Bachelorarbeit

Ein robustes Funkprotokoll für IoT-Anwendungen im Zug

A Robust Wireless Network Protocol for On-Train IoT Applications

Clemens Tiedt

Hasso-Plattner-Institut an der Universität Potsdam

29. Juni 2021

Bachelorarbeit

Ein robustes Funkprotokoll für IoT-Anwendungen im Zug

A Robust Wireless Network Protocol for On-Train IoT Applications

von
Clemens Tiedt

Betreuung
Prof. Dr. Andreas Polze, Lukas Pirl, Robert Schmid
Professur für Betriebssysteme und Middleware
Philippe Fuchs, Henry Hübler, Jenne Krey,
Gwyneth Mettendorf, Alexander Al Schmitt, Ingo Schwarzer
DB Systel GmbH

Hasso-Plattner-Institut an der Universität Potsdam

29. Juni 2021

Zusammenfassung

Das Projekt *Die IoT-Middleware im Zug* schafft die Basis für eine Plattform, mit deren Hilfe die DB Systel und ihre Kund:innen Sensordaten auf Zügen sammeln und verarbeiten können. Hierzu stellen wir Infrastruktur und Software-Bibliotheken bereit, die das Sammeln von Sensordaten und deren Weiterleitung an ein Backend ermöglichen. Sensordaten werden innerhalb eines Zuges kabellos mit *nRF24L01*-Funkmodulen verschickt. Diese haben wir aufgrund ihrer guten Verfügbarkeit und existierender Unterstützung durch Treiber-Bibliotheken gewählt. Aufbauend auf dem Protokoll der Funkmodule, *Enhanced Shockburst*, haben wir ein eigenes Netzwerkprotokoll, das *On-Train Transmission Protocol* (OTTP), entwickelt, das den Versand größerer Datenmengen als *Enhanced Shockburst* erlaubt, einen eigenen Adressvergabemechanismus nutzt und Mechanismen zur Verbesserung der Robustheit besitzt. Wir erläutern zudem, wie wir OTTP als Referenzimplementierung in Rust umgesetzt haben. Diese nutzt die moderne Syntax und Standardbibliothek von Rust, um eine einfach nutzbare und sichere Schnittstelle für Programmierer:innen zu bieten. Durch die Nutzung der Rust-Bibliothek *embedded-hal* unterstützen wir verschiedene Plattformen.

Inhaltsverzeichnis

1 Einleitung

Unser Projekt *Die IoT-Middleware im Zug* stellt die Basis für eine Plattform bereit, mit deren Hilfe die DB Systel und ihre Kund:innen Sensordaten auf Zügen erheben und für verschiedene Anwendungsfälle verarbeiten können.

1.1 Motivation

Sowohl die DB Systel selbst, als auch Kund:innen können in vielen Anwendungsfällen von der Sammlung von Sensordaten auf Zügen profitieren. Bis jetzt existiert allerdings keine einheitliche Plattform, um dies zu ermöglichen. Wer also Sensordaten sammeln und verarbeiten möchte, muss eine eigene Lösung schaffen, um Daten zu sammeln, vom Zug in die Cloud zu schicken und zu verarbeiten. Unser Ansatz sorgt dafür, dass die Sammlung der Daten durch dafür bereitgestellte Software-Werkzeuge vereinfacht wird und automatisiert das Versenden in die Cloud. Somit können Nutzer:innen sich fast vollständig auf die Verarbeitung der Daten konzentrieren.

Bei der Konzeption unserer Plattform haben wir uns auf einige Beispielanwendungsfälle aus Sicht der DB Systel und von Drittkund:innen bezogen.

Predictive Maintenance *Predictive Maintenance* ("vorausschauende Wartung") nutzt Daten wie beispielsweise Tonaufnahmen, um mittels maschinellem Lernen früher Wartungsbedarf zu erkennen und Reparaturen bereits vor dem Ausfall eines Bauteils zu ermöglichen. Dieser Themenbereich wird von Matrisch in [11] näher untersucht.

Live-Überwachung von Frachtgut Beim Transport bestimmter Güter (beispielsweise Impfstoffen) müssen für den gesamten Transport bestimmte Umweltbedingungen gesichert werden. Moderne Mikrocontroller mit geringem Energieverbrauch eröffnen die Möglichkeit, Messdaten innerhalb von beispielsweise Containern zu sammeln und nahezu in Echtzeit zu überwachen. Somit kann bereits während des Transports festgestellt werden, wenn es zu Schäden kommt.

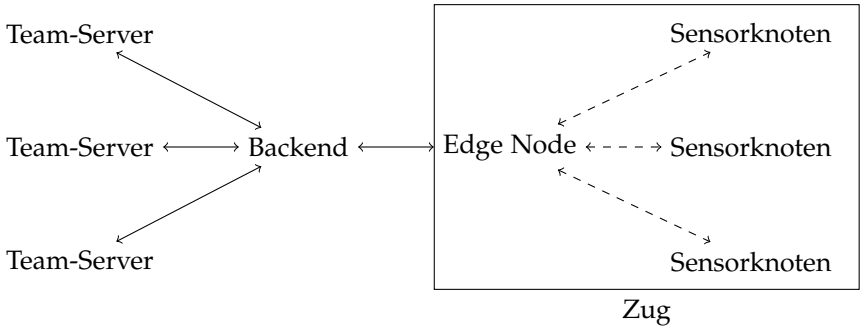
Messung von Fahrgastzahlen Auch Fahrgäste können von unserer Plattform profitieren. Anhand von WLAN-Signalen kann abgeschätzt werden, wie voll ein Wagen zu einem Zeitpunkt ist. Somit kann über eine App oder Webseite angezeigt werden, wo Fahrgäste einsteigen sollten, um einen Sitzplatz zu bekommen. Dieses Thema wird von Wanke in [17] näher untersucht.

1.2 Architektur

Das Projekt stellt eine Werkzeugsammlung bereit, mit deren Hilfe Kund:innen einfach Sensorknoten auf Basis von handelsüblichen Mikrocontrollern bauen können. Weiterhin

gibt es Software für *Edge Nodes*, die Sensordaten auf dem Zug sammeln und in die Cloud weiterleiten. Auf der Cloud-Seite gibt es einen zentralen Server sowie einzelne *Team-Server*, die von Kund:innen betrieben werden und die letztendliche Verarbeitung der Daten übernehmen.

Abbildung 1.1: Die Architektur des Gesamtsystems. Gestrichelte Linien sind nRF24-Funkverbindungen, durchgezogene sind Internetverbindungen.



Der Fokus dieser Arbeit ist die Kommunikation von Sensorknoten und Edge Nodes auf dem Zug (siehe Abbildung 1.1). Wir haben verschiedene Funktechnologien und Protokolle für den Einsatz im Zug evaluiert und basierend auf den gewonnenen Erkenntnissen, welche besonderen Anforderungen und Einschränkungen sich dort finden, ein eigenes Netzwerkprotokoll entwickelt.

1.3 Komponenten des Projekts

Wir geben nach dem Überblick über die gesamte Architektur nun noch einen genaueren Einblick in die einzelnen Komponenten.

Sensorknoten Nutzer:innen unserer Plattform können mithilfe von Sensorknoten Daten auf Zügen sammeln. Da Nutzer:innen ihre Sensorknoten selbst bereitstellen, können diese sehr verschiedene Hardware besitzen. Es ist denkbar, dass sie auf Mikrocontrollern aufbauen, sie könnten aber auch leistungstärkere Hardware wie einen *Raspberry Pi*[1] als Basis nutzen. Welche Sensoren darauf verbaut sind, ist komplett den Herstellern überlassen. Sensorknoten müssen die von Seibold in [13] beschriebene und im Rahmen dieses Projekts entwickelte Bibliothek zur Sammlung von Sensordaten nutzen.

Edge Nodes Für die Sammlung der Sensordaten auf einem Zug und deren Weiterleitung ins Backend sind *Edge Nodes* zuständig. Diese werden von uns bereitgestellt und nutzen Computer wie einen *Raspberry Pi*. Es wäre auch möglich, leistungsstärkere Computer bis ungefähr zur Leistungsklasse eines Intel Core i7 zu nutzen. Allerdings wird die höhere Leistung nicht unbedingt benötigt und verbraucht mehr Energie. Im Gegensatz zu Sensorknoten besitzen Edge Nodes eine Internetverbindung. Neben

[1] Teach, Learn, and Make with Raspberry Pi, `https://www.raspberrypi.org/` (abgerufen am 29. Juni 2021)

der reinen Sammlung von Sensordaten können Edge Nodes auch zur Verteilung von Aktualisierungen oder zur Verwertung von Daten bereits auf dem Zug genutzt werden. Zudem kann auch die Edge Node Daten sammeln, was unter anderem in [17] genutzt wird.

Backend Das Backend ist ein Verzeichnis der aktiven Team-Server. Wird ein neuer Sensorknoten auf einem Zug eingebaut, ermittelt die Edge Node beim Backend, zu welchem Team-Server dieser gehört.

Team-Server Die Team-Server werden von unseren Nutzer:innen betrieben. An die Team-Server werden die Daten letztendlich übermittelt. Diese Aufteilung in Backend und Team-Server sorgt dafür, dass Sensordaten nicht auf unseren Geräten gespeichert werden (auf Edge Nodes werden sie nur so lange gespeichert, bis sie an die Team-Server versandt werden können). Wie die Daten auf den Team-Servern verarbeitet werden, liegt komplett in den Händen der Nutzer:innen.

2 Grundlagen

2.1 IoT-Funktechnologien und existierende Protokolle

Im IoT-Bereich sind bereits verschiedene Netzwerkprotokolle für verschiedene Hardware und Anwendungsfälle etabliert. Wir werden nun erläutern, weshalb wir unser eigenes Transportprotokoll für dieses Projekt entwickelt haben und warum wir *nRF24* als Funktechnologie für die Referenzimplementierung gewählt haben.

Tabelle 2.1: Schichten im OSI-Modell[4]

Schicht	Name	Einheit
7	Anwendung	Daten
6	Darstellung	Daten
5	Sitzung	Daten
4	Transport	Segmente/Datagramme
3	Vermittlung/Paket	Pakete
2	Sicherung	Frames
1	Bitübertragung	Bits, Symbole

Über das gesamte Projekt verwenden wir Protokolle aus allen Schichten des OSI-Modells (siehe Tabelle 2.1), für diesen Teil des Projekts fokussieren wir uns jedoch auf die erste bis vierte Schicht.

Als erstes suchten wir eine Hardware-Basis für unser Netzwerk. An dieser Stelle geben wir nur einen kurzen Überblick, eine detailliertere Auswertung der verschiedenen Funktechnologien findet sich in [7]. Um eine passende zu finden, untersuchten wir eine Reihe von Funktechnologien nach folgenden Kriterien:

1. Ist die genutzte Frequenz in Deutschland legal? Bestimmte Funkfrequenzen dürfen in Deutschland nicht ohne weiteres genutzt werden. Deshalb muss unsere Funktechnologie für die Nutzung in Deutschland zugelassen sein.

2. Beträgt die Reichweite mindestens 30m? Wir treffen die Annahme, dass ein Wagen im Durchschnitt 30m lang ist. Ein Sensorknoten an einem Ende des Wagens muss also mit einer *Edge Node* am anderen Ende kommunizieren können.

