser ist es möglich Daten, die in einem Schlüssel-Wert-Paar vorliegen, in eine Datenbank zu schreiben. Dafür wird einmal die *config* Variable benötigt, die vom Typ RmonInfluxConfig ist. Diese beinhaltet die Konfigurationen der Datenbank, wie Host, Port, Benutzer, Passwort und Datenbankname. Dadurch kann ein *InfluxDBClient* erstellt werden, der durch das influxdb Paket mitgeliefert wird. Dieser *InfluxDBClient* besitzt eine Methode *write_points*, die Daten im JSON Format in die Datenbank, des erstellten Clients, schreiben kann. Diese Methode wird mit den übergebenen Variablen *measurements*, *tags* und *fields*

aufgerufen, welche den Namen des Messwertes, die gesetzten Tags und den Messwert selber enthalten. Die Methode gibt bei einer erfolgreichen Datenspeicherung *True* zurück. [17]

```
def write_to_influx_db(config: 'RmonInfluxConfig',
                       measurement: str,
                       tags: Dict[str, str],
                       fields: Dict[str, float]
                       ):
    logging.debug(f"-writing to influxDB of mac={measurement}")
    client = InfluxDBClient(host=config.db_host,
                            port=config.db_port,
                            username=config.db_user,
                            password=config.db_pass,
                            database=config.db_name
                            )
    json_body = [
        {"measurement": measurement,
         "tags": tags,
         "fields": fields
         }
    ]
    client.write_points(points=json_body)
```

Listing 4.6: InfluxDB Funktion

4.5 Integration ins bestehende Paket

In diesem Unterkapitel wird erörtert, wie die in den vorherig genannten Kapiteln erwähnten Komponenten, in das bestehende Paket eingebunden wurden, damit die Daten von den WiFi Geräten auf Grafana visualisiert werden können.

discover Diese Funktion dient dazu alle verbundenen WiFi Geräte über die vorher erwähnte *_get_mdns_services* Methode in einem lokalen Netzwerk zu finden. Von allen gefundenen Geräten wird dann jeweils ein Objekt der Klasse **wifiDevice** (4.2) erstellt.

```
def discover():
    ret = {}
    deviceapi = get_mdns_devices_dvl_deviceapi()
    plcnetapi = get_mdns_devices_dvl_plcnetapi()
```

```
5
6     for deviceapidict, deviceapiv in deviceapi.items():
7         ret[deviceapiv.get('SN')] = delosDevice(plcmac=plcnetapi[
    deviceapidict].get('PlcMacAddress') if deviceapidict in plcnetapi
    else "",
8                                         ip=deviceapiv.get('address'),
9                                         port=deviceapiv.get('port'),
10                                        mt=deviceapiv.get('MT'),
11                                        sn=deviceapiv.get('SN'),
12                                        firmware_version=deviceapiv.get('
    FirmwareVersion'),
13                                        firmware_date=deviceapiv.get("
    FirmwareDate"),
14                                        product=deviceapiv.get('Product'),
15                                        version=deviceapiv.get('Version'),
16                                        pathdeviceapi=deviceapiv.get('Path').
    split("/", 1)[0],
17                                        pathplcnetapi=plcnetapi.get(
    deviceapidict).get('Path').split("/", 1)[0] if deviceapidict in
    plcnetapi else "",
18                                        api=[deviceapiv.get('Path').split("/"
    , 1)[1], plcnetapi.get(deviceapidict).get('Path').split("/", 1)[1] if
     deviceapidict in plcnetapi else ""])
19     return ret
```

Listing 4.7: Discover Funktion

Diese wird einmal beim Start der main Funktion des Systems aufgerufen, um einen Überblick zu erhalten, welche Geräte im Netzwerk sind. Daraufhin wird für jedes Gerät ein Dashboard für Grafana erstellt, welches über die schon vorhandenen Funktionen abläuft. Diese mussten nur dementsprechend angepasst werden, dass ein String übergeben werden kann, der den Standort einer Templatedatei angeben kann, sodass die Möglichkeit besteht verschiedene Dashboard Templates zu nutzen, da es für die WiFi Geräte ein anderes Template gibt als für die PLC Geräte.
In der darauf folgenden Endlosschleife wird ein weiteres Mal eine Device Discovery (4.1) durchgeführt, damit falls in dem lokalen Netzwerk Geräte hinzugefügt oder entfernt werden diese auch im weiteren Verlauf berücksichtigt werden. Anschließend werden, über das in 4.2 erwähnte ubus Attribut, die ubus Calls abgesetzt und die Daten in Form eines Dictionarys zurückgegeben. Diese Daten werden wieder in einem Dictionary gespeichert, wo der Schlüssel

