Hochschule Esslingen
University of Applied Sciences

Fakultät Informationstechnik
im Studiengang Softwaretechnik und Medieninformatik

Bachelorarbeit

Evaluation der Programmiersprache Rust für den Entwurf und die Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens

Michael Watzko
Sommersemester 2018

Erstprüfer: Prof. Dr. Manfred Dausmann
Zweitprüfer: Dipl.-Ing. (FH) Kevin Erath M. Sc.

Firma: IT Designers GmbH
Betreuer: Dipl.-Inf. Hannes Todenhagen

[git] • master @ 0f78cf9 • 25. Mai 2018

Sperrvermerk

TODO: ?

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 25. Mai 2018 _____

Michael Watzko

Danksagungen

„This occasionally happens in Rust: there is a period of intense arguing with the compiler, at the end of which the code looks rather nice, as if it had been a breeze to write, and runs beautifully.“

– Jim Blandly und Jason Orendorff [**rust:orly_programming**]

Inhaltsverzeichnis

1 Einleitung

Der Begriff „autonomes Fahren“ hat spätestens seit den Autos von Tesla einen allgemeinen Bekanntheitsgrad erreicht. Damit ein Auto selbstständig fahren kann, müssen erst viele Hürden gemeistert werden. Dazu gehört zum Beispiel das Spur halten, das richtige Interpretieren von Verkehrsschildern und das Navigieren durch komplexe Kreuzungen.

Bevor ein autonomes Fahrzeug Entscheidungen treffen kann, benötigt es ein möglichst genaues Modell seines Umfelds. Hierzu werden von verschiedenen Sensoren wie Front-, Rück- und Seitenkameras und Abstandssensoren Informationen gesammelt und ausgewertet. Aber vielleicht kann ein Auto nicht immer selbständig genügend Informationen zu seinem Umfeld sammeln?

Externe Sensorik könnte Informationen liefern, die das Auto selbst nicht erfassen kann. Ein viel zu schneller Radfahrer hinter einer Hecke in einer unübersichtlicher Kreuzung? Eine Lücke zwischen Autos, die ausreichend groß ist, um einzufahren ohne zu bremsen? Die nächste Ampel wird bei Ankunft rot sein, ein schnelles und Umwelt belastendes Anfahren ist nicht nötig? Ideen gibt es zuhauf.

Aber was ist, wenn das System aussetzt? Die Antwort hierzu ist einfach: das Auto muss immer noch selbstständig agieren können, externe Systeme sollen nur optionale Helfer sein. Viel schlimmer ist es dagegen, wenn das unterstützende System falsche Informationen liefert. Eine Lücke zwischen Autos, wo keine ist; eine freie Fahrbahn, wo ein Radfahrer fährt; ein angeblich entgegenkommendes Auto, eine unnötig Vollbremsung, ein Auffahrunfall. Ein solches System muss sicher sein – nicht nur vor Hackern. Es muss funktional sicher sein, Redundanzen und Notfallsysteme müssen jederzeit greifen.

Aber was nützt die beste Idee, die ausgeklügelte Strategie, wenn nur ein einziges Mal vergessen wurde, einen Rückgabewert auf den Fehlerfall zu prüfen? Was nützt es, wenn Strategien für das Freigeben von Speicher in Notfallsituationen einen Sonderfall übersehen haben? Das System handelt total unvorhersehbar.

Was wäre, wenn es eine Programmiersprache geben würde, die so etwas nicht zulässt: die fehlerhaften Strategien zur Compilezeit findet und die Compilation stoppt; die trotz erzwungener Sicherheitsmaßnahmen, schnell und echtzeitnah reagieren kann und sich nicht vor Geschwindigkeitsvergleichen mit etablierten, aber unsicheren Programmiersprachen, scheuen muss?

Diese Arbeit soll zeigen, dass Rust genau so eine Programmiersprache ist und sich für sicherheitsrelevante, hoch parallelisierte und echtzeitnahe Anwendungsfälle bestens eignet.

1.1 Projektkontext

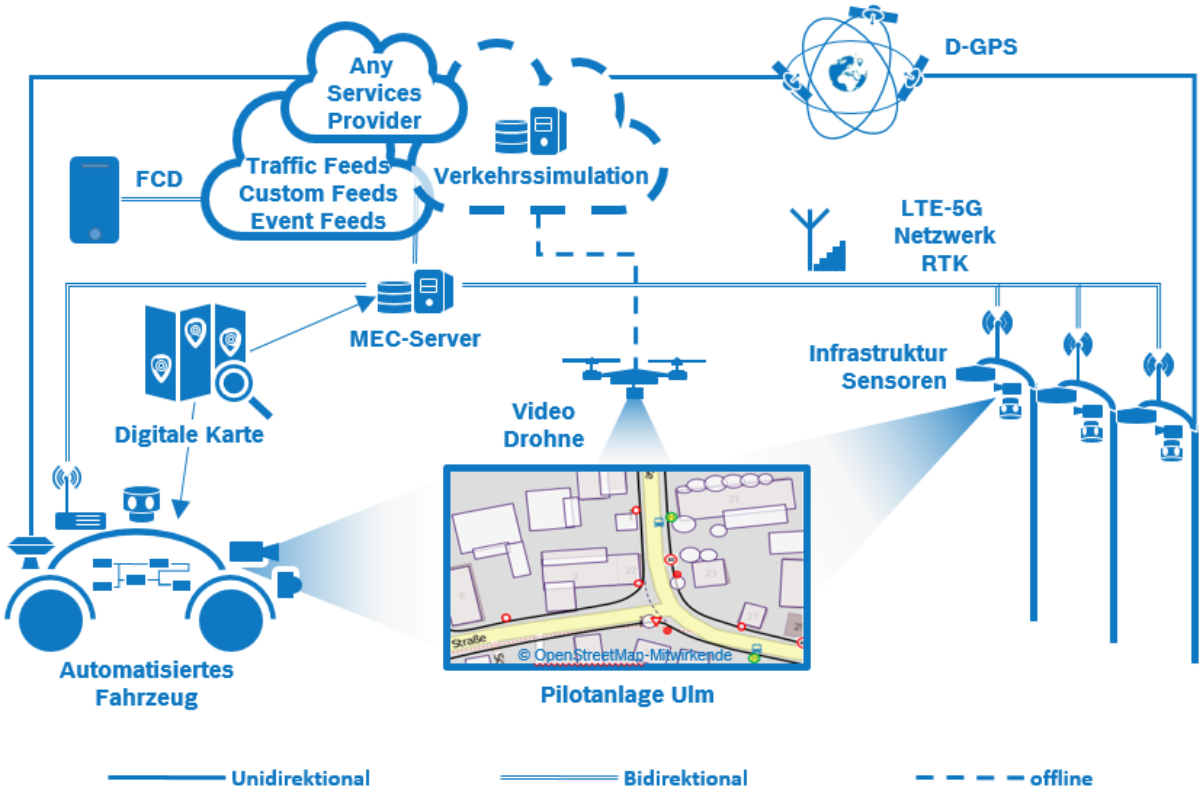


Abbildung 1.1: Übersicht über das Forschungsprojekt[1]

Diese Abschlussarbeit befasst sich mit dem Kommunikationsserver von Mobile Edge Computing (MEC)-View. Das MEC-View Projekt wird durch das Bundesministerium für Wirtschaft und Energie (BMWi) gefördert und befasst sich mit der Thematik hochautomatisierter Fahrzeuge. Es soll erforscht werden, ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist, um in eine Vorfahrtstraße automatisiert einzufahren.

Das Forschungsprojekt ist dabei ein Zusammenschluss mehrerer Unternehmen mit unterschiedlichen Themengebieten. Die IT-Designers Gruppe beschäftigt sich mit der Implementation des Kommunikationsservers, der auf der von Nokia zur Verfügung gestellten Infrastruktur im 5G Mobilfunk als MEC Server betrieben wird. Erkannte Fahrzeuge und andere Verkehrsteilnehmer werden von den Sensoren von Osram via Mobilfunk an den Kommunikationsserver übertragen. Der Kommunikationsserver stellt diese Informationen dem Fusionsalgorithmus der Universität Ulm zur Verfügung und leitet das daraus gewonnene Umfeldmodell an die hochautomatisierten Fahrzeuge der Robert Bosch GmbH und

[1] Quelle: https://www.uni-due.de/~hp0309/images/Arch_de_V1.png (Legende verschoben)

der Universität Ulm weiter. Durch hochgenaue, statische und dynamische Karten von TomTom und den Fahrstrategien von Daimler und der Universität Duisburg soll das Fahrzeug daraufhin automatisiert in die Kreuzung einfahren können.

1.2 Zielsetzung

Zu dem zuvor beschriebenen Verhalten des Kommunikationsservers besteht bereits eine Implementation in C++. Das Ziel dieser Bachelorarbeit ist es, eine alternative Implementierung des Kommunikationsservers in Rust zu schaffen. Durch die Garantien (**??**) von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert und somit eine fehlertolerantere und sicherere Implementation geschaffen werden kann.

Eine Ähnlichkeit in Struktur und Architektur zu der bestehender C++ Implementation ist explizit nicht vonnöten. Eventuelle Spracheigenheiten und einzigartige Features von Rust sollen im vollen Umfang genutzt werden können, ohne durch auferzwungene und unpassende Architekturmuster benachteiligt zu werden. Es ist erwünscht, eine kompetitive Implementation in Rust zu schaffen.

1.3 Aufbau der Arbeit

Diese Arbeit ist im Wesentlichen in die folgenden Themengebiete aufgeteilt: Grundlagen, Anforderungs- und Systemanalyse, Systementwurf und Implementation und Auswertung.

Im Themengebiet Grundlagen sollen wesentliche Bestandteile dieser Arbeit erläutert und erklärt werden. Hierzu zählt zum einen die Programmiersprache Rust in ihrer Entstehungsgeschichte, Garantien und Sprachfeatures (**??**). Zum anderen geht es um die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen (**??**) und dem Systemkontext, in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext, in dem der Server betrieben werden soll, genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt, sowie eine Übersicht über die Systeme gegeben, mit denen der Server interagiert wird.

Das Themengebiet Systementwurf und Implementation befasst sich mit dem theoretischen und praktischen Lösen der im vorherigen Kapitel aufgestellten Anforderungen. Aufgrund der Tatsache, dass es sich hierbei um eine alternative Implementation handelt, wird zur bestehenden C++ Implementation Bezug genommen. Architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede ergeben, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt.

TODO: entsprechend zu aktualisieren

2 Die Programmiersprache Rust

Rust hat als Ziel, eine sichere (siehe **??**) und performante Systemprogrammiersprache zu sei. Abstraktionen sollen die Sicherheit, Lesbarkeit und Nutzbarkeit verbessern aber keine unnötigen Performance-Einbußen verursachen (siehe **??**).

Aus anderen Programmiersprachen bekannte Fehlerquellen – wie vergessene `NULL`-Pointer Prüfung, vergessene Fehlerprüfung, „dangling pointers“ oder „memory leaks“ – werden durch strikte Regeln und mit Hilfe des Compilers verhindert (**??**). Im Gegensatz zu Programmiersprachen, die dies mit Hilfe ihrer Laufzeitumgebung[1] sicherstellen, werden diese Regeln in Rust durch eine statische Lebenszeitanalyse (**??**) und mit dem Eigentümerprinzip (**??**) bei der Kompilation überprüft und erzwungen. Dadurch erreicht Rust eine zur Laufzeit hohe Ausführgeschwindigkeit.

Das Eigentümerprinzip (siehe **??**) und die Markierung von Datentypen durch Merkmale (siehe **??**) vereinfacht es zudem, nebenläufige und sichere Programme zu schreiben.

Rust hat in den letzten Jahren viel an Beliebtheit gewonnen und ist 2018 das dritte Jahr in Folge als die beliebteste Programmiersprache in einer Umfrage auf Stack Overflow gewählt worden [**rust:stack_overflow:mose_loved**]. Die Programmiersprache scheint den sich selbst gesetzten Zielen, performant und sicher zu sein, gerecht zu werden:

> *„Again, Rust guides you toward good programs“* [**rust:orly_programming**]

> *„[..]Leute, die [..] sichere Programmierung haben wollen, [..] können das bei Rust haben, ohne [..] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen. “* [**rust:fefe**]

> *„[..] Rust makes it safe, and provides nice tools“* [**rust:c_is_hostile_mena**]

> *„Rust hilft beim Fehlervermeiden“* [**rust:c_is_hostile_golem**]

> *„Rust is [..] a language that cares about very tight control“* [**rust:tight_control**]

[1] u.a. Java Virtual Maschine (JVM), Common Language Runtime (CLR)

2.1 Geschichte

In 2006 begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln [**rust:faq**]. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Programmcode zu entwickeln. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“ [**rust:heise_interview_graydon**].

Auch Federico Mena-Quintero – Mitbegründer des GNOME-Projekts [**rust:gnome:federico**] – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [**rust:c_is_hostile_golem**]. In Vorträgen vermittelt er seither, wie Bibliotheken durch Implementationen in Rust ersetzt werden können [**rust:c_is_hostile_mena**].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, als mit einfachen Tests die Kernprinzipien demonstriert werden konnten. Die Entwicklung der Programmiersprache, des Compilers, des Buchs, von Cargo, von crates.io und von weiteren Bestandteilen findet öffentlich einsehbar auf GitHub[2] unter `https://github.com/rust-lang` statt. Dadurch kann sich jeder an Diskussionen oder Implementation beteiligen, seine Bedenken äußern oder Verbesserungen vorschlagen.

Durch automatisierte Tests (siehe **??**) in Kombination mit drei Veröffentlichungskanälen („release“, „stable“ und „nightly“) und „feature gates“ (siehe **??**) wird die Stabilität des Compilers und die der Standardbibliothek (**??**) gewährleistet.

Rust ist wahlweise unter MIT oder der Apache Lizenz in Version 2 verfügbar [**rust:copyright**].

2.2 Anwendungsgebiet

Das Ziel von Rust ist es, das Designen und Implementieren von sicheren und nebenläufigen Programmen möglich zu machen. Gleichzeitig soll der Spagat geschaffen werden, nicht nur ein sicheres aber lediglich theoretisches Konstrukt zu sein, sondern in der Praxis anwendbar zu sein. Als Beweis könnte hierbei auf die Umstellung von Firefox auf Rust und Servo – ein minimaler Webbrowser komplett in Rust geschrieben – verwiesen werden [**rust:faq**].

Interessant ist eine Diskussion von 2009, bei der „sicher aber nutzlos“ und „unsicher aber brauchbar“ gegenübergestellt wurde. Programmiersprachen scheinen auf der Suche nach

[2] Plattform zum Hosten von git-Repositories inklusive eingebautem Issue-Tracker und Wiki. Änderungen an Quellcode können vorgeschlagen und durch die Projektverantwortlichen übernommen werden. Bietet auch die Möglichkeit, eine kontinuierlichen Integrationssoftware einzubinden, um automatisierte Tests auf momentanen Quellcode und auch für Änderungen auszuführen. Eine vorgeschlagene Änderung kann somit vor Übernahme auf Kompatibilität überprüft werden.

dem nicht existierende „Nirvana“ zu sein, das sowohl sichere als auch brauchbare Programmierung verspricht [**rust:infoq:null**]. Rust möchte dieses Nirvana gefunden haben.

2.2.1 Kompatibilität

Da Rust den LLVM[3]-Compiler nutzt, erbt Rust auch eine große Anzahl der Zielplattformen die LLVM unterstützt. Die Zielplattformen sind in drei Stufen unterteilt, bei denen verschieden stark ausgeprägte Garantien vergeben werden. Es wird zwischen

- „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64),
- „Stufe 2: Compiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und
- „Stufe 3“ (u. a. Thumb (Cortex-Microcontroller))

unterschieden [**rust:platform_support**]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [**rust:github:128bit_integer**]).

2.2.2 Veröffentlichungszyklus

Es stehen Versionen in drei verschieden Veröffentlichungskanälen zur Verfügung:

- **nightly**: Version, die einmal am Tag mit dem aktuellen Stand des Quellcodes gebaut wird. Experimentelle und nicht fertige Features sind hier zwar enthalten, aber hinter „feature gates“ versteckt. Diese „Tore“ können durch entsprechende Attribute (siehe **??**) geöffnet werden.
- **beta**: Alle sechs Wochen wird die aktuellste Nightly zur Beta befördert und es werden nur noch Fehler aus dieser Version getilgt. Dieser Prozess könnte auch als Reifephase bezeichnet werden.
- **stable**: Nach sechs Wochen wird die aktuellste Beta zur Stable befördert und veröffentlicht. Gleichzeitig wird auch eine neue Beta veröffentlicht.

[3] Früher „Low Level Virtual Machine“ [**wiki:llvm**], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [**llvm:home**]. Unterstützt eine große Anzahl von Zielplattformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [**llvm:features**].
Bei der Compilierung wird der Programmcode zuerst in einen Assembler-ähnlichen Code übersetzt, der daraufhin von LLVM zu Maschinencode der Zielplattform compiliert und optimiert wird.

2.2.3 Ökosystem

Mit Rust wird nicht nur eine Programmiersprache, sondern auch ein umfassendes Ökosystem angeboten.

Cargo ist hierbei vermutlich das größte angebotene Werkzeug. Es löst Abhängigkeiten auf, indem es auf das öffentliche Verzeichnis unter `https://crates.io` zurückgreift und diese entsprechend herunterlädt und compiliert. Zum jetzigen Zeitpunkt (25. Mai 2018) sind über 14.000 Crates öffentlich erreichbar und nutzbar.

Eine Crate kann von jedem veröffentlicht werden, insofern derjenige ein GitHub-Konto besitzt, der Name der Crate noch nicht vergeben ist und der Programmcode compiliert. Die API-Dokumentation der jeweiligen Crate wird dabei automatisiert auf `https://docs.rs` veröffentlicht.

Unter `https://www.rust-lang.org` ist die Website von Rust erreichbar und unter `https://doc.rust-lang.org` sowohl die API-Dokumentation der Standardbibliothek als auch das hauseigene Rust Buch in Version 1 und 2. Die Entwicklung findet dagegen auf GitHub unter `https://github.com/rust-lang` statt.

Neue Bibliotheken für die Standardbibliothek werden zuerst unter `https://github.com/rust-lang-nursery` entwickelt. Dies ermöglicht eine ungestörte Diskussion und Entwicklung, ohne den Veröffentlichungszyklus und die Qualitätsstandards der Standardbibliothek einhalten zu müssen. Nach einer Bewährungsprobe können diese Crates in die Standardbibliothek übernommen werden [**rust:internals:1242_Recap**].

Unter `https://internals.rust-lang.org/` finden Diskussionen zu fundamentalen Herangehensweisen und Philosophien der Kernentwickler statt.

Kleine Testprogramme und Experimente können auf dem „Spielplatz“ unter `https://play.rust-lang.org` compiliert und ausgeführt werden, ohne eine lokale Installation zu benötigen. Unter `https://rustup.rs` ist dagegen das Installations- und Aktualisierungsprogramm zu finden.

2.3 Aufbau eines Projektverzeichnisses

Der Aufbau eines Rust Projektverzeichnis ist auf zwei verschiedene Arten möglich. Zum einen gibt es den klassische Aufbau, in dem lediglich der Programmcode liegt und der Compiler direkt aufgerufen und parametrisiert wird. Zum anderen wird der Aufbau als Crate (siehe **??**) empfohlen, da dadurch Abhängigkeiten automatisch aufgelöst werden können aber auch Metainformationen bezüglich des Autors, der Version und der Abhängigkeiten hinterlegt werden müssen. Ein klassischer Aufbau ist nur selten anzutreffen.

2.3.1 Klassisch

```
src/
|-- main.rs
|-- functionality.rs
|-- module/
    |-- mod.rs
    |-- functionality.rs
    |-- submodule/
        |-- mod.rs
        |-- functionality.rs
```

Listing 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager Cargo eine solche Benennung als Standardkonvention erwartet, kann bei manueller Nutzung des Compilers auch ein anderer Name für die Quelldatei vergeben werden.

Der Compiler startet in der Wurzeldatei und lädt weitere Module, die durch `mod module;` gekennzeichnet sind (ähnlich `#include "module.h"` in C/C++). Ein Modul kann dabei eine weitere Quelldatei oder ein ganzes Verzeichnis sein. Ein Verzeichnis wird aber nur als gültiges Modul interpretiert, wenn sich eine *mod.rs*-Datei darin befindet. Um Datentypen und Funktionen aus einem Modul nutzen zu können, ohne deren kompletten Pfad bei jeder Nutzung auszuschreiben, können sie durch zum Beispiel `use module::functionality::Data;` in dem aktuellen Namensraum bekannt gemacht werden. Dies ähnelt einem `import` aus Java oder einem `using` aus C#.

Wie bereits angedeutet, wird in Rust nicht eine „Klasse“, Datenstruktur oder Aufzählung pro Datei erwartet, sondern eine Quelldatei entspricht einem Modul. Dieses umfasst in vielen Fällen wenige aber mehrere Datenstrukturen, zugehörige Aufzählungen und Fehlertypen.

2.3.2 Als Crate

Eine „Crate“ (dt. Kiste/Kasten) erweitert den klassischen Aufbau um eine *Cargo.toml* Datei, in der Metainformationen zum Projekt hinterlegt werden. Durch die Benutzung des Werkzeugs „Cargo“ (dt. Fracht/Ladung) können Abhängigkeiten automatisch aufgelöst, heruntergeladen und compiliert werden.

