



Fakultät Informationstechnik
Studiengang Softwaretechnik und Medieninformatik

Bachelorkolloquium

Michael Watzko

Erstprüfer: Prof. Dr. Manfred Dausmann


Zweitprüfer: Dipl.-Ing. (FH) Kevin Erath M. Sc.

Firma: IT Designers GmbH


Betreuer: Dipl.-Inf. Hannes Todenhagen




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




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




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



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



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



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



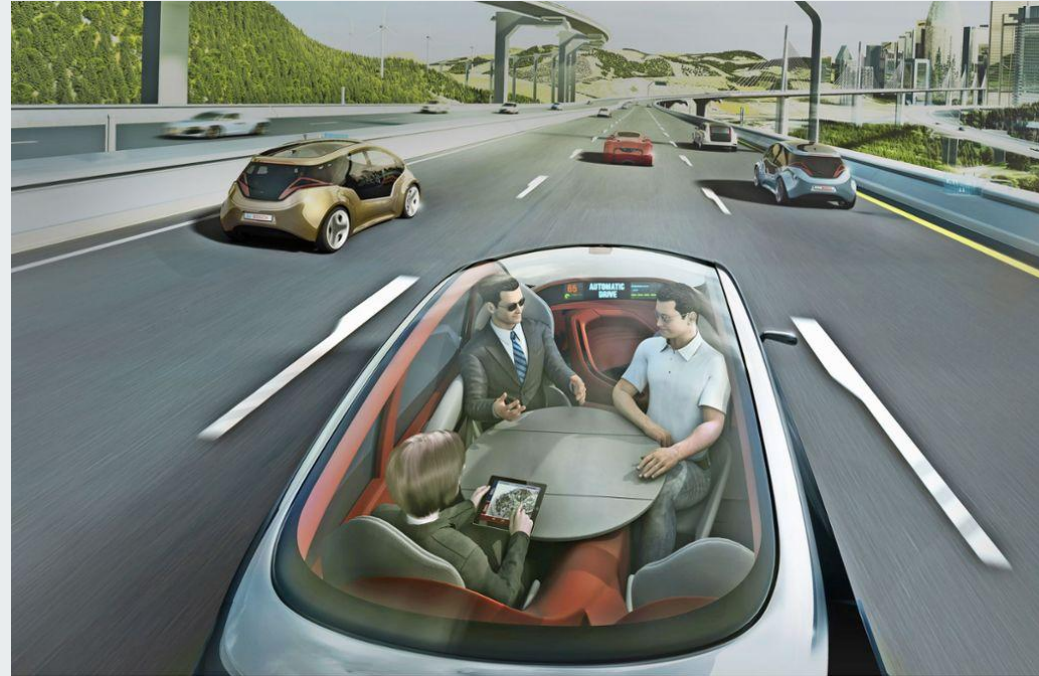
Inhalt

- Motivation
- Zielsetzung
 - MEC-View-Forschungsprojekt
 - Was ist MEC?
- Was ist Rust?
 - Was macht Rust besonders?
 - Beispiele
- Systemanalyse
- Umsetzung
- Auswertung
- Fazit



Motivation

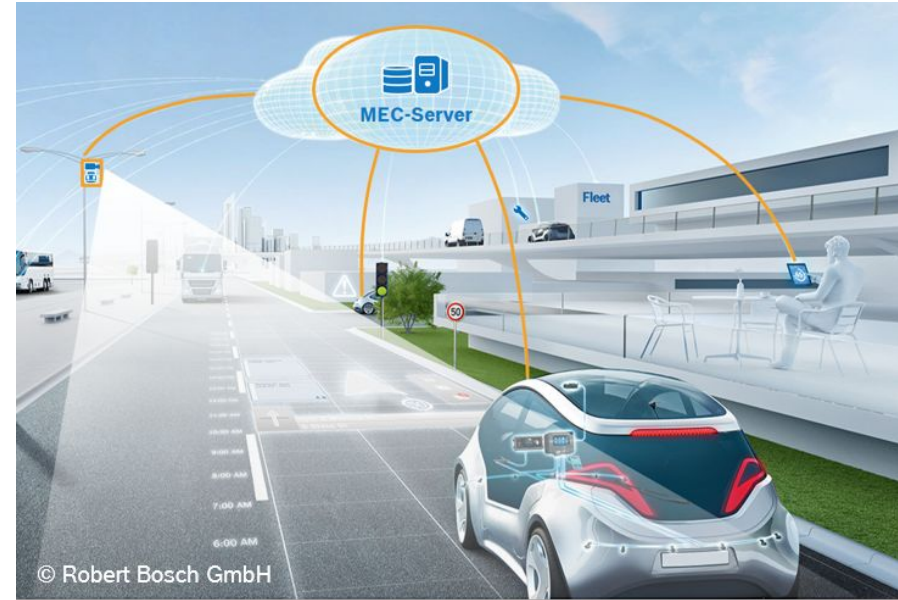
Motivation



<https://www.stuttgarter-zeitung.de/inhalt.autonomes-fahren-technisch-ausgereift-ab-er-verboten.dcb6d764-6c23-4bbe-94ef-c558890e2922.html?reduced=true>