die Seriennummer des Gerätes ist und der Wert das erhaltene Dictionary mit den Daten als Schlüssel-Wert Paare. Dies geschieht mit allen gefundenen Geräten. Nun hat man von allen in dem Netzwerk befindlichen Geräten die Daten erhalten und diese werden mit der in 4.4 erwähnten Methoden in die Datenbank geschrieben. Diese Schleife startet den nächsten Durchlauf in einem vorher festgelegten Zeitabstand erneut, sodass eine kontinuierliche Datenabfrage und -sicherung gewährleistet ist.

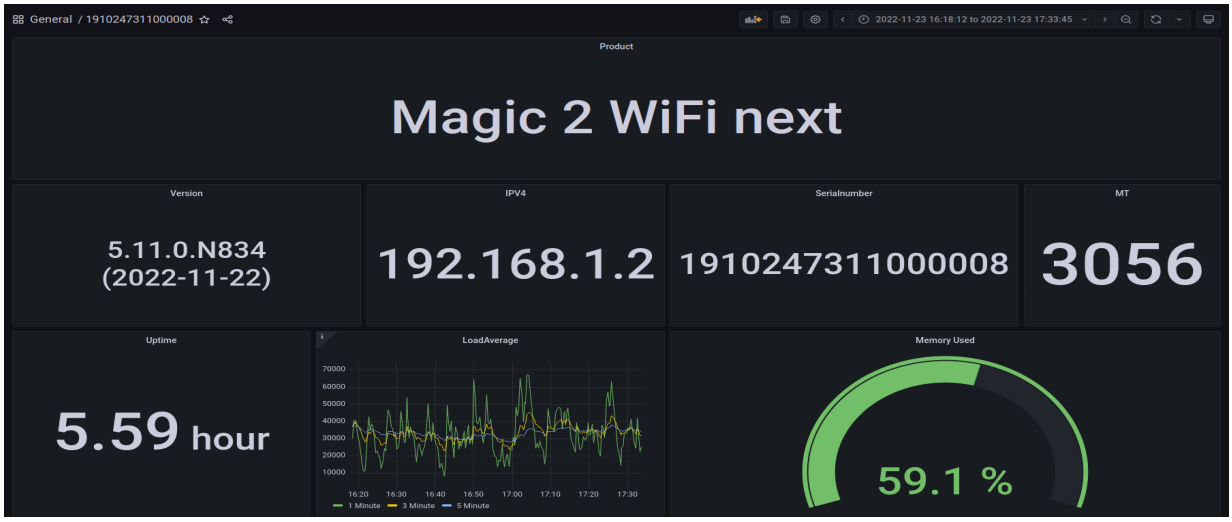

Abbildung 4.1: Grafana Dashboard für WiFi Geräte

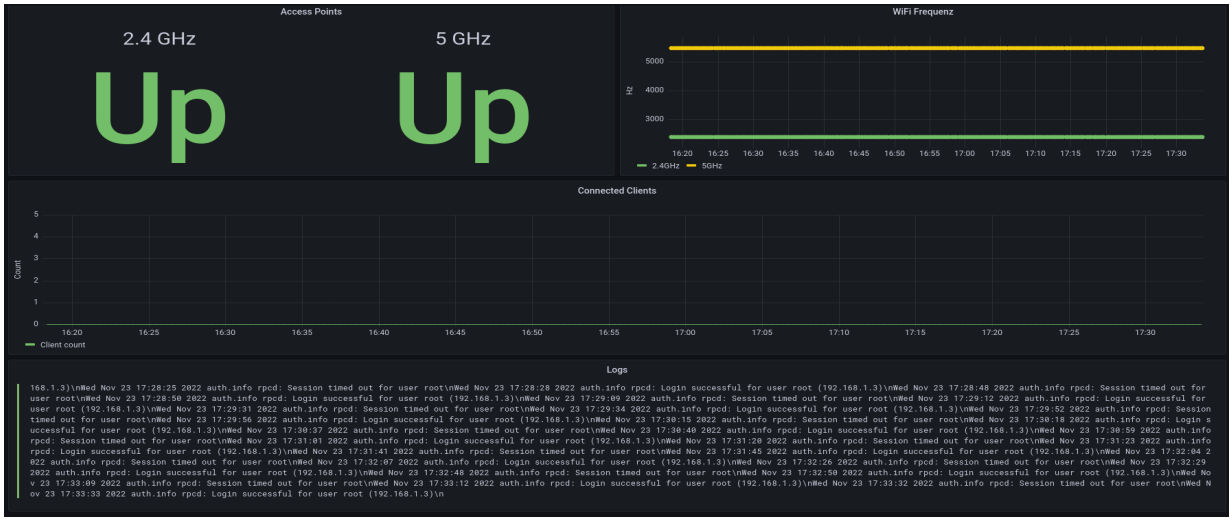

Abbildung 4.2: Grafana Dashboard für WiFi Geräte (unterer Teil)