Eine Crate kann entweder ein ausführbares Programm oder eine Bibliothek sein. Davon abhängig ist die Wurzeldatei *src/main.rs* (für ein ausführbares Programm) oder *src/lib.rs* (für eine Bibliothek). Mit dem Erzeugen einer Crate

```
crate/
|-- Cargo.toml
|-- src/
    |-- ...
```

Listing 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

(`cargo new --bin meinProg` bzw. `cargo new --lib meineBib`) wird auch gleichzeitig git[4] für das Verzeichnis initialisiert.

2.4 Hello World

```
1 fn main() {
2     println!("Hello World");
3 }
```

Listing 2.3: „Hello World“ in Rust

Der Programmcode in **??** gibt auf der Konsole `Hello World` aus. Dass `fn` die Funktion `main` definiert und diese der Startpunkt des Programms ist, wird vermutlich wenig überraschend sein. Viel überraschender ist vermutlich eher das Ausrufezeichen in Zeile 2, da es auf den ersten Blick dort nicht hingehören sollte. In Rust haben Ausrufezeichen und Fragezeichen besondere Bedeutungen, weswegen die Verwendung in Zeile 2 trotzdem richtig ist.

Die Bedeutung des Fragezeichens dient zum schnelleren Auswerten von `Rusult<_, _>`-Werten und wird in **??** genauer erklärt. Das Ausrufezeichen kennzeichnet, dass der ansonsten augenscheinliche Funktionsaufruf tatsächlich ein Aufruf einer Makrofunktion ist.

Eine Funktion `println` gibt es nicht, auch keine aus C erwarteten Funktionen wie `printf`, `fputs` oder `sprintf`. Eine Ausgabe erfolgt durch das `println!` Makro, welches einen `String` durch Nutzung des `format!` Makros erstellt und formatiert. Daraufhin wird das `writeln!` Makro verwendet, um die formatierte Zeichenkette auf die Standardausgabe zu schreiben.

2.5 Einfache Datentypen

Die Datentypen in Rust sind im wesentlichen die üblichen Verdächtigen: `bool` für boolische Ausdrücke; `char` für ein einzelnes Unicode Zeichen; `str` für eine Zeichenkette; `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, (bald `u128`, `i128` [**rust:github:128bit_integer:rfc**]) und `usize`, `isize` für ganzzahlige Werte; `f32`, `f64` für Fließkommazahlen in einfacher und zweifacher Präzision; Arrays und Slices [**rust:book:primitives**].

Ganzzahlige Datentypen mit einem führenden `u` sind vorzeichenlos (*unsigned*), vorzeichenbehaftete Datentypen (*signed*) sind dagegen mit einem `i` gekennzeichnet. Fließkommazahlen sind stattdessen mit einem führenden `f` (*floating point*) gekennzeichnet. Die darauf folgende Zahl gibt die Anzahl der Bits wieder, die für den Datentyp verwendet

[4] (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005 [**wiki:git**].

wird. Die einzige Ausnahme sind die ganzzahligen Datentypen `usize` und `isize`, da diese immer so groß sind, wie die Architektur der Zielplattform. Für die Indexierung eines Arrays oder einer Slice würden andere Datentypen, mit einer fest definierten Größe, keinen Sinn ergeben, da das Maximum an adressierbaren Elementen von der Architektur der Zielplattform abhängig ist.

Durch dieses Schema bei der Bezeichnung der Datentypen wird eine Verwirrung wie zum Beispiel in C unterbunden, wo die primitiven Datentypen (`short`, `int`, `long`, ..) keine definierte Größe haben, sondern abhängig vom eingesetzten Compiler und der Zielplattform sind [**deitel2013c**]. Erst ab C99 wurden zusätzliche, aber optionale, ganzzahlige Datentypen mit festen Größe definiert [**goll2014c**].

Konstanten können in Rust direkt einem Datentyp zugewiesen werden, indem dieser angehängt wird: `4711u16` ist vom Datentyp `u16`. Unterstriche dürfen an beliebiger Stelle Ziffern trennen, um die Lesbarkeit zu erhöhen: `1_000_000_f32`. Eine Schreibweise in Binär (`0b0000_1000_u8`), in Hexadezimal (`0xFF_08_u16`) oder in Oktal (`0o64_u8`) ist auch möglich. Konstante Zeichen und Zeichenketten können auch automatisch durch ein vorangestelltes *b* in Bytes gewandelt werden: `b'b'` entspricht `0x62_u8` und `b"abc"` entspricht `&[0x61_u8, 0x62_u8, 0x63_u8]`.

Arrays haben immer eine zur Compilezeit bekannte Größe und müssen auch immer mit einem Wert initialisiert werden (siehe **??**). Dynamische Arrays auf dem Stack gibt es (noch? [**rust:github:alloca**]) nicht, stattdessen wird auf die Vektor Implementation der Standardbibliothek verwiesen (siehe **??**). Die Notation für Arrays ist `[<Füllwert>; <Größe>]`, wobei die Größe ein konstanter Wert sein muss. `[0_u8; 128]` steht demnach für ein 128 Byte langes Array mit Werten vom Datentyp `u8`, das mit 0-en initialisiert ist.

„Slices“ (dt. Scheiben/Stücke) bezeichnet in Rust Referenzen auf Arrays oder auf Teilbereiche von Arrays und Slices. In einem so genannten „fat pointer“ wird der Startpunkt und die Größe der Slice gespeichert (siehe auch **??** auf Seite **??**). Hierdurch kann ein Zugriff außerhalb den Grenzen einer Slice oder eines Arrays verhindert werden, ein Buffer-Overflow ist in Rust daher nicht möglich.

Die Notation einer Slice ähnelt der eines Arrays: `&[<Datentyp>]`. Eine Slice kann zudem immer nur über eine eine Referenz angesprochen werden (siehe **??**). Um eine Slice auf ein Array oder eine andere Slice zu erhalten, muss der Start- und Endindex des Teilbereiches angegeben werden. Falls kein Start- oder Endindex angegeben wird, wird das jeweilige Limit übernommen.

Folgendes Beispiel soll die Notation von Arrays und Slices verdeutlichen:

```
1 fn main() {
2     let b : [u8; 10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
3     for b in &b[2..5] {
4         print!("{}, ", b);
```

```
    }
}
```

Listing 2.4: Beispiel eines Arrays und einer Slice

Das in ?? gezeigte Programm, gibt auf der Konsole `2, 3, 4,` aus.

Variablen werden durch das `let` Schlüsselwort gebunden, das heißt, der Variable wird die Eigentümerschaft für den Wert zugewiesen. Ausnahmen können Datentypen mit dem Merkmal `Copy` bilden, da diese ein implizite Kopie erlauben (siehe ??). Anstatt eine Variable optional als unveränderlich zu kennzeichnen (`const` in C, `final` in Java), wird eine Variable in Rust optional als veränderlich gekennzeichnet (`mut`). Standardmäßig ist eine Variable unveränderlich.

Lokale Typinferenz

Da Rust ein statisches Typensystem mit lokaler Typinferenz besitzt, muss der Datentyp einer Variable nicht notiert werden, sondern können automatisiert erkannt werden. Dies gilt aber nur lokal, also innerhalb von Funktionen und Closures. Parameterlisten und Rückgabewerte von Funktionen müssen die Datentypen explizit angegeben (siehe ??).

```
fn main() {
  let a = 10_u32;  // Datentyp wird durch Konstante bestimmt
  let b : u32 = a; // b muss vom Typ u32 sein
  let c = b;       // c ist vom Typ u32, weil b u32 ist
}
```

Listing 2.5: Beispiel für lokale Typinferenz

2.6 Zusammengesetzen Datentypen

Die Programmiersprache Rust kennt neben den einfachen Datentypen (??) weitere Möglichkeiten Daten zu organisieren:

- ein Tupel, das mehrere Werte namenlos zusammenfasst: `(f32, u8)` : `a.0 = 1.0_f32` ,
- eine Datenstruktur, die wie in C Datentypen namenbehaftet zusammenfasst: `struct Punkt { x: f32, y: f32 }` : `p.x = 1.0_f32` ,
- und Aufzählungen: `enum Bildschirm { Tv, Monitor, Leinwand }` .

Im Vergleich zu C kann ein Eintrag in einem `enum` gleichzeitig Daten wie eine Datenstruktur oder ein Tupel halten, oder lediglich einen Ganzzahlwert repräsentieren.

Mit dem `type` Schlüsselwort können Aliase erstellt oder im Falle von FFI (siehe **??**) aufgelöst werden: `type Vektor = (f32, f32);` Felder einer Struktur können zudem mit `pub` oder `pub(crate)` gekennzeichnet werden (siehe **??**).

Seit Version 1.19 ist auch der Datentyp `union` in Rust verfügbar [**rust:v1.19**]. Eine `union` kann aber nur in `unsafe`-Blöcken verwendet werden, da der Compiler eine ordnungsgemäße Nutzung nicht überprüfen kann. Für diese Abschlussarbeit hat der Datentyp aber keine Relevanz und wird daher nicht weiter erwähnt.

Referenzen

Auf alle Datentypen können Referenzen erstellt werden, um auf diese zuzugreifen, ohne sie zu konsumieren. In Rust spricht man dann oft davon, den Wert zu „leihen“, da sich der Eigentümer nicht ändert, sondern für den Gültigkeitsbereich der Referenz eine andere Variable auf den Wert verweist. Wie bei Variablen, wird zwischen Referenzen auf unveränderlichen und veränderlichen Werte unterschieden (siehe **??**). Die Notation für Referenzen auf unveränderliche Werte ist `&<Datentyp>`. Erwartungsgemäß ist `&mut <Datentyp>` die Notation für Referenzen auf veränderliche Werte. Referenzen auf Referenzen sind möglich. Eine manuelle Dereferenzierung einer Referenz ist in den allermeisten Fällen nicht nötig, sondern wird vom Compiler vorgenommen. In Fällen, in denen dies nicht wie erwartet automatisch geschieht, kann eine manuelle Dereferenzierung durch den `*`-Operator erzwungen werden.

2.7 Funktionen, Ausdrücke und Statements

Funktionen werden durch `fn` gekennzeichnet, gefolgt von dem Funktionsnamen, der Parameterliste und zuletzt der Datentyp für den Rückgabewert. Selbst wenn kein expliziter Rückgabetyp angegeben wird, wird formal `()` zurück gegeben; `()` entspricht etwa `void` aus bekannten Programmiersprachen. Die Parameterliste unterscheidet sich von bekannten Programmiersprachen wie C und Java, indem zuerst der Variablenname und darauf folgend der Datentyp notiert wird.

```
1 fn add(a: f32, b: f32) -> f32 {
2     a + b
3 }
```

Listing 2.6: Beispiel einer Funktion

Obwohl in Zeile 2 von **??** kein `return` zu sehen ist, wird trotzdem das Ergebnis der Addition zurückgegeben. Dies liegt daran, dass in Rust vieles ein Ausdruck ist und somit einen Rückgabewerte liefert [**rust:book:statements**]. Auch ein if-else ist ein Ausdruck und kann einen Rückgabewert haben. Ein bedingter Operator (?:) is somit unnötig, da stattdessen ein if-else verwendet werden kann: `let a = if b { c } else { d };`.

2.8 Implementierung einer Datenstruktur

Zu einer Datenstruktur oder Aufzählung kann ein individuelles Verhalten implementiert werden. In dieser Kombination ähneln diese Konstrukte sehr einer Klasse aus bekannten objektorientierten Programmiersprachen, wie zum Beispiel Java, C# oder C++ (siehe auch **??**).

Einen Konstruktor gibt es jedoch nicht; lediglich die Konvention, eine statische Funktion `new` stattdessen zu verwenden [**rust:book:constructors**]:

```
1  struct Punkt {
2      x: f32,
3      y: f32,
4  }
5
6  impl Punkt {
7      pub fn new(x: f32, y: f32) -> Punkt {
8          Punkt { x, y }
9      }
10 }
```

Listing 2.7: Punkt Datenstruktur mit einem „Konstruktor“

Manchmal wird auch `Default` implementiert (siehe **??**), wodurch eine statische Funktion `default()` als Konstruktor ohne Parameter bereitgestellt wird.

Da eine Funktionsüberladung nicht möglich ist, sollen stattdessen sprechende Name verwendet werden. Der `Vec<_>` der Standardbibliothek (siehe **??**) bietet zum Beispiel zusätzlich `Vec::with_capacity(capacity: usize)` an, um einen Vektor mit einer bestimmten Kapazität zu initialisieren.

Funktionen die sich auf eine Instanz beziehen, haben als ersten Parameter die Variable `self` (konsumierend), `&self` (lesend leihend) oder `&mut self` (exklusiv leihend) deklariert. Die `self`-Variable entspricht dabei dem `this` aus C++, C# oder Java: `fn x_eq_y(&self) -> bool { self.x == self.y }`. Ein großgeschriebenes `Self` bezieht sich auf den eigenen Typ, deshalb könnte die Funktion aus **??** auch folgende Signatur

haben: `pub fn new(x: f32, y: f32) -> Self ...`. Dies ist aber selten und oft nur in generischem Code anzutreffen (siehe **??**).

Für Funktionen können auch die Zugriffsmodifikatoren festgelegt werden (siehe **??**).

2.9 Generalisierung durch Traits

Ähnlich wie Java oder C# bietet Rust durch einen eigenen Typ die Möglichkeit, ein gewünschtes Erscheinungsbild zu generalisieren, ohne gleichzeitig eine Implementation vorzugeben. Im Rust wird dieser Typ „Trait“ (dt. Merkmal) genannt.

Für Merkmale werden Funktionen in einem entsprechenden `trait <Name> { }`-Block ohne Rumpf deklariert. Optional kann auch ein Standardrumpf implementiert werden, der bei einer Spezialisierung überschrieben werden darf. Auch auf ein Merkmal kann ein Zugriffsmodifikator gesetzt werden (siehe **??**).

Die Implementation eines Merkmals wird für jeden Datentyp in einem separaten Codeblock vorgenommen und entspricht der Notation `impl Merkmal for Datentyp { fn ... }`. Alternativ können Implementationen auch für ganze Gruppen von anderen Merkmalen vorgenommen werden: `impl<T> Merkmal for T where T: Clone { ... }` (entspricht: „implementiere `Merkmal` für all die, die `Clone`-bar sind“).

In Zukunft – oder jetzt in „nightly“ und hinter dem „feature gate“ `specialization` – wird es möglich sein, ein Standardverhalten für Gruppen zu implementieren und dieses, für einen speziellen Typ, zu überschreiben [**rust:github:specialization**].

Merkmale unterscheiden sich in ihrer Handhabung gegenüber anderen Datentypen, da sie im Allgemeinen keine bekannte Größe zur Compilezeit haben. Während dies in Programmiersprachen wie Java und C# automatisch durch die Darstellung abstrahiert und versteckt wird, hat ein Entwickler in Rust mehr Kontrolle über die Handhabung.

Dabei gibt es mehrere Vorgehensweisen:

- Die einfachste Art erfolgt über das Leihen mittels Referenz: `fn foo(bar: &Bar)` oder `fn foo(bar: &mut Bar)` – ein Unterschied zu anderen Datentypen ist nicht zu erkennen. Hierbei werden Funktionen aber dynamisch über eine „vtable“ aufgerufen, weswegen dies höhere Laufzeitkosten mit sich bringt. In Zukunft soll dieser Syntax eventuell durch `fn foo(bar: &dyn Bar)` und `fn foo(bar: &mut dyn Bar)` ersetzt werden, um deutlicher auf den dynamischen Aufruf hinzuweisen [**rust:github:dyn**].
- Alternativ kann das Objekt, das das geforderte Merkmal implementiert, auf den Heap verschoben und anschließend davon die Eigentümerschaft übertragen werden. Dies ist möglich, da das Verschieben auf den Heap die Größe der `Box` nicht beeinflusst. Eine `Box` ist letztendlich nur ein Pointer auf einen Speicherbereich auf dem

Heap. Ein Merkmal in einer `Box` wird „Trait-Object“ genannt und eine Funktionsdeklaration könnte so aussehen: `fn foo(bar: Box<Bar>)`.

- Die performanteste Alternative ist eine spezialisierte Funktion. Der Compiler dupliziert automatisch für jeden Datentyp die Funktion, setzt diesen ein und führt Optimierungen für den jeweiligen Datentyp durch (ähnlich einer Templateklasse in C++). In der Notation wird ein lokaler Typ deklariert, der als Bedingung ein oder mehrere Merkmale implementiert haben muss: `fn foo<T: Bar>(bar: T)`.

Eine Deklaration `fn foo(bar: Bar)` für das Merkmal `Bar` ist nicht möglich, da zur Compilezeit eine eindeutige Größe nicht bekannt ist. Der zu reservierende Speicher für die Variable kann nicht bestimmt werden, weswegen eine Übergabe über den Stack nicht möglich ist.

Im Folgenden werden oft anzutreffende und wichtige Merkmale aus der Standardbibliothek kurz erläutert:

- `Send`: Markiert einen Datentyp als zwischen Threads übertragbar. Automatisch für alle Datentypen implementiert, bei denen auch alle beinhalteten Datentypen von Typ Send sind. Manuelle Implementation ist nicht sicher [**rust:book:send_sync**].

  `!Send` verhindert dagegen, dass ein Wert zu anderen Threads übertragen werden darf. Somit können ansonsten rein textuell beschriebene Beschränkungen, wie zum Beispiel für der OpenGL-Kontext, durch den Compiler überprüft und erzwungen werden.

- `Sync`: Markiert einen Datentype als zwischen Threads synchronisierbar, d.h. mehrere Threads dürfen gleichzeitig lesend darauf zugreifen. `!Sync` verbietet dies hingegen. Automatisch für alle Datentypen implementiert, bei denen auch alle beinhalteten Datentypen von Typ `Sync` sind. Manuelle Implementation ist nicht sicher [**rust:book:send_sync**].

- `Sized`: Verlangt eine zur Compilezeit bekannte Größe. `?Sized` erlaubt dagegen eine unbekannte Größe zur Compilezeit.

- `Copy`: Markiert einen Datentyp, der durch einfaches Speicherkopieren (etwa „memcpy“) vervielfacht werden kann. Verlangt, dass alle beinhalteten Datentypen auch `Copy` sind. Alle einfachen Datentypen sind bereits `Copy`.

- `Clone`: Markiert einen Datentyp, der vervielfacht werden kann, dies jedoch nicht durch Kopieren des Speichers möglich ist – zum Beispiel da der Referenzzähler von `Arc` oder `Rc` erhöht werden muss. Stellt die Funktion `clone` bereit, die dafür explizit aufgerufen werden muss. Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Clone` sind. Alle einfachen Datentypen sind bereits `Clone`.

- `Debug` und `Display`: Erzwingt die Implementation von Funktionen, um einen Datentyp als Text darzustellen. Entweder mit möglichst vielen Zusatzinformationen (`Debug`) oder schön (`Display`). Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Debug` bzw `Display` sind.
- `Default`: Erzwingt die Implementation einer statische Methode `default()`, die wie ein leerer Standardkonstruktor von Java oder C# wirkt: Erzeugung einer neuen Instanz mit Standardwerten. Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Default` sind.
- `PartialEq`: Verlangt die Implementation einer Funktion, um mit Instanzen des gleichen Typs verglichen werden zu können. Im Vergleich zu `Eq` erlaubt `PartialEq`, dass Typen keine volle Äquivalenzrelation haben. Dies ist zum Beispiel für den Vergleich von Fließkommazahlen wichtig, da laut IEE754 `Nan` ungleich zu allem ist, auch zu sich selbst (`Nan != Nan`) [**wiki:nan**][**rust:orly_programming**][**rust:doc:partialeq**].
- `Eq`: Erlaubt dem Compiler einen Vergleich auf Bit-Ebene durchzuführen, ungeachtet des Datentyps [**rust:doc:eq**].
- `PartialOrd`: Verlangt die Implementation einer Funktion, damit Instanzen des gleichen Typs sortiert werden können. Erlaubt aber auch, dass Werte zueinander nicht sortierbar sind. Dies ist zum Beispiel für Fließkommazahlen wichtig, da laut IEE754 `Nan` nicht sortiert werden kann (weder `Nan <= 0` noch `Nan > 0` ergibt `true`) [**wiki:nan**][**rust:orly_programming**][**rust:doc:partialord**].
- `Ord`: Erzwingt im Gegensatz zu `PartialOrd`, dass Werte zueinander immer geordnet werden können.
- `Drop`: Verlangt die Implementation einer Funktion, die kurz vor der Speicherfreigabe eines Objekts aufgerufen wird (ähnlich Destruktor aus C++).

Mit dem Attribut `#[derive(..)]` ist eine automatisierte Implementation genannter Merkmale oft möglich, insofern die jeweiligen Bedingungen erfüllt sind. So kann im allgemeinen `#[derive(Clone)]` genutzt werden, um eine Datenstruktur oder eine Aufzählung automatisch klonbar zu machen oder `#[derive(Debug)]`, um automatisch alle Felder in Text wandeln zu können. Ein ergonomisches aber auch Fehler reduzierendes Feature.

2.10 Closure

Mit Closures bietet Rust ein entsprechendes Sprachkonstrukt um Lambdas aus Programmiersprachen wie Java, JavaScripit, C# und C++ darstellen zu können. Übergabeparameter befinden sich in der Notation zwischen zwei senkrechten Strichen, deren Typ nicht explizit angegeben werden muss. In geschweiften klammern ist anschließend der Rumpf zu

finden. Wenn der Rumpf nur eine Zeile groß ist, sind die geschweiften Klammern optional: `let adder = |a, b| a + b;`. Der Aufruf unterscheidet sich nicht von einem normalen Funktionsaufruf: `let sum = adder(1, 2);`.