3. Sind Übertragungsgeschwindigkeiten von mindestens 300kb/s möglich? Neben Sensordaten planen wir, auch Firmware-Updates oder neue Konfigurationen kabellos an Sensorknoten zu verteilen. Auch wenn diese für Embedded-Geräte kleiner ausfallen als für normale PCs, ist eine Datenrate von ca. 300kb/s notwendig, damit die Netzwerkgeschwindigkeit den Betrieb nicht negativ beeinflusst.

4. Ist die Technologie energieeffizient? Die Stromzufuhr auf dem Zug kann nicht durchgehend garantiert werden. Es ist also möglich, dass Sensorknoten eine Batterie zumindest als Rückfalloption nutzen. Dafür sollte die genutzte Funktechnologie möglichst wenig Energie benötigen.

Aufgrund dieser Kriterien nahmen wir vier Funktechnologien in die nähere Auswahl (siehe Tabelle 2.2).

Tabelle 2.2: Vergleich der untersuchten Funkstandards in der näheren Auswahl

Name	Funkfrequenz	Reichweite	Übertragungsgeschwindigkeit
Bluetooth LE (v5)[14]	2.4Ghz	400m	2Mbps
MiWi	868Mhz; 2.4Ghz[2]	100m	250kbps
nRF24[12]	2.4Ghz	55m	2Mbps
Weightless-W[15]	169 – 915Hz	5km	100kps

Von diesen schlossen wir *MiWi* und *Weightless-W* aus, da beide proprietär sind und entsprechend nur Unterstützung von ihren Herstellern haben. Insbesondere gibt es keine Bibliotheken, um Funkmodule dieser Hersteller in Rust anzusprechen. Auch *Bluetooth LE* verwarfen wir nach einiger Recherche. Dieses hat zwar den Vorteil, dass es von vielen modernen Smartphones unterstützt wird und Mikrocontroller mit Bluetooth-Unterstützung sehr verbreitet sind, ist allerdings nicht für klassischen Netzwerkanwendungen ausgelegt. Deshalb entschieden wir uns, *nRF24*-Funkmodule von *Nordic Semiconductor*[3] zu verwenden.

Diese haben den großen Vorteil, dass sie kostengünstig verfügbar und in der Maker-Szene sehr beliebt sind. Dadurch gibt es viele existierende Code-Beispiele und Unterstützung anderer Protokolle auf Basis von *nRF24*. Es stellte sich also die Frage, welches Protokoll wir in der Transport-Schicht einsetzen sollten. Offensichtliche Kandidaten hierfür waren TCP und UDP. Es gibt auch eine Implementierung von TCP/IP für *nRF24*[4] diese ist aber nicht mit Rust kompatibel. Zudem sind beide Protokolle nicht optimal für unseren Anwendungsfall. Das zugrundeliegende *Enhanced Shockburst*-Protokoll der *nRF24*-Module kann maximal 32 Bytes in einem Paket senden. Somit nimmt der TCP-Header bereits einen großen Anteil des Pakets ein. UDP hat den Nachteil, dass es nicht zuverlässig ist. Da wir wie erwähnt größere Daten auf mehrere Pakete aufteilen müssen, steigt die Chance, mit UDP Pakete zu verlieren und die ganze Übertragung unbrauchbar zu machen. Somit entschieden wir, ein eigenes Protokoll zu entwickeln, das speziell für die Anforderungen und Einschränkungen der Nutzung im Zug angepasst ist.

[2] 902 – 928Mhz auch in Nordamerika

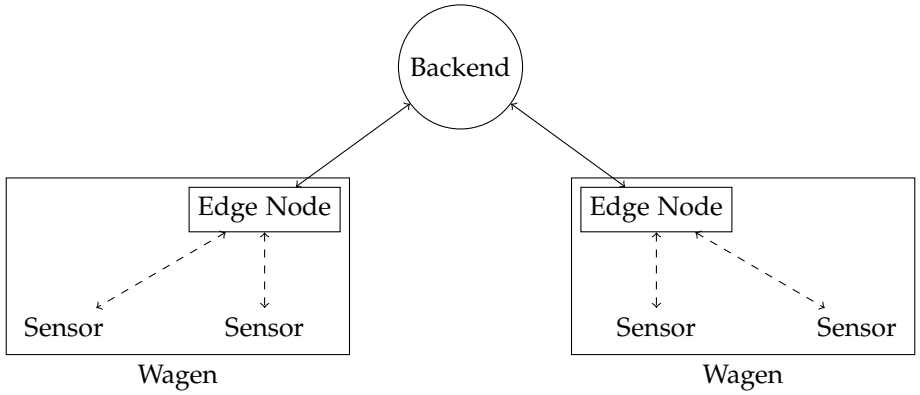
[3] Nordic Semiconductor - Home - nordicsemi.com, `https://www.nordicsemi.com/` (abgerufen am 29. Juni 2021)

[4] GitHub - nRF24/RF24Ethernet: OSI Layers 4&5 Wireless (not WiFi) TCP/IP networking on Arduino devices using nrf24l01+ radios, `https://github.com/nRF24/RF24Ethernet` (abgerufen am 29. Juni 2021)

2.2 Anforderungen an das Anwendungsszenario Zug

Das Anwendungsszenario besitzt einige besondere Anforderungen und erlaubt es uns, einige besondere Annahmen über den Charakter unseres Netzwerks zu treffen. Zuerst wollen wir diesen kurz beschreiben. Innerhalb eines Wagens besitzt es eine Sterntopologie (siehe Abbildung 2.1), in deren Zentrum sich eine Edge Node befindet. Diese haben wir gewählt, da sie für unseren Anwendungsfall am simpelsten ist. Wir gehen davon aus, dass jeder Sensorknoten mindestens eine *Edge Node* erreichen kann und grundsätzlich auch nur mit dieser kommunizieren muss. Somit vermeiden wir die Notwendigkeit von komplexeren und möglicherweise unsicheren Routing-Mechanismen. Beispielsweise vermeiden wir die kaskadierende Ausbreitung von Fehlern, wie sie in [3] beschrieben wird, da die Verbindung eines Sensorknotens nicht von anderen Sensorknoten abhängig ist. Die Sensorknoten, die mit der *Edge Node* verbunden sind, schicken in regelmäßigen Abständen ihre Messdaten an sie und erhalten gelegentlich neue Konfigurationen oder Firmware-Updates.

Abbildung 2.1: Aufbau des Netzwerks im Zug



Wir nehmen an, dass nur selten Geräte entfernt werden und auch nur unregelmäßig neue dem Netzwerk hinzugefügt werden. Allerdings kann nicht davon ausgegangen werden, dass Sensorknoten immer erreichbar sind. Die Stromversorgung im Zug ist nicht zu allen Zeitpunkten garantiert und es ist möglich, dass die Funkverbindung zeitweise überlastet oder unterbrochen wird. Unser Protokoll muss also mit der zeitweisen Abwesenheit von Geräten umgehen können.

2.3 Definition von Robustheit

Nach der Erläuterung, warum für unseren Anwendungsfall ein eigenes Netzwerkprotokoll sinnvoll ist und welchen Anforderungen es im Einsatzbereich Zug unterliegt, definieren wir nun den Begriff der Robustheit näher. Dieser ist zwar häufig im Kontext von Netzwerkprotokollen zu finden, aber oft nur ungenau definiert.

Eine Definition stellt das *Institute of Electrical and Electronical Engineers* (IEEE) bereit:
„The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.“ ([10]) (zu Deutsch: "Der Grad, zu dem ein System oder eine Komponente in der Anwesenheit von invaliden Eingaben oder anstrengenden Umgebungsbedingungen korrekt funktionstüchtig ist"). Eine ähnliche Definition findet sich bei Fernandez et al.: „Informally, robustness can be defined as the ability of a software to keep an 'acceptable' behavior [...] in spite of exceptional or unforeseen execution conditions“ ([6]) (zu Deutsch: “Informell kann Robustheit als die Fähigkeit einer Software, trotz außergewöhnlicher oder unvorhergesehener Ausführungsbedingungen ein 'akzeptables' Verhalten beizubehalten, definiert werden").

Neben diesen Definitionen, die sich auf Software im Allgemeinen beziehen, wollen wir hier auch das Gebiet der Netzwerktheorie einbeziehen. In diesem Kontext findet sich folgende Definition für Robustheit: „A system is robust if it can maintain its basic functions in the presence of internal and external errors. In a network context robustness refers to the system's ability to carry out its basic functions even when some of its nodes and links may be missing.“ ([2]) (zu Deutsch: “Ein System ist robust, wennn es seine grundlegenden Funktionen in der Anwesenheit interner und externer Fehler aufrecht erhalten kann. Im Kontext von Netzwerken bezeichnet Robustheit die Fähigkeit des Systems, seine grundlegenden Funktionen auszuführen, selbst wenn einige seiner Knoten und Verbindungen fehlen”). Hier spielen die Topologie des Netzwerks und die Widerstandsfähigkeit gegen kaskadierende Ausbreitung von Fehlern eine besondere Rolle. Fällt beispielsweise eine Edge Node in unserer Architektur aus, können Sensorknoten prinzipiell nicht mehr mit dem Internet kommunizieren. Durch die gewählten Abstände von je 30m zwischen zwei Edge Nodes hat jedoch jeder Sensorknoten zwei mögliche Edge Nodes zur Verbindung, was diese Schwäche ausgleicht.

Aus den genannten Definitionen leiten wir nun unsere Kriterien an Robustheit ab.

Zuverlässiger Datenaustausch Gesendete Daten sollen trotz umweltbedingten Einflüssen ihren Empfänger erreichen.

Schutz vor kaskadierenden Fehlern Ein Fehler auf einem Gerät darf nicht das ganze Netzwerk außer Kraft setzen.

Behebung von Fehlerzuständen Geräte sollten in der Lage sein, Fehlerzustände selbst ohne menschliche Eingriffe zu beheben.

2.4 Embedded-Programmierung mit Rust

Für dieses Projekt haben wir uns entschieden, für alle Komponenten, die auf Embedded-Geräten laufen sollen, die Programmiersprache *Rust* zu verwenden. Konventionell ist der Embedded-Bereich von C und C++ (zum Beispiel in Form des *Arduino*-Frameworks oder herstellereigener Frameworks wie der *ESP-IDF*-Werkzeugkette von *Espressif*) dominiert. Rust unterstützt im Jahr 2021 zwar noch weniger Mikrocontroller-Plattformen als beispielsweise *Arduino*, bietet dafür allerdings diverse Garantien und andere Vorteile.

Speichersicherheit Ein klassisches Problem von C und C++ sind Fehler in der Speicherverwaltung, da diese manuell erfolgt. Rust führt das Konzept von *Ownership* ein, durch das ein Speicherbereich zu einem Zeitpunkt immer genau von einer Variable verwaltet wird. Durch statische Analysen weiß der Rust-Compiler, wer wann auf welchen Speicher Zugriff hat und kann somit häufige Probleme wie Speicherlecks oder das Dereferenzieren eines Null-Pointers erkennen. Da dies beim Kompilieren passiert, beeinflusst es die Laufzeit eines Rust-Programms nicht.

Starkes Typsystem Rust bietet ein sehr ausdrucksstarkes Typsystem. Beispielsweise können Aufzählungen (`enums`) Strukturen als Variante haben, was Summentypen aus funktionalen Sprachen wie Haskell ähnelt. Gleichzeitig können Fehler durch inkompatible Typen bereits beim Kompilieren bemerkt werden, statt Probleme in der Ausführung zu verursachen.

