Der Raspberry Pi als EEBUS-Knoten in der Programmiersprache Go

Studienarbeit

des Studienganges Informationstechnik

an der Dualen Hochschule Baden-Württemberg Stuttgart

im 3. Studienjahr

von

Volkan Kilic

und

Tobias Kühn

13.05.2020

Bearbeitungszeitraum	3. Studienjahr
Matrikelnummer, Kurs	5186319, TINF17IN
	8626939, TINF17IN
Studienarbeitsbetreuer	Prof. Dr. Karl Friedrich Gebhardt

Selbstständigkeitserklärung

Name, Vorname: Kilic, Volkan

Matrikelnummer: 5186319

Studiengang/Kurs: Informationstechnik / TINF17IN

Name, Vorname: Kühn, Tobias

Matrikelnummer: 8626939

Studiengang/Kurs: Informationstechnik / TINF17IN

Titel der Arbeit: Der Raspberry Pi als EEBUS-Knoten in der Programmiersprache Go

Wir versichern hiermit, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen dieser Studienarbeit wird der EEBUS nach konkreter Aufgabenstellung (Abschnitt 1.1) eruiert.

1.1 Aufgabenstellung

„Ziel der Studienarbeit ist, in erster Linie ein tiefes Verständnis für den EEBUS zu erarbeiten und darzustellen. Darauf aufbauend könnte Software in Golang für einen Raspberry Pi als EEBUS-Knoten oder EEBUS-Gerät entwickelt werden.

- Recherche, Einarbeitung und Darstellung von EEBUS
- Einarbeitung in Golang
- Design der Software oder eines Anwendungsfalles
- Implementierung (erst Implementierung auf dem Laptop)
- Golang auf dem Raspberry Pi und Implementierung auf dem Pi
- Entwicklung einer Demo/Anwendungsfalles“ (Prof. Dr. K. F. Gebhardt)

1.2 Vorgehensweise

Für die Einarbeitung in die Thematik EEBUS wurde von der offiziellen EEBUS-Webseite die EEBUS-Spezifikation heruntergeladen, durchgelesen und die wichtigsten Aspekte zusammengefasst. Die Einarbeitung in die Programmiersprache Go erfolgte durch die offizielle Go-Tour von Golang. Zusätzlich wurden Lehrvideos, Tutorials und Foren miteinbezogen. Das Go-Programm wurde in der IDE Goland 2019 v3.2 entwickelt. Die Datei- und Versionsverwaltung erfolgte durch das Programm Git.

1.3 Aufbau der Arbeit

In dieser Arbeit werden zuerst die Grundlagen (s. Kapitel 2), die für diese Arbeit notwendig waren festgehalten. Anschließend wird der entworfene Anwendungsfallentwurf präsentiert (s. Kapitel 3). Danach werden die experimentellen Untersuchungen zur Implementierung des Anwendungsfalles beschrieben (s. Kapitel 4). Basierend auf dem dadurch gewonnen Wissen wurde eine Lösung entworfen (s. Kapitel 5). Darauffolgend wurde dieser Lösungsentwurf realisiert und die wichtigsten Aspekte festgehalten (s. Kapitel 6). Im Kapitel 7 ist eine ausführliche Anleitung zur Rekonstruktion der Lösung zu finden. Abschließend wird eine Reflektion der Arbeit durchgeführt und ein Ausblick für zukünftige Arbeiten gegeben (s. Kapitel 8).

2 Stand der Technik

In den folgenden Abschnitten werden die Grundlagen zum EEBUS und zu der Programmiersprache Go eruiert. Dabei werden einzelne Punkte in Rücksprache mit dem Studienarbeitsbegleiter ausführlicher oder nur stichpunktartig dokumentiert.

2.1 Grundlagen des EEBUS

Der EEBUS ist eine Kommunikationsschnittstelle, die auf mehreren Standards und Normen basiert. Somit kann der EEBUS von jedem Gerät und jeder technische Plattform unabhängig von Hersteller und Technologie frei genutzt werden. Das Ziel des EEBUS ist eine gemeinsame herstellerübergreifende Sprache für Energiemanagement im Internet of Things zu entwickeln und zu etablieren. Die (technische) EEBUS-Spezifikation ist freizugänglich und wird von der EEBUS Initiative standardisiert. Die Mitglieder des Vereins definieren zusammen konkrete branchenübergreifende Anwendungsszenarien, die dann im EEBUS als Anwendungsfälle standardisiert werden. Der EEBUS ist in EEBUS SPINE und EEBUS SHIP unterteilt. In den folgenden Unterpunkten wird die Ausgangssituation, der EEBUS SPINE und der EEBUS SHIP eruiert.

2.2 Ausgangssituation

„Künftige Systeme der elektrischen Energieversorgung werden stärker von volatilen Quellen wie Wind- oder Sonnenenergie geprägt und dezentraler organisiert sein als heute. Diese Entwicklung wird einen intensiveren Informationsaustausch in Form von intelligenten Stromnetzen zwischen allen Teilnehmern erforderlich machen. Insbesondere Privathaushalte werden dabei aktivere Rollen als bisher einnehmen. Derzeit diskutierte Möglichkeiten sind unter anderem:

- Sie treten nicht nur als Verbraucher, sondern auch als Erzeuger auf, z.B. mit eigenen Photovoltaikanlagen oder Blockheizkraftwerken. Diese Erzeugungsanlagen sind eventuell als virtuelle Kraftwerke organisiert. Das schließt die Möglichkeit ein, den benötigten Strom von vornherein vorzugsweise lokal zu erzeugen und damit zur Netzentlastung beizutragen.

- Sie stellen die Akkukapazität ihrer am Netz hängenden Elektrofahrzeuge als Puffer für den Ausgleich kurzfristiger Produktions- und Verbrauchsschwankungen zur Verfügung oder sie laden das Fahrzeug mit selbsterzeugtem Strom.
- Sie richten ihren Verbrauch an der aktuellen Stromerzeugung aus, indem der Betrieb leistungsstarker Verbraucher in Zeiten großen Stromangebots verlagert wird. Für diese Lastverschiebung eignen sich alle Haushaltsgeräte, deren Betriebszeitpunkte ohne Funktionseinbuße variiert werden können, z.B. Wärmepumpen, Kältegeräte, Spülmaschinen etc. Laut der BDEW-Jahresstatistik aus dem Jahre 2009 entfallen rund 80 % des häuslichen Stromverbrauchs auf diese Kategorie. […]

All diese Szenarien setzen voraus, dass sämtliche Teilnehmer untereinander Informationen zum Energiemanagement austauschen können:

- Lokal können sich alle zum Haushalt gehörenden Erzeuger und Verbraucher untereinander über Produktions- und Verbrauchswerte, Lastgänge und -prognosen etc. verständigen und ihre Betriebszustände aufeinander abstimmen.
- Zwischen Energieversorger und Haushalt findet ein bidirektionaler Austausch u.a. von Tarifinformationen und Messwerten statt, der weitgehend automatisiert und ohne persönliches Eingreifen des Nutzers erfolgt.“ [Wikipedia/EEBUS, 04.05.2020]

2.3 EEBUS SHIP

2.3.1. EEBUS SHIP Einleitung

In den letzten Jahrzehnten wurden verschiedene Hausautomationstechnologien entwickelt, die Geräte mithilfe digitaler Kommunikationstechnologien verbinden. Die meisten dieser Lösungen verfügen über eine eigene Infrastruktur, z. B. dedizierte Hausautomationskabel. Diese Ansätze sind für Gewerbe- und Industriegebäude akzeptabel, für Privathäuser jedoch zu komplex, insbesondere wenn eine Nachrüstung in bereits vorhandene Infrastrukturen erforderlich ist. In diesen Fällen wurden drahtlose Technologien eingeführt, um die Installation zu vereinfachen. In der Zwischenzeit wurden sogar Privathäuser mit IP-basierten (Internet Protocol) Installationen von Haus- oder Wohnungseigentümern erweitert. IP-basierte Geräte, die unterschiedliche Kundenanforderungen erfüllen, sind in den letzten Jahren immer beliebter geworden. Dies bedeutet, dass höchstwahrscheinlich in privaten Haushalten bereits eine Kommunikationsinfrastruktur vorhanden ist. Darüber hinaus gibt es viele potenzielle Erweiterungen für andere Bereiche als nur die Heimautomation, da Smartphones, PCs, Cloud-Kommunikation usw. den Horizont möglicher Anwendungen kontinuierlich erweitern. Es besteht jedoch ein Bedarf an einem sicheren standardisierten TCP / IP-Protokoll, das auf den Anforderungen für das Netzwerk der nächsten Generation im Internet der Dinge (IoT) basiert. Dinge im IoT können sich auf eine Vielzahl von Geräten beziehen und bringen viele zusätzliche Möglichkeiten mit sich, z.B. innerhalb der Hausautomation, Smart Grid, Smart Home oder Ambient Assisted Living (AAL). Die SHIP (Smart Home IP)-Spezifikation beschreibt einen IP-basierten Ansatz für die Plug-and-Play-Heimautomation, der problemlos auf zusätzliche Domänen erweitert werden kann. Die Lösung heißt SHIP Kommunikationsgeräte, die als SHIP-Knoten bezeichnet werden. [1]

2.3.2. Architekturansicht

Smart Home IP (SHIP) beschreibt einen IP-basierten Ansatz für die interoperable Konnektivität von Smart Home-Anwendungen, der lokale SHIP-Knoten im Smart Home sowie Webserver-basierte SHIP-Knoten und Remote-SHIP-Knoten abdeckt. SHIP-Knoten können auf verschiedenen Ansätzen der physikalischen Schicht

basieren, z.B. WiFi oder Powerline-Technologien. Man kann bspw. einen IP-Router verwenden, um verschiedene physische Netzwerke zu verbinden und den Zugang zum Internet zu ermöglichen. Das liegt jedoch außerhalb des Geltungsbereichs der SHIP-Spezifikation. Die Abbildung 1 stellt den Inhalt des Textes grafisch dar. [1]

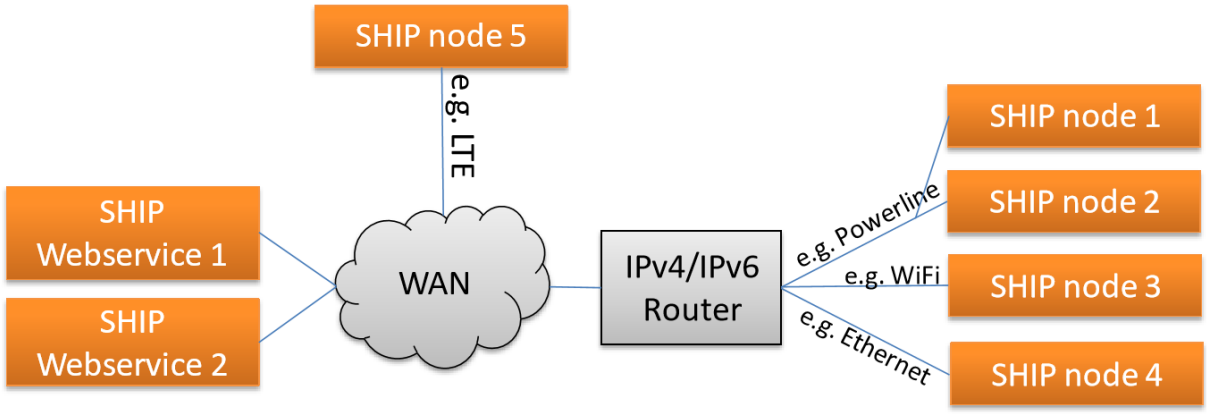

Abbildung 1: SHIP Architekturübersicht

Betrachtung des EEBUS SHIP im OSI-Modell:

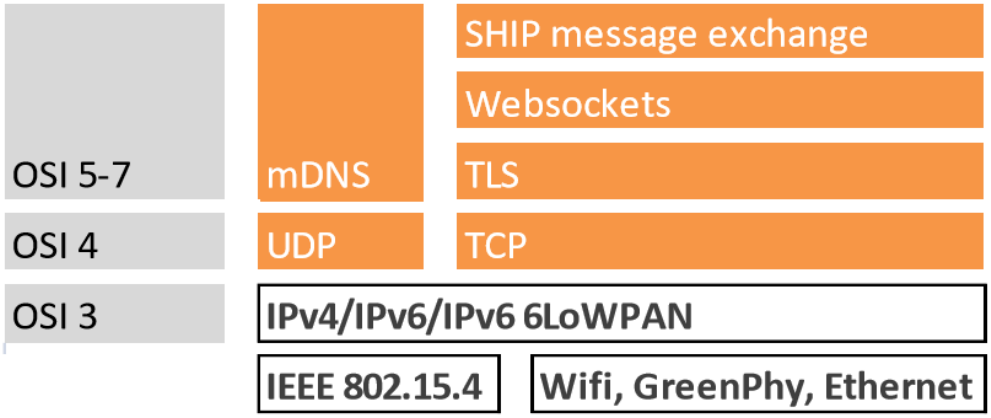

Abbildung 2: SHIP OSI-Modell

Auf der IP-Ebene sind sowohl IPv4 als auch IPv6 zulässig. IP-Adressen können vorkonfiguriert, über einen DNS-Server, mit SLAAC oder auf andere geeignete Weise zugewiesen werden. Ein SHIP-Knoten muss mDNS / DNS-SD für die Erkennung lokaler Geräte / Services unterstützen. Das SHIP-Protokoll basiert auf TCP, TLS und WebSockets. Ein SHIP-Knoten MUSS immer einen Server-Port bereitstellen. Nur ein SHIP-Knoten, der nur eine gleichzeitig aktive Verbindung unterstützt, kann den Server-Port schließen, um eine Client-Verbindung herzustellen. In SHIP ist es nicht wichtig, welcher SHIP-Knoten die Server- oder Client-Rolle

übernimmt. Wenn zwei SHIP-Knoten versuchen, praktisch gleichzeitig eine Verbindung herzustellen, werden Doppelverbindungen durch den Public-Key-Vergleich-Mechanismus verhindert. Auf diesen Mechanismus wird nicht genauer eingegangen. SHIP-spezifische Nachrichten, die über die hergestellte WebSocket-Verbindung übertragen werden, müssen mit JSON codiert werden. [1]