Eine Closure ist eine automatisch generierte Datenstruktur, die eines der folgenden Merkmale automatisch implementiert:

- `Fn<Args>` mit `fn call(&self, arg: Args) -> Self::Output`: Ein mehrfacher Aufruf ist möglich ohne das Nebeneffekte auftreten, da die Eigenreferenz unveränderlich ist.
- `FnMut<Args>` mit `fn call_mut(&mut self, arg: Args) -> Self::Output`: Da die Eigenreferenz veränderlich ist, können Nebeneffekte auftreten.
- `FnOnce<Args>` mit `fn call_once(self, arg: Args) -> Self::Output`: Ein Aufruf ist wegen des Eigenkonsums nur einmalig möglich.

Closures in Rust sind sehr performant (siehe **??**), weswegen eine Anwendung selbst in Zeitkritischen Anwendungsfällen problemlos möglich ist [**rust:orly_programming**].

2.11 Zugriffsmodifikatoren

Zugriffsmodifikatoren erlauben es in Rust, Module, Datenstrukturen, Aufzählungen, Merkmale und Funktionen gegenüber Nutzern einer Crate und anderen Modulen sichtbar zu machen. Der standardmäßige Zugriffsmodifikator limitiert die Sichtbarkeit auf das Modul, in dem die Deklaration stattgefunden hat und wird durch keine Notation eines Zugriffsmodifikators erreicht. Um die Sichtbarkeit auf die gesamte Crate zu erhöhen, wird ein `pub(crate)` vorangestellt. Mit `pub(super)` wird der entsprechende Typ nur für das hierarchisch darüber liegende Modul Sichtbar und mit `pub(foo::bar)` kann die Sichtbarkeit auch auf ein spezifiziertes Modul limitiert werden. Mit `pub` ist die Deklaration für alle sichtbar.

Zugriffsmodifikatoren können auch vor `use` Anweisungen geschrieben werden, um entsprechende Datentypen zusätzlich unter einem neuen Namensraum bekannt zu machen.

2.12 Musterabgleich

Der `match` Ausdruck ist ein sehr mächtiges Werkzeug in Rust und entspricht einem stark erweiterten `switch` aus Programmiersprachen wie C, Java oder C#. Mit ihm ist es nicht nur möglich, einen Wert einer Aufzählung aufzulösen, sondern Muster inklusive Konstanten zu vergleichen und gleichzeitig auf eventuell beinhaltete Werte zuzugreifen oder diese zu konsumieren. In einem `match` wird immer der erste kompatible Codepfad ausgeführt.

```
1 fn main() {
2     let value : Option<&str> = Some("text");
3     match value {
4         Some("test") => println!("Nur ein Test"),
5         Some(value) => println!("Wert ist: {}", value),
6         None => println!("Kein Wert"),
7     };
8 }
```

Listing 2.8: Kompletter `match` Ausdruck

Die Ausgabe des Programms aus **??** ist `Wert ist: text`. In dem Beispiel ist `value` aus Zeile 2 und 3 `Some("text")`. Sowohl Zeile 4 als auch Zeile 5 prüfen auf die Variation `Some`, aber nur der Codepfad in Zeile 5 wird ausgeführt. Dies liegt an der zusätzlichen Prüfung für den beinhalteten Wert, der für den Codepfad in Zeile 4 mit `"test"` übereinstimmen müsste. Da eine Übereinstimmung nicht vorliegt, trifft als nächstes Zeile 5 zu, in der nur die Variation `Some` übereinstimmen muss. Die Variable `value` bindet bei dieser Übereinstimmung den Wert, um ihn für den Programmcode ansprechbar zu machen. Falls dies nicht nötig wäre, könnte stattdessen auch die Wildcard `_` verwendet werden.

Das `match` Statement von Rust verlangt, dass eine Musterabgleichung immer zu einem Ergebnis führt. Dementsprechend müssen entweder alle Varianten einer Aufzählung aufgeführt sein oder ein Standardpfad vorhanden sein `_ => { }`. Hiermit wird verhindert, dass, nachdem eine Aufzählung um eine Variation erweitert wurde, eine Musterabgleichung nicht um das neue Element ergänzt wurde.

Wenn nur ein konkreter Fall von Bedeutung ist, kann dies in der verkürzten `if let` Schreibweise notiert werden:

```
1 fn main() {
2     let mut value : Option<u32> = Some(4);
3     if let Some(ref mut value) = value {
4         *value += 1;
5     }
6     println!("{:?}", value); // "Some(5)"
7 }
```

Listing 2.9: Vereinfachte `if let` Ausdruck

Ein weiterer Unterschied von **??** gegenüber **??** ist in Zeile 3 das Schlüsselwort `ref`, wodurch der Konsum des Wertes verhindert wird. Das Schlüsselwort `mut` erlaubt zudem eine

Änderung des Wertes, weswegen `value` in Zeile 4 vom Typ `&mut u32` ist. Die Dereferenzierung mit Addition wird somit ermöglicht. Auch hier kann die Wildcard verwendet werden: `if let Some(_) = value { println!("It's something!"); }`.

Weitere Möglichkeiten, Muster zu erkennen, sind ab Seite 221 in [**rust:orly_programming**] in detaillierter Ausführung zu finden. Dazu gehören unter anderem die „guard expression", „bindings" und „ranges". Aufgrund des Umfangs und die Irrelevanz für diese Arbeit wird hier auf eine weitere Vertiefung verzichtet.

2.13 Schleifen

Rust kennt die Schleifen `for`, `while` und `loop`. Eine `do-while` Schleife wie in anderen Programmiersprachen gibt es dagegen nicht. Die einfachste dieser Schleife ist `loop { }`, da der Rumpf der Schleife ohne Bedingung wiederholt wird. Durch diesen Schleifentyp wird dem Compiler mehr über den eigentlichen Verwendungszweck mitgeteilt und bei der Codeanalyse anders bewertet als eine `while true { }` Schleife [**rust:book:loops**]. Somit ist eine, auf den ersten Blick unverständliche Formulierung wie `while (true) { }` oder `for(;;) { }` unnötig. **??** zeigt, dass eine `loop`-Schleife zusätzlich auch einen Wert zurück geben kann (`break` in Zeile 9).

```
1  const VERBOTENES_ZEICHEN : &str = "#";
2
3  fn main() {
4      let name = loop {
5          let name = ...; // Lese Zeile von stdin
6          if name.contains(VERBOTENES_ZEICHEN) {
7              println!("Versuchs nochmal");
8          } else {
9              break name;
10         }
11     };
12
13     println!("Gültiger Name: {}", name);
14 }
```

Listing 2.10: Beispiel Verwendung einer `loop` Schleife

Die `for` Schleife erwartet immer etwas iterierbares und entspricht damit einer `foreach` aus anderen Programmiersprachen. Eine inkrementelle Laufvariable, zum Beispiel für die Indizierung eines Arrays, wird durch das Iterieren über einen Zahlenstrahl dargestellt. In

?? ist dies in Zeile 4 zu sehen, während in Zeile 8 direkt über die Werte des Arrays iteriert wird.

```
1  fn main() {
2      let array = [1, 2, 3, 4, 5, 6];
3
4      for i in 0..array.len() {
5          println!("Index: {}, Wert: {}", i, array[i]);
6      }
7
8      for a in &array {
9          println!("Wert: {}", a);
10     }
11 }
```

Listing 2.11: Beispiel Verwendung einer `for` Schleife

Die `while` Schleife ist die einfachste aller Schleifen, da das Verhalten dem aus anderen Programmiersprachen entspricht. Der Rumpf wird so lange wiederholt, wie die Bedingung `true` ergibt. Das Beispiel in ?? gibt eine Sekunde lang wiederholend `"Zeit noch nicht um"` auf der Konsole aus.

```
1 fn main() {
2     let start = std::time::Instant::now();
3     while start.elapsed().as_secs() < 1 {
4         println!("Zeit noch nicht um");
5     }
6 }
```

Listing 2.12: Beispiel Verwendung einer `while` Schleife

Auch in `while` Schleifen können verkürzte Musterabgleichungen durchgeführt werden. Die Notation ähnelt dem `if let` und ist in ?? zu sehen. Die eingelesene Zeile wird hierbei so lange wieder auf der Konsole ausgegeben, bis beim Einlesen ein Fehler auftritt.

```rust
use std::io::stdin;

fn main() {
    let mut eingabe = String::new();
    while let Ok(_) = stdin().read_line(&mut eingabe) {
        println!("Eingabe: {}", eingabe.trim());
        eingabe.clear();
    }
}
```

Listing 2.13: Beispiel Musterabgleichung in einer `while` Schleife

2.14 Attribute

In Rust können Funktionen, Datentypen und manche Codeblöcke mit Attributen versehen werden, um dem Compiler weitere Informationen bereitzustellen. Attribute können dabei bestimmte Merkmale automatisiert implementieren (siehe **??**), Unit-Tests markieren (siehe **??**), Bibliotheken spezifizieren (siehe **??**), „Tore zu Besonderheiten“ (engl. feature gates) öffnen (siehe **??**) oder Zielplattformen spezifizieren [**rust:book:attribute:cfg**].

Ein Attribut folgt der Notation `#[<Name>(<optionale Parameter>)]`. So compiliert eine Funktion mit dem Attribut `#[cfg(unix)]` nur für Unix Systeme, ein Attribut `#[cfg(not(unix))]` lässt die Funktion dagegen für alle Systeme compilieren, die nicht ein Unix-System sind. Dies ermöglicht zum Beispiel mehrere Funktionen mit dem gleichen Namen aber für unterschiedliche Plattformen zu schreiben. Der Compiler übernimmt dann nur die zur Zielplattform passende Funktion.

2.15 Unit- und Integrationstests

Unit-Tests und Integrationstests können in Rust ohne eine weitere Bibliothek durchgeführt werden. Für Unit-Tests müssen Module und Funktionen mit entsprechenden Attributen versehen sein, Integrationstests müssen im Verzeichnis `tests/` gespeichert sein [**rust:book:tests**].

Unit-Tests sind per Konvention immer in der Datei mit der zu testenden Funktionalität zu finden. Diese privaten, inneren Module, die konventionell „tests“ benannt sind, werden durch das Attribut `#[cfg(test)]` markiert. Durch diese Markierung wird der

beinhaltete Code nur für das Ausführen der Tests compiliert und spart bei normaler Kompilation Zeit. `cargo test` führt alle auffindbaren Funktionen mit dem Attribut `#[test]`, leeren Parameterlisten und keinen Rückgabewerten aus. Die Makros `assert!(a)`, `assert_eq!(a, b)` und `assert_ne!(a, b)` prüfen Ergebnisse und lösen `panic!`s aus (siehe **??**), falls Ergebnisse nicht den erwarteten Werten entsprechen. Ein Test gilt als bestanden, wenn keine `panic!` ausgelöst wurde.

Integrationstests unterscheiden sich von Unit-Tests, da sie die eigene, zu testende Crate, als externe Crate betrachten. Dadurch kann nur auf öffentliche Bestandteile zugegriffen und unzureichende Zugriffsrechte aufgespürt werden. Module mit dem Attribute `#[cfg(test)]` innerhalb von Integrationstests machen keine Sinn, da Integrationstests nur für die Tests compiliert werden. Test-Funktionen sind jedoch weiterhin mit `#[test]` markiert.

2.16 Namenskonvention und Formatierung

Rust bietet einen offiziellen Styleguide, der u.a. eine Namenskonvention für Funktionen, Datentypen, Variationen und Variablen beinhaltet [**rust:styleguide:naming**]. Auch über die Formatierung und Einrückungen werden bevorzugte Arten aufgezeigt [**rust:styleguide**].

```
1 enum MY_ENUM {
2     AN_ENTRY,
3     ANOTHER_ENTRY,
4 }
```

Listing 2.14: Beispiel für eine nicht konforme Aufzählung

Der Compiler überprüft einige dieser Konventionen und warnt bei Nichteinhaltung. Das Beispiel in **??** führt dabei zu den Warnungen in **??**. Von einer weiteren Vertiefung der verschiedenen Konventionen wird hier abgesehen, da diese sehr ausführlich und mit sprechenden Beispielen auf der offiziellen Website erklärt werden.

```
[warning]: type ‘MY_ENUM‘ should have a camel case name such as ‘MyEnum‘
[warning]: variant ‘AN_ENTRY‘ should have a camel case name such as ‘AnEntry‘
[warning]: variant ‘ANOTHER_ENTRY‘ should have a camel case name such as ‘AnotherEntry‘
```

Abbildung 2.1: Hinweise des Compilers bezüglich der Aufzählung aus **??**

2.17 Niemals nichts und niemals unbehandelte Ausnahmen

Rust kennt `NULL` (-Pointer) nicht und erlaubt auch keine Zugriff auf nicht initialisierte Variablen (siehe **??**), bietet aber einen `Option<_>`-Datentyp als Ersatz an. Dieser Datentyp erzwingt eine Prüfung vor dem Zugriff (siehe **??**).

Für die Fehlerbehandlung wird nicht auf ein Exception-Handling zurückgegriffen, sondern ein eigener Datentyp angeboten, der entweder den Rückgabewert enthält, oder aber einen Fehler: `Result<_, _>` (siehe **??**).

Durch den Fragezeichenoperator kann trotzdem ein ähnliches Verhalten wie beim Auftreten einer Ausnahme in Java oder C++ erzielt werden (siehe **??**).

Ein besonderer Fehlertyp ist die Panik, denn sie bedeutet in den meisten Fällen ein Logikfehler im Programmcode, der zur Laufzeit nur schwierig zu beheben ist. Eine Panik kann zum Beispiel beim Zugriffsversuch außerhalb der Grenzen einer Slice oder eines Arrays, beim Teilen durch 0 oder auch bei einem `.unwrap()` ausgelöst werden. Eine Panik durchläuft daraufhin, wie eine Exception in C#, Java oder C++, rückwärts alle Funktionsaufrufe und gibt den Speicher geordnet wieder frei, bis sie gefangen wird oder zuletzt der panische Thread endet. Falls dies im Main-Thread auftritt, wird danach der Prozess beendet. Wenn während einer Panik eine weitere Panik ausgelöst wird, verwandelt sich die Panik in einen nicht mehr aufzuhaltenden `abort` (dt. Abbruch), der den Prozess beendet [**rust:orly_programming**].

2.18 Standardbibliothek

Das Rust Entwicklerteam ist darum bemüht, die Standardbibliothek sehr leichtgewichtig zu halten. Nicht eindeutig als fundamental eingestufte Funktionalitäten werden lieber als Crate auf `https://crates.io` angeboten, anstatt sie in die Standardbibliothek zu übernehmen. Mit dieser Entscheidung soll auch eine Entwicklung unabhängig von den Releasezyklen von Rust ermöglicht werden [**rust:internals:1242**].

Die Standardbibliothek ist selbst eine Crate, auf die standardmäßige eine Abhängigkeit besteht. Für Fälle, in denen diese Abhängigkeit zu schwergewichtig ist, wie zum Beispiel im Embedded-Bereich, kann diese Abhängigkeit durch das Attribut `#![no_std]` unterbunden werden. Daraufhin sind nur noch die in der `core`-Crate zur Verfügung gestellten, fundamentalen Sprachkonstrukte verwendbar.

In dieser Abschlussarbeit wird der volle Funktionsumfang der Standardbibliothek genutzt. Wichtige, aber auch bekannte Datentypen sind hierbei:

- **std::vec::Vec**: Ein Vektor (wie eine Liste), bei dem die Werte in einem dynamisch groß allokierten Speicherbereich auf dem Heap liegen. Ist **der** Ersatz für dynamische

Arrays, da auch der `[]`-Operator überschrieben ist und sich daher ein Vektor wie ein Array oder eine Slice ansprechen lässt.

In **??** ist das Speicherlayout eines **Vec** und einer **Slice** auf dem Stack und dem Heap abgebildet. Zu sehen ist, dass eine **Slice** direkt auf die Elemente eines **Vec** zeigen kann und sich daher von einem Array-Pointer aus C und C++ nur durch die angehängte Längeninformation unterscheidet.

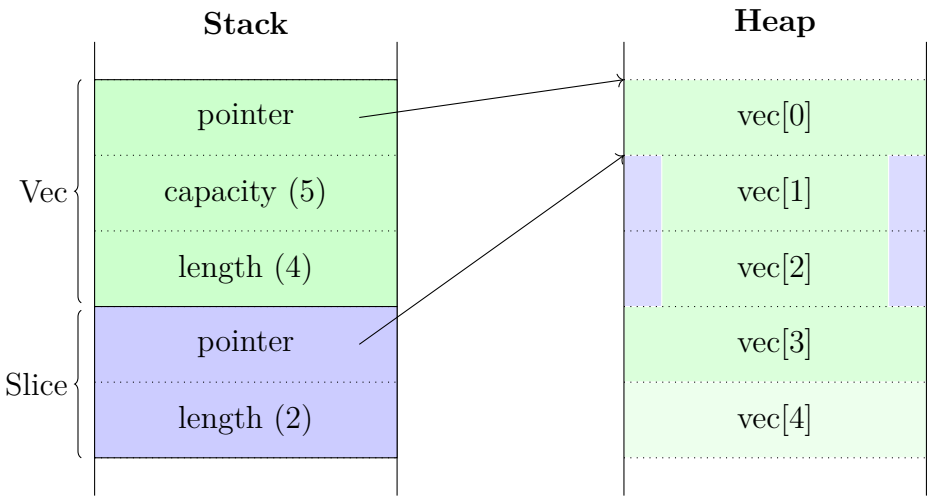


Abbildung 2.2: Speicherlayout Vec und Slice [**rust:orly_programming**]

- **std::boxed::Box**: Verweis auf einen Speicherbereich auf dem Heap eines beliebigen Datentyps. Erlaubt es, u.a. Eigentümerschaft über einen unbekannt großen Datentyp zu erlangen, da dies die Größe einer **Box** nicht beeinflusst (siehe **??**). Eine **Box** kann mit einem immer gültigen Heap-Pointer aus C und C++ verglichen werden.
- **std::string::String**: Eine UTF-8 codierte, vergrößer- und verkleinerbare Zeichenkette auf dem Heap.
- **std::rc::Rc**: Erweitert die **Box** um einen Referenzzähler und ermöglicht somit augenscheinlich mehrere Eigentümer, mit der Limitierung, nur noch lesend auf den beinhalteten Wert zugreifen zu können. Der beinhaltende Wert wird erst bei Lebensende der letzten **Rc** Instanz freigegeben. Verwendet einen mit wenig Mehraufwand verbundenen, nicht-atomaren Referenzzähler, weswegen eine **Rc** Instanz nicht zwischen Threads übertragen werden kann (`!Sync`, `!Send`).
- **std::sync::Arc**: Entspricht weitestgehend dem **Rc**, verwendet jedoch einen atomaren Referenzzähler. Dies ist zwar mit höheren Laufzeitkosten verbunden, erlaubt es aber, dass eine **Arc** Instanz zwischen Threads übertragen werden kann. Mehrere Threads können daher lesend auf den beinhalteten Wert zugreifen.

- **std::sync::Mutex**: Versichert einen threadübergreifenden, exklusiven Zugriff auf den beinhalteten Wert. In Rust schützt eine **Mutex** anstatt einen bestimmten Codeabschnitt die beinhalteten Daten [**rust:orly_programming**]. Ein Zugriff darauf ist erst möglich, nachdem andere Threads ausgesperrt werden konnten.
- **std::sync::RwLock**: Erlaubt mehreren Threads gleichzeitig lesend oder einem Thread schreibend auf den beinhalteten Wert zuzugreifen. Wie bei einer **Mutex** kann erst nach einem erfolgreichen Aussperren aller anderer Threads auf den geschützten Wert zugegriffen werden.

2.19 Speicherverwaltung

Rust benutzt ein „statisches, automatisches Speichermanagement – keinen Garbage Collector“ [**rust:youtube:goto2017**]. Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Geltungsbereichs ermittelt (siehe **??**). Durch diese statische Analyse findet der Compiler heraus, wann der Speicher einer Variable wieder freigegeben werden muss. Dies ist genau dann, wenn der Geltungsbereich des Eigentümers zu Ende ist. Weder ein Garbage Collector (GC), der dies zur Laufzeit nachverfolgt, noch ein manuelles Eingreifen durch den Entwickler (zum Beispiel durch `free(*void)`, wie in C/C++ üblich) ist nötig.

Falls der Compiler keine ordnungsgemäße Nutzung feststellen kann, wie zum Beispiel eine Referenz, die ihren referenzierten Wert überleben möchte, wird die Kompilation verweigert. Dadurch wird das Problem des „dangling pointers“ verhindert, ohne Laufzeitkosten zu erzeugen (siehe **??**).

Im folgenden **??** wird beispielhaft Speicher auf dem Heap allokiert. Dieser wird ordnungsgemäß freigegeben, ohne manuell eine Freigabe einzuleiten.

```
1 fn main() { // neuer Scope
2     let mut a = Box::new(5); // 5 kommt auf den Heap
3     { // neuer Scope
4         let b = Box::new(10); // 10 kommt auch auf den Heap
5         *a += *b; // a ist nun 15
6     } // Lebenszeit von b zu Ende, Speicher wird freigeben
7     println!("a: {}", a); // Ausgabe: "a: 15"
8 } // Lebenszeit von a zu Ende, Speicher wird freigegeben
```

Listing 2.15: Geltungsbereich von Variablen

Eine Variable kann auch vorzeitig durch den Aufruf von `std::mem::drop(_)` freigegeben werden. Die optionalen Implementation des `std::op::Drop`-Merkmals (siehe **??**) kommt der Implementation des Destruktors aus C++ gleich.

2.20 Eigentümer- und Verleihprinzip

Bereits 2003 beschreibt Bruce Powel Douglass im Buch „Real-Time Design Patterns“, dass „passive“ Objekte ihre Arbeit nur in dem Thread-Kontext ihres „aktiven“ Eigentümers tätigen sollen [**douglass2003real**]. In dem beschriebenen „Concurrency Pattern“ werden Objekte eindeutig Eigentümern zugeordnet, um so eine sicherere Nebenläufigkeit zu erlauben.

