Hochschule Esslingen
University of Applied Sciences

Fakultät Informationstechnik
im Studiengang Softwaretechnik und Medieninformatik

Bachelorarbeit

Entwurf und Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens in Rust

Michael Watzko

Sommersemester 2018
14.02.2018 - 22.06.2018

Erstprüfer: Prof. Dr. rer. nat. Dipl.-Inform. Manfred Dausmann
Zweitprüfer: ... Hannes Todenhagen

Firma: IT Designers GmbH
Betreuer: Dipl. Ing. (FH) Kevin Erath M.Sc.

[git] • master @ 0395152 • 19. Februar 2018

Sperrvermerk

U SHALL NOT PASS

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 19. Februar 2018 _____

Michael Watzko

Danksagungen

„Alle Zitate aus dem Internet sind wahr!“

Albert Einstein

„Rust is a vampire language, it does not reflect at all!“

https://www.youtube.com/watch?v=-Tj8Q12DaEQ

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Tesla Autos einen allgemeinen Bekanntheitsgrad erreicht. Um ein Auto selbstständig fahren lassen zu können, müssen erst viele Hürden gemeistert werden, zum Beispiel das Spur halten, auch bei fehlenden Fahrspurmarkierungen, das Interpretieren von Stoppschildern und navigieren durch komplexen Kreuzungen. TODO: ref tesla.com?

Bevor das Auto Entscheidungen treffen kann, muss es zuallererst ein Modell seines Umfelds erstellen oder zur Verfügung gestellt bekommen. Aber vielleicht kann ein Auto nicht immer selbständig genügend Informationen zu seinem Umfeld sammeln? TODO: (huhuhu Server implied huhuhu) TODO: fix 404