Zielsetzung

- MEC-Server in Rust implementieren
- Gegenüberstellung zu C++ Prototyp



ulm university universität
uulm



Offen im Denken

NOKIA



<http://mec-view.de/>

Bundesministerium
für Wirtschaft
und Energie



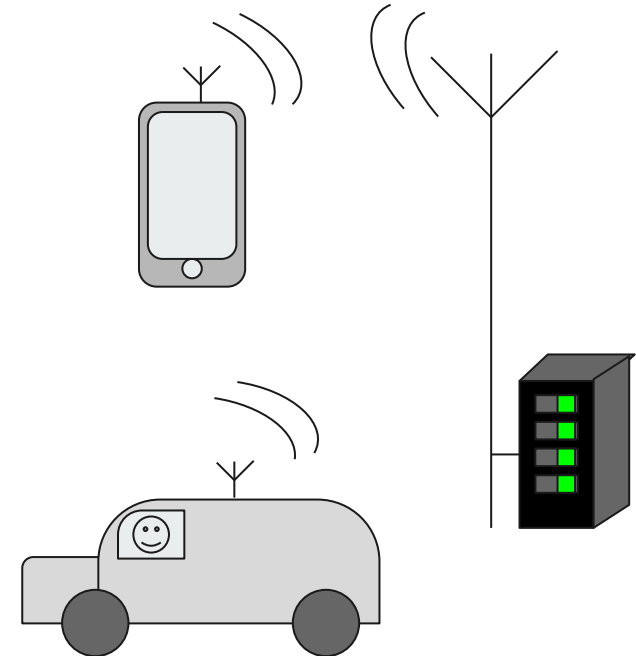
DAIMLER



Technik fürs Leben

Mobile Edge Computing (MEC)

- Schaffen eine Cloud-ähnliche Umgebung am Rande des Mobilfunknetzes
- Unterkategorie: Direkt an Funkmasten angeschlossene Recheneinheiten
- Ermöglichen niedrige Kommunikation mit niedriger Latenz
 - MEC: $\sim 20\text{ms}^1$
 - Cloudlösung: $\sim 100\text{ms}^1$



Kommunikation mittels ASN.1



- Seit den 90ern im Einsatz
- Nachrichtentypen in eigener Notationsform
- Wird zu nativen Datentypen übersetzt
- Definiert nur serialisierte Format (XER, BER, DER, PER, UPER)
- Serialisiert Programmiersprachenunabhängig

Kommunikation mittels ASN.1

- Seit den 90ern im Einsatz
- Nachrichtentypen in eigener Notationsform
- Wird zu nativen Datentypen übersetzt
- Definiert nur serialisierte Format (XER, BER, DER, PER, UPER)
- Serialisiert Programmiersprachenunabhängig

```
ClientRegistration ::= SEQUENCE {  
    type                ClientType,  
    covered-area        CoveredArea OPTIONAL,  
    minimum-message-period INTEGER (0..10000) OPTIONAL  
}
```



Was ist Rust?



Was ist Rust?



Was ist Rust?



https://de.wikipedia.org/wiki/Datei:Rost_Dose.jpg



<https://store.steampowered.com/app/252490/Rust>

Was ist Rust?



https://de.wikipedia.org/wiki/Datei:Rost_Dose.jpg



https://shop.europapark.de/cosmoshop/default/pix/r/eintrittskarten_parkeintritt.jpg



<https://store.steampowered.com/app/252490/Rust>

Was ist Rust?