Diese Philosophie setzt Rust direkt in der Sprache um, denn in Rust darf ein Wert immer nur einen Eigentümer haben. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer, kann der Wert auch ausgeliehen werden, um einen kurzzeitigen Zugriff zu erlauben; entweder exklusiv mit sowohl Lese- als auch Schreiberlaubnis, oder mehrfache mit nur Leseerlaubnis.

Eigentümerschaft kann auch übertragen werden, der vorherige Eigentümer kann danach nicht mehr auf den Wert zugreifen. Ein entsprechender Versuch wird mit einer Fehlermeldung durch den Compiler bemängelt.

Die statische Lebenszeitanalyse garantiert, dass es nur einen Eigentümer, eine exklusive Schreiberlaubnis oder mehrere Leseerlaubnisse auf eine Variable gibt. Da dies zur Compilezeit geschieht, ist eine Überprüfung zur Laufzeit nicht nötig und diese Philosophie keinen Laufzeitkosten mit sich bringt.

```rust
fn main() {
    let mut a = Box::new(1.0_f32); // Eigentümer der neuen
                                   // Heap-Variable ist a

    {
        let b = &a; // a wird an b mit Lesezugriff verliehen
        let c = &a; // a wird an c mit Lesezugriff verliehen

        println!("a: {}", a); // "a: 1"
        println!("b: {}", b); // "b: 1"
        println!("c: {}", c); // "c: 1"

        // let d = &mut a; // Nicht erlaubt: Es existieren
                           // verliehene Lesezugriffe

        // *a = 7_f32; // Nicht erlaubt: Es existieren
                       // verliehene Lesezugriffe

    } // Ende von b und c, a nicht mehr verliehen

    {
        let e = &mut a; // Leihe a mit Schreiberlaubnis
        **e = 9_f32;    // Setze Inhalt von a

        // println!("a: {}", a); // Nicht erlaubt: exklusiver
                                 // Zugriff an e verliehen

        println!("e: {}", e); // "e: 9"

    } // Ende von e, a nicht mehr verliehen

    println!("a: {}", a); // "a: 9"
    let f = a; // Neuer Eigentümer der Heap-Variable ist f
    // *a = 12.5_f32; // Nicht erlaubt: Nicht mehr Eigentümer
    // *f = 12.5_f32; // Nicht erlaubt: f nicht änderlich
    println!("f: {}", f); // "f: 9"
}
```

Listing 2.16: Eigentümer und Referenzen von Variablen

Das Beispiel in ?? zeigt verschiedene Möglichkeiten und Beschränkungen beim Verleihen und Übertragen von Werten.

Das Eigentümerprinzip unterbindet automatisch einen „dangling pointer“ (siehe **??**) und kann Optimierungen erlauben, die, bei ansonsten unzureichend detaillierten Nachforschung in der jeweiligen API Dokumentation, schnell zu Laufzeitfehlern führen kann. Am Funktionskopf ist zum Beispiel eindeutig ablesbar, ob ein Wert von einer Funktion konsumiert wird. Falls der Wert in solch einem Fall weiterhin benötigt wird, muss eine Kopie veranlasst werden. Eine falsche Nutzung der API wird durch eine Fehlermeldung des Compilers bemängelt, anstatt eines zur Laufzeit unerwarteten Verhaltens.

`String::from_utf8(vec: Vec<u8>)` nimmt zum Beispiel einen `Vec<u8>` entgegen und konsumiert diesen. Der Aufrufer kann danach nicht mehr auf diesen zugreifen, da die Eigentümerschaft an die Funktion `from_utf8` übertragen wurde. Dies erlaubt der Funktion, den Speicherbereich des `Vec<u8>` für den `String` wiederzuverwenden, ohne neuen Speicher zu allokieren oder zu kopieren [**rust:string:from_utf8**].

Das typische Problem, die Modifikation einer Kollektion während einer Iteration, wird durch das Eigentümerprinzip schon prinzipiell ausgeschlossen. Ein Wert kann während einer Iteration nicht hinzugefügt oder entfernt werden. Es kann kein exklusiver Zugriff für die Kollektion erlangt werden, da die Kollektion bereits lesend (oder exklusiv) an die Iteration verliehen ist. Eine `ConcurrentModificationException` (Java) wird deshalb nicht benötigt [**rust:orly_programming**].

2.21 Versprechen von Rust

> *„It's not bad programmers, it's that C is a hostile language“* [**rust:c_is_hostile_mena**]
>
> *„I'm thinking that C is actively hostile to writing and maintaining reliable code“* [**rust:c_is_hostile_mena**]

Rust wirbt mit Versprechen und Garantien, die dafür sorgen sollen, typische Fehler zu vermeiden. In einer perfekten Welt wären viele dieser Maßnahmen nicht nötig, da perfekte Wesen niemals einen Fehler machen und niemals etwas übersehen würden. Programmierer sind aber Menschen, Menschen machen Fehler. Deswegen hat Rust einige interessante Mechaniken eingeführt, bekannte Fehlerquellen zu unterbinden und erzwingt deren Einhaltung, indem andere Vorgehensweisen meist ausgeschlossen werden.

Dieses Kapitel beschäftigt sich mit den wichtigsten und bekanntesten dieser Mechaniken.

2.21.1 Kein undefiniertes Verhalten

Bei der Entwicklung von Rust wird ein sehr großer Fokus darauf gelegt, keine undefinierten Zustände zu erlauben. Daher ist es normalerweise nicht möglich, ein undefiniertes Verhalten oder einen undefinierten Zustand zu erzeugen. Die Ausnahme bilden einige Fälle

innerhalb von `unsafe` Blöcken, für zum Beispiel FFI (siehe **??**). Für diese Fälle gibt es eine überschaubare Liste von Szenarien, aus denen ein undefinierter Zustand bzw. undefiniertes Verhalten resultieren kann [**rust:book:undefined**].

Als einfaches Beispiel eines undefinierten Zustandes in C ist eine Variable, die deklariert wurde, der aber noch keinen Wert zugewiesen wurde. In manchen Szenarien hat die Variable dann den Wert, der in diesem Moment an der entsprechenden Stelle im Speicher steht, in anderen Szenarien wird der Speicher vom Betriebssystem, Allokator oder von vom Compiler eingefügten Befehlen mit 0en gefüllt – eine sichere Aussage ist nicht möglich. Sich darauf zu verlassen, dass neue Werte automatisch mit 0 initialisiert wurden, kann auf neuen Systemen oder mit anderen Compilern ein unvorhersehbares Verhalten provozieren.

Rust lässt deshalb keinen Zugriff auf Variablen zu, die nicht zuvor initialisiert wurden [**rust:orly_programming**]. Der Compiler stoppt mit einem Fehler:

```
error[E0381]: use of possibly uninitialized variable: 'a'
```

Abbildung 2.3: Der Compiler bemängelt die Nutzung einer nicht initialisierten Variable

2.21.2 Keine vergessene Null-Pointer Prüfung

> *„I call it my billion-dollar mistake. It was the invention of the null reference in 1965“* [**rust:infoq:null**] TODO: cant find moment in video / presentation of this quote!?

Wie in **??** bereits erwähnt, kennt Rust keinen `NULL`-Pointer. Daher ist es auch nicht möglich, durch Nachlässigkeit auf den falschen Speicher zuzugreifen. Eine Referenz ist immer gültig. Für Fälle, in denen es situationsbedingt keinen gültigen Wert gibt, bietet Rust stattdessen den `Option<_>` Datentyp an. `Option<_>` ist eine Aufzählung, die entweder `None` ohne einen Wert, oder `Some(_)` mit einem Wert ist. Auf den Wert kann nicht zugegriffen werden, ohne zu prüfen, ob wirklich die Variation `Some(_)` vorliegt. Dies kann durch `match` oder verkürzt durch ein `if let Some(wert) = optional { /* tu etwas mit wert */ }` geschehen (siehe **??**).

In vielen Fällen kann der `Option<_>` Datentyp in Maschinencode als `NULL`-Pointer dargestellt werden, weswegen durch diese Abstraktion keine weiteren Laufzeitkosten eingeführt werden [**rust:orly_programming**] (siehe **??**).

2.21.3 Keine vergessene Fehlerprüfung

```
1 #include <stdio.h>
2
3 void main(void) {
4    FILE *file = fopen("private.key", "w");
5    fputs("42", file);
6 }
```

Listing 2.17: Negativbeispiel: Fehlende Fehlerprüfung in C

In ?? sind mindestens zwei Fehler versteckt, die aber keinen Compileabbruch auslösen, sondern sich zur Laufzeit zeigen können. Der erste Fehler ist eine fehlende Überprüfung des Rückgabewertes von `fopen` in Zeile 4. Der Rückgabewert kann `NULL` sein, falls das Öffnen der Datei fehlgeschlagen ist. Der Versuch in die Datei zu schreiben in Zeile 5 kann daraufhin in einem Speicherzugriffsfehler resultieren und das Programm abstürzen lassen.

In Rust wird weder eine Ausnahme geworfen, noch ein Rückgabewert zurück gegeben, der ohne Prüfung verwendet werden kann:

```
1  use std::fs::File;
2  use std::io::Write;
3
4  fn main() {
5    match File::create("private.key") {
6      Err(e) => println!("Datei nicht erstellbar: {}", e),
7      Ok(mut file) => {
8        if let Err(e) = write!(file, "42") {
9          println!("Konnte nicht in Datei schreiben: {}", e);
10       }
11     }
12   }
13 }
```

Listing 2.18: Positivbeispiel: Keine fehlende Fehlerprüfung in Rust

Der Rückgabewert von `File::open("private.key")` in Zeile 5 von ?? ist vom Typ `Result<File, Error>`. Auf den eigentlichen Rückgabewert `File` kann nicht ohne eine

Fehlerprüfung zugegriffen werden, da dies `Result` verhindert. Eine Fehlerprüfung kann wie in Zeile 5 mit einem `match` oder verkürzt durch ein `if let` wie in Zeile 8 geschehen.

Durch die statische Lebenszeitanalyse (siehe **??**) in Rust ist der Geltungsbereich der `mut file` Variable bekannt, deshalb wird in dem Beispiel in Rust in **??** die Datei auch wieder ordnungsgemäß geschlossen. Dies ist im C Beispiel in **??** nicht der Fall. In einem größeren Programm könnte so zu unbekanntem Zeitpunkt das Limit an gleichzeitig geöffneten Dateien erreicht werden.

Da ein `match` oder ein `if let` für jeden Funktionsaufruf, der einen Fehler zurückgeben könnte, sehr umständlich und bereits für kleine Beispiele wie **??** unübersichtlich wird, kann dies durch den Operator `?` abgekürzt werden. Dazu muss die Funktion, die den Operator verwendet aber auch ein `Result` in einem kompatiblen Fehlertyp zurückgeben, wie in **??** zu sehen:

```
1  use std::fs::File;
2  use std::io::Write;
3  use std::io::Error;
4
5  fn main() {
6    if let Err(e) = schreibe_schluessel("private.key", "42") {
7      println!("Fehler aufgetreten: {}", e);
8    }
9  }
10
11 fn schreibe_schluessel(file: &str, content: &str) ->
       Result<(), Error>  {
12   let mut file = File::create(file)?;
13   write!(file, "{}", content)?;
14   Ok(())
15 }
```

Listing 2.19: Verkürzte Fehlerbehandlung in Rust

2.21.4 No dangling pointer

Durch das Eigentümerprinzip wird eine typische Fehlerquelle aus C und C++ verhindert, bei der durch einen Pointer auf bereits deallokierten Speicher zugegriffen wird. Ein Verhalten ist hierbei nicht vorhersehbar. Es könnte einfach nur auf den als „frei“ markierten Speicherbereich zugegriffen werden, der noch die vorherigen Werte enthält. Es könnte aber auch auf den nun neu zugewiesenen Speicherbereich geschrieben und dabei Werte anderer

Datenstrukturen überschrieben werden. Im besten Fall hat das Betriebssystem die Speicherseite dem Programm bereits entzogen und das Programm stürzt einfach nur ab.

Im Beispiel in **??** wird durch eine fehlerhafte Implementierung der Funktion `klone_computer` ein „dangling pointer“ in C provoziert. Daraufhin wird versucht, mit dem gleichen Beispiel in **??** ein „dangling pointer“ in Rust zu provozieren.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Computer {
  char* model;
} Computer;

Computer* erstelle_computer(char* model) {
    Computer* computer = malloc(sizeof(Computer));
    computer->model = malloc(strlen(model)+1);
    strcpy(computer->model, model);
    return computer;
}

Computer* klone_computer(Computer* original) {
    Computer* klon = malloc(sizeof(Computer));
    klon->model = original->model;
    return klon;
}

void loesche_computer(Computer* computer) {
    free(computer->model);
    free(computer);
}

void main() {
    Computer* c1 = erstelle_computer("Lenovo");
    Computer* c2 = klone_computer(c1);
    printf("Model Nr1=%s, Nr2=%s\n", c1->model, c2->model);
    loesche_computer(c1);
    printf("Model Nr2=%s\n", c2->model);
}
```

Listing 2.20: Negativbeispiel: Fehlerhafter Klon in C

```
1  struct Computer<'a> {
2      model: &'a str,
3  }
4
5  fn erstelle_computer<'a>(model: &'a str) -> Computer<'a> {
6      Computer { model }
7  }
8
9  fn klone_computer<'a>(original: &'a Computer) ->
       Computer<'a> {
10     Computer { model: original.model }
11 }
12
13 fn loesche_computer(_: Computer) {
14     // löschen durch Konsum
15 }
16
17 fn main() {
18     // Zeichenkette auf dem Heap
19     let model = String::from("Lenovo");
20
21     let c1 = erstelle_computer(&model);
22     let c2 = klone_computer(&c1);
23     println!("Model Nr1={}, Nr2={}\n", c1.model, c2.model);
24
25     loesche_computer(c1);
26     println!("Model Nr2: {}", c2.model);
27 }
```

Listing 2.21: Negativbeispiel: Fehlerhafter Klon in Rust

Ein äquivalentes Beispiel zu dem C-Beispiel in Rust zu schreiben, ist schwierig. Das Feld „model“ in der C-Struktur ist beim Erstellen der Eigentümer der Zeichenkette (Datentyp `String` in Rust) und beim Klonen der Entleiher (Datentyp `&str` in Rust). Im Rust-Beispiel ist die Zeichenkette deswegen in Zeile 19 außerhalb der `erstelle_computer`-Funktion, da der Geltungsbereich ansonsten beim Verlassen der Funktion bereits zu Ende wäre und das Leihen an die Datenstruktur nicht möglich wäre. Dieser Unterschied ist gleichzeitig auch die Fehlerquelle im C-Beispiel.

Das Beispiel in **??** entspricht dennoch weitestgehend dem Beispiel aus **??**, zumindest genügend, um zu zeigen, dass der Rust Compiler den Fehler erkennt und die Kompilation abbricht (für Zeile 25):

```
error[E0505]: cannot move out of ‘c1‘ because it is borrowed .
```

Abbildung 2.4: Der Compiler bemängelt den Versuch, eine Leihgabe länger als den Eigentümer leben zu lassen

2.21.5 Speichersicherheit

In Rust werden verschiedene Arten von Speichersicherheit garantiert. Zum einen, wird niemals auf einen `NULL`-Pointer zugegriffen (siehe **??**), zum anderen wird niemals auf einen bereits deallokierten Speicherbereich zugegriffen (siehe **??**).

Die Speichersicherheit umfasst aber auch den Zugriffe auf Puffer. So ist es nicht möglich einen Pufferüberlauf zu provozieren, da die Größe von einem Array und einer Slice immer bekannt ist. Der Compiler erzwingt eine Grenzüberprüfung beim Zugriff zur Laufzeit, falls die Einhaltung statisch nicht ersichtlich ist (**??**).

Ein weiterer Punkt zur Speichersicherheit ist im Bereich der Nebenläufigkeit zu finden: Ein Datenwettlauf wird durch Anwendung des Eigentümerprinzips verhindert (siehe **??**).

2.21.6 Sichere Nebenläufigkeit

Eine sichere Nebenläufigkeit wird in Rust durch das Eigentümerprinzip (siehe **??**) in Kombination mit den `Send` und `Sync` Merkmalen (siehe **??**) erreicht. Dabei ist diese sichere Nebenläufigkeit meist unsichtbar [**rust:orly_programming**], da der Compiler eine unsichere und damit syntaktisch falsche Verwendung nicht übersetzt. Ein Rust Programm das compiliert, ist daher, in vielerlei Hinsicht, sicher in der Nebenläufigkeit. Einzig ein „Deadlock“ kann nicht statisch ermittelt und verhindert werden.

Eine Wettlaufsituation (englisch „race condition“) um einen Wert ist in Rust nicht möglich. Das Eigentümer- und Leihprinzip verhindert dies, denn es kann nur exklusiv schreibend auf einen Wert zugegriffen werden (siehe **??**). Für einen Datenwettlauf muss dagegen, gleichzeitig zu einem schreibenden, ein lesender Zugriff erfolgen.

Datentypen, die einen gemeinsamen Zugriff auf veränderliche Werte ermöglichen (`Mutex`, `RwLock` und im erweiterten Sinne „channels“), liefern immer ein Ergebnis, ob der Versuch, einen exklusiven Schreib- oder Lesezugriff zu erhalten, erfolgreich war. Vor einer Fehlerauswertung kann die Sperre nicht genutzt werden (siehe **??**). Ein Sperrversuch wird mit einem Fehler beantwortet, wenn ein vorheriger Thread die Sperre nicht ordnungsgemäß, sondern mit einer `panic!`, beendet hat. In diesem Fall wird ein `PoisonError` zurückgegeben, der einen direkten Zugriff auf den Wert verhindert. Damit wird darauf hingewiesen, dass der geschützte Wert aufgrund er `panic!` eventuell in keinem konsistenten Zustand mehr ist.

In Rust wird auf einen geschützten Wert durch eine gültige Sperre zugegriffen. Es wird nicht ein Codebereich exklusiv betreten, sondern ein Speicherbereich exklusiv genutzt. Ein Zugriff ohne gültige Sperre ist deshalb nicht möglich:

```
use std::sync::Mutex;

fn main() {
    let mutex = Mutex::new(0);
    let lock  = mutex.lock();

    if let Ok(mut counter) = lock {
        *counter += 1;
        println!("Zähler: {}", counter); // "Zähler: 1"
    }
}
```

Listing 2.22: test2

2.21.7 Zero Cost Abstraction

Trotz der vielen verwendeten Abstraktionen möchte Rust dadurch möglichst keine weitere Laufzeitkosten erzeugen. Beim Übersetzen werden deshalb viele Abstraktionen durch Optimierungen für den Maschinencode unsichtbar.

2.21.8 Optional

Der `Option<_>` Datentyp kann zum Beispiel in vielen Fällen als Pointer dargestellt werden, der bei `NULL` `None` und ansonsten `Some(_)` ist [**rust:orly_programming**]. Somit wird eine Überprüfung erzwungen, ohne dabei Laufzeitkosten erzeugt zu haben.

2.21.9 Referenzzähler

Ein weiteres Beispiel sind die Referenzzählertypen `Rc` und `Arc<_>`. Der Zähler ist im Heap direkt vor dem beinhalteten Wert und nicht in einem extra Speicherbereich, weshalb ein weiterer, indirekter Speicherzugriff mit Laufzeitkosten verhindert werden kann.

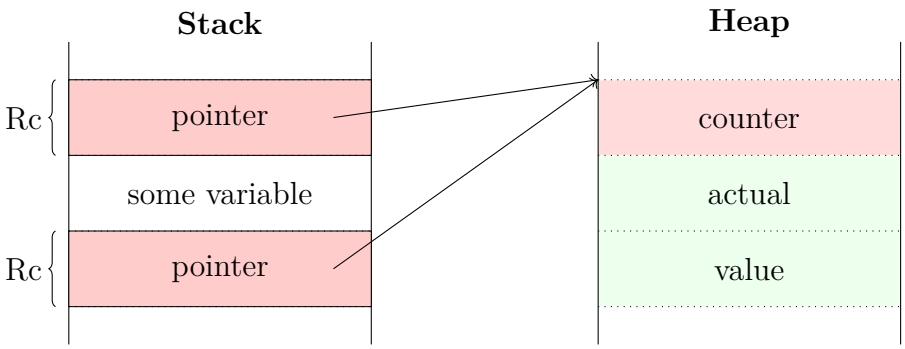


Abbildung 2.5: Speicherlayout Rc [**rust:orly_programming**]

2.21.10 Closures

Durch das Eigentümerprinzip (siehe **??**) und die statische Laufzeitanalyse (siehe **??**), sind Closures (siehe **??**) so sicher wie ähnliche Sprachkonstrukte in Programmiersprachen mit GC und so performant wie Lambdas in Programmiersprachen wie C++. Closures können sogar schneller als Funktionspointer sein [**rust:orly_programming**], da ein „Inlining“ möglich ist und da sie auf dem Stack allokiert werden können. Der Compiler kann diese und weitere umfassende Optimierungen durchführen, wenn ihm der genaue Typ bekannt ist.

2.22 Einbinden von externen Bibliotheken

Externe Datentypen

Rust bietet durch das Foreign Function Interface[5] (FFI) die Möglichkeit, Funktionalität anderer (System-)Bibliotheken zu nutzen. Entsprechende Strukturen und Funktionen werden durch einen `extern` Block oder im Falle von Strukturen stattdessen optional mit einem `#[repr(C)]` gekennzeichnet.

In einem Beispiel soll die Nutzung von Foreign Function Interface demonstriert werden.

[5] Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [**wiki:ffi**]