Moderne Syntax Um das starke Typsystem und andere Sprachfunktionen zu unterstützen, bietet Rust einige moderne syntaktische Konstrukte. Beispielsweise gibt es *Pattern-Matching* für Aufzählungen (`enums`), das den Zugriff auf Felder von Varianten ermöglicht und sicherstellt, dass jeder mögliche Wert durch einen Fall abgedeckt wird.

Für verschiedene Mikrocontroller-Plattformen stehen Bibliotheken für Hardware-Abstraktionen (englisch *Hardware Abstraction Layer*, HAL) zur Verfügung, die den Zugriff auf GPIO-Pins (*General Purpose Input/Output*) und Peripheriegeräte wie die Echtzeituhr oder *Analog Digital Converter* ermöglichen. Für die Programmierung mit Hardware bietet das starke Typsystem in Rust Unterstützung, da beispielsweise GPIO-Pins nicht nur durch eine Nummer dargestellt werden, sondern verschiedene Pins verschiedene Typen haben. Falls ein via GPIO angeschlossenes Gerät also eine Schnittstelle wie SPI (*Serial Peripheral Interface*) benutzt, die nur an bestimmten Pins zur Verfügung steht, kann das Typsystem bereits zur Compile-Zeit garantieren, dass nur korrekte Pins genutzt werden.

3 Funktionen des Protokolls

Nach der Erläuterung, in welchem Kontext wir ein Netzwerkprotokoll entwickelt haben, auf welcher Hardwarebasis es aufsetzt und welche Anforderungen es erfüllen soll, beschreiben wir nun seine Funktionen. Das Protokoll, das im Rahmen dieser Arbeit entwickelt wurde, haben wir *On-Train Transmission Protocol* (kurz *OTTP*) genannt. Es baut in seiner Referenzimplementierung auf das Protokoll *Enhanced Shockburst* (das native Protokoll der von uns genutzten *nRF24*-Funkmodule) auf, ließe sich aber problemlos auf andere Protokolle, die eine Paketgröße von mindestens 32 Bytes unterstützen, portieren.

3.1 Das Protokoll *Enhanced Shockburst*

Wie in Abschnitt 2.1 dargelegt, haben wir uns für *nRF24*-basierte Funkmodule von *Nordic Semiconductor* als Basis für unser eigenes Netzwerkprotokoll entschieden, speziell das Modell *nRF24L01*. Diese arbeiten mit einem Protokoll namens *Enhanced Shockburst*[16]. Da dessen Funktionalitäten und Einschränkungen maßgeblich das Design von *OTTP* beeinflusst haben, geben wir einen kurzen Überblick darüber.

Enhanced Shockburst funktioniert grundsätzlich Radio-artig. Jeder Teilnehmer legt seine Adresse selbst fest. Pakete werden auf einem Kanal im 2.4Ghz-Band gesendet und wenn ein Teilnehmer ein Paket mit der eigenen Adresse im Feld *Address* empfängt (siehe Tabelle 3.1), verarbeitet er dieses.

Tabelle 3.1: Aufbau eines Pakets in Enhanced Shockburst

Preamble	Address	PCF	Payload	CRC
1 Byte	3 – 5 Byte	9 Bit	1 – 32 Byte	1 – 2 Byte

Es wird eine dynamische Übertragungsgröße von bis zu 32 Byte pro Paket unterstützt. Zusätzlich wird eine CRC-Prüfsumme (*Cyclic Redundancy Check*) mitgesendet, die eine Validierung von Paketen erlaubt.

Tabelle 3.2: Aufbau des PCF-Felds in Enhanced Shockburst. Längen der Felder in Bit.

PAYLOAD_LEN	PID	NO_ACK
6	2	1

Das *Packet Control Field* (PCF, siehe Tabelle 3.2) enthält weitere Metadaten, genauer die Länge der übertragenen Daten (*PAYLOAD_LEN*) sowie eine Paket-ID und ein *Acknowledgement-Bit*, welche für die automatische Bestätigung von Paketen genutzt werden. Auch wenn das Protokoll mit dem Feld *PAYLOAD_LEN* Übertragungen von mehr

als 32 Byte unterstützt, sind die von uns genutzten *nRF24L01*-Funkmodule auf diese Übertragungsgröße beschränkt.

3.2 Übertragungen und Pakete

Wie bereits erläutert, können wir mittels *Enhanced Shockburst* maximal 32 Byte pro Paket senden. Für Messdaten unserer Sensoren oder andere Daten wie Konfigurationen oder Firmware-Updates reicht dies nicht aus. Deshalb arbeitet *OTTP* mit Übertragungen, die mit Segmenten bei TCP vergleichbar sind. Eine Übertragung besteht aus mindestens einem *OTTP*-Paket. Diese haben eine Länge von 32 Byte und werden als *Payload* eines *Enhanced Shockburst*-Pakets gesendet.

3.3 Aufbau eines Pakets

Im folgenden beschreiben wir, wie ein *OTTP*-Paket aufgebaut ist.

Tabelle 3.3: Aufbau eines OTTP-Pakets. Die Zahlen der oberen Reihe entsprechen Bytes.

0 – 1	2	3	4 – 5	6 – 7	8 – 31
Transmission-ID	Message Kind	Sender	Offset	Transmission Length	Payload

Ein Paket besitzt sechs Felder, von denen Metadaten acht Byte einnehmen.

Das erste Feld ist die *Transmission-ID* (Übertragungs-ID). Diese ist 16 Bit groß und für alle Pakete in einer Übertragung gleich.

Im zweiten Feld, das ein Byte einnimmt, ist die *Message Kind*, also die Funktion des Pakets, gespeichert. OTTP unterstützt neun verschiedene Pakettypen, deren Funktionen in Tabelle 3.4 erklärt sind.

Als nächstes folgt die Adresse. *Enhanced Shockburst* nutzt bis zu fünf Byte große Adressen. Da in unserem Anwendungsfall jedoch deutlich weniger Geräte ein Netzwerk bilden, reicht für *OTTP* ein Byte. Dabei ist die Adresse $0xFF$ für Router vorbehalten. Da *Enhanced Shockburst* die Adresse des Absenders nicht im Paket mit sendet, gibt es dieses Feld in *OTTP*.

Der *Offset* bestimmt die Reihenfolge der Pakete in einer Übertragung und beginnt immer bei null. Der *Offset* ist zwei Byte groß.

Nach dem *Offset* steht die Länge der gesamten Übertragung. Diese kann vom Empfänger genutzt werden, um festzustellen, ob eine Übertragung vollständig ist und um abzuschätzen, wie lange eine Übertragung dauern wird. Wie der *Offset* ist sie zwei Byte groß.

Tabelle 3.4: Paketypen und ihre Funktionen

Nummer	Name	Funktion
0	Data	Daten in einer Übertragung versenden
1	RequestAddress	Eine Adresse beim Router anfordern
2	PublishAddress	Als Router eine Adresse vergeben
3	AddressReceived	Nach Erhalt von (2) eine Adresse bestätigen
4	Approved	Adresse vom Router genehmigt
5	Abort	Fehler bei der Adressaushandlung
6	Connect	Verbindungsanfrage
7	Disconnect	Verbindung beenden
8	AcceptConnection	Verbindungsanfrage annehmen
9	RejectConnection	Verbindungsanfrage ablehnen

Die letzten 24 Byte enthalten die gesendeten Daten.

3.4 Fragmentierung

Wie bereits erwähnt, schränkt uns die verwendete *nRF24*-Funktechnologie auf eine Paketgröße von 32 Byte ein. Um größere Übertragungen vornehmen zu können, nutzt OTTP einen Fragmentierungsmechanismus. Beim Start einer neuen Übertragung legt der Sender zuerst die *Transmission-ID* fest. Diese wird zufällig gewählt, da bei einer ausreichenden Zufallsquelle und einer Größe von zwei Byte für die *Transmission-ID* die Chance einer Kollision bei $\frac{1}{2^{16}}$ liegt. Die zu sendenden Daten werden nun in bis zu 24 Byte große Teile aufgeteilt und daraus Pakete mit der festgelegten *Transmission-ID* und dem passenden *Offset* zur Bestimmung der Reihenfolge erstellt. Der Sender berechnet außerdem, wie viele Pakete die Übertragung enthalten wird und speichert dies im Feld *Transmission Length* für alle Pakete.

Der Empfänger wartet nun, bis entweder alle Pakete empfangen sind oder die Wartezeit für die Übertragung überschritten ist. Ob alle Pakete empfangen wurden, wird daran überprüft, ob die Übertragung *beendet* und *vollständig* ist. Ersteres bedeutet, dass ein Paket empfangen wurde, dessen *Offset* eins weniger als die Übertragungslänge ist. Zweiteres bedeutet, dass alle Pakete mit einem *Offset* kleiner dem maximalen *Offset* empfangen wurden.

Eine weitere Option wäre hier ein Präambel-Paket gewesen, das Metadaten wie die Länge der Übertragung im Vorhinein schickt. Wir haben uns allerdings dagegen entschieden, da wir in den meisten Fällen somit ein Paket mehr schicken müssten und nur je zwei Byte in den folgenden Paketen sparen. Mit einem weiteren Paket steigt die Wahrscheinlichkeit, dass Pakete fehlerhaft übertragen werden oder verloren gehen und die Implementierung auf der Empfängerseite wird komplexer, da sie für die Überprüfung der Vollständigkeit zuerst das Präambel-Paket finden muss.

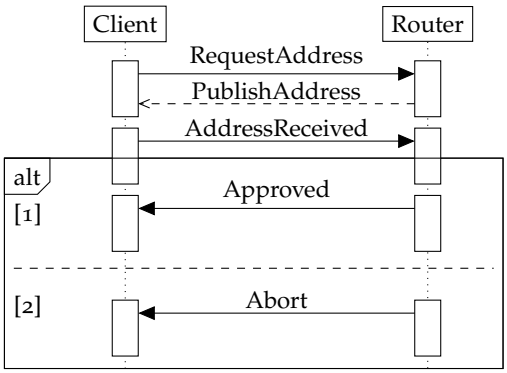
Außerdem kann das Präambel-Paket für Angriffe missbraucht werden. Beispielsweise könnte ein Angreifer die *Transmission-ID* und den Empfänger einer Übertragung abhören und dann ein Präambel-Paket mit einer falschen Übertragungslänge generieren. Je nachdem, wie der Empfänger diesen Fall behandelt (denkbar wäre beispielsweise das Verwerfen der Übertragung oder das Verwenden des zuerst empfangenen Präambel-Pakets), könnte ein Angreifer somit zumindest das Verwerfen einer Übertragung erzwingen. Wenn alle Pakete die Übertragungslänge enthalten, kann der Empfänger zumindest prüfen, welche Übertragungslänge häufiger vorkommt. Dies bietet auch keine komplette Sicherheit, erhöht aber bei einer gestörten Übertragung zumindest die Chance, diese zu rekonstruieren.

3.5 Aushandlung von Adressen

Um die Einrichtung von Geräten so unkompliziert wie möglich zu gestalten, bietet *OTTP* eine Form von automatischer Adressaushandlung an. Diese findet als Zwei-Phasen-Commit statt, sodass das Gerät zuerst eine Adresse anfragt und diese dann beim Router (im praktischen Einsatz also bei der *Edge Node*) bestätigt. Der Ablauf ist in Abbildung 3.1 dargestellt. Um eine Adresse zu erhalten, sendet ein Gerät zuerst ein Paket mit dem Nachrichtentypen *RequestAddress* an den Router. Daraufhin bestimmt der Router eine freie Adresse und sendet ein *PublishAddress*-Paket mit dieser Adresse als Daten an das Gerät zurück.