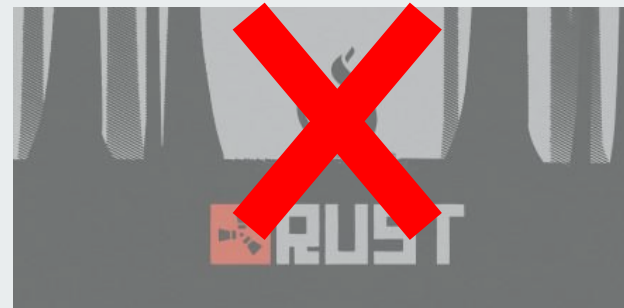


Michael Watzko - 25. Juni 2018

https://de.wikipedia.org/wiki/Datei:Rost_Dose.jpg



https://shop.europapark.de/cosmoshop/default/pix/r/eintrittskarten_parkeintritt.jpg



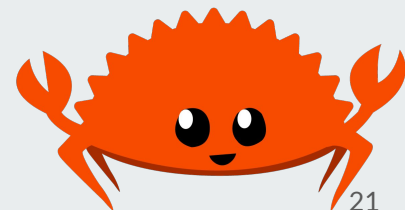
<https://store.steampowered.com/app/252490/Rust>

Was ist Rust?

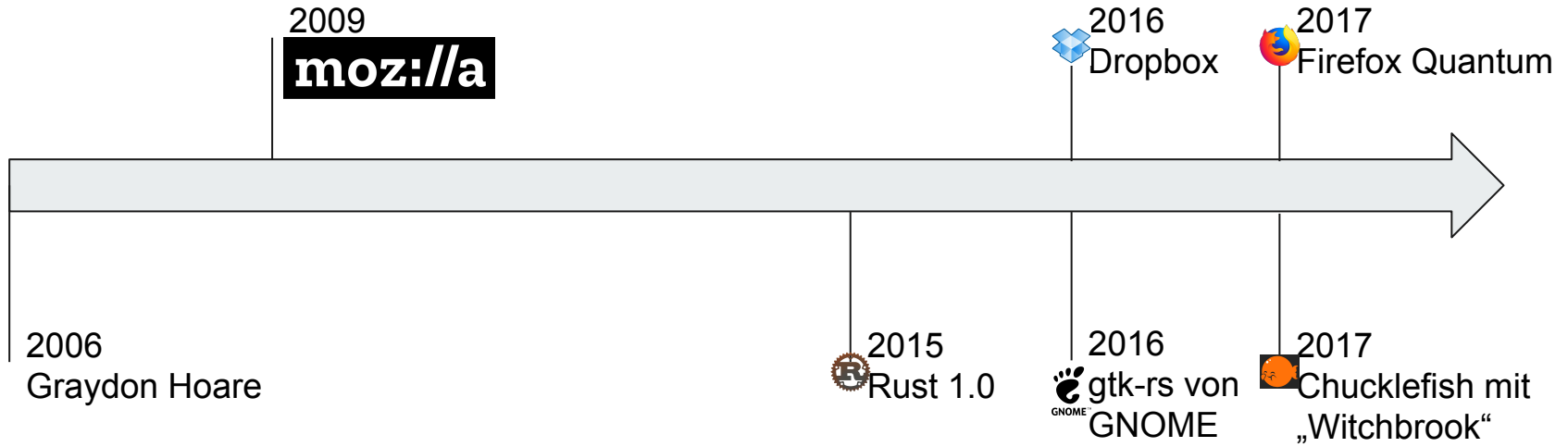


“**Rust** ist eine Systemprogrammiersprache die blitzschnell läuft, Speicherfehler vermeidet und Threadsicherheit garantiert.”

~ <https://www.rust-lang.org/de-DE/>



Geschichte



Was macht Rust besonders?



- Eigentümerprinzip
 - veränderlich / unveränderlich ausleihbar
 - statische Prüfung zur Compilezeit
- Threadsicher
 - garantiert: keine RaceCondition
- “Statisches, automatisches Speichermanagement” ohne Garbage-Collector
 - kein Speicherleck
 - kein dangling pointer
 - kein double-free
- Kein undefiniertes Verhalten
- Kein NULL(-Pointer)

Eigentümerprinzip - Beispiel



```
1 #[derive(Debug)]  
2 struct Position(f32, f32);
```


Eigentümerprinzip - Beispiel



```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn print(p: Position) {
5     println!("{:?}", p);
6 }
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn print(p: Position) {
5     println!("{:?}", p);
6 }
7
8 fn main() {
9     let p = Position(10.5_f32, 12.5_f32);
10    print(p);
11 }
```

```
Position(10.5, 12.5)
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn print(p: Position) {
5     println!("{:?}", p);
6 }
7
8 fn main() {
9     let p = Position(10.5_f32, 12.5_f32);
10    print(p);
11    print(p);
12 }
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn print(p: Position) {
5     println!("{:?}", p);
6 }
7
8 fn main() {
9     let p = Position(10.5_f32, 12.5_f32);
10    print(p);
11    print(p);
12 }
```

```
error[E0382]: use of moved value: `p`
  --> pos.rs:11:11
   |
10 |     print(p);
   |           - value moved here
11 |     print(p);
   |           ^ value used here after move
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn print(p: &Position) {
5     println!("{:?}", p);
6 }
7
8 fn main() {
9     let p = Position(10.5_f32, 12.5_f32);
10    print(&p);
11    print(&p);
12 }
```

```
Position(10.5, 12.5)
Position(10.5, 12.5)
```

Was macht Rust besonders?