```c
typedef struct PositionOffset {
    long position_north;
    long position_east;
    long *std_dev_position_north; // OPTIONAL
    long *std_dev_position_east;  // OPTIONAL

    // ...
} PositionOffset_t;
```

Listing 2.23: Ausschnitt von „PositionOffset“ (C-Code) aus der *libmessages-sys* Crate

Die Struktur in **??** muss zur Nutzung in Rust zuerst bekannt gemacht werden. Dabei gibt es mehrere Möglichkeiten:

- Falls der Aufbau der Struktur nicht von Bedeutung ist, kann es ausreichen, den Datentyp lediglich bekannt zu machen: `#[repr(C)] struct PositionOffset;`. In diesem Fall können aber nur Referenzen und Raw-Pointer auf die Struktur verwendet werden.
- Falls der Aufbau wie in **??** unbedeutend ist, es soll aber ausdrücklich auf einen externen Datentyp hingewiesen werden soll, kann dieser in einem `extern { }` Block bekannt gemacht werden: `extern { type PositionOffset; }` [**rust:github:extern_type**]. Dies ist zum jetzigen Zeitpunkt aber nur in „nightly“ und hinter dem „feature gate“ `extern_types` möglich.
- Der Inhalt der Struktur ist von Bedeutung, da darauf zugegriffen oder in Rust eine Instanz werden soll. In diesem Fall ist eine komplette Wiedergabe der Struktur unumgänglich:

```rust
use std::os::raw::c_long;

#[repr(C)]
pub struct PositionOffset {
    pub position_north: c_long,
    pub position_east: c_long,
    pub std_dev_position_north: *mut c_long,
    pub std_dev_position_east: *mut c_long,
    // ...
}
```

Listing 2.24: Ausschnitt von „PositionOffset“ (Rust-Code) aus der *libmessages-sys* Crate

In ?? ist die Struktur „PositionOffset“ deklariert, die durch das Attribut `#[repr(C)]` wie eine C-Struktur im Speicher organisiert wird. Damit die Struktur in Rust kompatibel zu der in C ist, müssen die Variablen von der selben Größe sein, ansonsten würde das Speicherlayout nicht übereinstimmen. Hierfür werden spezielle Datentypen (`c_long` , `c_void` , `c_char` , ...) angeboten, um die Kompatibilität mit verschiedenen Systemen und C-Compilern zu wahren.

Ein C-Pointer `*long` wird in Rust „Raw-Pointer“ genannt und entweder `*mut c_long` oder `*const c_long` geschrieben. Der Unterschied ist wie zwischen `&mut c_long` und `&c_long` und dient dem Rust Compiler zum Nachvollziehen, ob ein exklusiver Zugriff benötigt wird, oder nicht. Dies hilft zwar für die Fehlervermeidung durch eventuelle Compilefehler anstatt Laufzeitfehler, ist aber für die C-Funktion unbedeutend [**rust:book:raw_ptr**]:

Referenz in Rust	Raw-Pointer in Rust	C-Pointer
`&mut c_long`	`*mut c_long`	`long*`
`&c_long`	`*const c_long`	`long*`

Abbildung 2.6: Vergleich Rust Raw-Pointer und Referenz zu C-Pointer

Externer Funktionsaufruf

Externe Funktionen müssen im Gegensatz zu externen Strukturen immer in einem `extern {}` Block deklariert sein.

```
use std::os::raw::c_void;

#[link(name = "messages", kind = "static")]
extern {
    type asn_TYPE_descriptor_s;
    type asn_enc_rval_t;