Nun muss das Gerät mit einem *AddressReceived*-Paket die Adresse bestätigen, die es empfangen hat. Dem ersten Gerät, das eine Adresse bestätigt, sendet der Router nun ein *Approved*-Paket und beendet damit die Adressvergabe für dieses Gerät. Sollte ein anderes Gerät versuchen, die gleiche Adresse zu bestätigen, sendet der Router die Nachricht *Abort*, womit das Gerät die Adresse nicht akzeptieren darf und erneut eine Adresse anfordern muss.

Abbildung 3.1: Ablauf der Adressaushandlung

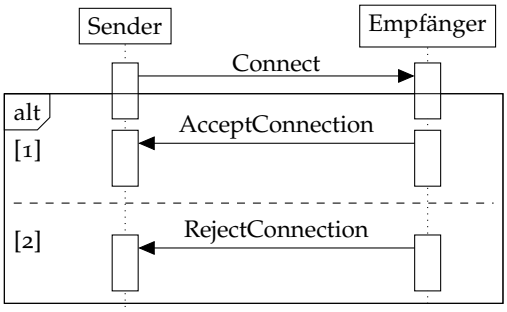


(1): Adresse wurde noch nicht bestätigt
(2): Adresse wurde bereits von einem Client bestätigt

3.6 Verbindungen

OTTP ist als verbindungsorientiertes Protokoll konzipiert, da dies das Ziel von größeren Übertragungen als 32 Byte deutlich vereinfacht. Der Empfänger weiß, mit welchem anderen Gerät er zu einem Zeitpunkt verbunden ist und muss empfangene Übertragungen nicht nach dem Sender filtern. Deshalb findet der Austausch von Übertragungen innerhalb von Verbindungen statt. Der Verständlichkeit halber bezeichnen wir die beiden Geräte in einer Verbindung als Sender und Empfänger, auch wenn das Senden und Empfangen bidirektional funktioniert. Der Ablauf des Verbindungsaufbaus ist in Abbildung 3.2 dargestellt. Um die Verbindung herzustellten, sendet der Sender ein *Connect*-Paket an den Empfänger. Unterhält der Empfänger gerade nicht eine anderen Verbindung, antwortet er mit *AcceptConnection*, sonst *RejectConnection*, da neue Verbindungen existierende nicht unterbrechen dürfen. Nun können Sender und Empfänger sich gegenseitig Daten schicken. Solange die Verbindung aktiv ist, werden beide Teilnehmer nur Daten des jeweils anderen akzeptieren. Ist die Kommunikation beendet, kann einer der Teilnehmer die Verbindung mit der Nachricht *Disconnect* trennen.

Abbildung 3.2: Ablauf beim Verbindungsaufbau



(1): Noch nicht verbunden
(2): Bereits eine andere aktive Verbindung

3.7 Sicherheit und Zuverlässigkeit

Eine wichtige Anforderung an *OTTP* ist Zuverlässigkeit. Darunter verstehen wir, dass Übertragungen vollständig und unverändert ankommen und die Netzwerkkommunikation durch fehlerhafte oder böswillige Geräte nicht blockiert werden kann. Eine verwendete Methode, um die Zuverlässigkeit zu garantieren, sind Zeitüberschreitungen. Wie bereits erläutert, kann es beispielsweise vorkommen, dass Geräte unerwartet die Stromzufuhr verlieren. Sollte dies während eines Ablaufs in *OTTP* wie der Adressaushandlung passieren, kann dadurch ein anderes Gerät in einem Zustand bleiben, in dem es auf eine Antwort wartet, diese aber niemals erhalten kann. Um dies zu vermeiden, gibt es immer Zeitüberschreitungen, wenn ein Gerät auf Daten von einem anderen wartet.

Eine weitere Methode, um *OTTP* robuster zu machen, wären Prüfsummen. Es gibt bereits auf der Ebene von *Enhanced Shockburst* Prüfsummen, die jedoch nur die Integrität eines *OTTP*-Pakets gewährleisten können. Beispielsweise können wir nicht komplett ausschließen, dass zwei Geräte gleichzeitig die selbe *Transmission-ID* wählen und durch einen Fehler oder böswilligerweise gleichzeitig an den selben Empfänger senden. Eine gewisse Sicherheit bieten hier bereits der *Offset* und die Übertragungslänge, da der Empfänger bemerken kann, wenn er verschiedene Pakete mit dem gleichen Offset empfängt. Dennoch könnten Prüfsummen hier helfen, die korrekte Übertragung zu identifizieren.

Abbildung 3.3: Schlüsselaustausch nach Diffie, Hellman und Merkle[9][5] zwischen Teilnehmern A und B

1. A legt die Primzahl p und eine Zahl $g < p$ fest und überträgt sie an B
2. A wählt eine geheime Zahl $a < p$ und berechnet $a' = g^a \mod p$
3. B wählt eine geheime Zahl $b < p$ und berechnet $b' = g^b \mod p$
4. A und B tauschen die Zahlen a' und b' aus
5. A berechnet $s = b'^a \mod p$
6. B berechnet $s = a'^b \mod p$
7. A und B haben den gemeinsamen geheimen Schlüssel s

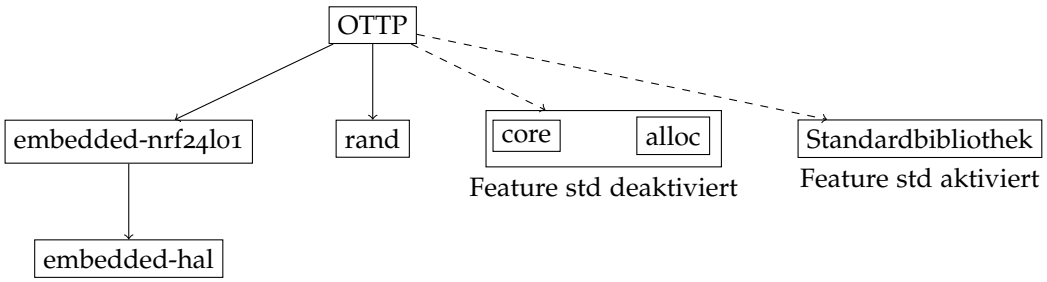
Neben der Zuverlässigkeit spielt auch Sicherheit eine wichtige Rolle. Dass unsere Sensornetzwerke auf Zügen eingesetzt werden und auf andere Funktechnologien als Bluetooth oder WLAN aufbauen, erhöht bereits ohne weitere Maßnahmen die Sicherheit, da somit spezielle Hardware und die Anwesenheit vor Ort für einen Angriff notwendig sind. Da wir auch Sensorknoten von Drittherstellern unterstützen, werden dennoch weitere Sicherheitsmaßnahmen benötigt. Da für den Prototypen im Rahmen dieses Projekts die grundlegende Funktionalität höhere Priorität hatte, sind diese aktuell noch nicht implementiert.

Ein Mechanismus dafür könnte der Schlüsselaustausch nach Diffie-Hellman-Merkle (Abbildung 3.3) sein. Damit können zwei Teilnehmer über einen unsicheren Kanal ein gemeinsames Geheimnis erzeugen, was wir beispielsweise zur Authentifizierung bei Verbindungen nutzen können. Beim Verbindungsaufbau erzeugen beide Seiten ihre eigenen Geheimzahlen sowie ein gemeinsames Geheimnis. Senden beide Seiten dann jeweils a' bzw. b' bei ihren Nachrichten mit, können sie somit sicher stellen, dass die Nachricht tatsächlich von dem anderen Teilnehmer kommt.

4 Implementierung für Mikrocontroller und Embedded-Linux-Systeme

In unserem Sensornetzwerk befinden sich Geräte verschiedener Leistungsklassen. Die meisten Sensorknoten basieren auf Mikrocontrollern, womit sie nur über Arbeitsspeicher in der Größenordnung weniger Megabytes verfügen. Andere Geräte, zum Beispiel *Edge Nodes*, nutzen Embedded-Linux-Systeme wie den Raspberry Pi 4.[5] Die Referenzimplementierung von *OTTP* ist in Rust geschrieben und stellt eine einzige Bibliothek zur Verfügung, die sowohl auf Mikrocontrollern als auch auf Computern mit Betriebssystem nutzbar ist. Um möglichst viele Plattformen unterstützen zu können, nutzt *OTTP* nur die in Abbildung 4.1 genannten Abhängigkeiten.

Abbildung 4.1: Abhängigkeiten der OTTP-Bibliothek. Gestrichelte Pfeile sind optionale Abhängigkeiten.



Um eine Anwendung, die *OTTP* nutzt, auf einer bestimmten Plattform ausführen zu können, gibt es drei Voraussetzungen.

Implementierung von *embedded-hal* für die Plattform Die von uns genutzte Bibliothek *embedded-nrf24l01*[6] abstrahiert durch die Nutzung von *embedded-hal*[7] von der konkreten Plattform und setzt eine Implementierung der dort definierten Schnittstellen (beispielsweise SPI) für die genutzte Hardwareplattform voraus.

Unterstützung der Standardbibliothek *oder* der Bibliotheken *core* und *alloc* Beispielsweise Mikrocontroller können nicht die komplette Rust-Standardbibliothek unterstützen. In diesem Fall kann *OTTP* mithilfe anderer Bibliotheken laufen, die jedoch unter anderem plattformspezifisch implementierte Speicherverwaltung benötigen. Genauer erläutern wir dies in Abschnitt 4.1

[5] Buy a Raspberry Pi 4 Model B – Raspberry Pi, `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/` (abgerufen am 29. Juni 2021)

[6] embedded-nrf24l01 - crates.io: Rust Package Registry, `https://crates.io/crates/embedded-nrf24l01` (abgerufen am 29. Juni 2021)

[7] embedded-hal - crates.io: Rust Package Registry, `https://crates.io/crates/embedded-hal` (abgerufen am 29. Juni 2021)

Implementierung plattformabhängiger *Traits* Einige Funktionalitäten können nicht von *embedded-hal* oder *core* abgedeckt werden. Wie diese durch *Traits* bereitgestellt werden können, wird in Abschnitt 4.2 erläutert.

Wir haben die Referenzimplementierung auf einem *Raspberry Pi Zero*[8] sowie auf einem *STM32F401RE*[9] getestet. Bei ersterem handelt es sich um einen ARM-basierten *Single Board Computer*, der verschiedene Linux-Distributionen unterstützt, zweiterer ist ein Mikrocontroller. In Quelltext 4.1 und Quelltext 4.2 ist die Initialisierung des Funkmoduls auf den verschiedenen Plattformen dargestellt. Dort lässt sich erkennen, dass das Funkmodul in beiden Fällen mit drei Variablen `ce`, `csn` und `spi` initialisiert wird. Mit den jeweiligen Implementierungen von *embedded-hal* (*rppal*[10] für den *Raspberry Pi* und *stm32f4xx-hal*[11] für den *STM32F401RE*) wird für die jeweilige Plattform eine Instanz des SPI-Interfaces und den zusätzlich benötigten Pins *Chip Enable* und *Chip Select* erzeugt. Wie sich sehen lässt, funktioniert dies auf beiden Plattformen unterschiedlich. Auf dem *Raspberry Pi* werden die Pins vergleichbar zu *Arduino* anhand ihrer Nummer initialisiert; für den *STM32F401RE* kann ein *Singleton* erzeugt werden, der alle GPIO-Pins als Variablen enthält. Dafür ist die Initialisierung des SPI-Interfaces auf dem Raspberry Pi einfacher, da die dafür genutzten Pins nicht vorher manuell initialisiert werden müssen.