Da die SHIP-Spezifikation von Mitgliedern der EEBus-Initiative e.V. definiert wird, kann sie zum Transport von EEBus-spezifischen Nutzdaten und zur Bereitstellung eines EEBus-IP-Backbones verwendet werden, es können jedoch auch andere Protokolle über SHIP verwendet werden. Um eine saubere Lösung bereitzustellen, wird SHIP ohne Abhängigkeiten vom EEBus-Datenmodell definiert. [1]

2.3.3. Registrierung

Die Registrierung eines SHIP-Knotens kann durch verschiedene Mechanismen ausgelöst werden, z. B. durch einen Knopfdruck („Auto Accept“), durch ein Inbetriebnahmewerkzeug (Inbetriebnahme) oder wenn ein Public-Key vom Benutzer eingegeben oder überprüft wurde (Benutzerüberprüfung, Benutzereingabe). [1]

2.3.3.1 Registrierung mit sicherer Bestätigung

SHIP bietet die Möglichkeit den Public-Key-Wert eines SHIP-Knotens mit einem anderen SHIP-Knoten zu vertrauen. In diesen Modus sucht ein SHIP-Knoten nach SHIP-Knoten mit vertrauenswürdigen Public-Key-Werten und versucht eine Verbindung zu ihnen herzustellen, um die Registrierung abzuschließen. Wenn die Registrierung bei einem SHIP-Knoten nicht erfolgreich abgeschlossen wird, so muss der SHIP-Knoten zyklisch erneut versuchen, eine Verbindung zum SHIP-Knoten mit der entsprechenden Public-Key-Wert herzustellen. [1]

2.3.3.2. Registrierung mit „Auto Accept“

Wenn der Modus "Auto Accept" aktiv ist, muss ein SHIP-Knoten sein eigenes Registerflag für die Service-Erkennung auf true setzen. Zusätzlich muss der SHIP-Knoten eine Service-Erkennung für andere SHIP-Knoten starten, die das Registerflag auf true gesetzt haben. Wenn beide SHIP-Knoten den "Auto Accept" -Modus verwenden, um eine Verbindung herzustellen, wird dies als gegenseitiges "Auto Accept" bezeichnet. Im Falle einer gegenseitigen "Auto Accept" überprüft keine

Seite einen Public-Key und daher kann ein sogenannter "Man-in-the-Middle" -Angriff nicht ausgeschlossen werden. Man sollte daher mindestens einen der anderen Überprüfungsmodi aktivieren, um gegenseitigen "Auto Accept" und potenzielle "Man-in-the-Middle" -Angriffe zu vermeiden. Wenn ein SHIP-Knoten mehr als einen anderen SHIP-Knoten mit einem auf "true" gesetzten Registerflag erkennt, muss er einen SHIP-Knoten mit einem geeigneten Mittel auswählen (z. B. könnte er zusätzliche Informationen interpretieren, die in der Service-Erkennung enthalten sind). Wenn der "Auto Accept" -Modus inaktiv ist, muss der SHIP-Knoten sein eigenes Registerflag für die Service-Erkennung auf "false" setzen und die Suche nach anderen SHIP-Knoten beenden, die das Registerflag auf "true" gesetzt haben. [1]

2.3.3.3. Die Registrierung zwischen einem SHIP-Knoten und einem Webserver-basierten SHIP-Knoten

Reihenfolge [1]:

1. IP-Adresse und Portnummer von DNS abrufen (nur wenn URL / DNS-Hostname verwendet wird; wenn IP-Adresse verwendet wird, kann dieser Schritt übersprungen werden)
2. Eine Verbindung zu IP-Adresse und Portnummer herstellen
3. Überprüfen des Public-Keys des Webserver-basierten SHIP-Knotens
4. Initialisierung des SHIP Message Exchange (SME) -Verbindungsmodus
5. Vorbereitung der Verbindungsdaten für den SHIP-Nachrichtenaustausch

2.3.3.4. Die Registrierung mit einem anderen lokalen SHIP-Knoten

Reihenfolge [1]:

1. Wenn "Auto Accept" aktiv ist, wird das Registerflag in der Service-Erkennung auf "true" gesetzt. Andernfalls muss es auf "false" gesetzt werden.
2. Wenn "Auto Accept" verwendet wird und der andere SHIP-Knoten das Registerflag in der Service-Erkennung auf "true" gesetzt hat, sollte in einen anderen Überprüfungsmodus gewechselt werden, um ein gegenseitiges "Auto Accept" zu verhindern. Wenn "Benutzerüberprüfung", "Benutzereingabe" oder

"Inbetriebnahme" verwendet wird, wird in der Service-Erkennung nach SHIP-Knoten mit entsprechenden Public-Key-Werten gesucht.

3. Verbindung zu IP-Adresse und Portnummer herstellen, die über die Service-Erkennung abgerufen wurden oder eingehende Verbindungen akzeptieren.
4. Überprüfen des Public-Keys des Kommunikationspartners
5. Initialisierung des SHIP Message Exchange (SME) -Verbindungsmodus
6. Vorbereitung der Verbindungsdaten für den SHIP-Nachrichtenaustausch

2.3.3.5. Erfolgreiche Registrierung

Mit der SHIP-Nachricht "Hallo" kann ein SHIP-Knoten die Vertrauenswürdigkeit des Kommunikationspartners bestätigen. Wenn ein SHIP-Knoten dem Kommunikationspartner vertraut, wird der SHIP-Knoten die Anmeldeinformationen des Kommunikationspartners speichern. Wenn beide Seiten ihre Vertrauenswürdigkeit untereinander mit einer "Hallo"-Nachricht für SHIP bestätigt haben, ist die Registrierung erfolgreich abgeschlossen. Jede neue Verbindung zwischen diesen beiden Geräten wird jetzt als erneute Verbindung und nicht als Registrierung angesehen. Dies gilt solange, bis einer der beiden SHIP-Knoten den "Hallo"-Handshake für SHIP absichtlich abbricht. [1]

2.3.4. Erneuter Verbindungsaufbau

Wenn zwei SHIP-Knoten zuvor erfolgreich eine Verbindung hergestellt haben, können beide Knoten jederzeit wieder miteinander verbunden werden. Es spielt keine Rolle, ob das Registerflag während der erneuten Verbindung "true" oder "false" ist. Das Registerflag ist nur für den Registrierungsprozess wichtig, wenn eine Verbindung zu neuen SHIP-Knoten hergestellt wird. Wenn der öffentliche Schlüssel immer noch mit dem zuvor (während der Registrierung) bereitgestellten, verifizierten und vertrauenswürdigen öffentlichen Schlüssel übereinstimmt, muss der SHIP-Knoten unverzüglich erneut akzeptiert werden. Optional kann vor dem Datenaustausch noch eine PIN-Überprüfung implementiert werden. [1]

2.3.5. Discovery

Ein Erkennungsmechanismus wird verwendet, um verfügbare SHIP-Knoten und ihre Services im lokalen Netzwerk zu finden, ohne deren Multicast-DNS-Hostnamen oder IP-Adressen zu kennen. Zu diesem Zweck muss mDNS / DNS-SD verwendet werden. DNS-SD-Einträge sollten eine „Time To Live“ (TTL) von 2 Minuten haben. MDNS / DNS-SD bietet Methoden für die Erkennung lokaler Services, die Erkennung von Ressourcen und die Auflösung von Multicast-DNS-Hostnamen zu IP-Adressen. Detaillierte Informationen zu mDNS sind im RFC 6762 zu finden; Informationen zu DNS-SD sind in der RFC 6763 zu finden. Ein SHIP-Knoten, der mDNS verwendet, muss einen Service namens "ship" anbieten. [1]

2.3.5.1. Service Instanz

Der SHIP-Knoten muss jedem von ihm angekündigten DNS-SRV / TXT-Eintragspaar ein <Instance>-Label von bis zu 63 Byte im UTF-8-Format zuweisen. In Übereinstimmung mit RFC 6763 und um Namenskonflikte zu vermeiden, muss dieses Etikett benutzerfreundliche und aussagekräftige Namen verwenden, z. B. Gerätetyp, Marke und Modell. Unter Verwendung einer hypothetischen Firma "BeispielUnternehmen" könnte eine Beispiel <Instanz> eines Produkts mit einem SHIP-Knoten "Kühlschrank BeispielUnternehmen EEB01M3EU" sein. Sollte dennoch ein Namenskonflikt auftreten, muss sich ein Knoten einen neuen Namen zuweisen, bis die Konflikte gelöst sind. Ein Konflikt sollte gelöst werden, indem eine Dezimalzahl in Klammern an die <Instanz> angehängt wird (z. B. "Name (2)" für den ersten Konflikt, "Name (3)" für den zweiten Konflikt usw. [1]

2.3.5.2. Service Name

Der mit DNS-SD verwendete Service-Name muss "ship" sein. Der <Service> -Teil eines Service-Instanznamens besteht aus dem Service-Namen, dem ein Unterstrich und ein Punkt vorangestellt sind. Zusätzlich muss ein zweites DNS-Label "_tcp" angegeben werden. [1]

Ein Beispiel für einen gültigen Service-Instanznamen wäre also: "Kühlschrank BeispielUnternehmen EEB01M3EU._ship._tcp.local." Dabei ist " Kühlschrank BeispielUnternehmen EEB01M3EU" der Teil <Instance> (im vorherigen Abschnitt

beschrieben), "ship" der Service-Name, "tcp" das Transportprotokoll und "local" der Teil <Domain>.

2.3.5.3. Multicast DNS-Name

Ein lokaler SHIP-Knoten muss einen eindeutigen Multicast-DNS-Hostnamen von bis zu 63 Byte zugewiesen bekommen. Um Namenskonflikte zu vermeiden, müssen eine eindeutige ID, wie sie in der TXT Tabelle definiert wird verwenden. Auf die TXT Tabelle wird nicht weiter eingegangen. [1]

2.3.6. TCP

Für die Kommunikation muss TCP verwendet werden. Eine Kommunikation über UDP ist bisher nicht spezifiziert. Die MTU-Größe darf 1500 Byte nicht überschreiten. [1]

2.3.7. TLS

Für die sichere Kommunikation muss mindestens TLS 1.2 verwendet werden. Dabei muss TLS wie in RFC 5246 angegeben verwendet werden. SHIP-Knoten müssen während des TLS 1.2-Handshakes die gegenseitige Authentifizierung verwenden. Daher MUSS der öffentliche Schlüssel / das Zertifikat auf Client- und Serverseite überprüft werden. Abbildung 3 veranschaulicht den TLS Handshake. [1]

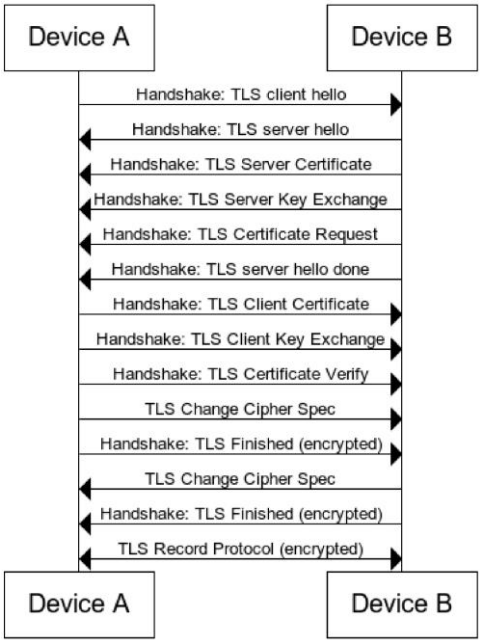

Abbildung 3: TLS Handshake

2.3.8. WebSocket

Zusätzlich zu TCP und TLS muss WebSocket verwendet werden. Eines der Ziele, die mit WebSocket erreicht werden, ist die Fähigkeit des Protokolls, Verbindungen über lokale Netzwerkgateways wie NAT-Geräte (Network Address Translation) oder Firewalls aufrechtzuerhalten. Dabei muss nach strikter Einhaltung von RFC 6455 gearbeitet werden. [1]

2.4 EEBUS SPINE

SPINE (Smart Premises Interoperable Neutral-message Exchange) ist das ausgearbeitete Datenmodel der EEBUS Initiative e.V. SPINE definiert sogenannte Anwendungsfälle, welche als Datensätze zusammenhängen. Diese sind so ausgearbeitet worden, so dass diese nicht von dem übertragenden Protokoll abhängig sind. Die Definition der SPINE Nachrichten geschieht auf dem Applikations-Layer 7 des ISO/OSI Modells, welche die Wahl des Transportprotokolls frei Verfügbar macht. Eine SPINE Nachricht wird in zwei Sektionen unterteilt [2]:

1. Ressourcen (separat Spezifiziert)
2. Protokolldaten

Die Ressourcen beschreiben die Funktion des Gerätes sowie dessen Design. Durch das Design kann eine Interoperabilität zu anderen Systemlandschaften ermöglicht werden. Die Protokolldaten beschreibt wie eine Vernetzung von SPINE-Kompatiblen Geräten, unabhängig vom Übertragungskanal, eingerichtet werden kann. Für das Kommunizieren mit anderen BUS-Protokollen ist ein sogenanntes Datenmapping erforderlich. Bei einem Datenmapping werden die EEBUS-SPINE-informationen auf die Spezifikationen des anderen Protokolls angepasst, so dass das EEBUS-fremde Gerät die Daten verarbeiten kann.

2.4.1. Data Model Specialization

Das SPINE-Konzept zielt darauf ab Definitionen wieder zu verwenden (bezüglich data model) -Gleiche Basisstruktur für verschiedene Funktionen-. Die Datenstruktur wird in XML modelliert [2].