- Eigentümerprinzip
 - veränderlich / unveränderlich ausleihbar
 - statische Prüfung zur Compilezeit
- Threadsicher
 - garantiert: keine RaceCondition
- “Statisches, automatisches Speichermanagement” ohne Garbage-Collector
 - kein Speicherleck
 - kein dangling pointer
 - kein double-free
- Kein undefiniertes Verhalten
- Kein NULL(-Pointer)

Eigentümerprinzip - Beispiel



```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn complex() -> &'static Position {
5     let pos = Position(4.0_f32, 2.0_f32);
6 }
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn complex() -> &'static Position {
5     let pos = Position(4.0_f32, 2.0_f32);
6     // complex algorithm
7     return &pos;
8 }
```


Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn complex() -> &'static Position {
5     let pos = Position(4.0_f32, 2.0_f32);
6     // complex algorithm
7     return &pos;
8 }
9
10 fn main() {
11     let p = complex();
12 }
```

Eigentümerprinzip - Beispiel

```
1 #[derive(Debug)]
2 struct Position(f32, f32);
3
4 fn complex() -> &'static Position {
5     let pos = Position(4.0_f32, 2.0_f32);
6     // complex algorithm
7     return &pos;
8 }
9
10 fn main() {
11     let p = complex();
12 }
```

```
error[E0597]: `pos` does not live long enough
--> pos.rs:7:13
   |
7 |     return &pos;
   |           ^^^ borrowed value does not live long enough
8 | }
   | - borrowed value only lives until here
```

Was macht Rust besonders?



- Eigentümerprinzip
 - veränderlich / unveränderlich ausleihbar
 - statische Prüfung zur Compilezeit
- Threadsicher
 - garantiert: keine RaceCondition
- “Statisches, automatisches Speichermanagement” ohne Garbage-Collector
 - kein Speicherleck
 - kein dangling pointer
 - kein double-free
- Kein undefiniertes Verhalten
- Kein NULL(-Pointer)

Rust erzwingt...



- “NULL-Pointer”-Prüfung:
 - Option-Typ
- Ergebnisprüfung:
 - Result-Typ
- Gültige Lebenszeiten von Referenzen
- Bounds-Check

Rust erzwingt...



- “NULL-Pointer”-Prüfung:
 - Option-Typ
- Ergebnisprüfung:
 - Result-Typ
- Gültige Lebenszeiten von Referenzen
- Bounds-Check

```
1 pub enum Option<T> {  
2     None,      // ~ NULL  
3     Some(T),   // ~ nicht NULL  
4 }
```

Rust erzwingt...



- “NULL-Pointer”-Prüfung:
 - Option-Typ
- Ergebnisprüfung:
 - Result-Typ
- Gültige Lebenszeiten von Referenzen
- Bounds-Check

Rust erzwingt...



- “NULL-Pointer”-Prüfung:
 - Option-Typ
- Ergebnisprüfung:
 - Result-Typ
- Gültige Lebenszeiten von Referenzen
- Bounds-Check

```
1 pub enum Result<T, E> {  
2     Ok(T),           // Ergebnis  
3     Err(E),          // Fehler  
4 }
```

“Schränkt doch alles nur ein!”



“Schränkt doch alles nur ein!”



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

“Schränkt doch alles nur ein!”



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

- Was wenn
 - fopen fehlschlägt -> SEGFAULT
 - fputs fehlschlägt -> falsche Annahme
- Wo wird die Datei geschlossen?
 - leak!

~~“Schränkt doch alles nur ein!”~~ Hilft bei der Fehlervermeidung



Rust Hilft bei der Fehlervermeidung



```
1 use std::fs::File;
2 use std::io::Write;
3
4 fn main() {
5     if let Ok(mut file) = File::create("private.key") {
6         write!(file, "42");
7     }
8 }
```

Rust Hilft bei der Fehlervermeidung

```
1 use std::fs::File;
2 use std::io::Write;
3
4 fn main() {
5     if let Ok(mut file) = File::create("private.key") {
6         write!(file, "42");
7     }
8 }
```