1.2 Projektkontext

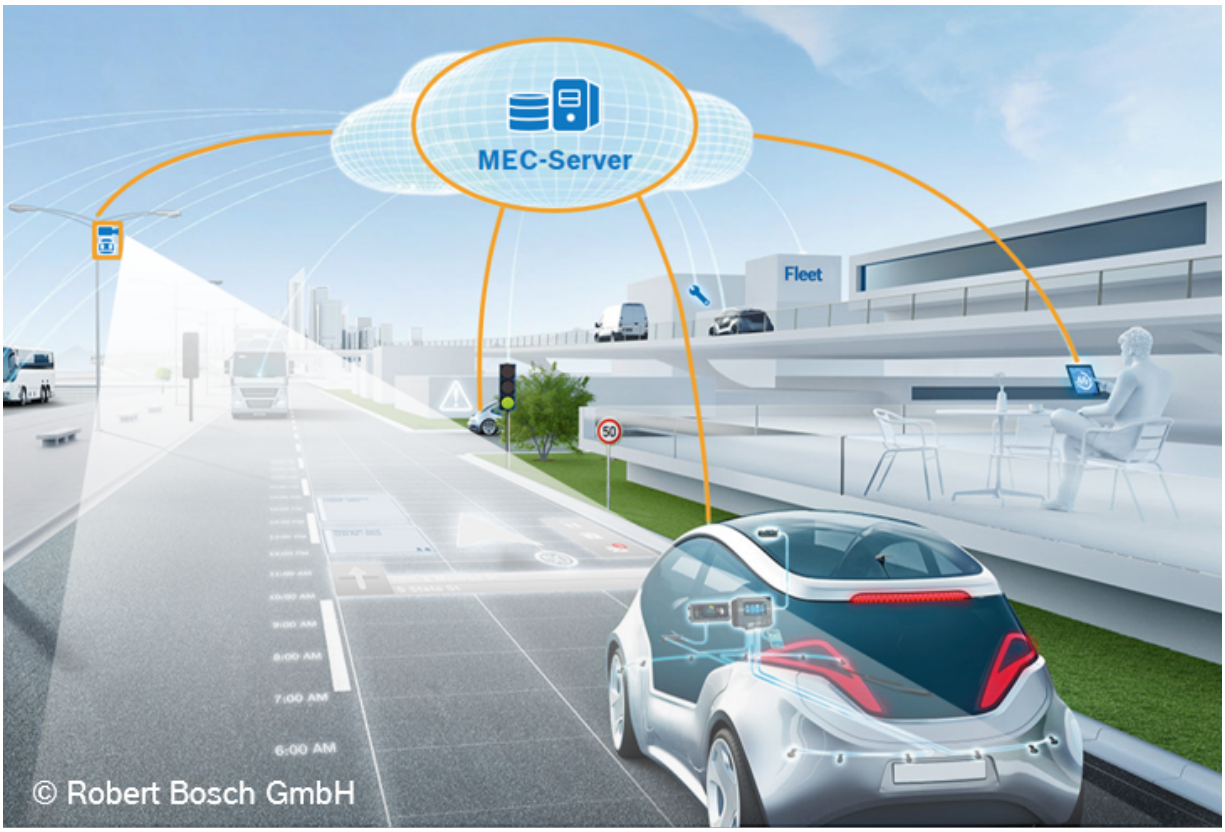

Abbildung 1.1: MEC-View Schaubild der Robert Bosch GmbH [9]

Quelle: https://www.uni-due.de/~hp0309/images/Schaubild_BoschStyle_V2.png

BMWi TODO: Forschungsprojekt MECView, BMWi oder so

1.3 Zielsetzung

Das Ziel ist es, eine alternative Implementierung des MEC-View Servers in Rust zu schaffen. Durch die Garantien TODO: ref von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert wird und somit eine fehlertolerantere und sicherere Implementation geschaffen wird. TODO: möglichst Beibehalt der Architektur?

1.4 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 TODO: ref, Garantien TODO: ref und Sprachfeatures TODO: ref, zum anderen die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen TODO: ref und dem Systemkontext in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext in dem das System betrieben werden soll genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt sowie eine Übersicht von Systemen mit denen 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. Auf architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt. TODO: michael.write_more();

2 Die Programmiersprache Rust

Rust ist eine Programmiersprache, die versucht performant – und daher durch Abstraktionen mit keinem zusätzlichen „Kosten“ TODO: ref zero cost abstractions – sichere Programmierung zu ermöglichen. Ziel ist eine TODO: Systemprogrammiersprache, die sowohl sicher TODO: cite chapter als auch performant ist und ohne eine Laufzeit ausgeführt werden kann. Verschiedene Fehlerquellen – wie „dangling pointers“, „double free“ oder „memory leaks“ TODO: ref – werden durch Abstraktionen und mit Hilfe des Kompilers verhindert. Anders als Programmiersprachen, die dies mit Hilfe einer Laufzeit ermöglichen (zbsp. Java oder C#), wird dies in Rust durch eine statische Analyse und einem Eigentümerprinzip bei der Kompilation gewährleistet.

2.1 Geschichte

In 2006 [14] begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Code zu schreiben. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“. [11]

Auch Federico Mena-Quintero – Mitbegründer des Gnome projekts TODO: cite `https://people.gnome.org/~federico/` or so – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [5]. In Vorträgen TODO: nix mehrzahl? vermittelt er seither, wie Bibliotheken durch Implementierungen in Rust ersetzt werden können [10].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, da einfache Tests und die Kernprinzipien demonstriert werden konnten. Die Entwicklung findet dabei öffentlich einsehbar auf GitHub unter `https://github.com/rust-lang/rust` statt und wird dabei nicht ausschließlich von Mozilla Angestellten koordiniert. Die Stabilität des Kompilers trotz hoher Flexibilität während der Entwicklung wird dabei durch Unterscheidung von drei Veröffentlichungskanälen – release, stable und nightly – in Kombination mit automatisierten Tests TODO: ref? gewährleistet. [14]

TODO: hobbyprojekt, mozilla, open-source, Entwicklung auf GitHub - jeder kann sich beteiligen, test(coverage), automatisierte builds, stable/beta/nightly

2.2 Ökosystem

2.2.1 Ganz ohne

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

Abbildung 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

TODO: files.rs, (nested)modules, compiler, use, function, macro

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für Ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager Cargo (Unterabschnitt 2.2.2) eine solche Benennung so als Standardkonvention erwartet, kann bei manueller Nutzung des Kompilers auch ein anderer Name für die Quelldatei vergeben werden.