2.4.2. Architektur Anforderung

In der Theorie schickt SPINE sogenannte Data Models entities, features, classes usw. Folgende Anforderungen müssen für jedes Gerät erfüllt sein [2]:

1. Ein Gerät hat Abhängigkeiten (Entitäten)
2. Jedes Gerät sollte einen sogenannten NodeManagement Instance implementieren
3. Jedes Feature sollte maximal eine Classe Implementieren. Es ist nicht zulässige mehrere Klassen auf einer Funktion zu haben
4. Ein Feature wird eine funktionelle Rolle zugeteilt. (Server or Client or special)

5. Als Serverrolle wird definiert, dass ein Gerät der Eigentümer der Daten ist. Bsp. LED-Licht an oder aus? Er stellt die Daten bereit und akzeptiert auch änderungen
6. Der Client wird nur zum empfangen von Serverdaten benutzt

2.4.3. Details zu den Adressebenen

Um bestimmte Funktionen eines Gerätes anzusteuern, ist eine Adressierung von Nöten. Die Adressierung von Funktionen, wird mit so genannten Entitäten beschrieben. Diese Entitäten können verschiedene Ebene besitzen. Somit ist es möglich das eine Entität mehrere Sub-Entitäten besitzt. Eine Entität kann mehrere Funktionen besitzen. Eine Funktion sollte immer eine Entität besitzen. Aus den Entitäten lässt sich dann ein Funktionsbaume erstellen, welcher zur Veranschaulichung der bereitgestellten Funktionen sehr gut geeignet ist. Die Zusammensetzung der Adressen wir in Abbildung 4 nachfolgend aufgezeigt und Erläutert.

```
"someDevice"
      +--- entity 0          (child of "someDevice")
      |        +--- feature 0
      +--- entity 1
      |        +--- entity 4 (child of "someDevice"/entity 1)
      |        |        +--- feature 7 (*1)
      |        +--- entity 5 (child of "someDevice"/entity 1)
      |        |        +--- feature 1 (*2)
      |        |        +--- feature 7 (*2)
      |        +--- feature 1 (child of "someDevice"/entity 1)
      |        +--- feature 4 (child of "someDevice"/entity 1)
      +--- entity 4           (child of "someDevice")
      |        +--- feature 1 (child of "someDevice"/entity 4)
```

Abbildung 4: SPINE Entity-Tree [2]

Die Adressen der Entitäten werden dann wie folgt erstellt und gelesen [2]:

1. Device „someDevice“
2. Device „someDevice“
3. Device „someDevice“ / entity 0
4. Device „someDevice“ / entity 0 / feature 0
5. …

Somit ist beim Programmieren der Anwendung auch klar welches Schema der Datenverarbeitung eingehalten werden muss um eine bestimmte Funktion ansprechen zu können.

2.4.4. SPINE Datagram

2.4.4.1 Struktur

Um den Austausch zwischen zwei Endpunkten dynamisch zu modellieren [2], definiert das SPINE-Datenmodell ein Element, welches sich Datagramm nennt. Dieses ist zusätzlich unterteilt in Header und Payload. Ein Datagramm beschreibt einen den Aufbau einer Information, welche später versendet werden kann.

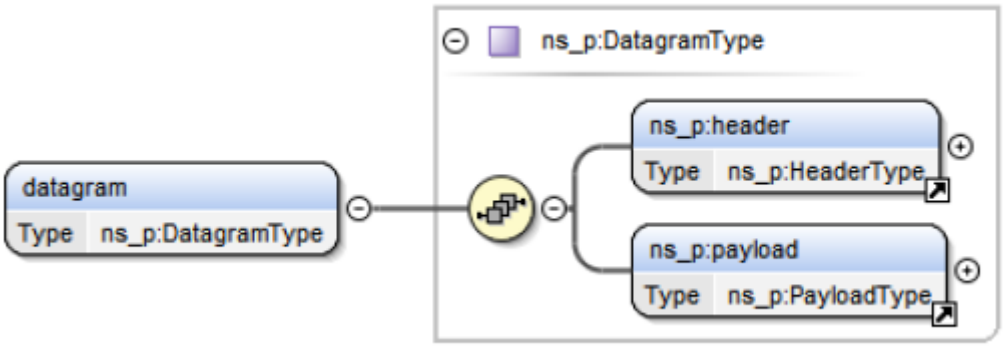

Abbildung 5: SPINE Datagram [2]

Um nachfolgende Tabelle zu verstehen sind folgende Informationen nötig:

1. M = mandatory
2. O = optional
3. NV = Not valid
4. C = choice,

Element	M/O/NV/C	Beschriebung
Datagram	M	Das Wurzelelement des SPINE-Datagramms MUSS immer vorhanden sein
Datagram.header	M	Das Kopfelement muss vorhanden sein.
Datagram.payload	M	Das Payload-Element muss vorhanden sein

Tabelle 1: SPINE Datagram [2]

2.4.4.2 SPINE Header

Der SPINE-Header ist für die Bereitstellung verschiedenster Grundinformationen bestimmt [2]. Zu den Informationen gehören unter anderem die Version des Datenmodells, die Adressen der Endpunkte sowie die Adressen der Funktionen.

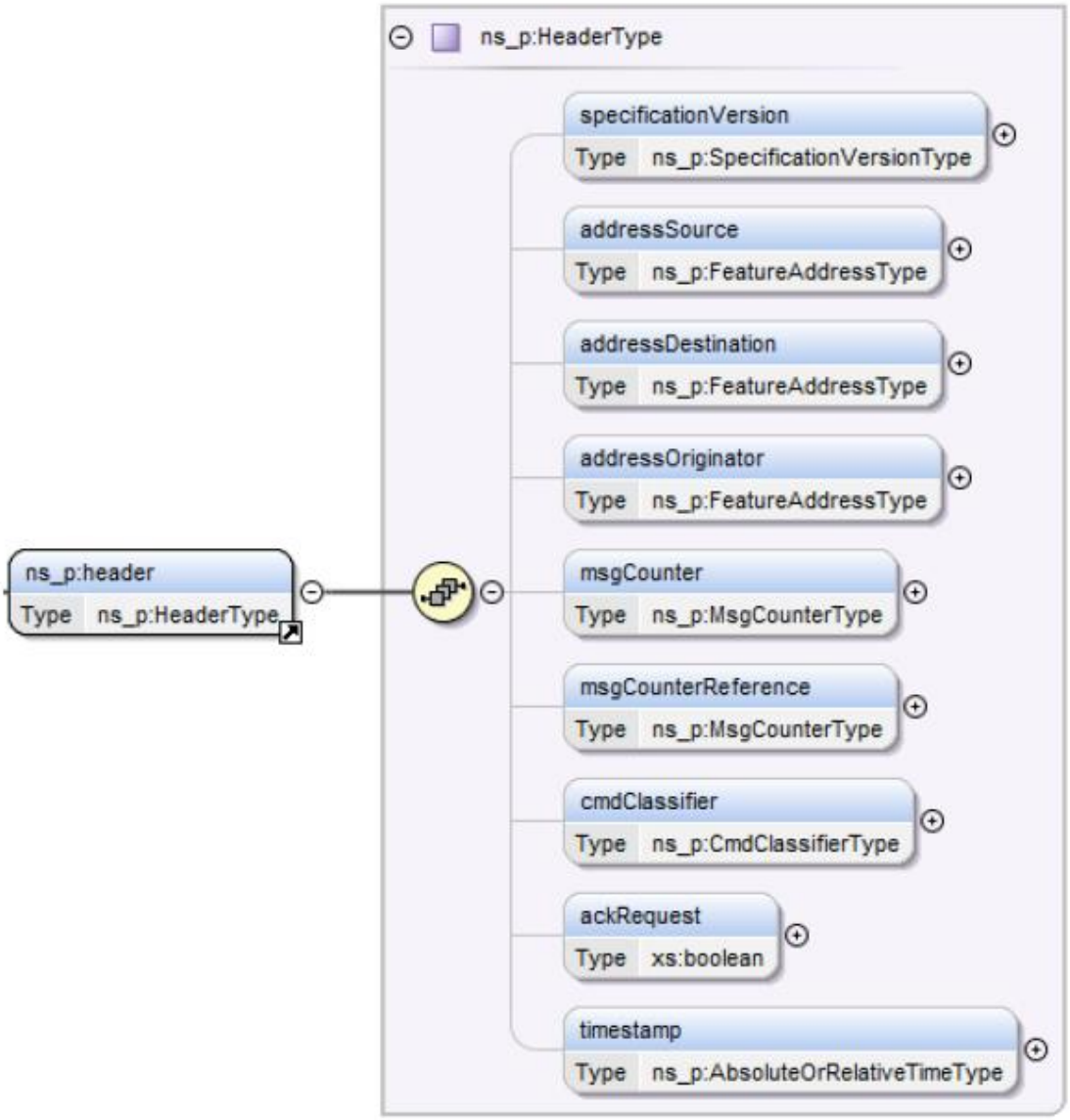

Abbildung 6: SPINE Datagram Header [2]

Abbildung 6 zeigt den vollständigen Aufbau des SPINE-Headers. Die genaue Beschreibung der Funktionen in SPINE-Header können der Spezifikation ab Seite 29 entnommen werden.

2.4.4.3. SPINE-Payload

Innerhalb des Elements Payload kann ein einzelner "Befehl" platziert werden. Er erlaubt oder erfordert sogar das Vorhandensein zusätzlicher Elemente, um eine Funktionalität auszudrücken oder zu identifizieren [2].

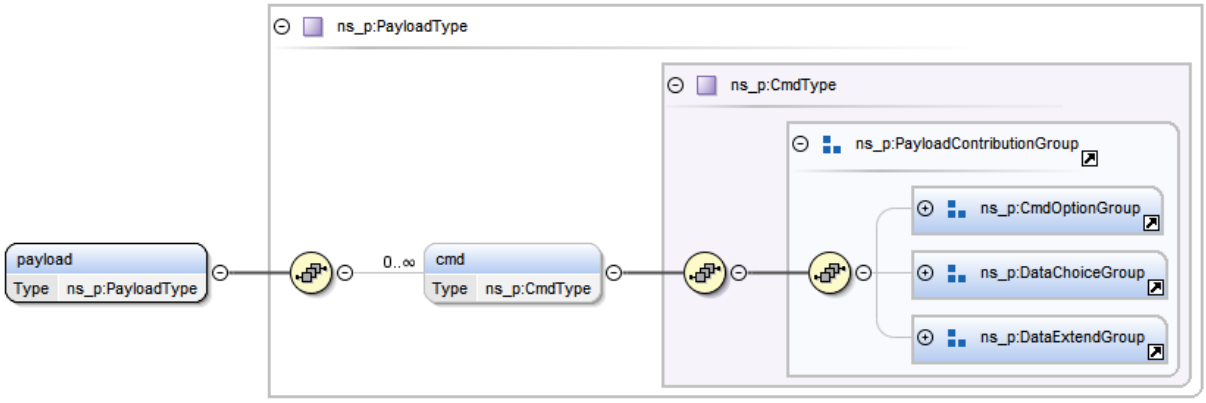

Abbildung 7: SPINE Datagram Payload [2]

Der Grundaufbau des Payloads zeigt Abbildung 7. Nach dem Element CMD werden sogenannte Gruppen erstellt, welche weitere Funktionen beinhalten. Diese können der Spezifikation ab Seite 38 Entnommen werden.

2.4.4.4. Kommunikationsarten

SPINE definiert derzeit zwei Kommunikationsarten in der Spezifikation.

Abbildung 8: SPINE Kommunikationsarten [2]

1. Gerät A kann direkt mit Gerät B kommunizieren.
2. Gerät A kann über Gerät B mit Gerät C kommunizieren.

Kommunikationsart 2 Verbindet Gerät A mit Gerät C. Gerät B leitet somit die Pakete von Gerät A zu Gerät C weiter. Diese Verbindungsart wird auch bei (SPINE-) Technologie-Gateways benötigt. Kommunikationsart 2 kommt eher einem Bus gleich [2].

2.5 Übersicht EEBUS SPINE/SHIP

Folgende Abbildung gibt eine Übersicht über den EEBUS SPINE und SHIP.

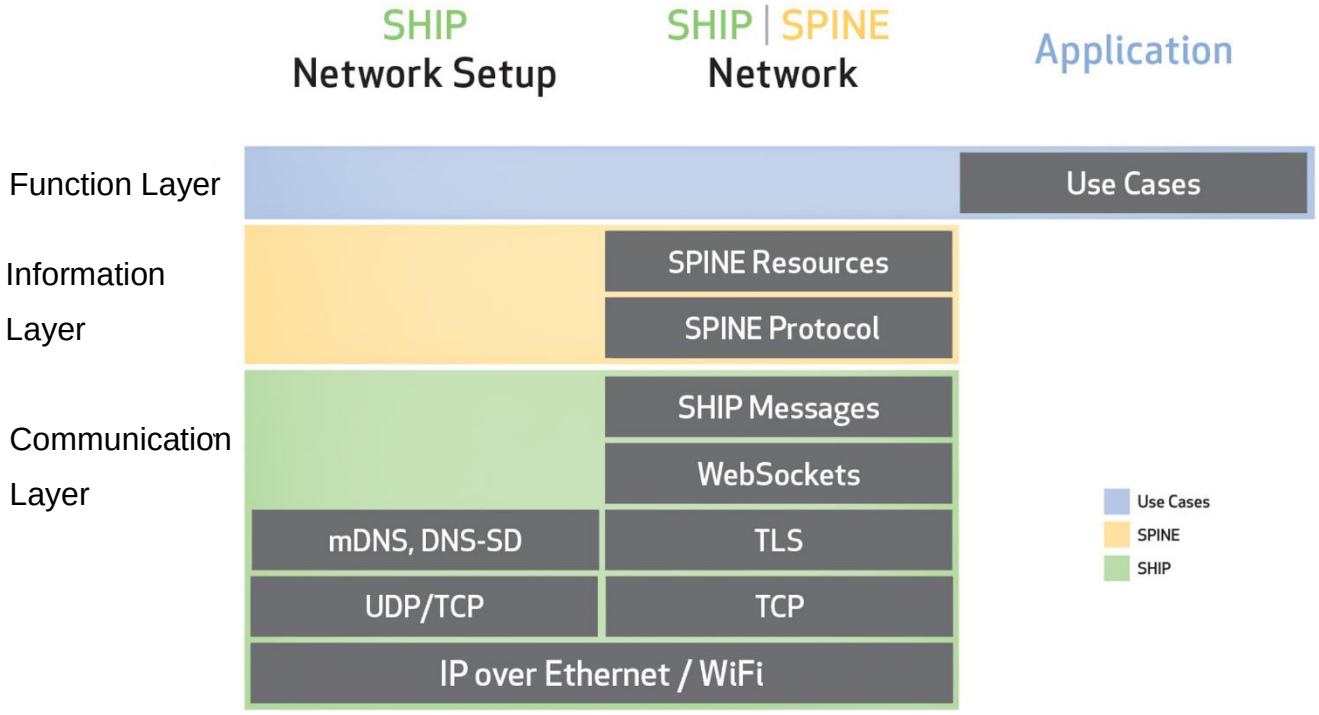

Abbildung 9: Übersicht SPINE/SHIP

2.6 Grundlagen der Programmiersprache Go

Go ist eine Open-Source-Programmiersprache, die ursprünglich von dem Unternehmen Google LLC entwickelt und im Jahr 2012 in der Version 1 freigegeben wurde. [3] Go ist ausdrucksstark, prägnant, sauber und effizient. Seine Parallelitätsmechanismen erleichtern das Schreiben von Programmen, die das Beste aus Multicore- und Netzwerkmaschinen herausholen, während das neuartige Typsystem eine flexible und modulare Programmerstellung ermöglicht. Go kompiliert schnell zu Maschinencode und bietet dennoch den Komfort der automatischen Speicherbereinigung und die Möglichkeit der Laufzeitreflexion. Es ist eine schnelle, statisch typisierte, kompilierte Sprache, die wie eine dynamisch typisierte, interpretierte Sprache wirkt. [4] In den folgenden Teilabschnitten werden die wichtigsten Go Grundlagen, Methoden, Interfaces und die Behandlung von Nebenläufigkeit in Go beschrieben.