5 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, die devolo WiFi Geräte in ein bestehendes Remote Network Monitoring System zu integrieren. Dies konnte erfolgreich umgesetzt werden.

5.1 Zusammenfassung

Im Rahmen dieser Seminararbeit wurde das bestehende Remote Network Monitoring System mit den devolo WiFi Geräten erweitert. Hierbei ist eine neue Art des Device Disovery hinzugefügt worden, welche über mDNS, die in einem lokalen Netzwerk befindlichen, Geräte findet. Um Informationen von diesen Geräten erhalten zu können, ist das System mit der Funktion erweitert worden, Ubus Calls an ein WiFi Gerät absetzten zu können. Die erhaltenen Daten werden dann, mithilfe der in dem System schon vorhandenen Funktion, in die Datenbank geschrieben. Hierbei wird als eindeutiger Schlüssel nicht mehr die MAC Adresse, sondern die Seriennummer des Gerätes verwendet. Zuletzt ist das automatische Aufbauen des Grafana Dashboards angepasst worden, sodass es nun verschiedene Dashboard Templates nutzen kann, um die zwei verschiedenen Dashboards für die PLC und WiFi Geräte aufbauen zu können.

5.2 Ausblick

Das System bietet noch viele Möglichkeiten es weiter zu entwickeln. So könnte man in Zukunft noch mehr Ubus Calls hinzufügen und diese Daten mit geeigneten Graphen auf dem Dashboard visualisieren um noch genauere Informationen von dem Gerät erhalten zu können.

Des Weiteren werden aktuell passwortgeschützte Geräte nicht unterstützt, da es keine Möglichkeit gibt komfortabel ein Gerätepasswort pro Gerät zu konfigurieren.
Außerdem besteht aktuell das Problem des IPv6 Supports. Das Paket *zeroconf* unterstützt zwar IPv6 jedoch hat man keinen Zugriff auf die Daten, die in den einzelnen Records stehen. Dazu sind Erweiterungen des Open Source Pakets notwendig, damit man sowohl auf die Information über die IPv4- als auch auf die IPv6-Adresse zugreifen kann.
Zudem wird momentan immer wieder eine neue Verbindung zu der Datenbank aufgebaut,

falls man etwas in diese schreiben möchte. Dies könnte in Zukunft so abgeändert werden, dass die Verbindung zu der Datenbank bestehen bleibt und erst bei einem auftretenden Fehler geschlossen wird. Dies spart etwas Zeit, da die Verbindung nicht immer neu aufgesetzt werden muss.

A Quellen

[1] developer.mozilla.org. *Request Header*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Request_header. Aufgerufen am: 26.10.2022.

[2] developer.mozilla.org. *HTTP response status codes*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. Aufgerufen am: 27.10.2022.

[3] developer.mozilla.org. *HTTP request methods*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Aufgerufen am: 10.11.2022.

[4] openwrt.org. *OpenWrt*. URL: https://openwrt.org/. Aufgerufen am 03.12.2022.

[5] openwrt.org. *ubus (OpenWrt micro bus architecture)*. URL: https://openwrt.org/docs/techref/ubus. Aufgerufen am 03.12.2022.

[6] jsonrpc.org. *JSON-RPC 2.0 Specification*. URL: https://www.jsonrpc.org/specification. Aufgerufen am 03.12.2022.

[7] voiceoverit.de. *Unterschied: Broadcast, Multicast, Unicast, Anycast*. URL: https://voiceoverit.de/blog/netzwerk/broadcast-multicast-unicast-anycast/121/. Aufgerufen am 21.11.2022.

[8] ionos.de. *Unicast: Gezielte Verbindung zwischen zwei Punkten*. URL: https://www.ionos.de/digitalguide/server/knowhow/unicast/. Aufgerufen am 21.11.2022.

[9] ionos.de. *Was ist ein Broadcast und wie funktioniert er?* URL: https://www.ionos.de/digitalguide/server/knowhow/broadcast/. Aufgerufen am 21.11.2022.