    fn uper_encode_to_buffer(
        type_descriptor: *const asn_TYPE_descriptor_s,
        struct_ptr: *const c_void,
        buffer: *mut c_void,
        buffer_size: usize,
    ) -> asn_enc_rval_t;
}
```

Listing 2.25: Externe Funktionsdefinition der ASN.1 Funktion zum Enkodieren

Wie in **??** zu sehen ist, können auch `extern {}` Blöcke mit Attributen (siehe **??**) versehen werden. Zwingend ist bei der Verwendung einer `#[link(..)]` Attributs der Name der Bibliothek, auf die sich der im `extern {}` Block stehende Code bezieht. Optional kann auch wie in **??** die Art der Verlinkung (dynamisch oder statisch) angegeben werden.

Die Art der Definition einer externen Funktion unterscheidet sich nicht von einer normalen Funktionsdefinition. Es sollten aber, wie in **??** beschrieben, zu C bzw. der externen Sprache kompatiblen Datentypen verwendet werden.

2.23 Unsafe

> *„To isolate unsafe code as much as possible, it's a good idea to enclose unsafe code within a safe abstraction and provide a safe API“* [**rust:book:unsafe**]

In manchen Situationen kann der Rust Compiler ein ordnungsgemäßes Speichermanagement nicht nachweisen. So ist zum Beispiel nicht nachvollziehbar, wie eine externe Funktion (siehe **??**) mit einem übergebenem Zeiger umgeht. Ebenso unklar ist, ob ein zurückgegebener Zeiger immer gültig (`&T`) oder auch ungültig (`NULL` / `Option<T>`) sein kann. Aus diesem Grund ist die Dereferenzierung von Zeigern als unsicher markiert. Eine Nutzung ist nur innerhalb `unsafe {}`-Blöcken möglich. Programmierer sollen dadurch besonders aufmerksam den (möglichst kleinen) Programmcode programmieren und prüfen.

Besondere Funktionen oder Datentypen, wie eine `Mutex`, die einen exklusiven Zugriff auf den inneren Wert ermöglicht, ohne selbst durch eine exklusive Referenz aufgerufen worden zu sein, sind nicht mit dem Eigentümerprinzip vereinbar. Die Implementation findet deswegen Teilweise auch in `unsafe {}`-Blöcken statt - für die Speichersicherheit sind dann die Entwickler (der Standardbibliothek) verantwortlich.

2.24 Beispiele der Verwendung von Rust

Der womöglich bekannteste Einsatzzweck von Rust ist im Webbrowser Firefox. Mehrere Versuche die Layoutberechnung in C++ zu parallelisieren sind aufgrund schwer auffindbaren Fehlern abgebrochen worden [**rust:example:firefox**]. Eine Parallelisierung im aktuellen Projekt „Quantum“ in Rust is dagegen mit ersten Erfolgen gekrönt [**rust:example:firefox_heise**]

Dropbox erreicht „Hunderte Millionen von Geräten“ mit Rust und das GNOME Projekt ermöglicht die Integration von Rust Code [**rust:example:two_years**].

Chucklefish, ein unabhängiges Spielestudio in London [**chucklefish:about**], hat bereits im Oktober 2017 bekanntgegeben, dass ihr nächstes Spiel „Witchbrook“ anstatt in C++ in Rust geschrieben wird [**chucklefish:rust:reddit**]. Im April 2018 wurde hierzu

ein Whitepaper veröffentlicht, in dem auch die Gründe für den Wechsel erläutert sind [**chucklefish:rust:whitepaper**].

3 Systemrelevante Technologien

Dieses Kapitel erläutert weitere, relevante Themen, die zur Umsetzung der hochperformante, serverbasierte Kommunikationsplattform wichtig sind.

3.1 Echtzeitsysteme

Echtzeitsysteme zeichnen sich im allgemeinen dadurch aus, eine Aufgabe in einem zuvor vorgegebenen Zeitraum bearbeiten zu können. Es existiert zu einer Aufgabe also immer eine Frist. Bei der Bewertung der Korrektheit eines Systems wird die Fähigkeit, eine Frist einhalten zu können, auch bewertet [**perf:buttazzo2006soft**]. Je nach Art des Echtzeitsystems, wird diese Frist jedoch unterschiedlich gewichtet:

- Bei einem harten Echtzeitsystem kann eine Überschreitung der Frist einen katastrophalen Ausgang haben. Selbst im schlimmsten Fall darf diese Frist nicht überschritten werden. Deswegen wird in einem harten Echtzeitsystem die maximale Reaktionszeit dem Zeitraum bis zur Frist gegenübergestellt [**douglass2003real**]. Ein Ergebnis nach Ablauf der Frist wird als nutzlos gewertet [**perf:wang2017real**].

  Zum Beispiel könnte eine zu späte Auswertung von Beschleunigungsdaten in einem Flugzeug zu einer verzögerten und mittlerweile falschen Reaktion und daraufhin zu einem Absturz führen [**perf:laplante2004real**].

- Bei einem weichen Echtzeitsystem resultiert die Überschreitung des vorgegebenen Zeitraums nicht in einer Katastrophe. Es wird die durchschnittliche Reaktionszeit dem Zeitraum bis zur Frist gegenübergestellt. Eine seltene und unter Last auftretende Überschreitung wird in Kauf genommen [**douglass2003real**]. Das System führt in so einem Fall weiterhin seine Aufgaben aus, die Performance wird daraufhin aber als unzureichend eingestuft.

3.2 Mobile Edge Computing

Mobile Edge Computing (MEC) bezeichnet Recheneinheiten, die eine Cloud-ähnliche Umgebung am Rande des Mobilfunknetzes schaffen [**etsi:mec**]. Wenn in dieser Arbeit auf

MEC Bezug genommen wird, sind damit explizit direkt an Funkmasten montierte Recheneinheiten gemeint. Dadurch, dass die Recheneinheiten direkt an einen Funkmast angeschlossen sind, können sie Anfragen aus dem Abdeckungsbereich der Antenne deutlich schneller beantworten (Latenz kleiner 20ms) als Cloudlösungen (Latenz ca 100ms) [**perf:mec:fraunhofer**]. Hierfür werden die Anfragen aus dem Mobilfunknetz direkt an die Recheneinheit weitergeleitet, anstatt über einen Provider eine Internetverbindung zu einer Cloudlösung aufzubauen.

Die Serverimplementierung des MEC-View Projekts wird in einer MEC-Recheneinheit ausgeführt, um die Sensordaten der Sensoren möglichst schnell an die Fahrzeuge des Abdeckungsbereichs des Funkamstes weiterleiten zu können.

3.3 ASN.1

> *„ ASN.1 has a long record of accomplishment, having been in use since 1984. It has evolved over time to meet industry needs, such as PER support for the bandwidth-constrained wireless industry and XML support for easy use of common Web browsers. “* [**asn:itu:asn.1**]

Die Notationsform Abstract Syntax Notation One (ASN.1) ermöglicht es abstrakte Datentypen und Wertebereich zu beschreiben [**asn:layman**]. Die Beschreibungen können anschließend zu Quellcode einer theoretisch[1] beliebigen Programmiersprache compiliert werden. Beschriebene Datentypen werden dadurch als native Konstrukte dargestellt und können mittels einer der standardisierten (oder auch eigenen [**asn:itu:ecn**]) Encodierungen serialisiert werden.

Um den Austausch zwischen verschiedenen Anwendungen und Systemen zu ermöglichen, sind durch die International Telecommunication Union (ITU) bereits einige Encodierungen standardisiert [**asn:itu:x691**]. Für diese Arbeit ist aber einzig der PER bzw. UPER Standard relevant, da der Server diese Encodierung verwenden muss, um mit den Sensoren und den Autos zu kommunizieren (siehe Anforderung in **??**).

Andere, bekanntere Verfahren werden hier nur kurz erwähnt:

- **BER** (Basic Encoding Rules): Flexible binäre Encodierung [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**] und ISO/IEC 8825-1 [**asn:iso**].
- **CER** (Canonical Encoding Rules): Reduziert BER durch die Restriktion, die Enden von Datenfelder speziell zu markieren anstatt deren Größe zu übermitteln und eignet sich gut für große Nachrichten [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**] und ISO/IEC 8825-1 [**asn:iso**].

[1] Es gibt keine Einschränkungen seitens des Standards, aber entsprechende Compiler zu finden erweist sich als schwierig (siehe **??**)

- **DER** (Distinguished Encoding Rules): Reduziert BER durch die Restriktion, Größeninformationen zu Datenfeldern in den Metadaten zu übermitteln und eignet sich gut für kleine Nachrichten [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**] und ISO/IEC 8825-1 [**asn:iso**].
- **XER** (XML Encoding Rules): Beschreibt den Wechsel der Darstellung zwischen ASN.1 und XML, spezifiziert in X.693 [**asn:itu:x693**] und ISO/IEC 8825-4 [**asn:iso**].

PER und UPER

Die Packed Encoding Rule ist in in X.691 [**asn:itu:x691**] und ISO/IEC 8825-2 [**asn:iso**] spezifiziert. Sie beschreibt eine Encodierung, die Daten kompakt – also in wenigen Bytes – serialisiert. Zu PER sind mehrere Variationen spezifiziert, für diese Arbeit ist jedoch nur UPER (unaligned PER) von Bedeutung. Im Gegensatz zu anderen Variationen bestehen Datenbausteine in UPER nicht aus ganzen Bytes, sondern aus unterschiedlich vielen Bits. Eine serialisierte Nachricht ist deswegen nicht N-Bytes sondern N-Bits lang. An den resultierenden Bitstring dürfen 0-Bits angehängt werden, um diesen in einen Bytestring wandeln zu können. Durch dieses Verfahren ist die Nachricht noch kürzer darstellbar.

Für Funkverbindungen ist dies von besonderer Bedeutung, da sich alle Teilnehmer das gleiche Übertragungsmedium teilen. Das Einsparen von wenigen Bytes pro Nachricht und je Teilnehmer ermöglicht einen höheren Gesamtdurchsatz.

3.4 Test-Driven Development

> *„Failure is progress.“* [**tdd**]

Bei der Test-getriebenen Entwicklung werden Tests in den Vordergrund gestellt. Die Implementierung einer neuen Funktionalität wird durch neue Tests, welche die Anforderung repräsentieren, eingeleitet. Erst nachdem ein Test erfolgreich feststellt, dass die geforderte Funktionalität noch nicht vorhanden ist, wird mit der Implementierung begonnen. Eine schnelle Implementierung hat hierbei die höchste Priorität und erlaubt temporär auch eine limitierende, „stinkende“ und naive Vorgehensweise [**tdd**]. Direkt im Anschluss wird ein Refactoring[2] durchgeführt, um die Qualitätsstandards wieder einzuhalten. Diese drei Phasen werden „red/green/refactor“ bezeichnet:

- **red**: Ein neuer Test wird erstellt, dieser stellt erfolgreich die Abwesenheit der Funktionalität fest, eine rote Fehlermeldung ist zu sehen.

[2] Verbesserung des Codes und der Struktur ohne Änderung der Funktionalität

- **green**: Der Test wird durch neuen Code zufriedengestellt; eine positive Ausgabe bestätigt dies. Eine schnelle Implementierung wird hierbei temporär einer hochwertigen bevorzugt [**tdd**], da ein erfolgreicher Test das Selbstvertrauen beim Refactoring stärkt und helfen würde, fehlerhafte Tests zu finden [**tdd**].
- **refactor**: Der neue Code wird aufgeräumt und verbessert, um den Qualitätsstandards gerecht zu werden.

Die Testgröße und der daraus resultierende Umfang der neuen Funktionalität wird durch die Zuversichtlichkeit des Entwicklers gesteuert [**tdd**]. Eine hohe Zuversicht führe zu größeren Tests, die etwas mehr Funktionalität auf einmal prüfen, während eine hohe Unsicherheit zu vielen kleinen Tests führen würden. Daraus resultiert, dass gerade komplexe Algorithmen mit vielen Tests abgesichert sein sollten. Bestehende Tests bilden ein Sicherheitsnetz, mit dem Fehler durch Änderungen detektiert werden können.

Test-getriebene Entwicklung verändert auch die Vorgehensweise bei der Implementation von Anforderungen. Anstatt zu fragen „Wie würde ich das implementieren?“, wird überlegt „Wie würde ich das testen?“ [**tdd**], womit auch implizit gefragt wird, wie die äußere Schnittstelle idealerweise aussehen sollen [**tdd**].

3.5 Funktionale Sicherheit

„Sicherheit“ ist im Deutschen kein eindeutiger Begriff. Sowohl „Sichersein vor Gefahr oder Schaden“ (*to be safe*), „Freisein von Fehlern oder Irrtümern“ (*to be confident*) oder „Schutz vor Gefahren, die von außen auf Systeme oder Personen einwirken“ (*security*) könnten mit „sicher sein“ gemeint sein [**safety**]. Deswegen ist es wichtig, den Begriff „funktionale Sicherheit“ kurz zu ergründen.

Bei funktionaler Sicherheit (*safety*) geht es um die Betriebssicherheit, eine „Freiheit von unvertretbaren Risiken“ [**safety**]. Unvertretbare Risiken sind in erster Linie Personenschäden, weswegen einheitliche Regularien in Normen wie der IEC 61508 bzw der DIN EN 61508 festgehalten sind. Für den Automobilbereich wurde die Norm in der ISO 26262 angepasst, um u.a. eine Einzelabnahme eines jeden Fahrzeuges, durch eine Gesamtabnahme des Produktes zu ermöglichen [**safety**].

Durch die Test-getriebene Entwicklung und der Verwendung von Rust und dessen Garantien (siehe **??**), sollen Fehler reduziert und eine möglichst sichere Implementierung geschaffen werden. Eine Entwicklung nach ISO 26262 findet nicht statt, da dies zum Einen nicht durch das Forschungsprojekt gefordert ist und zum Anderen den Umfang dieser Bachelorarbeit überschreitet.

3.6 Asynchrone Kommunikation und Datenströme

> *„However, for high-performance servers, you'll need to use asynchronous input and output.“* [**rust:orly_programming**]

Eine typische synchrone Kommunikation ist ein Funktionsaufruf. Zuerst werden die Parameter vorbereitet, dann die Funktion aufgerufen. Der weitere Code der aufrufenden Funktion wird nicht ausgeführt, bis der Aufruf der aufgerufenden Funktion beendet wurde.

Auch für Datenströme, bei zum Beispiel TCP-Verbindungen, bieten Standardbibliotheken der entsprechenden Programmiersprachen oft eine synchrone Kommunikation an. Hierbei beendet ein Funktionsaufruf für das Versenden oder Empfangen von Daten (bei TCP Bytes) erst, wenn die erwartete Anzahl an Elementen empfangen oder versendet wurde.

Für einen Client, der nur zu einem Server eine Verbindung aufbaut, ist diese Herangehensweise aufgrund der niedrigen Komplexität gut geeignet. Für einen Server, der mit vielen Datenströmen gleichzeitig kommunizieren muss, ist diese Herangehensweise nicht geeignet. Eine sequentieller Sendeaufruf würde die Latenz des Servers pro Client um einen von der Verbindungsgeschwindigkeit abhängigen Wert erhöhen. Einen Sende- und Empfangsthread je Client würde das Problem der Latenz lösen, jedoch aufgrund des je Thread benötigten Stacks und zusätzlichen Verwaltungsaufwand den Ressourcenverbrauch des Servers drastisch erhöhen.

Eine asynchrone Kommunikation ermöglicht dagegen die Kommunikation mit vielen Datenströmen gleichzeitig, ohne die zuvor erwähnte Probleme zu provozieren. Bei einem asynchronen Sende- oder Empfangsaufruf wird nicht auf dessen Fertigstellung gewartet, sondern in der darunterliegende Implementation wird die Anfrage notiert, sobald möglich abgearbeitet und der Aufrufende bei Fertigstellung informiert. Das erlaubt dem Aufrufer dem nächsten Datenstrom Daten zuzusenden, ohne das dies durch die Fertigstellung des ersten Aufrufs verzögert wird. Eine asynchrone Kommunikation erhöht jedoch die Komplexität des Aufrufers (hier: des Servers), da nun zusätzlich der Status einer Sende- und Empfangsanfrage nachverfolgt werden muss. Im Gegenzug wird die Verwaltung von vielen Datenströmen bzw. Verbindungen mit wenigen Threads ermöglicht.

4 Anforderungen

In diesem Kapitel sind die Anforderungen an das umzusetzende System gelistet. Die Anforderungen sind dabei in funktionale und nichtfunktionale Anforderungen sortiert. Funktionale Anforderungen beschreiben das gewünschte Verhalten des Systems, nichtfunktionale Anforderungen zeigen dagegen Rahmenbedingungen bei der Umsetzung auf [**goll2012methoden**].

4.1 Funktionale Anforderungen

- **Anforderung 1: TCP Server**
  Auf Port 2000 sollen neue TCP-Verbindungen angenommen werden. Jedem Client soll eine eigenen TCP Verbindung zugewiesen sein.
- **Anforderung 2: Client als Sensor**
  Ein Client soll sich nach dem Verbindungsaufbau als Sensor registrieren können. Ein Client soll sich nicht mehrmals registrieren können. Vor einer Registrierung soll der Typ des Clients unbekannt sein und er soll nicht als Sensor agieren können.
- **Anforderung 3: Client als Fahrzeug**
  Ein Client soll sich nach dem Verbindungsaufbau als Fahrzeug registrieren können. Ein Client soll sich nicht mehrmals registrieren können. Vor einer Registrierung soll der Typ des Clients unbekannt sein und er soll nicht als Fahrzeug agieren können.
- **Anforderung 4: Fahrzeug initialisieren**
  Der Server soll dem Fahrzeug nach Registrierung alle bekannten Sektoren übermitteln.
- **Anforderung 5: Sensor initialisieren**
  Der Server soll den Sensor nach Registrierung deabonnieren.
- **Anforderung 6: Sensoren abonnieren**
  Sensoren sollen bei verbunden Fahrzeugen abonniert sein.
- **Anforderung 7: Sensoren deabonnieren**
  Sensoren sollen bei keinen verbunden Fahrzeugen deabonniert sein.