2.6.1. Go Grundlagen

Jedes Go-Programm besteht aus Paketen. Go-Programme starten immer in dem *main*-Paket in der *main*-Funktion. Über den *import*-Befehl können andere Pakete importiert werden. Dadurch kann der Benutzer auf *exportierte Namen* der anderen Pakete zugreifen. Auf jegliche nicht exportierte Namen kann von außerhalb des Ursprungspaketes nicht zugegriffen werden. Ein Name wird dann exportiert, wenn er mit einem Großbuchstaben beginnt. [5] Im folgenden Codebespiel (Abbildung 9) wird ein Programm ausgeführt, welches die Zahl Pi ausgibt.

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func main() {
9      fmt.Println(math.Pi)
10 }
11
```

```
3.141592653589793

Program exited.
```

Abbildung 10 Beispielcode: Ausgabe Pi

In dem Codebeispiel (Abbildung 9) sind die exportierten Namen auf die zugegriffen werden eine Konstante (*Pi*) und eine Funktion (*Println*). Die Konstante *Pi* gibt den Wert von Pi und die Funktion *Println* gibt die Ausgabe von der Konstante *Pi*.

Eine Funktion kann keine oder mehrere Argumente entgegennehmen und eine beliebige Anzahl von Rückgabewerten zurückgeben. In diesem Codebeispiel (Abbildung 9) nimmt die Funktion eine Konstante entgegen.

Konstanten werden mit dem Schlüsselwort *const* deklariert. Konstanten können Buchstaben, Strings, boolesche oder numerische Werte sein, wie in folgendem Codebeispiel (Abbildung 10) ersichtlich wird. [5]

```
1  package main
2
3  import "fmt"
4
5  const Pi = 3.14
6
7  func main() {
8      const World = "世界"
9      fmt.Println("Hello", World)
10     fmt.Println("Happy", Pi, "Day")
11
12     const Truth = true
13     fmt.Println("Go rules?", Truth)
14 }
15
```

```
Hello 世界
Happy 3.14 Day
Go rules? true

Program exited.
```

Abbildung 11 Beispielcode: Konstanten

Die Basistypen in Go sind: *bool, string, int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr, byte* (anderer Name für *uint8*), *rune* (anderer Name für *int32*), *float32, float64, complex 64, complex128*. Die Datentypen *int, uint* und *uintptr* belegen auf 32-bit-Systemen 32 bits und auf 64-bit-Systemen 64 bits. [5]

Mit der Anweisung *var* wird eine Liste von Variablen deklariert. Variablen können auch mit einem Initializer (einem pro Variable) versehen werden. Wenn ein Initializer benutzt wird, muss der Variablentyp nicht angegeben werden. Die Variable übernimmt den Typ des Initializers. Go bietet die Möglichkeit innerhalb einer Funktion statt der *var*-Anweisung mit Initializer die *:=* Anweisung für die Deklaration und Zuweisung zu verwenden. Außerhalb einer Funktion ist die := Anweisung nicht verfügbar. Im folgenden Codebeispiel (Abbildung 11) wird dies exemplarisch für die Datentypen *int, bool* und *string* vorgeführt. [5]

```go
package main

import "fmt"

var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"
    fmt.Println(i, j, c, python, java)
}
```

```
1 2 true false no!

Program exited.
```

Abbildung 12 Codebeispiel: Variablen

Variablen, die ohne expliziten Wert deklariert werden, werden implizit mit ihrem jeweiligen *Nullwert* initialisiert. *Nullwert* für numerische Typen ist *0*, für boolesche Werte ist *false* und für *strings* ist “ “ (Leerstring). [5]

Go bietet die Kontrollstrukturen *for, if, else* und *switch* an. Außerdem können mit der *defer*-Anweisung Funktionen solange mit der Ausführung verzögert werden, bis die umgebenden Funktionen ausgeführt wurden. [5]

Weitere wichtige Typen die Go anbietet sind: *Pointer, Structs, Arrays, Slices* und *Maps*. Ein *Pointer* hält die Adresse einer Variablen im Speicher, ähnlich wie in C. Im Gegensatz zu C gibt es in Go keine Pointerarithmetik. Ein *struct* ist ein Verbund von Elementen. Ein *Array* hat eine fixe Größe, ein *Slice* dagegen hat eine dynamische Größe und bietet einen flexiblen Zugriff auf die Elemente eines Arrays. Eine *Map* ordnet Schlüsseln Werte zu. Folgendes Codebeispiel (Abbildung 12) zeigt eine Anwendung für eine *Map*. [5]

```go
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m map[string]Vertex

func main() {
    m = make(map[string]Vertex)
    m["Bell Labs"] = Vertex{
        40.68433, -74.39967,
    }
    fmt.Println(m["Bell Labs"])
}
```

```
{40.68433 -74.39967}

Program exited.
```

Abbildung 13 Codebeispiel: Map

2.6.2. Go Funktionen

Funktionen können keine, ein, mehrere oder eine variable Anzahl von Argumenten enthalten. Dabei muss die Anzahl der eingegangenen Variablen, nicht mit der ausgehenden Variablen übereinstimmen. Bei der Erstellung von Funktionen ist es wichtig bestimmte Konventionen von Golang zu kennen. Die wichtigste ist, dass Fehler erst zum Schluss zurückgegeben werden. Eine einfache Deklaration einer Funktion kann folgendermaßen aussehen:

```
func (v Vertex) Abs() float64 {
    fmt.Println(math.Sqrt(v.X*v.X + v.Y*v.Y))
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Abbildung 14: Golang Funktion [5]

Wie in der Abbildung 13 zu sehen ist, muss eine Funktion nicht benannt werden, wichtig ist das ein Datentyp deklariert wird. Wenn eine Funktion später, also zu einem bestimmten Zeitpunkt, muss für die Funktion ein Name bereitgestellt werden. Die in Abbildung 13 gezeigte Funktion enthält eine Variable v welche als eigener Typ Vertex (Vektor) deklariert wird. Im Anschluss soll das Ergebnis als Betrag zurückgegeben werden.

Eine Funktion kann im Header recht variabel aufgebaut sein. Man kann Pointer sowie weitere Logik, oder aber auch Eingabe und Ausgabe definieren. Mit Ein- und Ausgabe ist gemeint, ob Go etwas in die Funktion mit hereinbringt, oder einen Wert herausgibt.

Bei der Benutzung von Funktionen muss stets bewusst sein welcher Code wann und wo abgearbeitet werden soll. Dabei kommt man schnell mit den Themen clean code oder Optimierung in Berührung.

2.6.3. Go Methoden

In Go gibt es keine Klassen, wie in anderen Programmiersprachen das der Fall ist, sondern Methoden. Eine Methode kann eine Vielzahl von Funktionen besitzen. Somit kann man auch sagen, dass Methoden eine Art von Funktionen sind [5]. Dennoch unterscheiden sich Funktionen und Methoden in einem Punkt. Eine Methode enthält immer ein Empfänger-Argument. Dieses Argument kann in der Methode, nachdem die Funktion abgelaufen ist, aufgerufen werden. Methoden können innerhalb eines Paketes definiert werden.

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  type Vertex struct {
9      X, Y float64
10 }
11
12 func Abs(v Vertex) float64 {
13     return math.Sqrt(v.X*v.X + v.Y*v.Y)
14 }
15
16 func main() {
17     v := Vertex{3, 4}
18     fmt.Println(Abs(v))
19 }
20
```

Abbildung 15: Golang Methode [5]

In Abbildung 14 ist ein typischer Ablauf einer Methode zu sehen. Es wird wieder ein Vektor mit den Variablen X und Y als float64 in einem struct deklariert. Danach erstellt die Methode Abs(v Vertex) eine Berechnung und Speicher diese in v vom Typ Vertex. Später in der Main wird v mit Werten befüllt und die Methode in einem fmt.Println(abs(v)) aufgerufen. Als Ausgabe ist dann das Ergebnis der Berechnung auf der Konsole zu sehen.

Bei der Benutzung von Methoden können die gleichen Mittel benutzt werden wie bei Funktionen.

2.6.4. Go Nebenläufigkeit

Eine der größten straken von Go ist die Behandlung der Nebenläufigkeit. Nebenläufigkeit beschreibt das abarbeiten von Prozessen parallel. Dabei werden innerhalb der Prozesse sogenannte Threads genutzt. Da ein Prozesswechsel oder Threadwechsel für die Hardware sehr rechenintensiv ist möchte man in der Regel häufige Thread oder Prozesswechsel vermeiden. Die Nebenläufigkeit in Go werden mit so genannten Go-Routinen behandelt [5]. Die Go-Routinen werden durch einen Scheduler auf einen so genannten Threadpool verteilt. Ein Threadpool enthält bereits geöffnete Threads und behält diese für weitere Bearbeitungen offen [6]. Somit muss ein Thread nicht teuer erzeugt und wieder geschlossen werden. Die Go-Routinen wurden so implementiert, dass diese bei einem Threadwechsel sehr performant sind.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func say(s string) {
9      for i := 0; i < 5; i++ {
10         time.Sleep(100 * time.Millisecond)
11         fmt.Println(s)
12     }
13 }
14
15 func main() {
16     go say("world")
17     say("hello")
18 }
19
```

```
world
hello
hello
world
hello
world
hello
world
```

Abbildung 16: Go Nebenläufigkeit [5]

Die Methode in Zeile 8 gibt lediglich den String hello in einem bestimmten abstand ab. Die main Funktion beinhaltet den Ausdruck go say(„world“). Durch das Kommando go wird die Methode say im Hintergrund ausgeführt. Es taucht also immer dann ein world auf, wenn say(“hello“) wartet. Somit lässt sich auch die Ausgabe in Abbildung 15 erklären. Die Go-Routine kümmert sich also automatisch um die Abarbeitung im Hintergrund.

3 Anwendungsfallentwurf

In diesem Kapitel wird ein exemplarischer Anwendungsfallentwurf für den EEBUS präsentiert. Der folgende Anwendungsfall wurde mit den zur Verfügung stehenden Ressourcen und in Rücksprache mit dem Studienarbeitsbegleiter entworfen.

Anwendungsfall: „LED“

In diesem Anwendungsfall soll über eine Webseite eine LED über den EEBUS gesteuert werden. Dabei fungiert ein Raspberry Pi (i.F. EEBUS-Knoten genannt) als EEBUS-Knoten und ein weiterer Raspberry Pi (i.F. LED-Steuereinheit genannt) als LED-Steuereinheit. Auf dem EEBUS-Knoten wird ein Webserver laufen gelassen, über den der Benutzer die LED an- und ausschalten kann. An die GPIO der LED-Steuereinheit wird eine LED elektrisch verbunden. Der Webserver und die zwei Raspberry Pi´s kommunizieren über den EEBUS. Folgende Abbildung (Abbildung 16) stellt den beschriebenen Anwendungsfall grafisch dar.

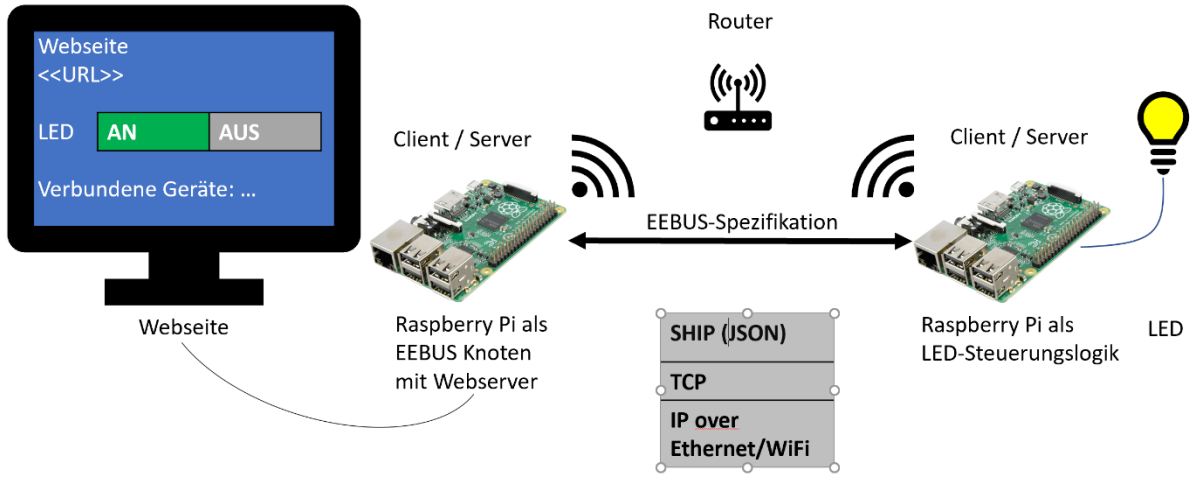

Abbildung 17: Anwendungsfall: LED

4 Experimentelle Untersuchungen

In diesem Kapitel wird untersucht, ob der als kritisch eingestufte Teil des Codes auf einem beliebigen Rechner oder nur auf dem Raspberry Pi ausgeführt werden kann. Als kritischer Code wird dabei nur der Teil vom Anwendungsfall zur Ansteuerung der LED betrachtet.