[10] ionos.de. *Multicast: Mehrpunktverbindungen für effiziente Datenübertragung*. URL: https://www.ionos.de/digitalguide/server/knowhow/broadcast/. Aufgerufen am 21.11.2022.

[11] ionos.de. *DNS-Records: Wie funktionieren DNS-Einträge?* URL: https://www.ionos.de/digitalguide/hosting/hosting-technik/dns-records/. Aufgerufen am 21.11.2022.

[12] ionos.de. *Multicast DNS: Alternative Namensauflösung im kleinen Stil*. URL: https://www.ionos.de/digitalguide/server/knowhow/multicast-dns/. Aufgerufen am 18.11.2022.

[13] influsdata.com. *Time series database (TSDB) explained*. URL: https://www.influxdata.com/time-series-database/. Aufgerufen am 03.12.2022.

[14] grafana.com. *Grafana*. URL: https://grafana.com/. Aufgerufen am 04.12.2022.

[15] Klaus Ostermann. *Sofware Engineering Anforderungsanalyse*. URL: https://ps.informatik.uni-tuebingen.de/teaching/ss15/se/2_Anforderungsanalyse.pdf. Aufgerufen am 17.11.2022.

[16] Jakub Stasiak Paul Scott-Murphy William McBrine. *zeroconf 0.39.4*. URL: https://pypi.org/project/zeroconf/. Aufgerufen am 08.12.2022.

[17] influxdb-python. *API Documentation*. URL: https://influxdb-python.readthedocs.io/en/latest/api-documentation.html#influxdbclient. Aufgerufen am 05.12.2022.

B Abkürzungsverzeichnis

ITU International Telecommunication Union

PLC Powerline Communication

HTTP Hypertext Transfer Protocol

URL Uniform Resource Locator

mDNS Multicast Domain Name System

DNS Domain Name System

DHCP Dynamic Host Configuration Protocol

IGMP Internet Group Management Protocol

MLD Multicast Listener Discovery

TSDB Time series database

JSON Javascript Object Notation

JSON-RPC Javascript Object Notation Remote Procedure Call

C Listings

D Abbildungsverzeichnis

E Anhang 1

E.1 Beispiele

```
Anfrage:
curl -d
'{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "call",
    "params": [
        "00000000000000000000000000000000",
        "session",
        "login",
        {
            "username": "root",
            "password": "secret"
        }
    ]
}'  http://your.server.ip/ubus

Antwort:
{
    "jsonrpc":"2.0",
    "id":1,
    "result":[
        0,
        {
            "ubus_rpc_session":"c1ed6c7b025d0caca723a816fa61b668",
            "timeout":300,
            "expires":299,
            "acls":{
                "access-group":{
                    "superuser":["read","write"],
                    "unauthenticated":["read"]
                },
                "ubus":{
                    "*":["*"],
                    "session":["access","login"]
                },
```

```
37                  "uci":{
38                      "*":["read","write"]
39                  }
40              },
41              "data":{
42                  "username":"root"
43              }
44          }
45      ]
46  }
```

Listing E.1: Beispiel Ubus Call

Abbildung E.1: Bespiel Dashboard

E.2 Helper Funktionen

```
1 def _get_mdns_devices(service_type: str):
2     if service_type == "_dvl-plcnetapi":
3         services = _get_mdns_services(service_type="_dvl-plcnetapi._tcp.
    local.")
```

```
4     elif service_type == "_dvl-deviceapi":
5         services = _get_mdns_services(service_type="_dvl-deviceapi._tcp.
      local.")
6     else:
7         return None
8
9     found_devices = {}
10    for value in services.values():
11        properties = {}
12        for k, v in value.properties.items():
13            k = k.decode("UTF-8")
14            v = v.decode("UTF-8")
15            properties[k] = v
16        properties["port"] = value.port
17        properties["address"] = socket.inet_ntoa(value.addresses[0])
18        found_devices[value.server] = properties
19    return found_devices
20
21 def get_mdns_devices_dvl_plcnetapi():
22    return _get_mdns_devices("_dvl-plcnetapi")
23
24 def get_mdns_devices_dvl_deviceapi():
25    return _get_mdns_devices("_dvl-deviceapi")
```

Listing E.2: mDNS Funktionen zur Separierung von plcnetapi und deviceapi

```
1 def get_mdns_devices_dvl_plcnetapi():
2     return _get_mdns_devices("_dvl-plcnetapi")
3
4 def get_mdns_devices_dvl_deviceapi():
5     return _get_mdns_devices("_dvl-deviceapi")
```

Listing E.3: Discover Funktion