Quelltext 4.1: Initialisierung des nRF24l01-Moduls auf dem Raspberry Pi

```
let gpio = Gpio::new().unwrap();
let spi = rppal::spi::Spi::new(
    rppal::spi::Bus::Spi0,
    rppal::spi::SlaveSelect::Ss0,
    8_000_000,
    rppal::spi::Mode::Mode0,
)
.unwrap();
let ce = gpio.get(25).unwrap().into_output();
let csn = gpio.get(8).unwrap().into_output();

let mut nrf = NRF24L01::new(ce, csn, spi).unwrap();
```

Quelltext 4.2: Initialisierung des nRF24l01-Moduls auf dem STM32F401RE

```
let device_peripherals = stm32f4xx_hal::stm32::Peripherals::take().unwrap();

let gpioa = device_peripherals.GPIOA.split();
let rcc = device_peripherals.RCC.constrain();
let clocks = rcc.cfgr.sysclk(48.mhz()).pclk1(8.mhz()).freeze();

let tx = gpioa.pa2.into_alternate_af7();
```

[8] Buy a Raspberry Pi Zero – Raspberry Pi, https://www.raspberrypi.org/products/raspberry-pi-zero/ (abgerufen am 29. Juni 2021)

[9] STM32F401RE - STM32 Dynamic Efficiency MCU, Arm Cortex-M4 core with DSP and FPU, up to 512 Kbytes of Flash memory, 84 MHz CPU, Art Accelerator - STMicroelectronics", https://www.st.com/en/microcontrollers-microprocessors/stm32f401re.html (abgerufen am 29. Juni 2021)

[10] rppal - crates.io: Rust Package Registry, https://crates.io/crates/rppal (abgerufen am 29. Juni 2021)

[11] stm32f4xx-hal - crates.io: Rust Package Registry, https://crates.io/crates/stm32f4xx-hal (abgerufen am 29. Juni 2021)

```
let rx = gpioa.pa3.into_alternate_af7();
let cfg = hal::serial::config::Config::default().baudrate(115200.bps());

let ce = gpioa.pa8.into_push_pull_output();
let csn = gpioa.pa4.into_push_pull_output();

let sck = gpioa.pa5.into_alternate_af5();
let miso = gpioa.pa6.into_alternate_af5();
let mosi = gpioa.pa7.into_alternate_af5();

let spi = Spi::spi1(
    device_peripherals.SPI1,
    (sck, miso, mosi),
    MODE_0,
    2.mhz().into(),
    clocks,
);

let mut nrf = NRF24L01::new(ce, csn, spi).unwrap();
```

4.1 Unterstützung der Standardbibliothek

Die Referenzimplementierung von *OTTP* unterstützt auch Systeme, auf denen die Rust-Standardbibliothek nicht zur Verfügung steht, da diese das Vorhandensein einer *libc* wie *glibc* oder *musl* voraussetzt. Da die Standardbibliothek im Gegensatz zu beispielsweise C standardmäßig in ein Rust-Programm eingebunden wird, muss das Attribut `no_std` für die Bibliothek gesetzt werden. Ein großer Teil der Standardbibliothek basiert auf der Bibliothek *core*, die nur in Rust ohne externe Abhängigkeiten geschrieben ist und somit auf jeder von Rust unterstützten Plattformen genutzt werden kann. Darin fehlen vor allem Typen und Funktionen, die das Vorhandensein eines klassischen Betriebssystems (zum Beispiel Multithreading und Synchronisationsprimitive) oder Speicherverwaltung voraussetzen. Insbesondere letzteres stellt eine große Einschränkung ein, da hiermit Datentypen wie Vektoren und Maps fehlen. Dafür gibt es jedoch die Bibliothek *alloc*, die nach der Implementierung eines eigenen `Allocator`, der Heap-Allozierungen für die genutzte Plattform durchführt, einen Großteil der Datentypen aus dem `collections`-Modul der Rust-Standardbibliothek bereitstellt.

Quelltext 4.3: Unterschiedliche Imports je nach aktivierten Features

```
#[cfg(feature = "std")]
use std::vec::Vec;
#[cfg(not(feature = "std"))]
use alloc::vec::Vec;

let v: Vec<u32> = Vec::new();
```

Der Rust-Compiler unterstützt *Conditional Compilation* in Form von *Features* (siehe Quelltext 4.3). Dies ermöglicht es uns, mit der Standardbibliothek zu arbeiten, wenn diese verfügbar ist und sonst die Implementierungen aus *core* und *alloc* zu verwenden.

4.2 Plattformabhängiger Code in *Traits*

In einigen Fällen sind wir auf plattformspezifische Implementierungen angewiesen, die nicht durch *core* abgedeckt werden können. Ein Beispiel hierfür sind Timer. An einigen Stellen wird die Ausführung eines Ablaufs kurzzeitig pausiert oder für Zeitüberschreitungen muss die aktuelle Systemzeit bekannt sein. Wie diese Operationen implementiert werden, ist jedoch stark vom verwendeten System abhängig. Auf einem Embedded-Linux-System kann der aktuelle Thread pausiert werden, auf einem Mikrocontroller muss hingegen über das plattformspezifische HAL die Ausführung pausiert werden. An diesen Stellen setzen wir *Trait*-Objekte ein. Mit *Traits* können wir festlegen, welche Methoden eine Struktur besitzen muss, geben aber keine Implementierung vor. Ein Beispiel dafür ist in Quelltext 4.4 zu sehen, wo der *Timer-Trait* aufgeführt ist und einmal mit Funktionen aus der Standardbibliothek und dann spezifisch für die *STM32F401RE*-Plattform implementiert ist.

Quelltext 4.4: Beispiel für einen plattformabhängig implementierten Trait

```
trait Timer {
    fn now(&self) -> u64;
    fn delay(&self, duration: Duration);
}

struct StdTimer;

impl Timer for StdTimer {
    fn now(&self) -> u64 {
        std::time::SystemTime::now()
            .duration_since(std::time::UNIX_EPOCH)
            .unwrap().as_secs()
    }

    fn delay(&self, duration: Duration) {
        std::thread::sleep(duration)
    }
}

struct Stm32Timer {
    rtc: stm32f4xx_hal::stm32::RTC,
    delay: stm32f4xx_hal::delay::Delay,
}

impl Timer for Stm32Timer {
    fn now(&self) -> u64 {
        let tr = self.rtc.tr.read().bits();
        let dr = self.rtc.dr.read().bits();
        ((dr as u64) << 32) + tr as u64
    }

    fn delay(&self, duration: core::time::Duration) {
        self.delay.delay_ms(duration.as_millis() as u32)
    }
}
```

Wie der Vergleich zeigt, unterscheiden sich die Implementierungen stark. Mit der Standardbibliothek können alle Funktionen durch existierende Typen, Konstanten und Methoden abgebildet werden. Auf dem Mikrocontroller muss der Timer hingegen auf Hardware-Komponenten zugreifen und im Fall der Echtzeituhr beispielsweise ihre Register auslesen. Dennoch besitzen beide nach außen die gleiche Schnittstelle, sodass sie für *OTTP* genutzt werden können.

4.3 Verwaltung des Funkmoduls

Es gibt bereits einen Rust-Treiber für die von uns verwendeten *nRF24L01*-Funkmodule, der auf der Bibliothek *embedded-hal* aufsetzt. Durch die Verwendung von *embedded-hal* können wir damit auf allen von *OTTP* unterstützten Plattformen arbeiten. Die konkreten Zustände des Funkmoduls werden durch drei Datentypen abgebildet: `StandbyMode` bei der Initialisierung, `RxMode` zum Empfangen und `TxMode` zum Senden. Da unsere Abstraktionen Zugriff auf das Funkmodul benötigen, ist ein Handle darauf als Variable in den entsprechenden Strukturen abgelegt. Dies führt jedoch zu Schwierigkeiten mit Rusts *Ownership*-Konzept. Wenn wir den Modus des Funkmoduls wechseln, wird ein neuer Wert aus dem gespeichertern Modus erzeugt und die Variable, in der das Funkmodul gespeichert war, wird ungültig. Da dies nicht passieren darf, können wir das Funkmodul nicht einfach als Variable in einer Struktur speichern. Stattdessen legen wir dafür eine `Option` an. Hierbei handelt es sich um einen eingebauten Datentypen von Rust, der als `Option<T>` die möglichen Werte `Some(T)` und `None` annehmen kann. Wir können also einen Wert des generischen Typen `T` in der `Option` speichern und zu einem beliebigen Zeitpunkt entnehmen, ohne dass die `Option` ungültig wird. Methoden, die das Funkmodul benutzen, bewegen es also in eine lokale Variable der Methode und schreiben es am Ende zurück in die Variable der Struktur, zu der sie gehören. Dies verpflichtet Methoden zur vom Compiler nicht prüfbaren Invariante, die Variable des Funkmoduls nach der Benutzung und idealerweise auch nach dem Auftreten eines Fehlers nicht leer zu lassen. Da dies nur innerhalb des Codes von *OTTP* passiert, kann es keine Probleme für Nutzer:innen des Protokolls verursachen, solange unser Code die Invariante aufrecht erhält.

4.4 Netzwerkpakete

Den Aufbau eines Pakets haben wir bereits in Abschnitt 3.3 beschrieben. In der Referenzimplementierung sind sie grundsätzlich als Byte-Array repräsentiert, die sich jedoch in einem Wrapper befinden, der dem Rust-Konzept von *Smart Pointers* folgt. Ein *Smart Pointer* ist eine Struktur, die einen Speicherbereich besitzt und besondere Zugriffe darauf ermöglicht oder Garantien darüber gibt. Für Pakete ist hier der Zugriff auf einzelne Felder besonders interessant. Einige Felder speichern beispielsweise 16-Bit-Zahlen in zwei benachbarten Bytes. Würde man direkt auf das darunterliegende Array zugreifen, könnten Werte falsch ausgelesen werden. Die Zugriffsfunktionen des Typen `Packet` garantieren jedoch, dass die Werte an den korrekten Stellen gelesen und richtig zusammengesetzt werden.

Trotzdem muss an einigen Stellen, beispielsweise in der Funktion zum Senden, auf das eigentliche Array zugegriffen werden. Hierzu implementiert `Packet` den *Trait* `Deref`. Dieser *Trait* wird genutzt, um einen *Smart Pointer* wie eine Referenz auf seinen Inhalt zu behandeln. Wird ein `Packet` dereferenziert, kann anderer Code also auf das interne Array zugreifen. Da `Deref` nur unveränderlichen Zugriff erlaubt (es gibt für veränderliche Dereferenzierung auch `DerefMut`), kann damit das Paket aber nicht ungültig gemacht werden.

4.5 Aussagekräftige Typen

Das starke und flexible Typsystem von Rust unterstützt uns darin, die Bedeutung von Datentypen in ihrem Aufbau zu kodieren. Dies zeigt sich beispielsweise in der Fehlerverarbeitung. In Rust gibt es im Gegensatz zu anderen Sprachen keine *Exceptions*. Stattdessen haben Funktionen, in denen ein Fehler auftreten kann, ein `Result<T, E>` als Rückgabewert, welches als Aufzählung mit den Varianten `Ok(T)` und `Err(E)` implementiert ist. Somit muss der Code, der die Funktion aufgerufen hat, einen möglichen Fehler behandeln, zum Beispiel, indem er selbst wieder ein `Result` zurück gibt. Wir haben für Fehler in Funktionen von OTTP den Typen `OttpError`, der als Aufzählung implementiert ist (siehe Quelltext 4.5).