4.1 Experimenteller Code

Einfacher Code, der den GPIO-Pin 18 toggeln lässt. Dies sollte die an diesen PIN angeschlossene LED zum blinken bringen.

```
package main

import (
        "fmt"
        "time"

        "github.com/stianeikeland/go-rpio"
)

func main() {
        fmt.Println("opening gpio")
        err := rpio.Open()
        if err != nil {
                panic(fmt.Sprint("unable to open gpio",
err.Error()))
        }

        defer rpio.Close()

        pin := rpio.Pin(18)
        pin.Output()

        for x := 0; x < 20; x++ {
                pin.Toggle()
                time.Sleep(time.Second / 5)
        }
}
```

Abbildung 18: Code zur GPIO-Pin Ansteuerung [7]

4.2 Untersuchung der Implementierung auf einem beliebigen Rechner

Zu Beginn wurde untersucht, ob eine Ausführung des erstellten Codes nach Anwendungsfallentwurf auf einem Laptop-Rechner möglich ist. Das Ergebnis war, dass beim kompilieren ein Systemaufruf nicht kompiliert werden konnte, da der Rechner den Systemaufruf zur Ansteuerung der GPIO-Pins nicht kannte.

4.3 Untersuchung der Implementierung auf dem Raspberry Pi

Aufgrund des ARM-Prozessors auf dem Raspberry Pi konnte der erstellte Code mit den Systemaufrufen zur Ansteuerung der GPIO-Pins kompiliert werden und die LED blinkte.

5 Lösungsentwurf

In diesem Kapitel wird ein Lösungsentwurf für den beschriebenen Anwendungsfall (Kapitel 3) dargestellt.

5.1 Konzeptaufbau

Als erstes war das Konzept wie in Abschnitt 5.1.1. beschrieben aufgebaut, allerdings führte dieser Aufbau zu betriebssystemabhängigen Problemen. Deshalb wurde ein neues Konzept wie in Abschnitt 5.1.2. beschrieben entworfen und implementiert.

5.1.1 Konzept 1

Das erste Konzept sah vor, dass nur ein einziges Programm geschrieben wird, welches auf beiden Raspberry Pi´s läuft. Dabei wäre während der Laufzeit -durch den Benutzer- entschieden worden, ob der Pi nun Client oder Server Logik ausführt. Dieser Aufbau hat zur Folge, dass der Code nur auf Prozessoren mit einem Systemaufruf zur Ansteuerung von GPIO-Pins kompiliert werden kann. D.h. der Code könnte nicht auf einem „normalen“ Windows- oder MAC-PC kompiliert werden. Allerdings wurde in Rücksprache mit dem Studienarbeitsbegleiter festgelegt, dass eine Kompilierung auf besagten Betriebssystemen möglich sein solle. Deshalb wurde ein weiteres Konzept (s. Abschnitt 5.1.2) entworfen. Folgende Abbildung visualisiert das erste Konzept.

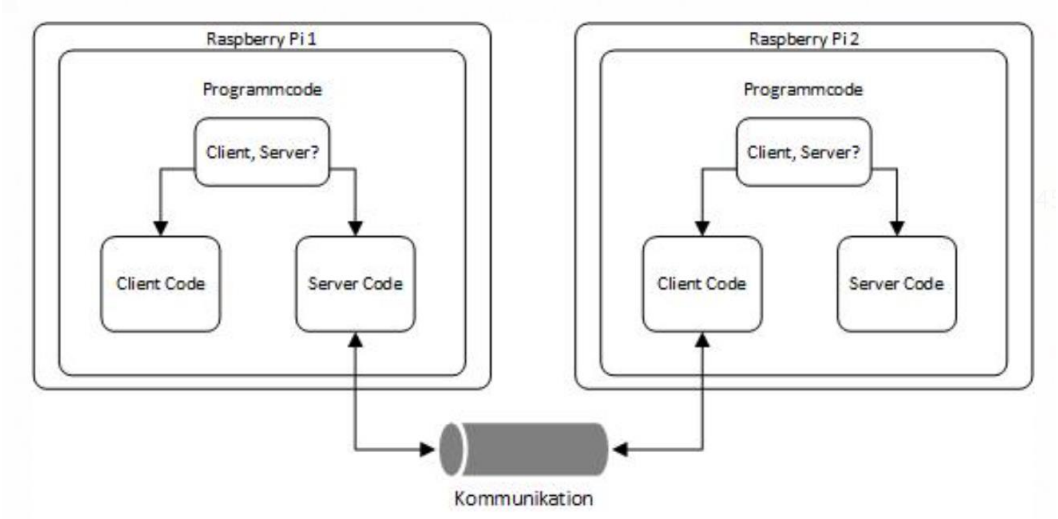

Abbildung 19: Konzept 1

5.1.2 Konzept 2

Das zweite Konzept sieht vor, dass die Applikation in zwei unterschiedliche Programme aufgeteilt wird. Dabei wird ein Programm auf dem Raspberry Pi laufen gelassen und das andere Programm, kann auf einem beliebigen Rechner (Windows, MAC) laufen gelassen werden. Das Programm auf dem Pi stellt dabei die Client-Anwendungen mit der LED-Ansteuerungslogik da. Das andere Programm stellt den Webserver auf Basis der Programmiersprache Go da. Da der Go-Compiler auf allen gängigen Betriebssystemen verfügbar ist, ist dieses Programm auch auf allen gängigen Betriebssystemen kompilier- und ausführbar. Folgende Abbildung veranschaulicht das Konzept.

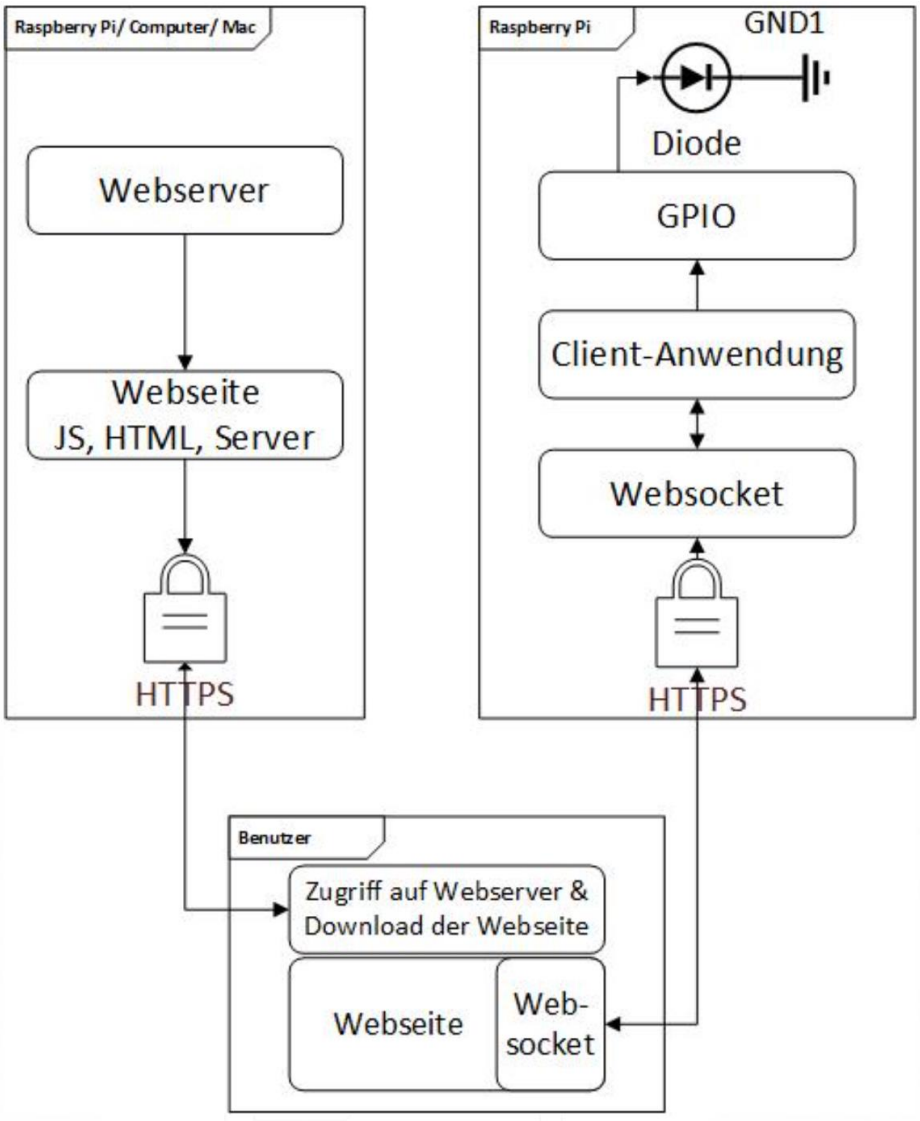

Abbildung 20: Konzept 2

6 Entwicklung und Implementierung des Lösungsentwurfs

6.1 Struktur des Server-Codes

Das Paket setzt sich wie folgt zusammen:

- EEBUS_Server
  - assets
    - index.html
    - index.js
  - docs
  - tools
    - lc-tlscert.go
  - EEBus_Server.go
  - readme.md

Die oben aufgelistete Aufzählung beginnt mit dem Root-Verzeichnis EEBUS_Server. In diesem Verzeichnis, auch Projektverzeichnis genannt, befinden sich die drei Ordner assets, docs und tools. In dem assets Ordner befinden sich die HTML und JavaScript Dateien, welche später für den Betrieb der Webseite benötigt werden. Im Ordner docs befindet sich sowohl die EEBus Spezifikation für SHIP und SPINE als auch diese Ausarbeitung als PDF. In dem Ordner tools ist eine Datei namens lc-tlscert.go abgelegt. Diese ist für die Zertifikatserstellung notwendig. Die Zertifikate werden später für die HTTPS-Verbindung benötigt. Im Rootordner ist die EEBus_Server.go Datei abgespeichert. Diese Datei enthält die Main-Funktion, die den Startpunkt des Programmes vorgibt. Zusätzlich ist eine readme.md Datei in dem EEBUS_Server Ordner abgespeichert. Diese Datei enthält eine kurze Beschreibung des Git-Repository‘s.

6.2 Funktion des Server-Codes

In diesem Abschnitt wird die Funktionsweise des Server-Codes anhand von Code-Ausschnitten erläutert.

Code-Ausschnitt EEBus_Server.go:

Code Tabelle	Datei: EEBus_Server.go

```
1  package main
2
3  import (
4     "log"
5     "net/http"
6  )
7  func main() {
8     http.Handle("/",
9     http.FileServer(http.Dir("./assets")))
10    err := http.ListenAndServeTLS(":7070", "server.crt",
11    "server.key", nil)
12    if err != nil {
13       log.Fatal("ListenAndServe: ", err)
14    } else {
15       log.Println("Successfully started.")
16    }
17 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 2: EEBus_Server.go

Wie in der Tabelle 2 zu sehen ist, werden zuerst die benötigten Pakete log und net/http importiert. Danach wird die Main-Funktion ausgeführt. Diese stellt einen http handler bereit, welcher die index.html Datei in dem assets Ordner dem Benutzer bei einem Aufruf des Webserver als Webseite zur Verfügung stellt. Für einen gesicherten und EEBus konformen Verbindungsaufbau über HTTPS, stellt der Webserver eine TLS-Verbindung dem Benutzer bereit. Diese Verbindung ist mit einem Zertifikat und einem Schlüssel versehen, so dass sie gegen Man-in-the-Middle Angriffe geschützt ist. Außerdem wurde eine Fehlerbehandlung implementiert. Dabei

wird so lange die Fehlervariable err gleich nil ist, also kein Fehler aufgetreten ist, der Webserver ausgeführt.

Code-Ausschnitt lc-tlscert.go:

Der Code-Ausschnitt für die Datei lc-tlscert.go befindet sich, auf Grund der Länge des Codes, im Anhang. Bei dem Code handelt es sich um den Ablauf einer Routine mit eingabegesteuerten Funktionen. Wie bei der Datei EEBus_Server.go, behandelt auch dieser Code Fehler und gibt diese dem Benutzer aus. Des Weiteren werden die in Golang implementierten Bibliotheken für die Zertifikaterstellung eingebunden. Der Code erstellt die Client- und Server-Zertifikate, die für eine verschlüsselte Verbindung benötigt werden. Diese Zertifikate sind selbst signiert und haben lediglich im privaten (LAN) Umfeld eine Legitimität. Die Routinen erstellen eine Verschlüsselung mit ECDSA (Elliptic Curve Digital Signature Algorithm), ursprünglich RSA. Dieses Verfahren nutzt gegenüber dem herkömmlichen Verfahren elliptische Kurven für die Erstellung eines Privaten Schlüssels. Das ECDSA Verfahren ist im Gegensatz zu RSA, bei kleinerer Schlüssellänge, gleichbleibend sicher. Zusätzlich ist deshalb ECDSA, bei der Schlüsselgenerierung und dem Schlüsselaustausch performanter. Die Datei wurde so abgeändert, so dass ein Benutzer lediglich den Anweisungen folgen muss. Der Code erstellt dann eine Zertifikatsdatei und eine Schlüsseldatei welche von dem EEBus_server.go und dem Client benötigt werden. Der Code wurde aus der [8] Quelle entnommen und für dieses Projekt angepasst.

Code-Ausschnitt index.html:

Code Tabelle		Datei: index.html
1	<!DOCTYPE html>	
2	<html lang="en">	
3	<head>	
4	<meta charset="UTF-8">	
5	<title>Studienarbeit EEBus</title>	
6	</head>	
7	<body>	
8	<input type="text" id="ip">	
9	<input type="number" id="pinnumber"	
10	autocomplete="off" min="0">	
11	<label for="status">Status: </label><input	
12	type="checkbox" id="status">	
13	<label for="active">Aktiv: </label><input	
14	type="checkbox" id="active">	
15	<button id="send">Send</button>	
16		
17	<script src="index.js"	
18	type="application/javascript"></script>	
19	</body>	
20	</html>	
Ersteller: Kilic, Kühn		Datum: 08.05.2020

Tabelle 3: index.html

Eine HTML Datei besitzt eine Grundstruktur, welche zum Beispiel <html>, <head> und <body> sind. Der Hauptteil <body> besitzt sogenannte HTML Forms, welche mit input deklariert sind. Die forms sind dafür da, dass man IP-Adresse, GPIO Port und den Port an oder aus schalten kann. Zum schluss wird noch eine JavaScript Datei eingebunden, welche für die Logik zuständig ist.

Code-Ausschnitt index.js

Code Tabelle	Datei: index.js