Die Kompilierung nutzt TODO: diese Datei als Wurzel, in der Module durch `mod module;` und Quelldateien durch `mod functionality;` „inkludiert“ werden können. Eine Datei *mod.rs* ist die Wurzeldatei eines Moduls.

Rust nutzt den LLVM[1]-Kompiler und erbt daher auch eine große Anzahl an Zielplattformen für die Rust kompiliert werden kann. Es wird aber zwischen drei Stufen unterschieden, bei denen verschieden stark ausgeprägte Garantien vergeben sind. Es wird zwischen „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64), „Stufe 2: Kompiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und „Stufe 3“ (u. a. Thumb) unterschieden [15]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [4]).

2.2.2 Mit Cargo

TODO: dependencies

TODO: Cargo init –bin <name> TODO: missing .gitignore / .git mention / git alltogether TODO: Cargo.toml TODO: [crates.io]

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

Abbildung 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

[1] Früher „Low Level Virtual Machine“ [16], heute Eigenname; ist eine „Ansammlung von Modularen und Wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [12]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [13].

2.3 Hello World

TODO: official format/naming convetion

2.4 Rust als funktionale Programmiersprache

TODO: functional programming -> no global state, no exceptions, find literature TODO: prove via code

2.5 Rust als Objekt-Orientierte Programmiersprache

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

Abbildung 2.3: „Hello World“ in Rust

TODO: prove via design patterns, a few? from faq:: Is Rust object oriented? It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but not everything, and not always using the same abstraction you're accustomed to.

2.6 Versprechen von Rust

2.6.1 Zero Cost Abstraction

2.6.2 Kein undefiniertes Verhalten

TODO: ref oreilly

2.6.3 Kein Null-Pointer

TODO: explain option

2.6.4 Kein vergessene Fehlerprüfung

TODO: explain result

2.6.5 No dangling pointer

TODO: src https://www.youtube.com/watch?v=d1uraoHM8Gg

2.6.6 Statische Speicher- und Lebenszeitanalyse

TODO: while compiling, does not compile on error / unprovable code

2.6.7 type safety langauge

Rust ist...

TODO: Rust -> MIR -> assembler
TODO: MIR/assemblerbeispiele?
[6]

2.7 Warum Rust?

> *„[..]Leute, die [..] sichere Programmierung haben wollen, [..] können das bei Rust haben, ohne die [von D] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen. “* [7]

> *„It's not bad programmers, it's that C is a hostile language“* (Seite 54, [10])

> *„I'm thinking that C is actively hostile to writing and maintaining reliable code“* (Seite 129, [10])

> *„[..] Rust makes it safe, and provides nice tools“* (Seite 130, [10])

> *„Rust hilft beim Fehlervermeiden“* [5]

> *„Rust is [..] a language that cares about very tight control“* [3]

TODO: unused orly rust [1]

2.8 Kernfeatures

https://www.youtube.com/watch?v=d1uraoHM8Gg
TODO: no need for a runtime, all static analytics
TODO: memory safety
TODO: data-race freedom
TODO: active community
TODO: concurrency: no undefined behavior
TODO: ffi binding Foreign Function Interface[2]
TODO: zero cost abstraction
TODO: package manager: cargo

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: static type system with local type inference
TODO: explicit notion of mutability
TODO: zero-cost abstraction *(do not introduce new cost through implementation of abstraction)
TODO: errors are values not exceptions TODO: no null
TODO: ßtatic automatic memory management no garbage collection
TODO: often compared to GO and D (44min)

2.9 Schwächen

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: compile-times
TODO: Rust is a vampire language, it does not reflect at all!
TODO: depending on the field -> majority of libraries?