Quelltext 4.5: Aufbau des OttpError-Typen

```
pub enum OttpError {
    GenericError,
    CannotRead,
    ConfigError,
    DeviceUnavailable,
    SendFail,
    IncompleteTransmission,
    ConnectionRefused,
    NotConnected,
    ConnectionClosed,
    RouterAbort,
    Timeout,
}
```

Tritt ein Fehler im Protokoll auf, gibt es also eine semantische Information darüber, was passiert ist. Es wäre auch möglich, den genauen Fehler beispielsweise als String zu speichern, die Implementierung als Aufzählung hat allerdings den Vorteil, dass das bereits erwähnte *Pattern-Matching* darauf funktioniert und die Information über den Fehler direkt im Datentypen kodiert ist.

Ein ähnliches Entwurfsmuster nutzen wir für die Pakettypen. Diese sind auch als Aufzählung definiert, haben jedoch noch weitere Funktionen.

Quelltext 4.6: Implementierung von MessageKind mit Konvertierung von und zu Bytes

```
#[derive(PartialEq, Eq, Clone, Copy)]
pub enum MessageKind {
```

```
    Data,
    RequestAddress,
    PublishAddress,
    AddressReceived,
    Approved,
    Abort,
    Connect,
    Disconnect,
    AcceptConnection,
    RejectConnection,
    Invalid,
}

impl From<u8> for MessageKind {
    fn from(val: u8) -> Self {
        match val {
            0 => MessageKind::Data,
            1 => MessageKind::RequestAddress,
            2 => MessageKind::PublishAddress,
            3 => MessageKind::AddressReceived,
            4 => MessageKind::Approved,
            5 => MessageKind::Abort,
            6 => MessageKind::Connect,
            7 => MessageKind::Disconnect,
            8 => MessageKind::AcceptConnection,
            9 => MessageKind::RejectConnection,
            _ => MessageKind::Invalid,
        }
    }
}

impl From<MessageKind> for u8 {
    fn from(val: MessageKind) -> Self {
        val as u8
    }
}
```

Um den Pakettyp in einem Paket als Byte speichern zu können, implementieren wir für 8-Bit-Integer den Trait `From<MessageKind>` (siehe Quelltext 4.6). Da Aufzählungen in Rust als Integer repräsentiert werden, wenn sie nicht wie `Result` oder `Option` Werte in den Varianten speichern, können wir hier einen primitiven *Cast* nutzen. Umgekehrt können wir Bytes zu `MessageKind` konvertieren, wofür wir *Pattern-Matching* nutzen. Da es nicht für alle möglichen Bytes einen Pakettypen gibt, geben wir im Standardfall den Pakettypen `Invalid` aus. Neben diesen manuell implementierten *Traits* lassen wir auch einige automatisch implementieren. Besonders wichtig sind hier `PartialEq` und `Eq`. Diese erlauben es uns, den Gleichheitsoperator auf `MessageKind` zu benutzen (siehe Quelltext 4.7). Da `MessageKind` eine einfache Aufzählung ist, kann der Rust-Compiler die Implementierung automatisch generieren.

Quelltext 4.7: Beispielhafte Verwendung des Pakettypen

```
let data = Packet::from(payload.as_ref());
```

```
if data.message_kind() == MessageKind::PublishAddress {
    break data;
}
```

Es wäre auch möglich gewesen, den Pakettypen einfach als Byte darzustellen, unsere Implementierung bietet jedoch vor allem für die Lesbarkeit Vorteile. Da nur die Werte der Aufzählung valide Pakettypen sind, ist der Code in Quelltext 4.7 auf jeden Fall sicher. Würde die Methode `Packet.message_kind()` hier eine Zahl zurück geben, könnte man diese beispielsweise versehentlich mit einem vermeintlichen Pakettypen vergleichen, der gar nicht existiert.

4.6 Streams und Listeners

Die Schnittstelle, über die Programmierer:innen meist mit OTTP interagieren werden, sind `OttpStream` und `OttpListener`. Diese orientieren sich grob an dem Design von `TcpStream` und `TcpListener` aus der Rust-Standardbibliothek.[12]

Quelltext 4.8: Beispielhafte Verwendung eines OttpStream

```
let mut nrf = NRF24L01::new(ce, csn, spi).unwrap(); // Initialisierung gekürzt

let mut stream = OttpStream::new(nrf, 21, Box::new(StdTimer))
                            .expect("Configuration failed");
if stream.connect(42).is_ok() {
    stream.write("Hello, world!");
}
```

Bei der Erstellung eines `OttpStream` müssen ihm ein Handle auf ein *nRF24L01*-Gerät, eine initiale Adresse und ein Timer übergeben werden. Danach kann mit der Methode `connect` die Verbindung zu einem Gerät, auf dem ein `OttpListener` läuft, hergestellt werden. Der `OttpStream` unterstützt die *Traits* `Read` und `Write` aus der Standardbibliothek, die beispielsweise auch zur Interaktion mit Dateien oder *Unix-Sockets* genutzt werden. Läuft *OTTP* auf einem System ohne Standardbibliothek, werden stattdessen `Read` und `Write` aus einer eigenen Nachimplementierung genutzt. Es könnte auch die Bibliothek `core_io`[13] genutzt werden, die alle kompatiblen Strukturen, *Traits* und Funktionen aus dem `io`-Modul der Standardbibliothek für `no_std`-Projekte anbietet. Allerdings funktioniert diese nur mit bestimmten Versionen des Rust-Compilers, weshalb wir sie nicht standardmäßig einsetzen. Ist der `OttpStream` verbunden, kann mit den Methoden `read` und `write` aus den genannten *Traits* aus dem *Stream* gelesen oder in ihn geschrieben werden. Die darunterliegenden Mechanismen wie Fragmentierung werden dadurch komplett vor Nutzer:innen der Bibliothek versteckt.

Quelltext 4.9: Beispielhafte Verwendung eines OttpListener

```
let mut nrf = NRF24L01::new(ce, csn, spi).unwrap(); // Initialisierung gekürzt
```

[12] std::net - Rust, `https://doc.rust-lang.org/stable/std/net/index.html` (abgerufen am 29. Juni 2021)
[13] core_io - crates.io: Rust Package Registry, `https://crates.io/crates/core-io` (abgerufen am 29. Juni 2021)

```
let listener = OttpListener::new(device, 42, Box::new(StdTimer));
let stream = listener.listen()
                .expect("Error during connection process");
let mut buf = [0; 50];
stream.read(&mut buf).unwrap();
```

Das Gegenstück zum `OttpStream` bildet der `OttpListener`. Nach seiner Initialisierung wartet er auf eine Verbindungsanfrage. Wurde eine Verbindung erfolgreich hergestellt, gibt er selbst einen `OttpStream` zurück, der mit dem anderen Gerät verbunden ist. Die Verbindung ist dabei bidirektional, es können also beide Teilnehmer lesen und schreiben.

Als Erweiterung des `OttpListener` gibt es noch den `OttpRouter`. Dieser kann bis auf zwei Unterschiede genau wie ein `OttpListener` genutzt werden. Zum einen nutzt er immer die Adresse $0xFF$, da diese wie in Kapitel 3 beschrieben für Router reserviert ist. Zum anderen hört er beim Aufruf der Methode `listen` auch auf Adressanfragen. Wir haben uns für diese Variante entschieden, da sie im gegebenen zeitlichen Rahmen am schnellsten die Implementierung eines Routers ermöglichte. Eine andere Option wäre es, den Router als Option im `OttpListener` bereitzustellen. In diesem Fall würde bei der Initialisierung ein Argument übergeben, ob der `OttpListener` auch Adressen vergeben soll. Dann könnte das Funkmodul so konfiguriert werden, dass es auf der Adresse $0xFF$ Nachrichten empfängt und der `OttpListener` auf Adressanfragen reagieren. Allerdings wird ein Großteil aller Geräte, die *OTTP* nutzen, nicht als Router agieren, sodass für Programmierer:innen die aktuelle Aufteilung in `OttpListener` und `OttpRouter` kürzeren und einfacher verständlichen Code ergeben sollte.

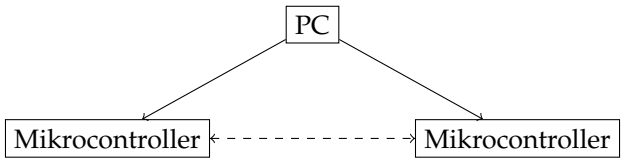
5 Messungen

Neben der Spezifikation und der Beschreibung unserer Referenzimplementierung werden wir *OTTP* nun durch einige Messungen charakterisieren.

5.1 Experimenteller Aufbau

Um die Messungen reproduzierbar durchzuführen, benötigten wir einen experimentellen Aufbau. Wir entschieden uns, hierfür auf die *STM32F401RE*-Mikrocontroller und *Raspberry Pis* zurückzugreifen, die wir bereits eingesetzt hatten. Beide dieser Plattformen bieten eigene Vor- und Nachteile. Auf einem *Raspberry Pi* läuft ein vollständiges Linux-Betriebssystem. Dieses verwaltet Faktoren wie das Scheduling unserer Anwendung und Zugriff auf die GPIO-Pins. Es ist also möglich, dass der *Raspberry Pi* durch die Arbeit des Betriebssystems unsere Messungen verfälscht. Im Gegensatz dazu haben wir die komplette Kontrolle über die Hardware eines Mikrocontrollers. Dafür können wir auf einem *Raspberry Pi* die Rust-Standardbibliothek einsetzen und haben deutlich mehr Arbeitsspeicher zur Verfügung.

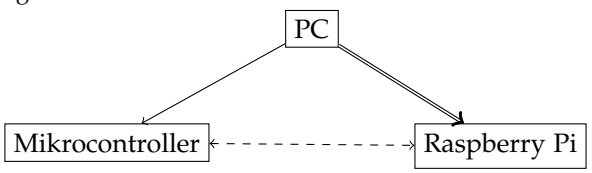
Abbildung 5.1: Aufbau mit zwei Mikrocontroller für Messungen. Durchgezogene Pfeile sind serielle Verbindungen, gestrichelte sind nRF24-Verbindungen.



Für unsere Messungen sind präzise Zeitstempel wichtig. Dies ist auf den Mikrocontrollern ein Problem, da diese zwar über eine Echtzeituhr verfügen, diese aber nicht kalibriert ist. Der naive Ansatz, auf dem sendenden Mikrocontroller den Zeitpunkt des Sendens festzuhalten und auf dem anderen den Zeitpunkt des Empfangs, wird aufgrund der fehlenden Synchronisierung also nicht funktionieren. Um dies zu umgehen, könnten wir beide Zeitmessungen auf einem Mikrocontroller durchführen. Wir würden also nicht den Empfangszeitpunkt, sondern den Zeitpunkt der Antwort festhalten. Da wir für das Auslesen von Ausgaben der Mikrocontroller sowieso die serielle Verbindung zu einem Computer benötigen, haben wir uns stattdessen entschieden, die Zeitmessungen an diesem durchzuführen, was zu dem in Abbildung 5.1 dargestellten Aufbau führt. Wir halten den Zeitstempel auf dem Computer fest, sobald der Sender über die serielle Verbindung mitteilt, dass die Übertragung gesendet wurde und nehmen einen weiteren Zeitstempel auf, wenn der Empfänger den vollständigen Empfang der Übertragung mitteilt. Die

geringe Verzögerung, die durch die serielle Verbindung entstehen könnte, halten wir für vernachlässigbar, da beide Mikrocontroller baugleich sind und deshalb die gleiche Verzögerung haben sollten.