```
1  const pinNumber = document.querySelector('#pinnumber');
2  const status = document.querySelector('#status');
3  const active = document.querySelector('#active');
4  const button = document.querySelector('#send');
5  const responseText =
6  document.querySelector('#responseText');
7  const ip = document.querySelector('#ip');
8  let socket;
9  button.onclick = () => {
10     const data = {
11         event: "message",
12         data: {
13             pinNumber: Number(pinNumber.value),
14             status: status.checked,
15             active: active.checked,}
16     }
17     if (socket === undefined) {
18         socket = new
19 WebSocket(`wss://${ip.value}:7070/ws`);
20         socket.onmessage = (message) => {
21             const data = JSON.parse(message.data);
23             if (data && data.status) {
24                 responseText.textContent = data.status;
25             }
26         }
27         socket.onopen = () => {
28             socket.send(JSON.stringify(data));}
29     }
30     socket.send(JSON.stringify(data));}
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 4: index.js

Die index.js ist für das einsammeln der Daten aus den HTML forms zuständig. Die gesammelten Daten werden dann in ein JSON Format umgewandelt. Nach der Umwandlung wird mit Hilfe der eingetragenen IP ein Websocket zum Ziel aufgebaut und die JSON-Daten übermittelt. Der geöffnete Websocket wird solange offengehalten, bis die Webseite verlassen oder der Browser geschlossen wird.

6.3 Struktur des Client-Codes

Das Paket setzt sich wie Folgt zusammen:

- EEBUS_Client
  - Src
    - EEBus_Client
      - events.go
      - GPIO_Raspi.go
      - websocket.go
  - tools
    - Lc-tlscert.go
  - EEBus_Client.go
  - Makefile

Die oben genannte Auflistung beginnt mit dem Root-Verzeichnis EEBUS_Client. Darunter befinden sich die Ordner src ->EEBus_Client und tools. In src->EEBus_Client befinden sich Komponenten-Dateien welche für das ausführen des Programms notwendig sind. In dem Ordner tools befindet sich, wie auch im EEBus_server, die Datei lc-tlscert.go. Unter dem Root-Verzeichnis ist zusätzlich die EEBus_Client.go Datei gespeichert. Diese Datei enthält die Main-Funktion, die den Startpunkt des Programmes vorgibt. Im selben Root-Verzeichnis ist auch ein Makefile gespeichert. Welche das Repository und die notwendigen Abhängigkeiten automatisch downloadet.

6.4 Funktion des Client-Codes

Code-Ausschnitt events.go

Code Tabelle	Datei: events.go

```
1  package EEBus_Client
2
3  import (
4     "encoding/json"
5     "log"
6  )
7
8  type EventHandler func(*Event)
9
10 /*
11 Example Event:
12 {
13    "event": "message",
14    "data": {
15       "pinnumber": 16,
16       "active": true,
17       "status": false,
18    }
19 }
20 */
21
22 type Event struct {
23    Name string `json:"event"`
24    Data Data   `json:"data"`
25 }
26
27 type Data struct {
28    Pinnumber int  `json:"pinnumber"`
29    Active    bool `json:"active"`
```

```
30      Status    bool `json:"status"`
31  }
32
33  func NewEventFromRaw(rawData []byte) (*Event, error) {
34      log.Printf("%s\n", rawData)
35      event := new(Event)
36      err := json.Unmarshal(rawData, event)
37      return event, err
38  }
39
40  func (e *Event) Raw() []byte {
41      raw, _ := json.Marshal(e)
42      return raw
43  }
44
45  func handleEventMessage(d Data) error {
46      err := Gpio(d.Pinnumber, d.Active)
47      if err != nil {
48          return err
49      }
50      return nil
51  }
```

Ersteller: Kilic, Kühn	Datum:10.05.2020

Tabelle 5: events.go

Der eben aufgeführte Code wurde unter Zuhilfenahme der Quelle [9] erstellt. Zuerst wird ein struct Event erstellt. Dieses stellt die erste Hierarchieebene des JSON-Formats dar. Sie definiert in dieser Ebene ein event vom Typ String und Data vom Typ Data. Der Data struct-Typ stellt die zweite Hierarchieebene des JSON-Formats dar. In ihr werden die Variablen: Pinnumber, Active und Status definiert. Diese empfangen die gesetzten Werte der Webseite.

Durch die Funktion NewEventFromRaw wird ein neues Event erzeugt und die empfangenen JSON Daten wieder in ihre ursprüngliche Form zurückübersetzt. Wird dabei ein Fehler festgestellt, zum Beispiel durch ein fehlerhaftes Format, wird dieser ausgegeben und die Funktion beendet.

Mit der Funktion handleEventMessage werden die Werte aus Pinnumber und Active in die Variable d vom Typ struct gespeichert und der Datei GPIO_Raspi.go übergeben.

Code-Ausschnitt GPIO_Raspi.go

Code Tabelle		Datei: GPIO_Raspi.go
1	`package EEBus_Client`	
2		
3	`import (`	
4	`"fmt"`	
5	`"github.com/stianeikeland/go-rpio" // you have to`	
6	`Import "go get github.com/stianeikeland/go-rpio for`	
7	`including the package for gpio on raspi "`	
8	`)`	
9		
10	`func Gpio(pinnumber int, active bool) error {`	
11	`fmt.Println("opening gpio")`	
12	`err := rpio.Open()`	
13	`if err != nil {`	
14	`return err`	
15	`}`	
16		
17	`defer rpio.Close()`	
18		
19	`pin := rpio.Pin(pinnumber)`	
20	`pin.Output()`	
21		
23	`if active {`	
24	`pin.High()`	
25	`} else {`	
26	`pin.Low()`	
27	`}`	
28	`return nil`	
29	`}`	
Ersteller: Kilic, Kühn		Datum: 08.05.2020

Tabelle 6: GPIO_Raspi.go

Die Funktion nimmt die Pin-Nummer, zur Ansteuerung des GPIO Pins des Raspberry Pis's, entgegen und setzt dabei in Abhängigkeit der Variable active den angegebenen Pin auf High oder Low.

Mit dem importierten Packet aus Quelle [10], sind noch weitere Funktionen möglich. Die Art der Funktionen können der Dokumentation im Repository entnommen werden. Derzeit ist anzumerken, dass der PWM Modus nicht ohne root-Berechtigung auf dem Raspberry Pi ausführbar ist.

Code-Ausschnitt websocket.go

Der Code-Ausschnitt websocket.go wird zur Lesbarkeit in mehrere Code-Tabellen unterteilt.

Code Tabelle	Datei: websocket.go

```
1  package EEBus_Client
2
3  import (…)
4
5  var upgrader = websocket.Upgrader{
6     HandshakeTimeout: 0,
7     ReadBufferSize:   2048,
8     WriteBufferSize:  2048,
9     WriteBufferPool:  nil,
10    Subprotocols:     nil,
11    Error:            nil,
12    CheckOrigin: func(r *http.Request) bool {
13       return true
14    },
15    EnableCompression: false,
16 }
17
18 type WebSocket struct {
19    Conn    *websocket.Conn
20    Out     chan []byte
21    In      chan []byte
23    Events map[string]EventHandler
24 }
25
26 type ServerResponse struct {
27    Status string `json:"status"`
28 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 7: websocket.go_1

Damit eine einfachere Implementierung von Websockets möglich ist, wird die fertige Go Implementation von Gorilla [11] genutzt. Gorilla ist ein web toolkit für Golang welches Websockets nach der Definition der RFC 6455 implementiert.

Damit eine http Verbindung Websockets benutzen kann, muss die Verbindung mit weiteren Parametern erweitert werden. Diese sind in der Variable upgrader definiert.

Das WebSocket struct definiert eine Verbindung Conn einen Out Kanal als channel Byte Array sowie ein In als channel Byte Array. Events wird als map mit dem Schlüsseltyp string und dem EventHandler aus der Datei events.go verknüpft.

Code Tabelle	Datei: websocket.go

```
1  func NewWebSocket(w http.ResponseWriter, r *http.Request)
2  (*WebSocket, error) {
3     conn, err := upgrader.Upgrade(w, r, nil)
4     if err != nil {
5        log.Printf("An error accourde while upgrading the
6        connection: %v", err)
7        return nil, err
8     }
9
10    ws := &WebSocket{
11       Conn:   conn,
12       Out:    make(chan []byte),
13       In:     make(chan []byte),
14       Events: make(map[string]EventHandler),
15    }
16
17    go ws.Reader()
18    go ws.Writer()
19    return ws, nil
20 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 8: websocket.go_2

NewWebSocket ist ein Konstruktor welcher als Übergabeparameter den Responswriter sowie die http.Request bekommt und den WebSocket zurückgibt. Wie in Zeile drei bis acht beschrieben, wird die bestehende Verbindung erweitert und auf Fehler geprüft. Anschließen wird die Websocket Instanz mit den vorher definierten Werten Conn, Out, In und Events erstellt. Damit der Schreiben- und Empfangskanal (Reader() & Writer()) parallel arbeiten können, werden diese mit dem Parameter im Hintergrund abgearbeitet.

Code Tabelle	Datei: websocket.go

```
1  func (ws *WebSocket) Reader() {
2     defer func() {
3        _ = ws.Conn.Close()
4     }()
5     for {
6        _, message, err := ws.Conn.ReadMessage()
7        if err != nil {
8           if websocket.IsUnexpectedCloseError(err,
9  websocket.CloseGoingAway, websocket.CloseAbnormalClosure)
10 {
11              log.Printf("WS Message Error: %v", err)
12           }
13           break
14        }
15        event, err := NewEventFromRaw(message)
16        if err != nil {
17           log.Printf("Error parsing message: %v", err)
18        } else {
19           //
20           log.Printf("MSG: %v", event)
21        }
22        if action, ok := ws.Events[event.Name]; ok {
23           action(event)
24        }
25     }
26 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 9: websocket.go_3

Websocket.go_3 definiert nun den Reader. Eine geöffnete Verbindung muss am ende immer geschlossen werden. Mit defer wird sichergestellt, dass die Funktion zum Schluss der for-Schleife ausgeführt wird. In Zeile sechs bis 14 wird empfangene Nachricht ausgelesen und auf Fehler überprüft. Ist ein Fehler aufgetreten so wird

dieser auf der Konsole ausgegeben und die Funktion abgebrochen. Damit nun aus der Nachricht ein Event erstellt werden kann, muss die Nachricht, der Funktion in events.go NewEventFromRaw, übergeben werden. Tritt kein fehler auf so wird das event auf der Konsole ausgegeben. Ist der Vorgang korrekt abgelaufen so wir die Aktion in Zeile 23 ausgeführt.

Code Tabelle	Datei: websocket.go

```
1  func (ws *WebSocket) Writer() {
2     for {
3        select {
4        case message, ok := <-ws.Out:
5           if !ok {
6              _ =
7  ws.Conn.WriteMessage(websocket.CloseMessage, make([]byte,
8  0))
9              return
10          }
11          w, err :=
12 ws.Conn.NextWriter(websocket.TextMessage)
13          if err != nil {
14             return
15          }
16          _, _ = w.Write(message)
17          _ = w.Close()}}}
18
19 func (ws *WebSocket) On(eventName string, action
20 EventHandler) *WebSocket {
21    ws.Events[eventName] = action
22    return ws
23 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 10: websocket.go_4

Der Inhalt des Out channel wird mit dem Code in Zeile 4 dem Benutzter auf der Webseite angezeigt. Dabei wird die Nachricht vom Backend ins Frontend gesendet und auf Fehler überprüft. Ist kein Fehler aufgetreten, so wird die Nachricht dem Writer übergeben und beim nächstmöglichen Zeitpunkt ins Frontend gesendet. Die Funktion in Zeile 19 wird benötigt, damit Events definiert werden können.