- **Anforderung 8: Sensordaten weitergeben**
  Empfangene Sensordaten sollen dekodiert und an den Fusions-Algorithmus übergeben werden.

- **Anforderung 9: Fahrzeug kann Umgebungsmodell abonnieren**
  Ein Fahrzeug kann ein Abonnement für Umgebungsmodelle erstellen.

- **Anforderung 10: Fahrzeug kann Abonnement aufkündigen**
  Ein Fahrzeug kann ein bestehendes Abonnement für Umgebungsmodelle aufkündigen.

- **Anforderung 11: Umgebungsmodell weitergeben**
  Ergebnisse des Fusions-Algorithmus sollen enkodiert un an die verbundenen Fahrzeuge mit einem Abonnement versendet werden.

- **Anforderung 12: Fusions-Algorithmus**
  Für den Testbetrieb soll der Fusions-Algorithmus aus der C++ Referenzimplementierung („SampleAlgorithm“) in Rust nachempfunden werden. Dieser Fusions-Algorithmus ist eine „dummy“ Implementation, d.h. eingehende Daten werden nach einem einfachen Schema wieder verschickt: jede empfangende *SensorFrame*-Nachricht (siehe **??**) wird mit einer *EnvironmentFrame*-Nachricht (siehe **??**) an alle Fahrzeuge beantwortet. Der Zeitstempel der *SensorFrame*-Nachricht wird hierbei auf die *EnvironmentFrame*-Nachricht übertragen.

- **Anforderung 13: TODO: Konfigurationsdateien?**
  TODO: /etc/MECViewServer/...

- **Anforderung 14: TODO: Startparameter?**
  TODO: sample: send_delay_ms, sensor_timeout_ms, environment_frame_file, ...

4.2 Nichtfunktionale Anforderungen

- **Anforderung 15: Implementation in Rust**
  Die Implementation des Servers soll in der Programmiersprache Rust vorgenommen werden.

- **Anforderung 16: Kommunikationsprotokoll ist ASN.1/UPER**
  Das Protokoll für die Kommunikation zwischen dem Server und den Clients soll ASN.1 mit der Encodierung UPER sein. Es sollen die bereits definierten Nachrichten verwendet und keine neuen Nachrichten definiert werden. Das Kommunikationsverhalten soll die Anforderungen der C++ Referenzimplementation erfüllen, sprich den Clients soll nicht ersichtlich sein, ob die Rust oder die Referenzimplementation des Servers ausgeführt wird.

- **Anforderung 17: Plattform MEC**
  Die Implementation des Servers soll in compilierter Form auf einem MEC Server mit dem Betriebssystem Ubuntu 16.04 LTS Server und der Architektur x86-64 ausführbar sein.

- **Anforderung 18: Reaktionszeit des Servers**
  Die Implementation des Servers soll Nachrichten von 6 Videosensoren mit einem Nachrichtenintervall von 100ms, 7 Lidarsensoren mit einem Nachrichtenintervall von 50ms und zwei gleichzeitig vebundenen Fahreugen 5ms beantworten können. Dies umfasst Nachricht dekodieren, Fusions-Algorithmus darbieten, Resultat enkodieren und versenden. Die Bearbeitungszeit des Fusions-Algorithmus zählt nicht dazu.

- **Anforderung 19: Kein Echtzeitsystem**
  Trotz vorgegebener Reaktionszeit wird das System nicht als hartes Echtzeitsystem gewertet. Eine Analyse für die maximale Reaktionszeit ist nicht verlangt. Stattdessen soll das System als weiches Echtzeitsystem gewertet werden, weswegen die durchschnittliche Reaktionszeit betrachtet wird.

- **Anforderung 20: Implementation in Rust**
  Die Implementation soll in der Programmiersprache Rust vorgenommen werden.

- **Anforderung 21: Einhaltung des Styleguides**
  Der Quellcode der Serverimplementation soll sich an den offiziellen Styleguide (siehe **??**) halten.

- **Anforderung 22: Widerstand gegen Nachrichtenüberflutung**
  Die Funktionalität des Servers gegenüber anderen Clients soll durch eine Überflutung von Daten eines einzelnen Sensors nicht beeinträchtigt werden.

- **Anforderung 23: Widerstand gegen Nachrichtenrückstau**
  Die Funktionalität des Servers gegenüber anderen Clients soll durch Fahrzeuge, für die sich ein Nachrichtenrückstau gebildet hat oder von einzelnen langsamen Verbindungen, nicht beeinträchtigt werden.

- **Anforderung 24: Ob die Einbindung einer externen C++ Bibliothek als Fusions-Algorithmus möglich ist, soll überprüft werden.**

5 Systemanalyse

TODO: .. [**goll2012methoden**]

5.1 Systemkontextdiagramm

In der folgenden Abbildung sind die Systemgrenzen aufgezeigt.

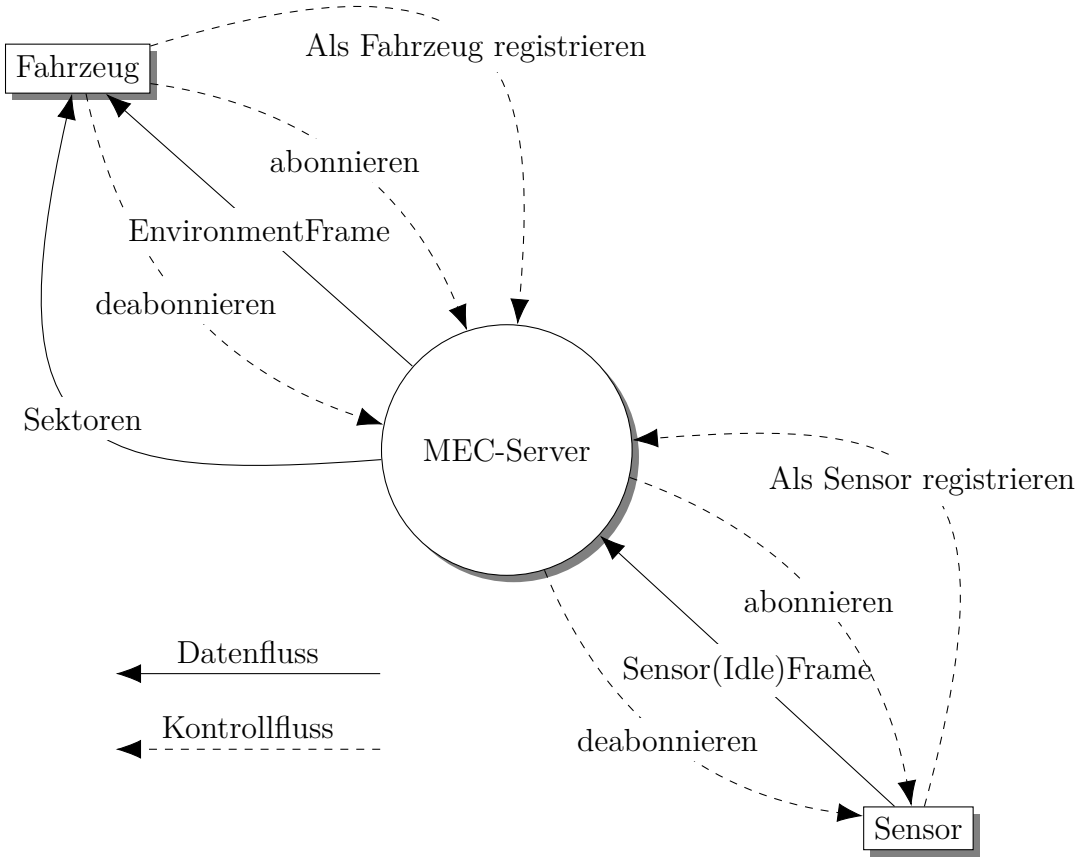


Abbildung 5.1: Systemkontextdiagramm

5.2 Anwendungsfalldiagramme

TODO: was wirklich umgesetzt sein wird

5.2.1 MEC-Server

Das Anwendungsfalldiagramm in **??** zeigt die Funktionalität des Servers, die gegenüber einem Fahrzeug un einem Sensor zur Verfügung stellt werden soll. Darauf folgend sind diese Anwendungsfälle genauer erklärt.

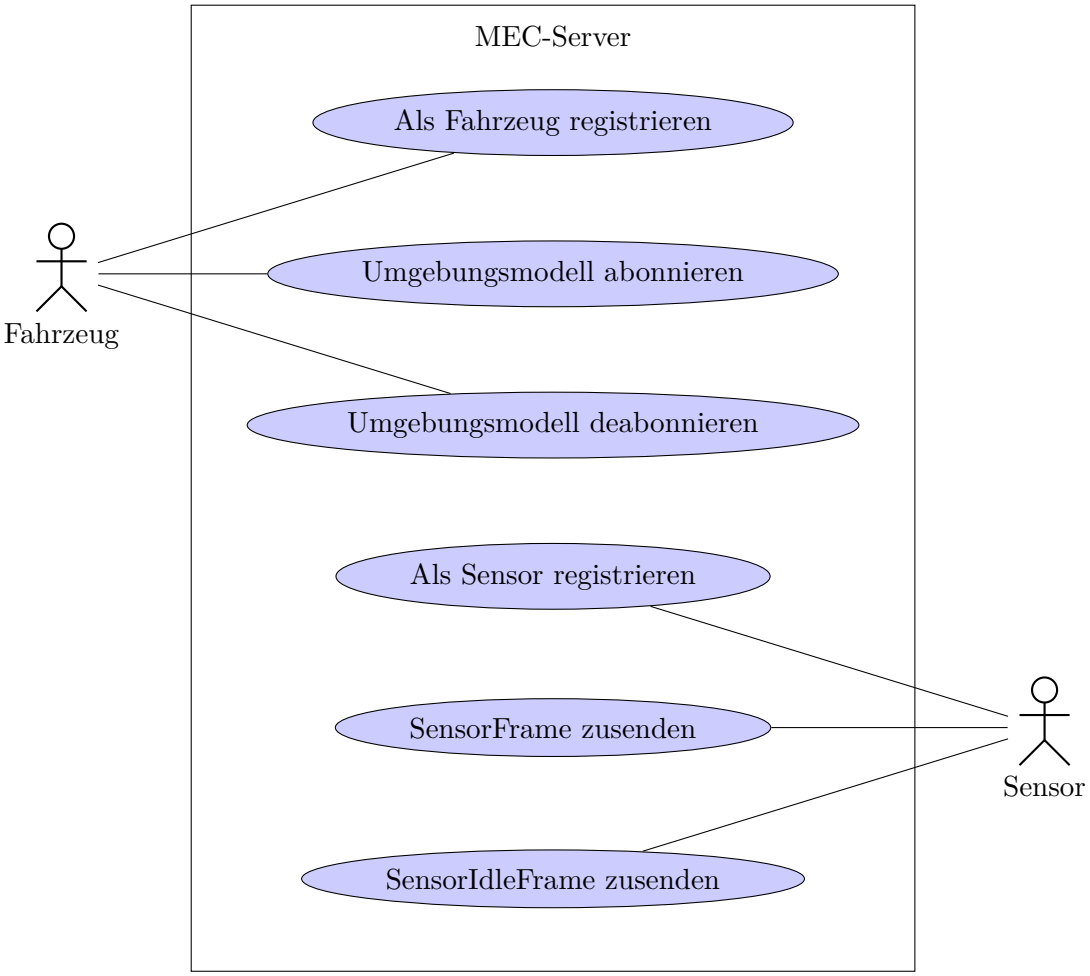


Abbildung 5.2: Anwendungsfalldiagramm des MEC-Servers

5.2.2 Als Fahrzeug registrieren

Initiator:	Fahrzeug
Beteiligte Akteure:	MEC-Server, Fahrzeug
Vorbedingung:	Noch nicht registriert
Basisablauf:	Ein neu verbundenes Fahrzeug kann sich dem Server gegenüber als Fahrzeug registrieren. Eine Registrierung kann für jede Verbindung nur einmal vorgenommen werden und wird durch die Übermittlung einer *ClientRegistration*-Nachricht durchgeführt (siehe **??**).
Alternativablauf:	–
Nachbedingung:	Das Fahrzeug kann alle weiteren, dem Fahrzeug zugeordneten, Use-Cases ausführen.

5.2.3 Umgebungsmodell abonnieren

Initiator:	Fahrzeug
Beteiligte Akteure:	MEC-Server, Fahrzeug
Vorbedingung:	Ist als Fahrzeug registriert
Basisablauf:	Ein Fahrzeug kann das Umgebungsmodell abonnieren, woraufhin neue Modelle vom Server an das Fahrzeug übermittelt werden sollen. Ein Abonnement wird durch eine *UpdateSubscription*-Nachricht aktualisiert (siehe **??**)
Alternativablauf:	–
Nachbedingung:	Neue Umgebungsmodelle werden an das Fahrzeug übermittelt.

5.2.4 Umgebungsmodell deabonnieren

Initiator:	Fahrzeug
Beteiligte Akteure:	MEC-Server, Fahrzeug
Vorbedingung:	Umgebungsmodell ist abonniert
Basisablauf:	Ein Fahrzeug kann das Umgebungsmodell deabonnieren, woraufhin keine neuen Modelle mehr vom Server an das Fahrzeug übermittelt werden sollen. Ein Abonnement wird durch eine *UpdateSubscription*-Nachricht aktualisiert (siehe **??**).
Alternativablauf:	–
Nachbedingung:	Es werden keine weiteren Umgebungsmodelle an das Fahrzeug übermittelt.

5.2.5 Als Sensor registrieren

Initiator:	Sensor
Beteiligte Akteure:	MEC-Server, Sensor
Vorbedingung:	Noch nicht registriert
Basisablauf:	Ein neu verbundener Sensor kann sich dem Server gegenüber als Sensor registrieren. Eine Registrierung kann für jede Verbindung nur einmal vorgenommen werden und wird durch die Übermittlung einer *ClientRegistration*-Nachricht durchgeführt (siehe **??**).
Alternativablauf:	–
Nachbedingung:	Der Sensor kann alle weiteren, dem Sensor zugeordneten, Use-Cases ausführen.

5.2.6 SensorFrame zusenden

Initiator:	Sensor
Beteiligte Akteure:	MEC-Server, Sensor
Vorbedingung:	Ist als Sensor registriert
Basisablauf:	Ein Sensor kann dem Server eine *SensorFrame*-Nachricht übermitteln. Der Server soll diese Nachricht dem Fusions-Algorithmus weiterleiten.
Alternativablauf:	–
Nachbedingung:	–

5.2.7 SensorIdleFrame zusenden

Initiator:	Sensor
Beteiligte Akteure:	MEC-Server, Sensor
Vorbedingung:	Ist als Sensor registriert
Basisablauf:	Ein Sensor kann dem Server eine *SensorIdleFrame*-Nachricht übermitteln.
Alternativablauf:	–
Nachbedingung:	–

5.2.8 Sensor

In **??** sind Anwendungsfälle aufgezeigt, die durch den Sensor dem MEC-Server angeboten werden.

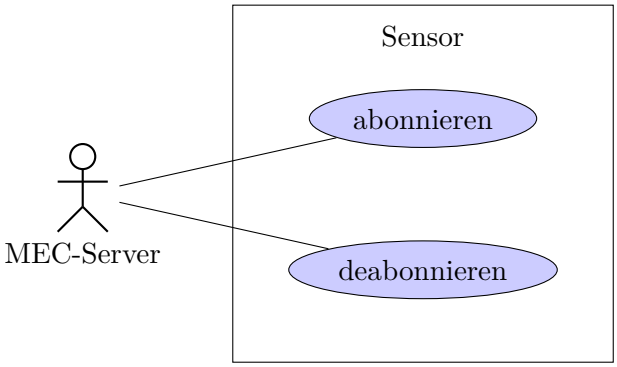


Abbildung 5.3: Anwendungsfalldiagramm des Sensors

5.2.9 Abonnieren

Initiator:	MEC-Server
Beteiligte Akteure:	MEC-Server, Sensor
Vorbedingung:	Ist als Sensor registriert
Basisablauf:	Der MEC-Server kann einen Sensor abonnieren um *SensorFrame*-Nachrichten (siehe **??**) zu erhalten. Ein Abonnement wird durch eine *UpdateSubscription*-Nachricht aktualisiert (siehe **??**).
Alternativablauf:	–
Nachbedingung:	Der MEC-Server erhält *SensorFrame*-Nachrichten.

5.2.10 Deabonnieren

Initiator:	MEC-Server
Beteiligte Akteure:	MEC-Server, Sensor
Vorbedingung:	Ist als Sensor registriert
Basisablauf:	Der MEC-Server kann ein Abonnement aufkündigen um keine weiteren *SensorFrame*-Nachrichten (siehe **??**) zu erhalten. Ein Abonnement wird durch eine *UpdateSubscription*-Nachricht aktualisiert (siehe **??**).
Alternativablauf:	–
Nachbedingung:	Der MEC-Server erhält keine *SensorFrame*-Nachrichten.

5.3 Fahrzeug

In **??** sind Anwendungsfälle aufgezeigt, die durch das Fahrzeug dem MEC-Server angeboten werden.

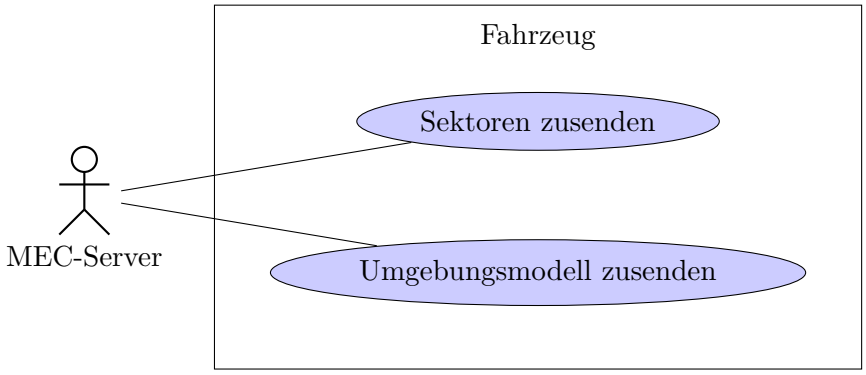


Abbildung 5.4: Anwendungsfalldiagramm des Fahrzeugs

5.3.1 Sektoren zusenden

Initiator:	MEC-Server
Beteiligte Akteure:	MEC-Server, Fahrzeug
Vorbedingung:	Ist als Fahrzeug registriert, Sektoren noch nicht übermittelt
Basisablauf:	Der MEC-Server soll dem Fahrzeug einmalig nach Registrierung alle bekannten Sektoren in einer *InitMessage*-Nachricht (siehe **??**) zusenden.
Alternativablauf:	–
Nachbedingung:	Sektoren können nicht erneut zugesendet werden

5.3.2 Umgebungsmodell zusenden

Initiator:	MEC-Server
Beteiligte Akteure:	MEC-Server, Fahrzeug
Vorbedingung:	Fahrzeug ist abonniert
Basisablauf:	Der MEC-Server kann dem Fahrzeug Umgebungsmodelle mit *EnvironmentFrame*-Nachrichten (siehe **??**) zusenden.
Alternativablauf:	–
Nachbedingung:	–

5.4 Nachrichtenanalyse

Eine ASN.1 Nachricht wird, wie in **??** zu sehen, mit vorangestellten Kopfdaten versendet. Diese Kopfdaten enthalten die Länge der ASN.1 Nachricht und dessen Typ, der bei der Dekodierung beim Empfängers bekannt sein muss. Die Nachrichtenlänge und der Nachrichtentyp werden als 32-Bit lange und vorzeichenlose Ganzzahlen übermittelt und entsprechen damit dem Datentyp `u32` in Rust. Sie werden als „Big-Endian“ auf dem Datenstrom dargestellt.

Die Nachrichtenlänge gibt die Länge der ASN.1 Nachricht in Bytes an.

Für den Nachrichtentyp sind nur die in **??** gelisteten Werte gültig und werden im Anschluss erklärt.

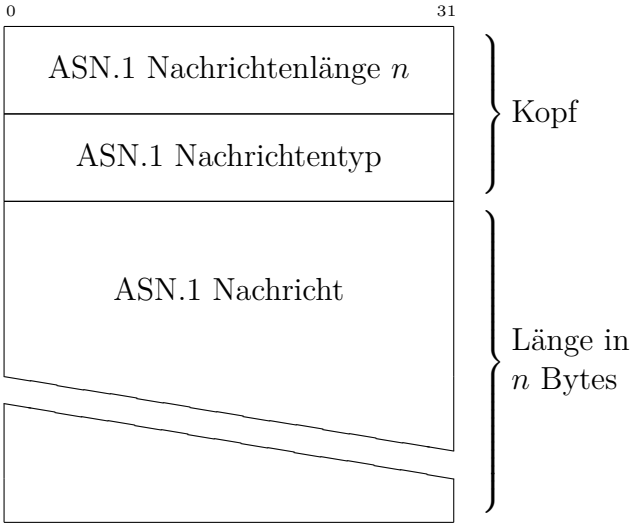


Abbildung 5.5: ASN.1 Nachricht mit Kopfdaten

Typ	Nachricht	Aus Sichtweise des Servers
0	„Nicht definiert“	–
1	ClientRegistration	eingehend
2	SensorFrame	eingehend
3	EnvironmentFrame	ausgehend
4	UpdateSubscription	bidirektional
5	InitMessage	ausgehend
6	RoadClearanceFrame	ausgehend
7	SensorIdleFrame	eingehend

Abbildung 5.6: Gültige ASN.1 Nachrichtentypen

5.4.1 ClientRegistration

Die ClientRegistration-Nachricht sendet der Client nach dem Verbindungsaufbau dem Server zu, um mitzuteilen, ob es sich um einen Sensor oder ein Fahrzeug handelt.

5.4.2 SensorFrame

Die SensorFrame-Nachricht wird vom Sensor an den Server versendet und beschreibt Objekte, die der Sensor erkannt hat.

5.4.3 EnvironmentFrame

Die EnvironmentFrame-Nachricht wird vom Server an das Fahrzeug versendet und enthält das Ergebnis des Fusions-Algorithmus.

5.4.4 UpdateSubscription

Die UpdateSubscription-Nachricht wird vom Server an den Sensor oder vom Fahrzeug an den Server gesendet. Der Server kann sich damit am Sensor an einem SensorFrame-Abonnement anmelden oder abmelden, während das Fahrzeug sich am Server am EnvironmentFrame-Abonnement anmelden oder abmelden kann.

5.4.5 InitMessage

Die InitMessage-Nachricht wird vom Server, nach einer Fahrzeuganmeldung, an das Fahrzeug gesendet und beinhaltet Koordinaten über die Sektoren, die die Sensoren beobachten.

5.4.6 RoadClearanceFrame

Die RoadClearanceFrame-Nachricht wird vom Server an das Fahrzeug versendet und kann Verkehrs-, Wetter- und andere Informationen über die Sektoren enthalten.

5.4.7 SensorIdleFrame

Die SensorIdleFrame-Nachricht wird vom Sensor in regelmäßigen Zeitintervallen an den Server versendet, wenn der Sensor nicht abonniert ist. Die Anwesenheit des Sensors wird somit festgestellt.

5.5 Schnittstellenanalyse

In diesem Kapitel werden die Schnittstellen analysiert, über die der MEC-Server mit dem Sensor und dem Client kommuniziert. Für die Darstellung werden Sequenzdiagramme verwendet. Aktionen in „loop“-Abschnitten werden so lange wiederholt, bis die erste darauf folgende Aktion eintritt.

5.5.1 Sensor und MEC-Server

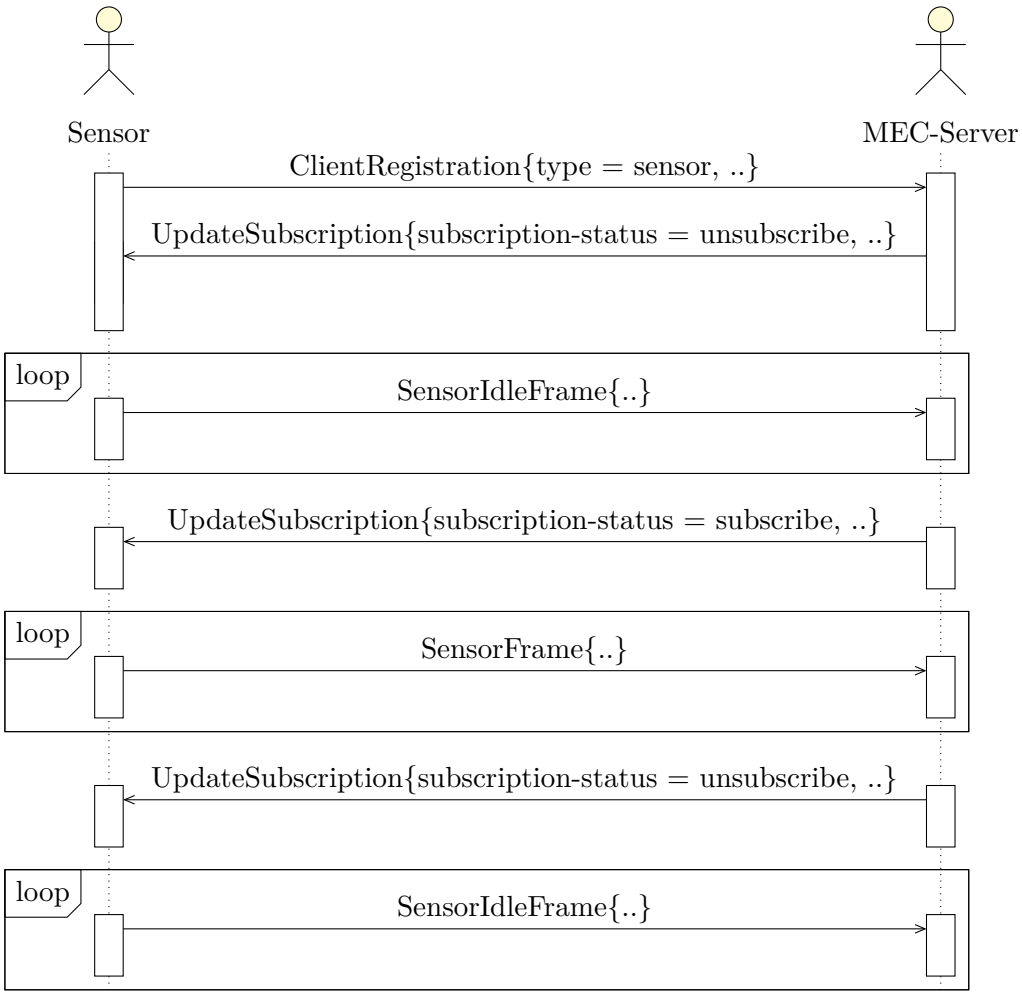


Abbildung 5.7: Kommunikation zwischen Sensor und MEC-Server

In **??** ist die Kommunikation zwischen einem Sensor und dem MEC-Server zu sehen. Nach Verbindungsaufbau versendet der Sensor eine *ClientRegistration*-Nachricht (siehe **??**) um sich am Server als Sensor zu registrieren. Der Server beantwortet dies mit einer *UpdateSubscription*-Nachricht (siehe **??**) um klarzustellen, dass kein Abonnement gewünscht ist. Der Sensor versendet darauf folgend in regelmäßigen Abständen *SensorIdleFrame*-Nachrichten (siehe **??**). Nach unbekannter Zeit kann der Server ein Abonnement erstellen, indem eine entsprechende *UpdateSubscription*-Nachricht versendet wird. Daraufhin sendet der Sensor dem Server anstatt *SensorIdleFrame*-Nachrichten *SensorFrame*-Nachrichten (siehe **??**) zu, bis dieser das Abonnement mit einer *UpdateSubscription*-Nachricht wieder aufkündigt.

5.5.2 Fahrzeug und MEC-Server

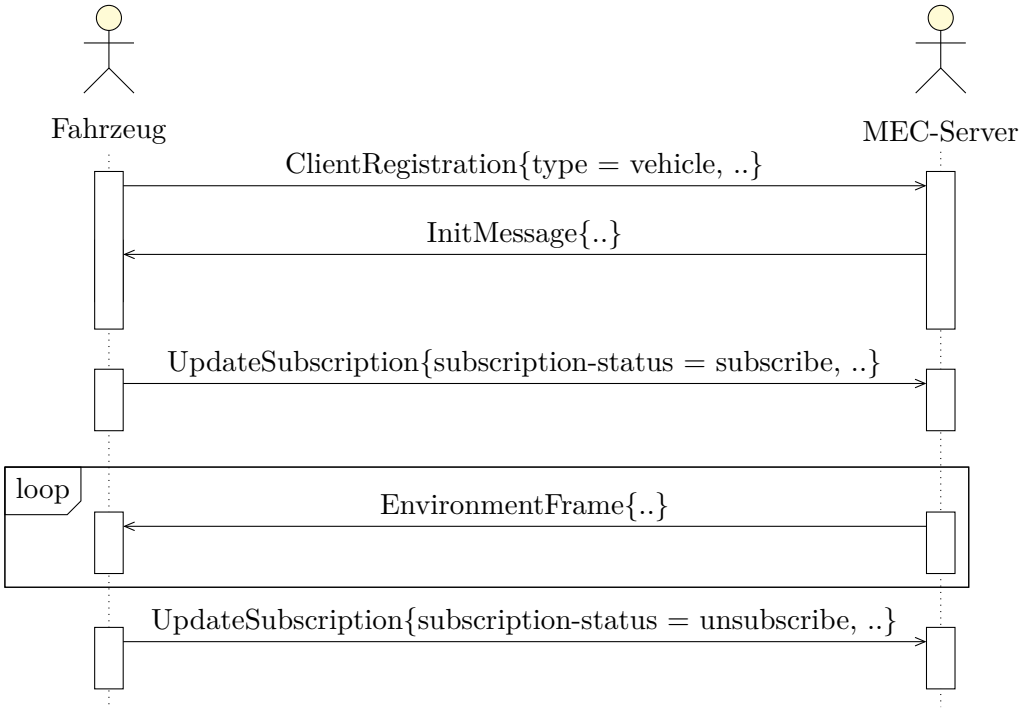


Abbildung 5.8: Kommunikation zwischen Fahrzeug und MEC-Server

In **??** ist die Kommunikation zwischen einem Fahrzeug und dem MEC-Server zu sehen. Nach Verbindungsaufbau versendet das Fahrzeug eine *ClientRegistration*-Nachricht um sich am Server als Fahrzeug zu registrieren. Der Server beantwortet dies mit einer *InitMessage*-Nachricht (siehe **??**) mit Informationen zu bekannten Sektoren. Bis das Fahrzeug ein Abonnement mittels einer *UpdateSubscription*-Nachricht erstellt, werden keine weiteren Nachrichten ausgetauscht. Während eines gültigen Abonnements sendet der Server Umfeldmodelle in *EnvironmentFrame*-Nachrichten (siehe **??**) an das Fahrzeug. Durch eine *UpdateSubscription*-Nachricht kann das Fahrzeug jederzeit das Abonnement aufkündigen und erhält daraufhin keine weiteren *EnvironmentFrame*-Nachrichten.

5.5.3 Sensor, Fahrzeug und MEC-Server

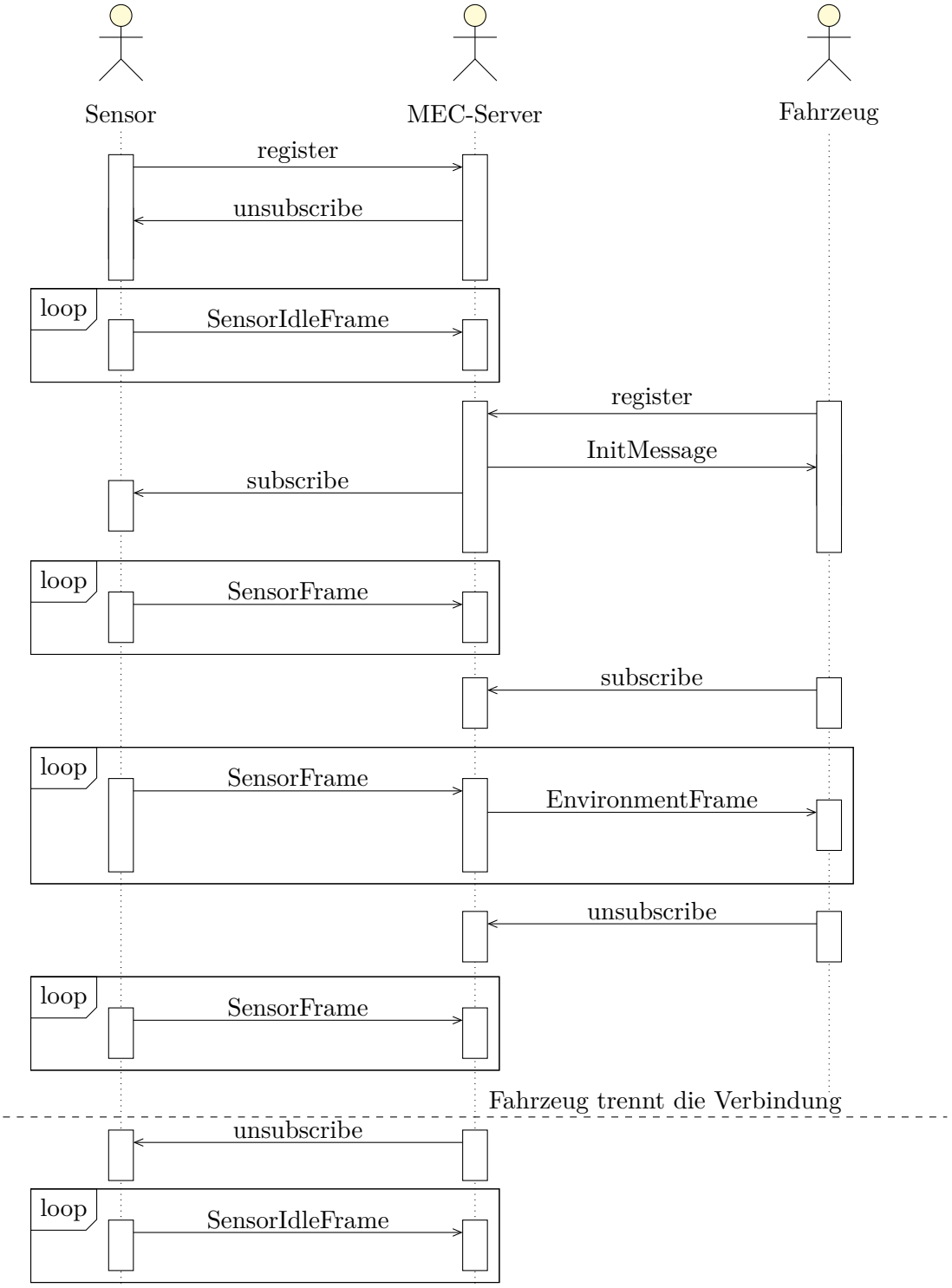


Abbildung 5.9: Interaktion des MEC-Servers mit einem Fahrzeug und einem Sensor

In **??** ist das Zusammenspiel des Sensor aus **??**, des Fahrzeugs aus **??** und des MEC-Servers zu sehen. Um die Übersichtlichkeit zu wahren, wurden die Nachrichtentypen durch Anweisungen ersetzt. Es ist zu erkennen, das der Server bereits ein Abonnement gegenüber dem Sensor erstellt, sobald sich ein Fahrzeug registriert. Dieses Abonnement ist so lange gültig, wie zumindest ein Fahrzeug verbunden ist.

6 Systementwurf

In diesem Kapitel wird eine Entwurf für die Implementation des Systems aufgezeigt. Die Struktur des Systems wird durch einen Architekturentwurf visualisiert und die zu verwendeten Technologien erläutert.

6.1 Kein ASN.1 Compiler für Rust

Eine frühe Recherche hat ergeben, dass kein ASN.1 Compiler für Rust verfügbar ist. Mit *yasna*, *raisin* und *rust-asn1* wird teilweise ASN.1 Funktionalität in Rust zur Verfügung gestellt, aber keine unterstützt die benötigte UPER Codierung. Auch kann keine der genannten Crates die in ASN.1 Notation definierten Nachrichten in native Rust Datenstrukturen übersetzten. Die von der ITU genannten Compiler[1] unterstützen Rust auch nicht. Dieser Umstand erzwingt die Nutzung der C-Datenstrukturen, wie die Referenzimplementierung, mittels Foreign Function Interface (siehe **??**). Zur besseren Kapselung ist die Einbindung der ASN.1 Nachrichten deswegen in mehreren Schritten in zwei Crates ausgelagert.

6.2 Framework Tokio

Das Framework Tokio wird als Grundlage für die Umsetzung der Architektur genutzt. Tokio[2] bietet eine „Laufzeit zum schreiben von zuverlässigen, asynchrone und schlanke Anwendungen in Rust“ [**rust:crate:tokio**].

Tokio nutzt die Crates *mio* und *Future* um eine ereignisorientierte, asynchrone und daher nicht blockierende Ein-/Ausgabeplattform zu bieten. Bearbeitungsschritte sind in „Futures“ (dt. Zukunft, hier in etwa: „ein zukünftiger Wert“) aneinandergereiht und verpackt.

Neben den genannten technischen Funktionen ist hervorzuheben, dass das Framework von vielen Entwicklern aus dem Rust-Entwicklerteam mindestens mitentwickelt wird[3]. Aufgrund dessen ist ein hoher Grad an Qualität und Aktualität des Frameworks zu erwarten.

[1] https://www.itu.int/en/ITU-T/asn1/Pages/Tools.aspx
[2] https://crates.io/crates/tokio
[3] u.a. Alex Crichton, Carl Lerche, Aaron Turon: https://github.com/tokio-rs/tokio/graphs/contributors