Abbildung 5.2: Aufbau mit einem Raspberry Pi und einem Mikrocontroller für Messungen. Durchgezogene Pfeile sind serielle Verbindungen, gestrichelte sind nRF24-Verbindungen und doppelte sind SSH-Verbindungen über WLAN.



Für die Messung der Übertragungsdauer ergab sich das Problem, dass die *STM32F401RE*-Mikrocontroller für Übertragungen ab 100 Byte nicht über ausreichend Arbeitsspeicher verfügen. Aus diesem Grund führten wir einen zweiten Aufbau (Abbildung 5.2) ein, der auf der Empfängerseite einen *Raspberry Pi* einsetzt. Den Mikrocontroller können wir weiterhin über eine serielle Schnittstelle auslesen, mit dem *Raspberry Pi* kommunizieren wir mittels SSH. Ein Vorteil des *Raspberry Pi* besteht darin, dass er eine Uhr besitzt, die über das Netzwerk gestellt werden kann. Somit können wir die Zeitmessung des Mikrocontrollers auf dem Computer vornehmen und die des *Raspberry Pi* direkt auf diesem. Durch die synchronisierten Uhren wird die Messung höchstens geringfügig durch das Verwenden verschiedener Geräte verfälscht.

5.2 Adressaushandlung

Als erste Messung betrachten wir, wie lange das Aushandeln einer Adresse benötigt. Hierzu nutzen wir die maximale mögliche Sendegeschwindigkeit der *nRF24L01*-Module, 2Mbps. Die in Tabelle 5.1 gelisteten Daten stammen aus zehn Messungen.

Tabelle 5.1: Dauer der Adressaushandlung in Sekunden

Minimum	Median	Durchschnitt	Varianz	Maximum
0.8431	1.0913	1.1205	0.0201	1.2893

Hier ist bereits ein Muster zu erkennen, das in Abschnitt 5.4 noch deutlicher wird. Wie bereits erläutert, findet die Adressaushandlung in einem Zwei-Phasen-Commit statt. Dadurch müssen beide Teilnehmer Pakete des anderen abwarten. Durch die Funktionsweise der *nRF24L01*-Module muss der Empfänger jeweils etwa 100ms warten, bevor er wieder prüfen kann, ob ein Paket vorliegt. Ob die Pakete den Empfänger zu günstigen Zeitpunkten erreichen, beeinflusst die Länge der Adressaushandlung insgesamt maßgeblich, insbesondere, da in diesem Prozess nur vier Pakete versandt werden.

5.3 Verbindungsaufbau

Als nächstes messen wir, wie lange es dauert, eine Verbindung zwischen zwei Geräten herzustellen. Wir messen wieder mit 2Mbps Übertragungsgeschwindigkeit. Die gelisteten Daten stammen aus zehn Messungen.

Tabelle 5.2: Dauer des Verbindungsaufbaus in Sekunden

Minimum	Median	Durchschnitt	Varianz	Maximum
0.5713	0.7740	0.7927	0.0180	1.0054

Wie Tabelle 5.2 zeigt, dauert der Verbindungsaufbau etwa zwischen einer halben und einer ganzen Sekunde, wobei die meisten unserer zehn gesamten Messwerte in der Nähe des Medians, also zwischen 0.7s und 0.8s liegen. Der Sender der Verbindungsanfrage muss ein Paket mit der Bestätigung oder Ablehnung der Verbindung seitens des Empfängers abwarten, was die Varianz erklärt, die hier zwar geringer als in Abschnitt 5.2, aber höher als für ein einziges Datenpaket in Abschnitt 5.4 ist.

5.4 Senden einer Übertragung

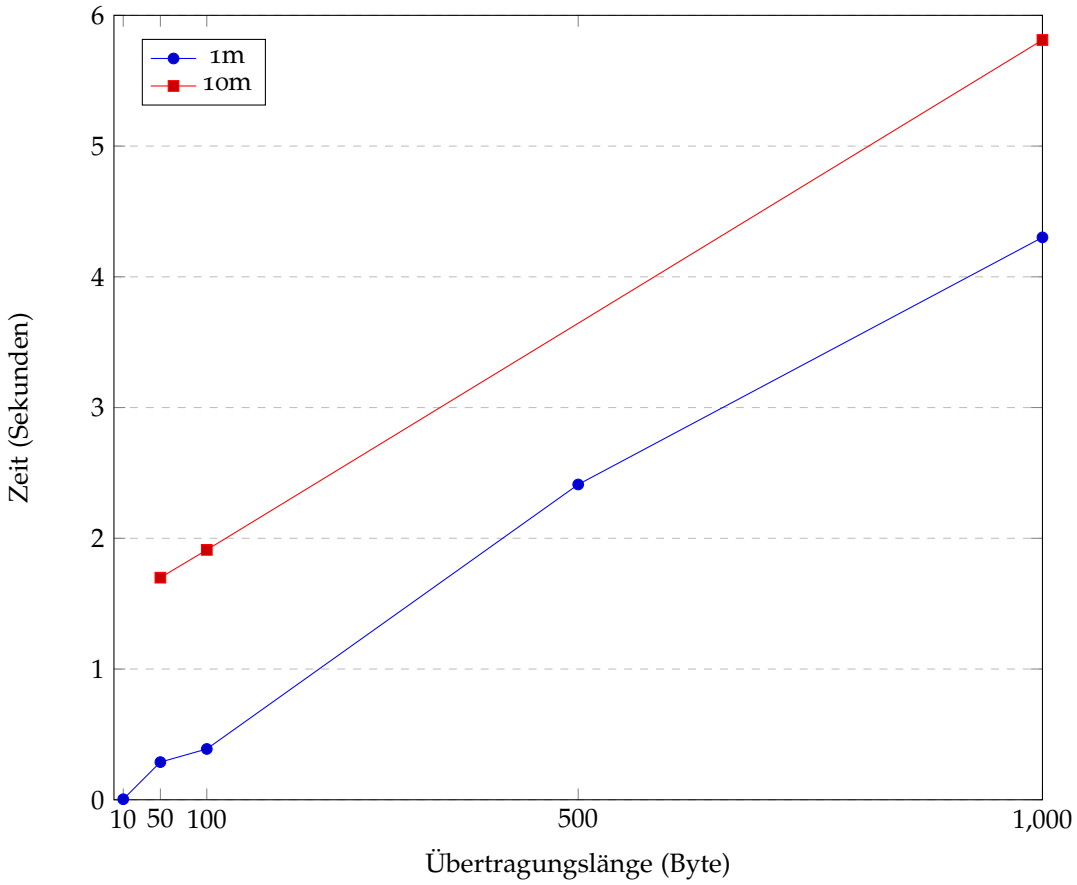
Im praktischen Einsatz ist die Dauer einer Übertragung am häufigsten von Interesse, da sie im Gegensatz zur Adressaushandlung und dem Verbindungsaufbau eine dynamische Länge besitzt. Für diese Messung mussten wir aufgrund des beschränkten Arbeitsspeichers unserer Mikrocontroller auf einen *Raspberry Pi* als Empfänger zurückgreifen.

Tabelle 5.3: Dauer einer Übertragung in Sekunden

Übertragungslänge (Byte)	Minimum	Median	Durchschnitt	Varianz	Maximum
10	0.0040	0.0041	0.0044	0.0000	0.0049
50	0.2149	0.2880	0.2681	0.0013	0.2968
100	0.3586	0.3886	0.4596	0.0279	0.7602
500	2.3654	2.4115	2.3964	0.0007	2.4138
1000	4.2751	4.3017	4.2949	0.0002	4.3037

Wir haben jeweils fünf Messungen mit unterschiedlichen Datenmengen von 10 bis 1000 Byte durchgeführt. Die Ergebisse sind in Tabelle 5.3 festgehalten. Wie Abbildung 5.3 zeigt, steigt die Übertragungsdauer nach 50 Byte annähernd linear. Der große Unterschied zwischen 10 und 50 Byte lässt sich dadurch erklären, dass ersteres nur ein Paket benötigt. Es ist ebenfalls auffällig, dass die Varianz der gemessenen Zeiten bei 100 Byte am höchsten ist. Die notwendige Wartezeit vor dem Empfang jedes Pakets kann die Zeit für die gesamte Übertragung bei einem ungünstigen zeitlichen Versatz zwischen den beiden Geräten signifikant erhöhen. Da für 50 Byte nur zwei Pakete versandt werden, kann der zeitliche Versatz hier besonders stark die Gesamtdauer beeinflussen. Mit mehr Paketen werden mehr davon sowohl zu günstigen als auch zu ungünstigen Zeitpunkten empfangen, sodass die Varianz insgesamt sinkt.

Abbildung 5.3: Vergleich der Übertragungsdauer im Median auf 1m und 10m Abstand



Die Messungen in Tabelle 5.3 haben wir auf einer Distanz von unter einem Meter zwischen Sender und Empfänger durchgeführt. Wie bereits erläutert, besitzen die Funkmodule allerdings eine Reichweite von etwa 50m. Um einen besseren Eindruck zu schaffen, welche Übertragungsgeschwindigkeiten beim Einsatz im Zug zu erwarten sind, haben wir die Datenübertragung auch bei einer Reichweite von 10m getestet.

Tabelle 5.4: Dauer einer Übertragung in Sekunden bei einer Distanz von ca. 10m

Übertragungslänge (Byte)	Minimum	Median	Durchschnitt	Varianz	Maximum
50	1.6706	1.6989	1.6874	0.0003	1.7021
100	1.8832	1.9114	1.9006	0.0003	1.9163
1000	5.7825	5.8114	5.7994	0.0003	5.8129

Wie zu erwarten war, erhöht sich die benötigte Zeit für Übertragungen, allerdings nur um einen konstanten Betrag von etwa 1.4s. Es ist also davon auszugehen, dass der Einfluss einer größeren Distanz insbesondere bei größeren Datenmengen vernachlässigbar wird. Auffällig ist auch die extrem geringe Varianz bei allen Übertragungslängen. Für 1000 Byte ist die Varianz bereits auf kurze Distanz sehr gering, aber für die kleineren Übertragungen sinkt sie hier noch einmal merklich. Wir vermuten, dass dies durch den konstanten Anteil, um den die Übertragungsdauer sich verlängert, zu erklären ist, da dieser für Übertragungslängen bis 100 Byte deutlich größer als die in Tabelle 5.3 gemessene Zeit ist.

6 Schlussbetrachtung

Abschließend betrachten wir, wie *OTTP* noch erweitert werden könnte und evaluieren den aktuellen Stand des Protokolls.

6.1 Mögliche Erweiterungen

OTTP ist in seinem aktuellen Zustand bereits grundsätzlich nutzbar. Es gibt dennoch Funktionen, die aufgrund von zeitlichen oder technischen Einschränkungen aktuell nicht implementiert sind.

Eine davon ist die Erneuerung von Adressen. Diese hat das Ziel, Sicherheit und Zuverlässigkeit zu verbessern. Wenn beispielsweise ein Gerät die Stromzufuhr verliert und neu gestartet werden muss, erhält es eine neue Adresse. Seine alte Adresse ist dann unbesetzt, aber der Router kann dies nicht wissen. Somit können über die Zeit immer mehr Adressen blockiert werden. Wenn Adressen nur eine begrenzte Gültigkeit hätten, könnte der Router Adressen, die nicht erneuert wurden, neu vergeben und dieses Problem vermeiden. Für die praktische Umsetzung müsste diese Idee noch deutlich ausgearbeitet werden. Beispielsweise fehlt noch ein Mechanismus, damit andere Geräte beim Router die Gültigkeit einer Adresse prüfen können und somit Verbindungen von abgelaufenen Adressen ablehnen können. Außerdem könnten Geräte bei der Adressaushandlung einen individuellen Wert, der auch nach einem Neustart gleich bleibt, mitsenden, sodass der Router gleiche Adressen erneut vergeben kann.