2.10 Performance Fallstricke

TODO: [8]

[2] Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [2]

2.11 Beispiele von Verwendung von Rust

TODO: firefox

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: GTK binding heavily to rust

3 Stand der Technik (c++ Version)

3.1 Hochperformant -> parallel?

3.2 Serverbasierte Kommunikationsplattform

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

3.4 ASN.1

3.5 PER

3.6 MEC-View Server und Umgebung

4 Anforderungen

4.1 Funktionale Anforderungen

4.2 Nichtfunktionale Anforderungen

4.3 Kein Protobuf weil

5 Systemanalyse

5.1 Systemkontextdiagramm

5.2 Schnitstellenanalyse

5.3 C++ Referenzsystem

6 Systementwurf

6.1 Änderungen bedingt durch Rust

7 Implementierung

8 Auswertung

9 Zusammenfassung und Fazit

Literatur

[1] Jim Blandy. Why Rust? Trustworthy, Concurrent System Programming. Englisch. 2015. URL: http://www.oreilly.com/programming/free/files/why-rust.pdf (besucht am 01. 06. 2017).

[2] Wikipedia contributors. Foreign function interface — Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Foreign_function_interface&oldid=825105351 (besucht am 14. 02. 2018).

[3] fgilcher. Subreddit Rust. fgilcher kommentiert. Englisch. 3. Nov. 2017. URL: https://www.reddit.com/r/rust/comments/7amv58/just_started_learning_rust_and_was_wondering_does/dpb9qew/ (besucht am 14. 02. 2018).

[4] GitHub. Tracking issue for 128-bit integer support (RFC 1504). Englisch. URL: https://github.com/rust-lang/rust/issues/35118#issuecomment-278078118 (besucht am 19. 02. 2018).

[5] Sebastian Grüner. „C ist eine feindselige Sprache“. Der Mitbegründer des Gnome-Projekts Fede Deutsch. 22. Juni 2017. URL: https://www.golem.de/news/rust-c-ist-eine-feindselige-sprache-1707-129196.html (besucht am 14. 02. 2018).

[6] Jason Orendorff Jim Blandy. Programming Rust. Fast, Safe Systems Development. O'Reilly Media, Dez. 2017. ISBN: 1491927283.

[7] Felix von Leitner. Fefes Blog. D soll Teil von gcc werden. Deutsch. 22. Juni 2017. URL: https://blog.fefe.de/?ts=a7b51cac (besucht am 14. 02. 2018).

[8] Llogiq. Llogiq on stuff. Rust Performance Pitfalls. Englisch. URL: https://llogiq.github.io/2017/06/01/perf-pitfalls.html (besucht am 14. 02. 2018).

[9] MEC-View. Deutsch. URL: http://mec-view.de/ (besucht am 19. 02. 2018).

[10] Federico Mena Quintero. Replacing C library code with Rust. What I learned with librsvg. Englisch. URL: https://people.gnome.org/~federico/blog/docs/fmq-porting-c-to-rust.pdf (besucht am 14. 02. 2018).

[11] Julia Schmidt. Graydon Hoare im Interview zur Programmiersprache Rust. Deutsch. 12. Juli 2013. URL: https://www.heise.de/-1916345 (besucht am 16. 02. 2018).

[12] The LLVM Compiler Infrastructure Project. LLVM Overview. Englisch. URL: https://llvm.org/ (besucht am 19. 02. 2018).

[13] The LLVM Compiler Infrastructure Project. LLVM Features. Englisch. URL: https://llvm.org/Features.html (besucht am 19. 02. 2018).

[14] The Rust Programming Language. Englisch. URL: https://www.rust-lang.org/en-US/faq.html (besucht am 16. 02. 2018).

[15] The Rust Programming Language. Rust Platform Support. Englisch. URL: https://forge.rust-lang.org/platform-support.html (besucht am 19. 02. 2018).

[16] Wikipedia. LLVM — Wikipedia, Die freie Enzyklopädie. 2017. URL: \url{https://de.wikipedia.org/w/index.php?title=LLVM&oldid=169299719} (besucht am 19. 02. 2018).

Glossar

Foreign Function Interface Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [2] . 8

LLVM Früher „Low Level Virtual Machine“ [16], heute Eigenname; ist eine „Ansammlung von Modularen und Wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [12]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [13]. . 5

Abkürzungsverzeichnis

BMWi Bundesministerium für Wirtschaft und Energie. 2

Abbildungsverzeichnis