6.3 Architektur

In **??** ist der Entwurf der Architektur visualisiert. Blau markierte Objekte haben als Aufgabengebiet die Kommunikation, grün markierte Objekte bilden die Geschäftslogik ab und rot markierte Objekte sind Spezialisierungen für ASN.1.

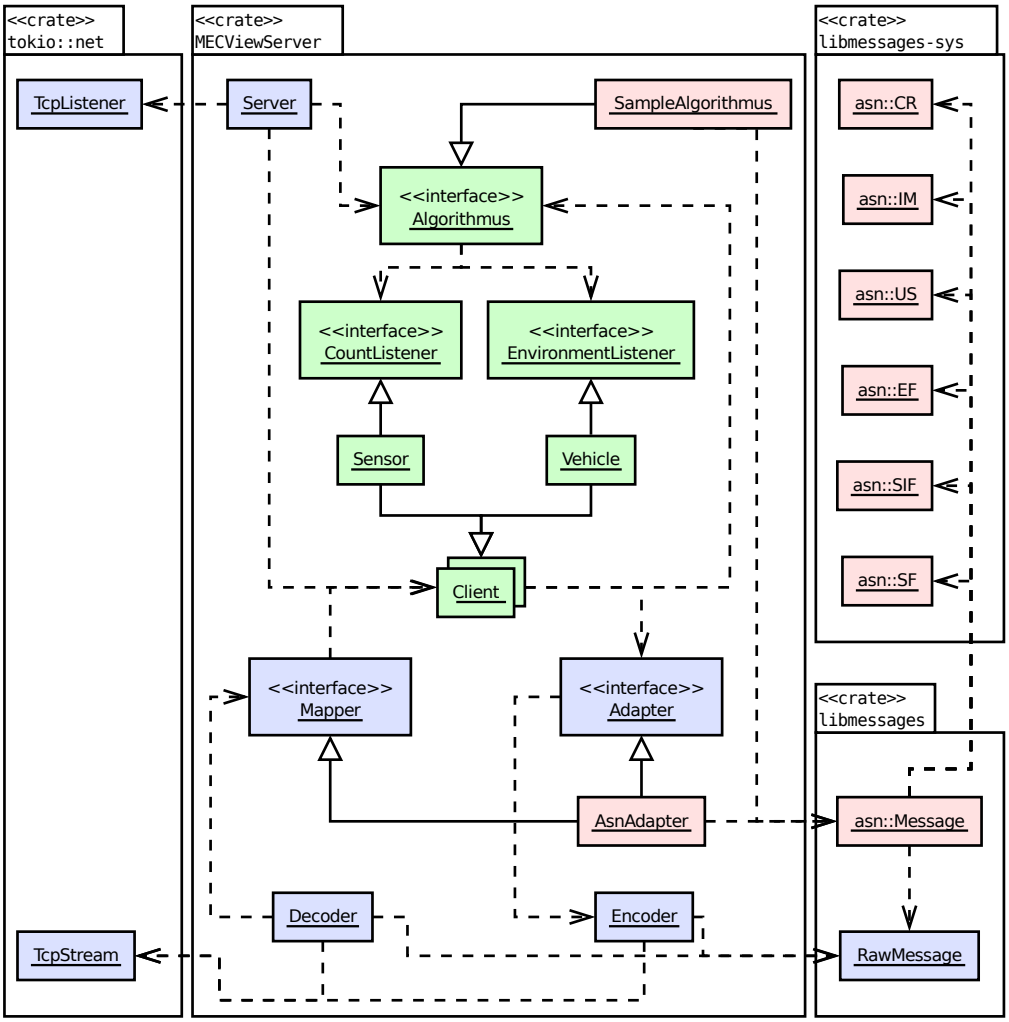


Abbildung 6.1: Architekturentwurf

Die Implementation ist in drei Crates aufgeteilt:

- **libmessages-sys**: Enthält bindings für die ASN.1 Datenstrukturen und Funktionen der C-Bibliothek (siehe **??**). Unsichere Funktionsaufrufe nach C, wie zur Serialisierung und Deserialisierung, sind durch sichere Rust Funktionen gekapselt. Eine ordnungsgemäße Allokation und Deallokation der Nachrichten wird hier sichergestellt.

- **libmessages**: Stellt Datenstrukturen und Implementationen bereit um unabhängig von der eingesetzten Nachrichtentechnologie eine Nachricht abbilden zu können. Hierzu wird in *RawMessage* der Kopf (siehe **??**) und der binären Inhalt einer Nachricht dargestellt.

  Zusätzlich ist in dieser Crate für jede Verwendete Technologie (in dieser Bachelorarbeit nur ASN.1) die Wandlung zwischen einer *RawMessage* und einer entsprechenden Datenstruktur implementiert sein.

- **MECViewServer**: Enthält die Logik für den Sensor, das Fahrzeug und den Algorithmus und bündelt sie mit der Kommunikationslogik zu einem ausführbaren Kompilat.

Erklärung Server

Der Server lädt zu beginn eine Algorithmus Implementation, öffnet einen TCP-Port und wartet auf dem daraus resultierenden *TcpListener* auf neue Clients. Jedem neuen Client wird eine Referenz auf den Algorithmus mitgegeben.

Erklärung Client, Sensor und Vehicle

Der Client hält eine Referenz auf einen Algorithmus und einen Adapter. Im Falle eines *Vehicle*s kann ein *EnvironmentListener* am Algorithmus registriert werden um neue Umgebungsmodelle zu erhalten. Im Falle eines *Sensor*s kann ein *CountListener* am Algorithmus registriert werden um über Änderungen bei der Anzahl der registrierten Fahrzeuge informiert zu werden.

Erklärung Adapter

Der Adapter bietet Schnittstelle, damit eine Clientinstanz Nachrichten versenden kann, ohne die eingesetzte Nachrichtentechnologie zu kennen. Hierzu wird der Clientinstanz Funktionen bereitgestellt, die von der Adapterimplementation in entsprechende Nachrichten übersetzt werden.

Erklärung Mapper

Der Mapper ruft für Empfangene Nachrichten der eingesetzten Nachrichtentechnologie Funktionen der Clientinstanz auf und agiert somit als Gegenstück zum Adapter.

Erklärung Encoder / Decoder

Der Encoder schreibt *RawMessage*s auf den TCP-Datenstrom, während der Encoder *RawMessage*s vom TCP-Datenstrom liest. Hierzu wird eine Wandlung, wie in **??** beschrieben, vorgenommen.

Nachrichtentechnologie kapseln

Innerhalb der MECViewServer Crate sollen möglichst wenige Klassen die eingesetzt Nachrichtentechnologie kennen. Hierzu zählen die Algorithmus- (*SampleAlgorithmus*), Mapper- und Adapterimplementationen (*AsnAdapter*). Dies ist unumgänglich, weil sowohl der eingesetzte Algorithmus, der Mapper und der Adapter Felder einer Nachricht sowohl lesen als auch schreiben muss. Zudem muss der Server mindestens indirekt die eingesetzte Technologie kennen, um die richtigen und zueinander kompatiblen Algorithmus-, Mapper- und Adapterimplementation zu instantiieren. Weitere Klassen agieren dagegen unabhängig von der eingesetzten Nachrichtentechnologie.

6.3.1 Kommunikationsarchitektur

Um ein Mehrkernsystem effizient zu nutzen, arbeiten einige Klassen asynchron zueinander. Hierzu zählt der Server, der fortlaufend neue Verbindungen annimmt. Die daraus resultierenden Clientinstanzen bearbeiteten asynchron eingehende Anfragen. Letztendlich aktualisiert auch der Algorithmus asynchron zu den Clientinstanzen und dem Server das Umfeldmodell.

Die Kommunikation zwischen einem TCP-Datenstrom und einer Clientinstanz soll mittels des „Channel Architektur Pattern“ [**douglass2003real**] verwirklicht werden. Das Muster eignet sich besonders gut, da auf eingehende und ausgehende Datensätze eine immer gleiche Transformation angewandt werden muss:

?? beschreibt die nötigen Transformationen bis eine Anfrage bearbeitet werden kann und die nötigen Transformationen um eine Antwort zu versenden:

- **Daten empfangen**: Die eingehenden Daten (hier Bytes) müssen angesammelt werden, bis sie eine komplette Anfrage abbilden.
- **RawMessage dekodieren**: Die angesammelten Daten werden in eine RawMessage gewandelt. Nachrichtentyp und Korpus werden hierbei extrahiert.
- **asn::Message dekodieren**: Aus dem Korpus wird eine ASN-Nachricht des entsprechenden Typs dekodiert.
- **Anfrage zuordnen**: Die ASN-Nachricht wird einer Anfrage zugeordnet.
- **Anfrage berabeiten**: Die Anfrage wird bearbeitet und eine Antwort generiert.

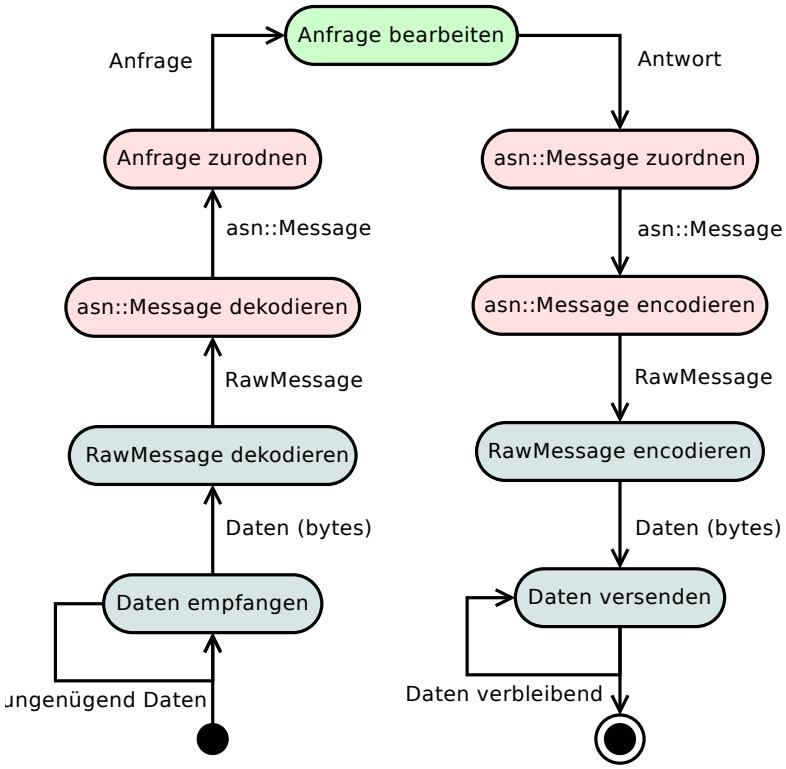


Abbildung 6.2: „Channel Architektur Pattern“ zwischen einem TCP-Datenstrom (unten) und einer Clientinstanz (oben). Zu lesen von links unten, nach oben, nach rechts unten.

- **asn::Message zuordnen**: Die Antwort wird eine ASN-Nachricht eingebettet.
- **asn::Message encodieren**: Die ASN-Nachricht wird in eine RawMessage encodiert. Der Nachrichtentyp wird entsprechend gesetzt.
- **RawMessage encodieren**: Die RawMessage wird in Bytes gewandelt.
- **Daten versenden**: Die Bytes werden über den TCP-Datenstrom versandt.

Nach dem „Channel Architektur Pattern“ ist sequentielle Bearbeitung zwischen dem TCP-Datenstrom und der Clientinstanz in jede Richtung als Kanal interpretierbar. Da jeder Zwischenschritt innerhalb eines Kanals lediglich von dem Ergebnis des vorhergehenden abhängig ist, können die einzelnen Schritte parallel zueinander bearbeiten werden. TODO: Durch einen zusätzlichen, vorangestellten Multiplexer und nachfolgenden Demultiplexer könnten in einer späteren Version mehrere Kanäle parallel den Client oder Datenstrom speisen. Hiervon wird in der ersten Implementierung jedoch abgesehen, da je Client keine derart große Datenflut erwartet wird, die dies verlangen würde. Stattdessen soll lediglich jeder Kanal eines Clients parallel zum dazugehörigen Client und zu allen anderen Kanälen ausgeführt werden.

Tokio (siehe **??**) ermöglicht durch die Kombinationsmöglichkeiten von *Futures* nicht blockierenden Kommunikationskanäle nach diesem Prinzip zu erstellen und zu betreiben.

Ausgliederung von libmessages weil ASN-Bindings + weil von Protokolländerungen unabhängig sein soll

Prüfen tokio, queues, async + proxy pattern

Abbildung vereinfacht, Kommunikation via Queues im tokio context

[**goll2012methoden**], example [**goll2012methoden**]

TODO: Component diagrams tikz-uml: server, algorithm, encoder/...

6.4 Sequenzdiagramme

6.4.1 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?

TODO: Hochperformant -> parallel?

TODO: Design Pattern, Gamma et al, four important aspects

TODO: Real Time Design Patterns Buch: Ab Seite 141, verschiedene Systempatterns, microkernel [**douglass2003real**]? channel architektur pattern [**douglass2003real**]?

TODO: Message Queuing Pattern [**douglass2003real**]

TODO: kein hartes Echtzeitsystem, gerade mal weiches Echtzeitsystem **??**

TODO: Clean Architecture / Clean Code

7 Implementierung

In diesem Kapitel wird auf Besonderheiten bei der Implementierung eingegangen.

TODO: diagramm message decode steps

TODO: proxy communication

7.1 Bindgen für ASN

TODO: schnell problem: kein asn->rs compiler, c bindings aufwending -> autogen via bindgen

TODO: link issues fixed in commit d5d694c

TODO: impl Drop / free structs, important but do not overengineer

7.1.1 Continuous Integration mittels Jenkins

Mittels der Continuous Integration Software Jenkins wird die Rust-Implementation nach jeder Aktualisierung der Quellcode-Repository gebaut. In einem Jenkinsfile ist hierzu die folgende Pipeline definiert:

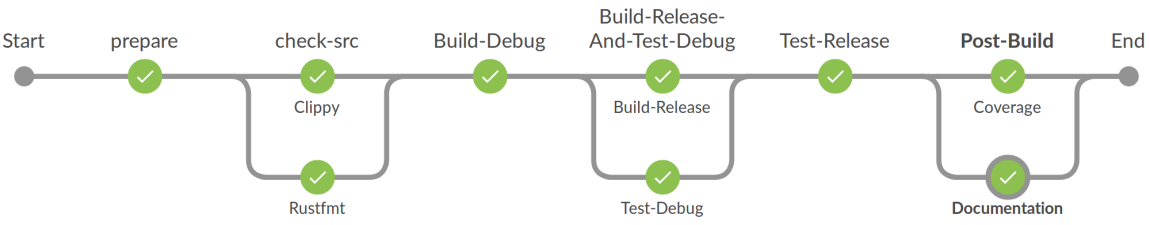


Abbildung 7.1: Jenkins Pipeline Graph

Im folgenden sind die einzelnen Schritte genauer erklärt.

Schritt „prepare“

In diesem Schritt wird überprüft, ob die richtige Version von Rust installiert ist. Sowohl der Rust Compiler als auch Abhängigkeiten wie Cargo, Clippy oder Tarpaulin werden falls nötig in diesem Schritt nachinstalliert und aktualisiert.

Schritt „check-src“

Hier wird der Quellcode auf einen schlechtem Programmierstil oder schlechte Programmierpraktiken durch Clippy und auf eine nicht standardkonforme Formatierung überprüft. Ein erkannter Mangel resultiert in einem abgebrochenem Bauversuch und einem entsprechendem roten Icon auf der Projektseite auf Jenkins. Ein Quellcode, der diese Qualitätsstandards nicht erfüllt, wird somit schnellstmöglich abgelehnt.

Schritt „Build-Debug“

In diesem Schritt wird das Projekt und Abhängigkeiten im Debug-Modus kompiliert. In einem Debug-Modus werden nahezu keine Optimierungen durchgeführt, weswegen die Ausführgeschwindigkeit meist mangelhaft ist. Im falle eines Fehlers in einem Unit-Test, können in aus einem Debug-Artefakt jedoch nützlich Informationen gewonnen werden.

Schritt „Build-Release-And-Test-Debug“

Wie der Name bereits vermuten lässt, wird in diesem Schritt das Projekt im Release-Modus übersetzt. Gleichzeitig wird das zuvor erstellte Debug-Artefakt auf Fehler überprüft, indem Unit-Tests ausgeführt werden. Ein Release-Artefakt ist stark optimiert und ist für den Produktivbetrieb gedacht.

Schritt „Test-Release“

In diesem Schritt wird das Release-Artefakt mittels Unit-Tests auf Fehler überprüft. Hierdurch soll sichergestellt werden, dass kein Fehler durch Optimierungen des Compilers aufgedeckt oder verursacht wurden. TODO: . Auch soll kein Artefakt für den Produktivbetrieb empfohlen werden, das nicht auf Fehler überprüft wurde.

Schritt „Post-Build“

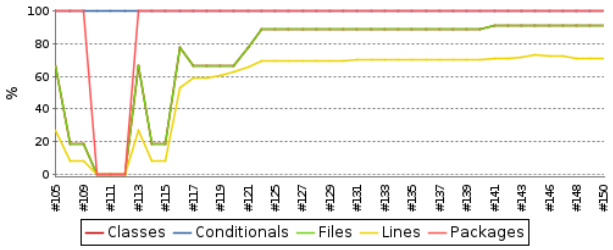
Dieser Schritt dient zur Zusammenstellung der Testabdeckung der Unit-Tests und der automatisierten Erstellung der Dokumentation.

Ergebnis des erfolgreichen Durchlaufs

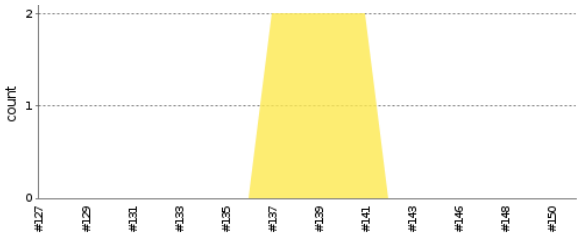
Jede Aktualisierung der Quellcoderepository resultiert bei erfolgreicher Kompilation und durchlaufen der Tests in den folgenden Artefakten:

- **Debug-Artefakt**: Kompilation für eine manuelle Fehlersuche mit vielen auf den Quellcode bezogenen Symbolen und ohne Optimierungen durch den Compiler
- **Release-Artefakt**: Kompilat mit Optimierungen für den Produktivbetrieb
- **TODO: Unit-Test-Abdeckungsbericht**: Bericht über wie viele Zeilen, Dateien, Klassen und Verzweigungen durch die Unit-Tests getestet wurden.
- **Dokumentation**: TODO: .

Für jeden erfolgreichen Bauversuch, werden die erhobenen Informationen in die folgenden Graphen auf der Projektseite auf Jenkins übernommen:



(a) Jekins Coverage Graph



(b) Jekins Warnings Graph

?? zeigt die Testabdeckung seit Aufzeichnungsbeginn. Die frühen und starken Schwankungen resultieren aus Testdurchläufen bei der Einrichtung des Graphen. **??** zeigt Warnungen die der sehr TODO: .. Rust Compiler für die verschiedenen Builds ausgegeben hat.

7.2 ...

TODO: tdd schwierig weil viel integration mit framework / async, schnell wird daraus integrationstest(?, unerwünscht)

TODO: rust-clippy: https://github.com/rust-lang-nursery/rust-clippy

7.3 Strategien für Performance

TODO: „bypass“ for algorithm-»many vehicles

TODO: diagramm: jede future ein eigenes paket", queues dazwischen für kommunikation

TODO: catch panic? mention how in **??**

TODO: SRP, impl Object separat from impl CommandProcessor for Object, testability?

7.3.1 Tokio

7.3.2 Generalisierung mittels Aufzählung für nicht erweiterbare Anzahl von Elementen

7.3.3 libmessages make unsafe libmessages-sys safe

7.3.4 Vorgehen, bindgen tests, C/unsafe/wrapper -> nach sicher -> architektur entwickeln

7.3.5 Unerwartete Schwierigkeiten

TODO: Trailing zeroes issue, commit a02496d + tagged, libmessages/src/asn.rs:17 bzw :31 bzw :32 Ok((result.encoded as usize + 7) / 8

TODO: unterschiedlicher Heap libc / rust

TODO: jenkins clippy nightly often broken

7.3.6 –help

TODO: bei der entwicklung nie segfault/deadlock gehabt, nur einmal heap problem wegen libc

7.4 Valgrind: Race Conditions all over the place

8 Auswertung

In diesem Kapitel wird die Implementation des MEC-View-Servers in Rust mit der Referenzimplementation in C++ verglichen. Hierzu wird die Textumgebung aus dem C++ Projekt auch auf die Rust Implementation angewandt: Fahrzeuge und Sensoren werden simuliert.

test für durchsatz nur bedingt aussagekräftig

gleicher dummy alg

TODO: Während den tests zeigte sich ein Fehler im C++ Server. Race-conditional missing cleanup for closed sockets in certain error cases?

8.1 Testumgebung

Alle Tests wurden auf dem selben Ubuntu 16.04.4 LTS Server mit zwei Intel Xeon CPUs (E5-2620 v4 @ 2.10GHz) ausgeführt. Der CPU Gvernor wurde für alle Tests auf „performance“ gestellt, um Schwankungen in der CPU Taktung durch Energiesparmaßnahmen und damit Verfälschungen der Testergebnisse zu unterbinden.

Für jede Kombination aus Sensoren, Fahrzeugen und Serverimplementierung wurde 10 Tests durchgeführt. Während jedem Test ist für jede Empfangene Nachricht am Fahrzeug, die Latenz seit Versand am Sensor in eine Datei geschrieben worden. Nach einem Testdurchlauf wurden die Anzahl an Nachrichten gezählt und der Durchschnitt und Boxplots für die Latenzen gebildet.

Zu jedem Zeitpunkt ist maximal ein Test gleichzeitig auf dem System ausgeführt worden.

Um Latenzen durch das Netzwerk auszuschließen, wurden auch die virtuellen Sensoren und Fahrzeuge auf dem gleichen Server wie die Serverimplementation ausgeführt und durch die Loopback-Netzwerkschnittstelle mit der Serverimplementation verbunden. Ein kurzer Test ergab hierbei einen Durchsatz von ca 40Gbit/s (mittels *iperf*) und eine Latenz von ca 24us (mittels *ping*) auf der Loopback-Netzwerkschnittstelle. Auswirkungen auf die Testergebnisse sind damit vernachlässigbar klein.

Beide Implementationen sind im Release-Modus und ohne „RoadClearanceModule“ compiliert worden.

TODO: no RCM

TODO: release mode

8.2 Vorgehen

Das Vorgehen war für alle Tests gleich und wurde bis auf den Start des Servers durch ein Skript ausgeführt. Variiert wurden lediglich die Anzahl an Fahrzeugen und Sensoren, die als Übergabeparameter an das Skript übergeben werden. Das Vorgehen eines Testdurchlaufs sieht folgendermaßen aus:

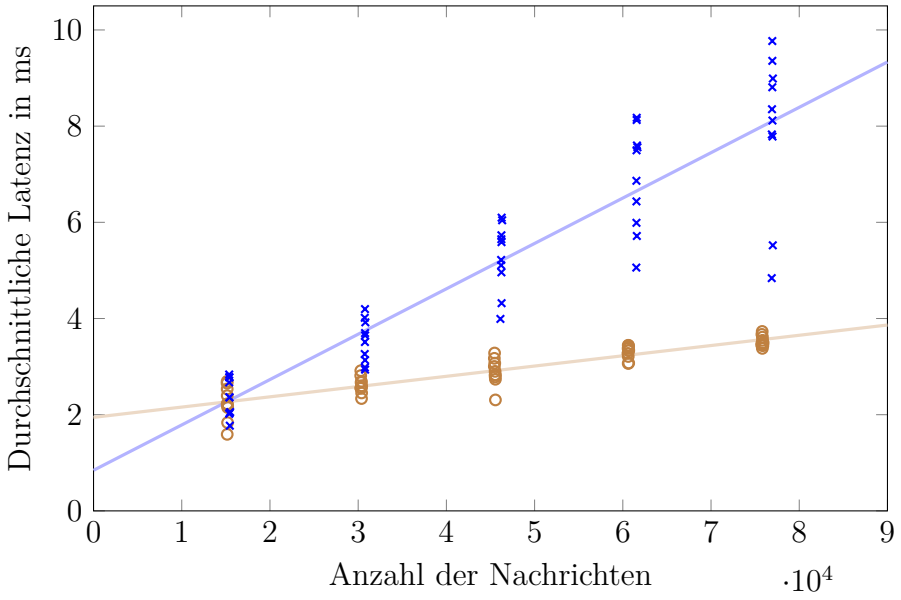
- Server mit Log-Level „info“ starten, falls noch nicht gestartet
- Sensoren starten
- Eine kurze Pause (2 Sekunden) machen, um genügend Zeit für den Verbindungsaufbau der Sensoren zu geben
- Fahrzeugen starten. 5 Sekunden nach dem Verbindungsaufbau abonnieren diese das Umgebungsmodell für 60 Sekunden.
- Auf ende aller Fahrzeuge warten.
- Alle Sensoren beenden
- Kurze Pause (2 Sekunden) vor erneutem Test-Durchlauf einräumen

Für jedes Umgebungsmodell, das von einem Fahrzeug empfangen wird, wird auf der Konsole die Latenz in ganzen Millisekunden ausgegeben.

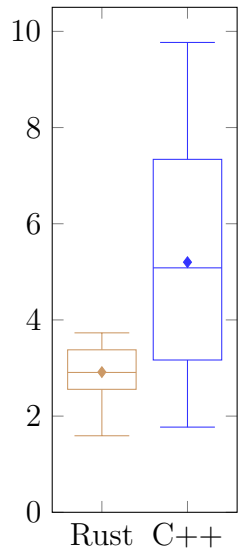
Die Latenz errechnet der C++ Clientsimulator, indem beim Versenden eines *SensorFrames* durch den simulierten Sensor die aktuelle Zeit in der Nachricht hinterlegt wird. Der Dummy-Algorithmus überträgt diesen TODO: Timestamp in das *EnvironmentFrame* und versendet dieses an die Fahrzeuge. Die simulierten Fahrzeuge errechnen die Differenz zwischen der aktuellen Zeit und dem TODO: timestamp und geben diesen auf der Konsole aus.

Alle ausgegebenen Latenzen werden mittels *bash*-Pipe in eine Datei übertragen. Bei der Auswertung wird der Durchschnitt errechnet.

8.3 Ergebnisse



(a) Latenz-Nachrichtenanzahl Diagramm



(b) Boxplot für die Latenz

8.3.1 profiler?

9 Zusammenfassung und Fazit

TODO: war rust toll?

TODO: Eigentümerprinzip gewöhnungsbedürftig, learning curve, einem wird sehr deutlich unter die nase gehalt, wo etwas performance kostet

TODO: messen ist nicht einfach

Abkürzungsverzeichnis

ASN.1 Abstract Syntax Notation One. 43

BMWi Bundesministerium für Wirtschaft und Energie. 2

GC Garbage Collector. 26, 37

ITU International Telecommunication Union. 43

MEC Mobile Edge Computing. 2, 42, 43

Abbildungsverzeichnis

Listings