Eine andere Erweiterung wäre der Umbau von *OTTP* auf eine asynchrone Benutzerschnittstelle. Der Vorteil hiervon wäre, dass Netzwerkkommunikation nicht immer blockieren müsste. Beispielsweise blockiert aktuell das Senden einer Übertragung immer den aktuellen Ausführungsstrang, auch wenn es theoretisch im Hintergrund ablaufen könnte. Die größte Hürde ist hier die Bibliothek *embedded-nrf24l01*, die wir für die Interaktion mit dem Funkmodul nutzen. Da diese nicht asynchron arbeitet, können wir keine echte asynchrone Schnittstelle darauf aufbauen.

Eine nicht funktionale Erweiterung stellt die Umsetzung von ausführlicheren Tests dar. Aktuell testen wir in *OTTP*, ob das Kodieren und Dekodieren von Paketen korrekt funktioniert. Die tatsächliche Netzwerkkommunikation zu testen stellt eine Herausforderung dar, weil dies echte Hardware benötigt. Möglicherweise lässt sich der in Abschnitt 5.1 beschriebene Aufbau hierfür adaptieren. Dafür müssten wir abwägen, welche Eigenschaften des Protokolls wir auf echter Hardware testen wollen. Beispielsweise könnte ein einfacher Test daraus bestehen, eine zufällig generierte Übertragung zu versenden und deren

korrekten Erhalt zu prüfen. Es sind jedoch auch komplexere Tests möglich, die beispielsweise durch eine elektronisch kontrollierte Steckdose Stromausfälle simulieren. Aktuell verwalten wir das Projekt mittels *GitLab*.[14] Reine Software-Tests lassen sich dort mittels *Continuous Integration* automatisieren; mit einer eigenen, dort registrierten, Testmaschine mit einem entsprechen Testaufbau wären aber auch Hardware-Tests denkbar. Ohne einen Testaufbau mit echter Hardware wären noch Tests mit Mock-Objekten, also einer simulierten Netzwerkverbindung, möglich. Der Aufwand, diese zu implementieren, wäre jedoch angesichts der Möglichkeit, zumindest lokal mit echten Geräten zu testen, im Rahmen dieses Projekts unverhältnismäßig gewesen.

6.2 Evaluierung des entstandenen Protokolls

Auch wenn noch nicht alle möglichen Funktionen von *OTTP* implementiert sind, steht am Ende dieses Projekts ein funktionierendes Protokoll mit einer Referenzimplementierung, die eine vergleichbare Nutzerschnittstelle zu TCP in der Rust-Standardbibliothek bietet. Insbesondere stellt sich die Frage, ob die in Abschnitt 2.3 aufgestellten Bedingungen an Robustheit erfüllt wurden.

Zuverlässigen Datenaustausch erreichen wir durch die genutzten *nRF24L01*-Funkmodule sowie den Fragmentierungsmechanismus. *Enhanced Shockburst* sorgt dafür, dass die einzelnen Netzwerkpakete korrekt den Empfänger erreichen und die Fragmentierung garantiert auf Empfängerseite, dass Übertragungen komplett empfangen werden. Hier wäre es jedoch denkbar, mehr Sicherheitsmaßnahmen in *OTTP* selbst einzubauen, um in dieser Hinsicht hardwareunabhängig zu sein. Auch kaskadierende Fehler vermeiden wir. Da jeder Sensorknoten direkt mit einer *Edge Node* verbunden ist, kann sein Ausfall nur ihn selbst und keine anderen Sensorknoten betreffen. Somit ist noch die Behebung von Fehlerzuständen als Kriterium offen. Es wäre beispielweise möglich, dass eine *Edge Node* nach dem Ausfall eines Sensorknotens in einer Verbindung bleibt, die nie geschlossen wird. Dieses Problem lösen wir durch Zeitüberschreitungen. Fällt die *Edge Node* selbst aus, hilft unsere Verteilung von *Edge Nodes* auf dem Zug. Da wir 30m als Abstand zwischen *Edge Nodes* annehmen, kann jeder Sensorknoten sich mit zwei *Edge Nodes* verbinden. Fällt eine aus, besteht also immer noch ein Maß an Redundanz.

Es lässt sich also abschließend festhalten, dass die aktuelle Implementierung von *OTTP* die Anforderungen an Robustheit erfüllt, auch wenn diese durch Erweiterungen noch besser garantiert werden können.

6.3 Evaluierung der Arbeit mit Rust

Wie in Abschnitt 2.4 erläutert, haben wir für die Implementierung dieses Protokolls Rust als Programmiersprache gewählt. Da Rust im Embedded-Bereich noch sehr neu ist, wollen wir kurz evaluieren, wie gut die Arbeit damit funktioniert hat.

[14] Iterate faster, innovate together | GitLab, `http://gitlab.com` (abgerufen am 29. Juni 2021)

Das Embedded-Rust-Ökosystem ist für sein verhältnismäßig junges Alter (Rust ist seit 2015 stabil[1], *Arduino* gibt es seit 2005[8]) bereits sehr umfangreich. Bibliotheken, um den *Raspberry Pi*, *STM32F401RE* und *nRF24L01* zu verwenden, existierten bereits und auch andere beliebte Hardware-Plattformen wie *ESP32*[15] werden unterstützt. Es fällt allerdings auf, dass die von Rust bereitgestellten Werkzeuge noch nicht komplett für die Programmierung von Mikrocontrollern geeignet sind. Mit `rustc` und `cargo` können zwar Programme für diese kompiliert werden, für den *STM32F401RE* war aber trotzdem noch das externe Werkzeug `objcopy` (aus dem Paket `binutils`[16] für die genutzte ARM-Architektur) notwendig, um ein ladbares Abbild zu erzeugen.

Die Arbeit mit Hardware fällt durch die vereinheitlichten Schnittstellen in *embedded-hal* einfach und ermöglicht die Unterstützung verschiedener Plattformen mit dem gleichen Code. Für unseren Entwicklungsprozess war es sehr hilfreich, die gleiche Bibliothek nur mit kleinen Änderungen sowohl auf Mikrocontrollern als auch auf dem Raspberry Pi nutzen zu können. Ein Nachteil ist jedoch, dass sich die Implementierungen von *embedded-hal* teils stark in ihrem Design unterscheiden. Beispielsweise initialisieren wir auf dem *Raspberry Pi* GPIO-Pins über ihre Nummer, was immer noch Fehler wie die Initialisierung einer Schnittstelle, die nur bestimmte Pins nutzt, mit den falschen Pins ermöglicht.

Abschließend lässt sich festhalten, dass Rust viel Potenzial für die Entwicklung eingebetteter Systeme bietet und bereits jetzt sehr gut nutzbar ist. Für sehr viele Mikrocontroller und Sensoren gibt es Bibliotheken, die dank *embedded-hal* zu großen Teilen interoperabel sind. Die grundlegenden Konzepte von Rust wie *Ownership* machen es auch auf eingebetteten Plattformen einfach, die Verwaltung von Ressourcen wie dem Zugriff auf das Funkmodul nachzuvollziehen. Die Werkzeuge für die eingebettete Entwicklung sind an einigen Stellen noch nicht ausgereift, aber hier macht die Arbeitsgruppe für eingebettetes Rust[17] große Fortschritte.

[15] ESP32 Wi-Fi & Bluetooth MCU I Espressif Systems, `https://www.espressif.com/en/products/socs/esp32` (abgerufen am 29. Juni 2021)

[16] Binutils - GNU Project - Free Software Foundation, `https://www.gnu.org/software/binutils/` (abgerufen am 29. Juni 2021)

[17] GitHub - rust-embedded/wg: Coordination repository of the embedded devices Working Group, `https://github.com/rust-embedded/wg` (abgerufen am 29. Juni 2021)

Literaturverzeichnis

[1] *Announcing Rust 1.0 | Rust Blog*. en. 2015. URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html (besucht am 23. Juni 2021).

[2] Albert-László Barabási. *NETWORK SCIENCE NETWORK ROBUSTNESS*. en.

[3] Morteza Biabani, Nasser Yazdani und Hossein Fotouhi. „REFIT: Robustness Enhancement Against Cascading Failure in IoT Networks“. In: *IEEE Access* 9 (2021). Conference Name: IEEE Access, Seiten 40768–40782. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3065293.

[4] J.D. Day und H. Zimmermann. „The OSI reference model“. In: *Proceedings of the IEEE* 71.12 (1983), Seiten 1334–1340. DOI: 10.1109/PROC.1983.12775.

[5] Whitfield Diffie und Martin E. Hellman. *New Directions in Cryptography*. 1976.

[6] Jean-Claude Fernandez, Laurent Mounier und Cyril Pachon. „A Model-Based Approach for Robustness Testing“. en. In: *Testing of Communicating Systems*. Herausgegeben von Ferhat Khendek und Rachida Dssouli. Band 3502. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, Seiten 333–348. ISBN: 978-3-540-26054-7 978-3-540-32076-0. DOI: 10.1007/11430230_23. URL: http://link.springer.com/10.1007/11430230_23 (besucht am 23. Juni 2021).

[7] Marcel Garus. *Implementierung eines dynamischen, linearen Funknetzwerks mit heterogenen Thin Clients*. 2021.

[8] Posted 26 Oct 2011 | 19:05 GMT. *The Making of Arduino - IEEE Spectrum*. en. 2011. URL: https://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino (besucht am 23. Juni 2021).

[9] S. L. Graham, R. L. Rivest und Ralph C. Merkle. *Secure Communications Over Insecure Channels*. 1978.

[10] „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), Seiten 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[11] Heinrich Matrisch. *Classfication and Preprocessing of Sensor Data in Networks*. 2021.

[12] *NRF24L01 2.4GHz Antenna SPI Interface Wireless Transceiver Empfänger*. de. URL: https://eckstein-shop.de/NRF24L01-24GHz-Antenna-SPI-Interface-Wireless-Transceiver-Empfaenger-Funk-Modul-for-Arduino (besucht am 23. Juni 2021).

[13] Leonard Seibold. *A Flexible Toolbox to Stream Sensor Data from Microcontrollers to the Edge*. 2021.

[14] Jon Gunnar Sponås. *Things You Should Know About Bluetooth Range*. en-gb. URL: https://blog.nordicsemi.com/getconnected/things-you-should-know-about-bluetooth-range (besucht am 23. Juni 2021).

[15] *Ubiik | Weightless | High Performance LPWAN*. en. URL: https://www.ubiik.com/lpwan-technology (besucht am 23. Juni 2021).

[16] Daniel Veilleux. *Intro to ShockBurst/Enhanced ShockBurst*. en. Nov. 2015. URL: https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/intro-to-shockburstenhanced-shockburst (besucht am 14. Juni 2021).

[17] Jonas Wanke. *From Measurement to Visualization: Live Railroad Car Occupancy as an Example Use Casefor a Sensor Middleware*. 2021.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Ein robustes Funkprotokoll für IoT-Anwendungen im Zug“ („A Robust Wireless Network Protocol for On-Train IoT Applications“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 29. Juni 2021,

(Clemens Tiedt)