Code Tabelle	Datei: websocket.go

```
1  func WebConn(w http.ResponseWriter, req *http.Request) {
2     ws, err := NewWebSocket(w, req)
3     if err != nil {
4        log.Printf("Error creating websocket connection:
5        %v\n", err)
6        return
7     }
8     ws.On("message", func(e *Event) {
9        log.Printf("Message received: %v\n", e.Data)
10       err = handleEventMessage(e.Data)
11       resp := ServerResponse{}
12       if err != nil {
13          resp.Status = "Error"
14       } else {
15          resp.Status = "Success"
16       }
17       raw, _ := json.Marshal(resp)
18       ws.Out <- raw
19    })
20 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020

Tabelle 11: websocket_5

Mit der Funktion WebConn wird ein neuer Websocket erstellt. Dabei wird die Variable ws mit dem ResponseWriter gefüllt und auf Fehler überprüft. Ist kein Fehler aufgetreten, so wird ein neues Event erstellt und handleEventMessage aus der Datei events.go aufgerufen. Dieser Handler ist für das Abfangen von Fehlern an der GPIO Schnittstelle zuständig. ServerResponse gibt dem Benutzer eine Rückmeldung, ob die Kommunikation zwischen Benutzer und Client via JSON erfolgreich war. Wurde die Verbindung erfolgreich aufgebaut und die Nachricht erfolgreich übermittelt, so wird Success als JSON übersetzt und in den ws.Out channel geschrieben. Dabei wird zum nächstmöglichen Zeitpunkt die Nachricht an den Benutzer gesendet und auf der Webseite ausgegeben.

Code-Ausschnitt EEBus_client.go

Code Tabelle		Datei: EEBus_Client.go
1	`package main`	
2		
3	`import (`	
4	`"EEBus_Client"`	
5	`"log"`	
6	`"net/http"`	
7	`"time"`	
8	`)`	
9		
10	`func main() {`	
11	`http.HandleFunc("/ws", EEBus_Client.WebConn)`	
12	`err := http.ListenAndServeTLS(":7070", "server.crt",`	
13	`"server.key", nil)`	
14	`if err != nil {`	
15	`log.Fatal("ListenAndServe: ", err)`	
16	`}`	
17	`for true {`	
18	`time.Sleep(time.Second)`	
19	`}`	
20	`}`	
Ersteller: Kilic, Kühn		Datum: 08.05.2020

Tabelle 12: EEBus_Client.go

Die Datei EEBus_client.go hat die gleiche Funktion wie die Datei EEBus_server.go. Der Code erstellt einen http Handler, welcher einen Webserver erzeugt. Der Webserver ist dann unter der IP-Adresse des Gerätes und der Zuhilfenahme des Ports 7070 erreichbar. Für einen gesicherten und EEBus konformen Verbindungsaufbau über HTTPS, stellt der Webserver eine TLS-Verbindung dem Benutzer bereit. Auch diese Verbindung ist mit einem Zertifikat und einem Schlüssel versehen, so dass sie gegen Man-in-the-Middle Angriffe geschützt ist. Außerdem wurde auch hier eine Fehlerbehandlung implementiert. So lange die Fehlervariable err gleich nil ist, also kein Fehler aufgetreten ist, wird der Webserver ausgeführt.

7 Anleitung

In diesem Kapitel wird eine Anleitung zur Implementierung des Programms präsentiert.

7.1 Voraussetzungen

Zur Kompilierung der Server-Anwendung muss der Systemabhängige Compiler, wie in der Tabelle 13 aufgeführt, genutzt werden.

Architektur	Betriebssystem	Compiler
x86-64	Windows (ab windows 7)	go1.14.2.windows-amd64.msi
x86-64	macOS (ab macOS 10.11)	go1.14.2.darwin-amd64.pkg
x86-64	Linux (ab Linux 2.6.23)	go1.14.2.linux-amd64.tar.gz
arm	Linux	go1.14.2.linux-armv6l.tar.gz

Tabelle 13: Compiler-Übersicht [12]

Die Client-Anwendung wird auf einem Raspberry Pi kompiliert, da dieser die Syscalls für die GPIO Schnittstelle anbietet. Hierfür wird der ARM Compiler genutzt.

7.1.1. Installation des Compilers auf einem Raspberry Pi

Zum Herunterladen und installieren des Compilers wird wie folgt vorgegangen:

- Herunterladen des Compilers mit:
  wget https://dl.google.com/go/go1.14.2.linux-armv6l.tar.gz
- Entpacken des Archivs:
  tar -C /usr/local -xzf go1.14.2.linux-armv6l.tar.gz
- Hinzufügen der Pfadvariable in die .bashrc am Ende der Datei:
  export PATH=$PATH:/usr/local/go/bin permanent
- Überprüfung der Installation, durch Eingabe in ein Terminal:
  go version

Ausgabe bei erfolgreicher Installation:

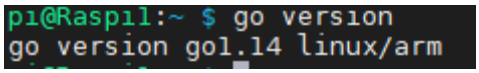

Abbildung 21: go_version check

7.1.2. Installation des Compilers auf Windows

Die Installation des Go-Compilers auf Windows 10 erfolgt durch:

- Installation des Compilers mit Hilfe des Installationsdatei
- Erstellen eines Arbeitsverzeichnisses mit der Strucktur Bsp.:
    - C:\Projects\GO\
        - bin
        - pkg
        - src
- Erstellung der GOPATH Benutzervariable

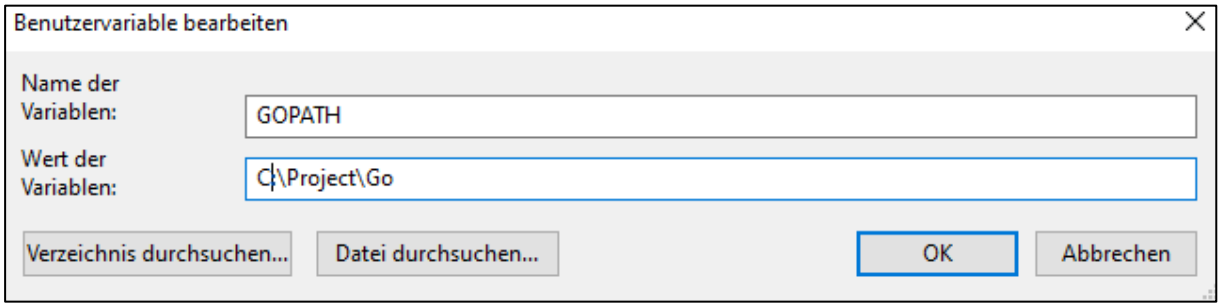

Abbildung 22: Benutzervariable

- Überprüfung der Installation, durch Eingabe in die Eingabeaufforderung: *go version*

Ausgabe bei erfolgreicher Installation:

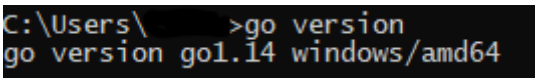

Abbildung 23: go_version check win

Die Installation auf Linux und macOS erfolgt einem ähnlichen Prinzip.

7.1.3 Installation von Git für Linux

Die Installation wird nur für die Client-Anwendung benötigt. Hierzu wird im Terminal folgender Befehl abgesetzt:

sudo apt-get install git

Nach abschicken des Befehlt wird das Administratorkennwort benötigt. Nach der Eingabe des Kennworts wird Git heruntergeladen und automatisch installiert.

7.2 Download und Kompilierung der Client-Anwendung

Zuerst wird mit folgendem Befehl:

git clone https://github.com/tkuehn1/EEBus_Client.git

nach Eingabe der GitHub Benutzerdaten wird die Client-Anwendung heruntergeladen. Nach dem Download muss in das heruntergeladene Verzeichnis gewechselt werden.

Danach wird mit dem Befehl:

make

eine Bash-Datei ausgeführt, die die Abhängigkeiten herunterlädt, das Programm und die Datei zur Zertifikatserstellung kompiliert. Anschließend muss in den Ordner EEBus_Client/tools navigiert und mit dem Befehl:

./lc-tlscert

die Zertifikate erstellt werden.

Danach muss für den Common name die *IP-Adresse des EEBus-Clients* eingetragen werden. Als erste DNS or IP-Eintrag wird **localhost** eingetragen. Beim zweiten Eintrag ist dieselbe IP-Adresse wie beim Common name einzutragen. Die Abfrage nach der 3 Adresse muss mit einer *leeren Eingabe* bestätigt werden. Anschließend muss die Gültigkeitsdauer in Tagen angegeben werden, bspw. *17.* Nach dem Drücken einer beliebigen Taste werden die Zertifikate in dem darüberliegenden Ordner abgespeichert. Bei erfolgreicher Zertifikatserstellung erfolgt folgende (s. Abb. 24) Ausgabe:

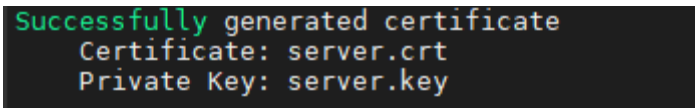

Abbildung 24: Erfolgreiche Zertifikatserstellung

7.3 Download und Kompilierung der Server-Anwendung

1. Download ZIP-Datei: https://github.com/tkuehn1/EEBus_Server
2. Entpacken der ZIP-Datei
3. Kompilierung der Komponenten tools/lc-tlscert.go und EEBus_server.go mit den Befehlen:

   go build -o /path/to/dir/EEBus_Server/tools/lc-tlscert[Dateiendung]
   path/to/dir/EEBus_server/lc-tlscert.go

   go build -o /path/to/dir/EEBus_Server/EEBus_server[Dateiendung]
   /path/to/dir/EEBus_server/EEBus_server.go

   Anmerkung: Unter Windows muss die Dateiendung .exe mit angegeben werden.
4. Erstellung der Zertifikate wie im Abschnitt 7.2 beschrieben wurde. Jedoch mit der IP-Adresse des Servers
5. Kopieren der Dateien toos/server.key und tools/server.crt in den überliegenden Ordner (das manuelle Kopieren der Zertifikate wird für ein betriebssystemunabhängigen Code benötigt)

7.4 Nutzung des Programms

Da der Websocket Verbindungsaufbau über einen privaten Schlüssel verschlüsselt wird, muss zuerst die Zertifikatsdatei (server.crt) der Client-Anwendung heruntergeladen werden und auf den zu benutzenden Rechner installiert werden. Die Zertifikatsdatei kann über unterschiedliche Medien (z.B. USB-Stick, E-Mail oder SSH) übertragen werden.

Nachdem die Zertifikatsdatei erfolgreich auf dem Zielrechner übertragen wurde, muss die Datei nun im Browser hinzugefügt werden. Die folgenden Schritte zur Einbindung des Zertifikats werden ausschließlich für den Webbrowser Firefox in der

Version 76.0.0.1 beschrieben. Unter der Rubrik Einstellungen → Datenschutz & Sicherheit → Sicherheit → Zertifikate → Zertifikate anzeigen… muss das Zertifikat hinzugefügt werden.

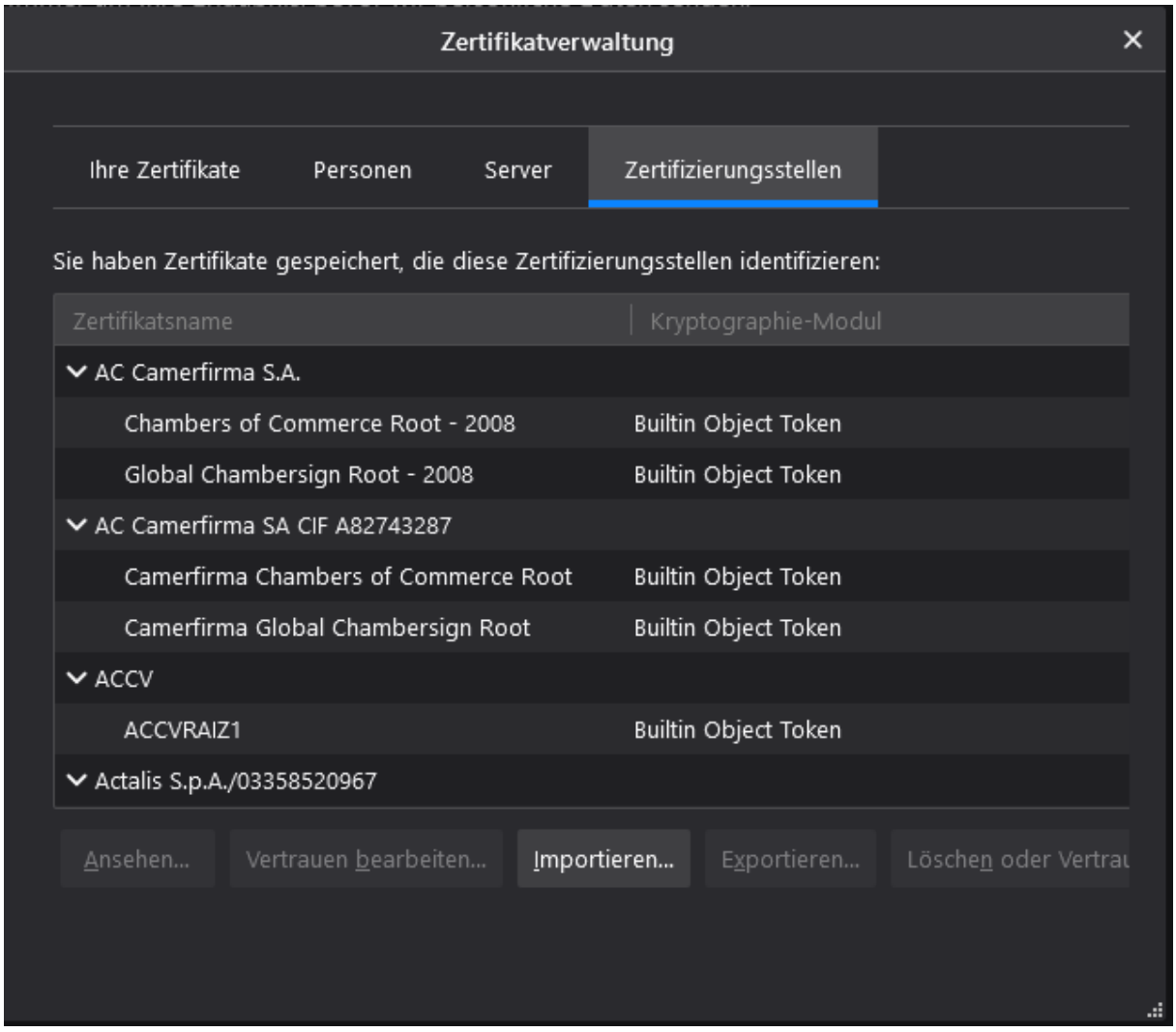

Abbildung 25: Firefox_Zertifikatverwaltung

Unter der Rubrik Zertifizierungsstellen (s. Abb. 25) muss über Importieren… die Zertifikatsdatei ausgewählt werden. Bei der anschließenden Vertrauensabfrage muss „Dieser CA vertrauen, um Webseiten zu identifizieren“ ausgewählt und bestätigt werden.

Jetzt können die Server- und Client-Anwendung gestartet werden. Für den erfolgreichen Websocket-Aufbau muss eine Ausnahme hinzugefügt werden. Hierzu muss die Client IP-Adresse mit dem Port 7070 in das Eingabefeld eingegeben werden:

https://[Client-IP]:7070/

Nach dem Absenden der Adresse muss einmalig das Risiko akzeptiert werden.

Nun verbindet man sich mit der Webseite des Servers, in dem man folgende Adresse in die Adressleiste des Browsers eintippt:

https://[Server-IP]:7070/

Auch hier muss das Risiko einmalig akzeptiert werden. Nach dem Bestätigen der Meldung erscheint folgende Ansicht im Browser:

Abbildung 26: Webseite

In das erste Feld wird die Client IPv4 Adresse eingegeben. Im zweiten Feld wird der GPIO Pin an dem die LED angeschlossen ist angegeben. Die Belegung der GPIO Pinleiste kann aus folgender Abbildung entnommen werden:

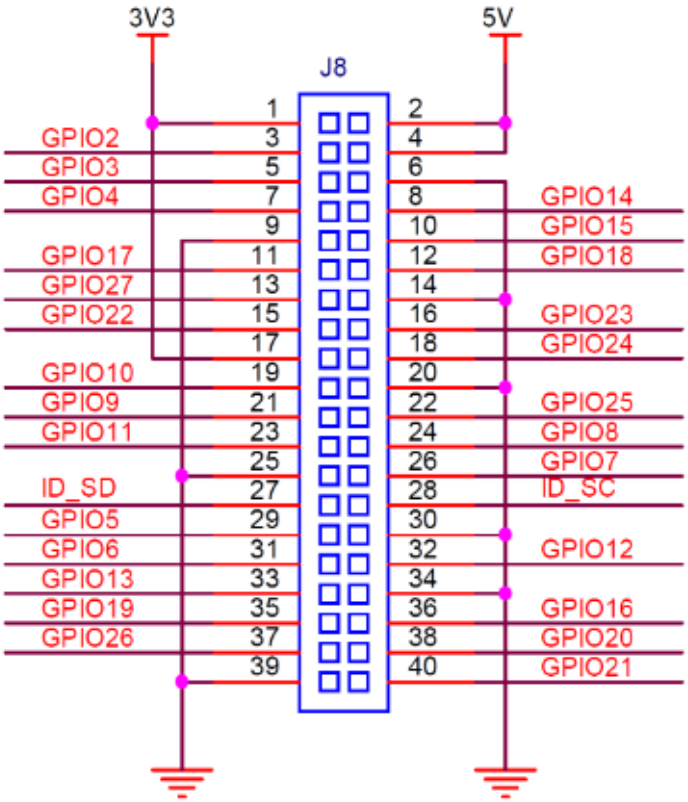

Abbildung 27: GPIO-Pinbelegung [13]

Die Funktion des Status Felds ist derzeit noch im Aufbau und hat keine Funktion. Beim Absenden der Daten wird eine Websocket-Verbindung von dem Webbrowser des Benutzers zum Client aufgebaut. Abhängig von der Aktiv-Checkbox wird der angegebene Pin auf High oder Low gesetzt.

8 Reflektion und Ausblick

Mit dieser Arbeit sollte der EEBUS untersucht und eine Beispielanwendung, die den EEBUS implementiert, in der Programmiersprache Go entworfen werden.

Mit dem implementierten Lösungsentwurf wurde die EEBUS-Spezifikation weitestgehend erfüllt und das vorgegebene Ziel erreicht. Folgende Funktion wurden nur behandelt aber nicht oder anders umgesetzt: Registrierung (s. Abschnitt 2.3.3) und Discovery (s. Abschnitt 2.3.5).

Für zukünftige Projekte kann auf dieser Arbeit aufgesetzt und z.B. Registrierung und Discovery nach exakten Vorgaben umgesetzt werden. Sodass der Raspberry Pi nicht mehr manuell über den Webserver angesteuert werden muss, sondern direkt sobald der Pi aktiv ist auf der Webseite angezeigt wird. Zusätzlich könnte die Webseite um einen Log, der alle SHIP-Nachrichten der SHIP-Teilnehmer anzeigt, erweitert werden.

Rückblickend lässt sich sagen, dass das EEBUS Protokoll ein noch sehr junges und sich noch im Aufbau befindendes Protokoll ist. Deshalb gestaltete sich die Informationsbeschaffung und der Wissensaufbau als sehr aufwändig. Mit dieser Arbeit wurden die Grundlagen für einen schnellen Einstieg in das Thema EEBUs und zukünftige Arbeiten geschaffen. Falls zusätzlich die Programmiersprache Go weiterhin genutzt werden sollte, dann sind auch die technischen Grundlagen für zukünftige Projekte gesetzt.

Unter folgenden Links können die Git-Repositories aufgerufen werden:

Server-Anwendung: https://github.com/tkuehn1/EEBus_Server

Client-Anwendung: https://github.com/tkuehn1/EEBus_Client

9 Literaturverzeichnis

[1] SHIP, „eebus.org,“ 04 11 2019. [Online]. Available: https://www.eebus.org/media-downloads/. [Zugriff am 06 05 2020].

[2] SPINE, „eebus.org,“ 17 12 2018. [Online]. Available: https://www.eebus.org/media-downloads/. [Zugriff am 05 05 2020].

[3] golang.org, „golang.org,“ [Online]. Available: https://blog.golang.org/go-version-1-is-release. [Zugriff am 12 03 2020].

[4] Golang, „golang.org,“ 05 07 2020. [Online]. Available: https://golang.org/doc/. [Zugriff am 07 05 2020].

[5] G. Tour, „go-tour-de.appspot.com,“ [Online]. Available: https://go-tour-de.appspot.com/methods/2. [Zugriff am 07 05 2020].

[6] F. Müller, „jaxenter.de,“ 25 06 2018. [Online]. Available: https://jaxenter.de/nebeneinander-und-doch-miteinander-70957. [Zugriff am 07 05 2020].

[7] F. M. Syariati, „medium.com,“ 19 06 2018. [Online]. Available: https://medium.com/@farissyariati/go-raspberry-pi-hello-world-tutorial-7e830d08b3ae. [Zugriff am 08 05 2020].

[8] driskell, „github.com,“ 20 11 2015. [Online]. Available: https://github.com/driskell/log-courier/blob/master/lc-tlscert/lc-tlscert.go. [Zugriff am 10 05 2020].

[9] zekro, „youtube.com,“ 14 10 2018. [Online]. Available: https://www.youtube.com/watch?v=norUcMSJRtQ. [Zugriff am 10 05 2020].

[10] GPIO, „github.com,“ 14 02 2019. [Online]. Available: https://github.com/stianeikeland/go-rpio. [Zugriff am 10 05 2020].

[11] „gorillatoolkit.org,“ 19 03 2020. [Online]. Available: https://www.gorillatoolkit.org/. [Zugriff am 10 05 2020].

[12] „golang.org,“ Google, [Online]. Available: https://golang.org/dl/. [Zugriff am 11 05 2020].

[13] R. P. (. Ltd., „raspberrypi.org,“ 21 06 2019. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf. [Zugriff am 13 05 2020].

[14] Wikipedia, „de.wikipedia.org,“ 02 09 2019. [Online]. Available: https://de.wikipedia.org/wiki/EEBUS. [Zugriff am 07 05 2020].

10 Anhang

Code Tabelle	Datei: index.html

```
1  /*
2  * Copyright 2014-2015 Jason Woods.
3  *
4  * Licensed under the Apache License, Version 2.0 (the
5  "License");
6  * you may not use this file except in compliance with the
7  License.
8  * You may obtain a copy of the License at
9  *
10 * http://www.apache.org/licenses/LICENSE-2.0
11 *
12 * Unless required by applicable law or agreed to in
13 writing, software
14 * distributed under the License is distributed on an "AS
15 IS" BASIS,
16 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
17 express or implied.
18 * See the License for the specific language governing
19 permissions and
20 * limitations under the License.
21 *
22 * Derived from Golang src/pkg/crypto/tls/generate_cert.go
23 * Copyright 2009 The Go Authors. All rights reserved.
24 * Use of this source code is governed by a BSD-style
25 * license that can be found in the LICENSE file.
26  */
27
28 package main
29
30 import (
```

```
31      "bufio"
32      "crypto/rand"
33      // "crypto/rsa"
34      "crypto/ecdsa"
35      "crypto/elliptic"
36      "crypto/x509"
37      "crypto/x509/pkix"
38      "encoding/pem"
39      "fmt"
40      "math/big"
41      "net"
42      "os"
43      "strconv"
44      "time"
45  )
46
47  var input *bufio.Reader
48
49  func init() {
50      input = bufio.NewReader(os.Stdin)
51  }
52
53  func readString(prompt string) string {
54      fmt.Printf("%s: ", prompt)
55
56      var line []byte
57      for {
58          data, prefix, _ := input.ReadLine()
59          line = append(line, data...)
60          if !prefix {
61              break
62          }
63      }
```

```
64
65      return string(line)
66  }
67
68  func readNumber(prompt string) (num int64) {
69      var err error
70      for {
71          if num, err = strconv.ParseInt(readString(prompt),
72  0, 64); err != nil {
73              fmt.Println("Please enter a valid numerical
74  value")
75              continue
76          }
77          break
78      }
79      return
80  }
81
82  func anyKey() {
83      input.ReadRune()
84  }
85
86  func main() {
87      var err error
88
89      template := x509.Certificate{
90          SerialNumber:          big.NewInt(1),
91          Subject:              pkix.Name{
92                                  Organization: []string{"Log
93  Courier"},
94                               },
95          NotBefore:             time.Now(),
96          SignatureAlgorithm:    x509.ECDSAWithSHA384,
```

```
97          KeyUsage:           x509.KeyUsageCertSign |
98  x509.KeyUsageKeyEncipherment |
99  x509.KeyUsageDigitalSignature,
100         ExtKeyUsage:
101 []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
102         BasicConstraintsValid: true,
103         IsCA:               true,
104     }
105
106     fmt.Println("Specify the Common Name for the
107 certificate. The common name")
108     fmt.Println("can be anything, but is usually set to
109 the server's primary")
111     fmt.Println("DNS name. Even if you plan to connect via
112 IP address you")
113     fmt.Println("should specify the DNS name here.")
114     fmt.Println()
115
116     template.Subject.CommonName = readString("Common
117 name")
118     fmt.Println()
119
120     fmt.Println("The next step is to add any additional
121 DNS names and IP")
122     fmt.Println("addresses that clients may use to connect
123 to the server. If")
124     fmt.Println("you plan to connect to the server via IP
125 address and not DNS")
126     fmt.Println("then you must specify those IP addresses
127 here.")
128     fmt.Println("When you are finished, just press
129 enter.")
130     fmt.Println()
```

```
131
132     var cnt = 0
134     var val string
135     for {
136         cnt++
137
138         if val = readString(fmt.Sprintf("DNS or IP address
139 %d", cnt)); val == "" {
140             break
141         }
142
143         if ip := net.ParseIP(val); ip != nil {
144             template.IPAddresses =
145 append(template.IPAddresses, ip)
146         } else {
147             template.DNSNames = append(template.DNSNames,
148 val)
149         }
150     }
151
152     fmt.Println()
153
154     fmt.Println("How long should the certificate be valid
155 for? A year (365")
156     fmt.Println("days) is usual but requires the
157 certificate to be regenerated")
158     fmt.Println("within a year or the certificate will
159 cease working.")
160     fmt.Println()
161
162     template.NotAfter =
163 template.NotBefore.Add(time.Duration(readNumber("Number
164 of days")) * time.Hour * 24)
```

```
165
166     fmt.Println("Common name:",
167 template.Subject.CommonName)
168     fmt.Println("DNS SANs:")
169     if len(template.DNSNames) == 0 {
170         fmt.Println("    None")
171     } else {
172         for _, e := range template.DNSNames {
173             fmt.Println("   ", e)
174         }
175     }
176     fmt.Println("IP SANs:")
177     if len(template.IPAddresses) == 0 {
178         fmt.Println("    None")
179     } else {
180         for _, e := range template.IPAddresses {
181             fmt.Println("   ", e)
182         }
183     }
184     fmt.Println()
185
186     fmt.Println("The certificate can now be generated")
187     fmt.Println("Press any key to begin generating the
188 self-signed certificate.")
189     anyKey()
190
191     priv, err := ecdsa.GenerateKey(elliptic.P384(),
192 rand.Reader)
193     if err != nil {
194         fmt.Println("Failed to generate private key:", err)
195         os.Exit(1)
196     }
197
```

```
198     serialNumberLimit := new(big.Int).Lsh(big.NewInt(1),
199 128)
200     template.SerialNumber, err = rand.Int(rand.Reader,
201 serialNumberLimit)
202     if err != nil {
203         fmt.Println("Failed to generate serial number:",
204 err)
205         os.Exit(1)
206     }
207 
208     derBytes, err := x509.CreateCertificate(rand.Reader,
209 &template, &template, &priv.PublicKey, priv)
210     if err != nil {
211         fmt.Println("Failed to create certificate:", err)
212         os.Exit(1)
213     }
214 
215     certOut, err := os.Create("server.crt")
216     if err != nil {
217         fmt.Println("Failed to open server.pem for
218 writing:", err)
219         os.Exit(1)
220     }
221     pem.Encode(certOut, &pem.Block{Type: "CERTIFICATE",
222 Bytes: derBytes})
223     certOut.Close()
224 
225     keyOut, err := os.OpenFile("server.key",
226 os.O_WRONLY|os.O_CREATE|os.O_TRUNC, 0600)
227     if err != nil {
228         fmt.Println("failed to open server.key for
229 writing:", err)
230         os.Exit(1)
```

```
231     }
232     b, err := x509.MarshalECPrivateKey(priv)
233     pem.Encode(keyOut, &pem.Block{Type: "EC PRIVATE KEY",
234 Bytes: b})
235     keyOut.Close()
236 
237     fmt.Println("Successfully generated certificate")
238     fmt.Println("    Certificate: server.crt")
239     fmt.Println("    Private Key: server.key")
240     fmt.Println()
241     fmt.Println("Copy and paste the following into your
242 Log Courier")
243     fmt.Println("configuration, adjusting paths as
244 necessary:")
245     fmt.Println("    \"transport\": \"tls\",")
246     fmt.Println("    \"ssl ca\":
247 \"path/to/server.crt\",")
248     fmt.Println()
249     fmt.Println("Copy and paste the following into your
250 LogStash configuration, ")
251     fmt.Println("adjusting paths as necessary:")
252     fmt.Println("    ssl_certificate =>
253 \"path/to/server.crt\",")
254     fmt.Println("    ssl_key         =>
255 \"path/to/server.key\",")
256 }
```

Ersteller: Kilic, Kühn	Datum: 08.05.2020