```
warning: unused `std::result::Result` which must be used
--> fopen.rs:6:5
   |
6 |     write!(file, "42");
   |     ^^^^^^^^^^^^^^^^^^
```

Rust Hilft bei der Fehlervermeidung

- Guter Programmierstil verlangt Fehlerprüfung

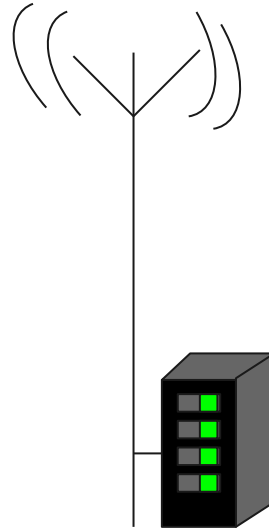
```
1 use std::fs::File;
2 use std::io::Write;
3
4 fn main() {
5     if let Ok(mut file) = File::create("private.key") {
6         if let Err(e) = write!(file, "42") {
7             println!("Konnte nicht in Datei schreiben: {}", e);
8         }
9     }
10 }
```

- Kein Zugriff auf 'file' bei einem Fehler
- Schreibergebnis wird überprüft oder mit Compilerwarnung bemängelt
- Datei wird wieder ordnungsgemäß geschlossen

Systemanalyse

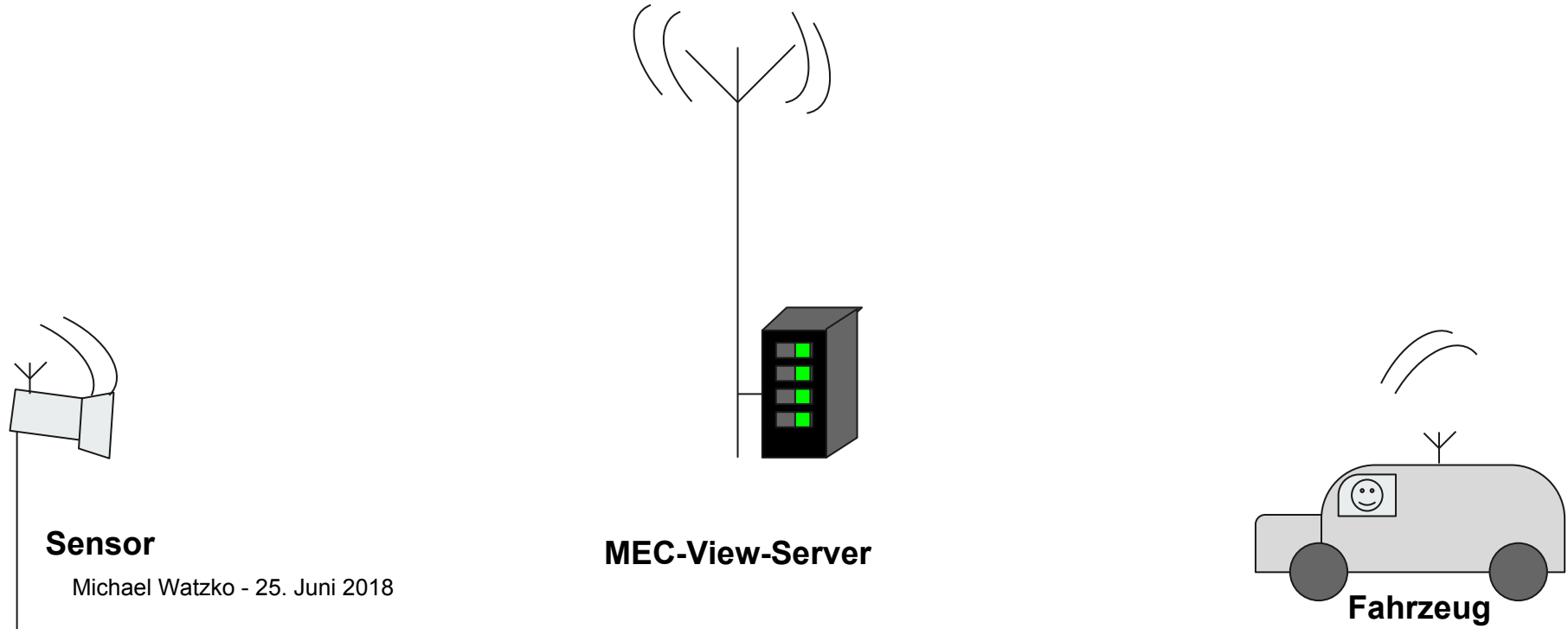


Systemanalyse

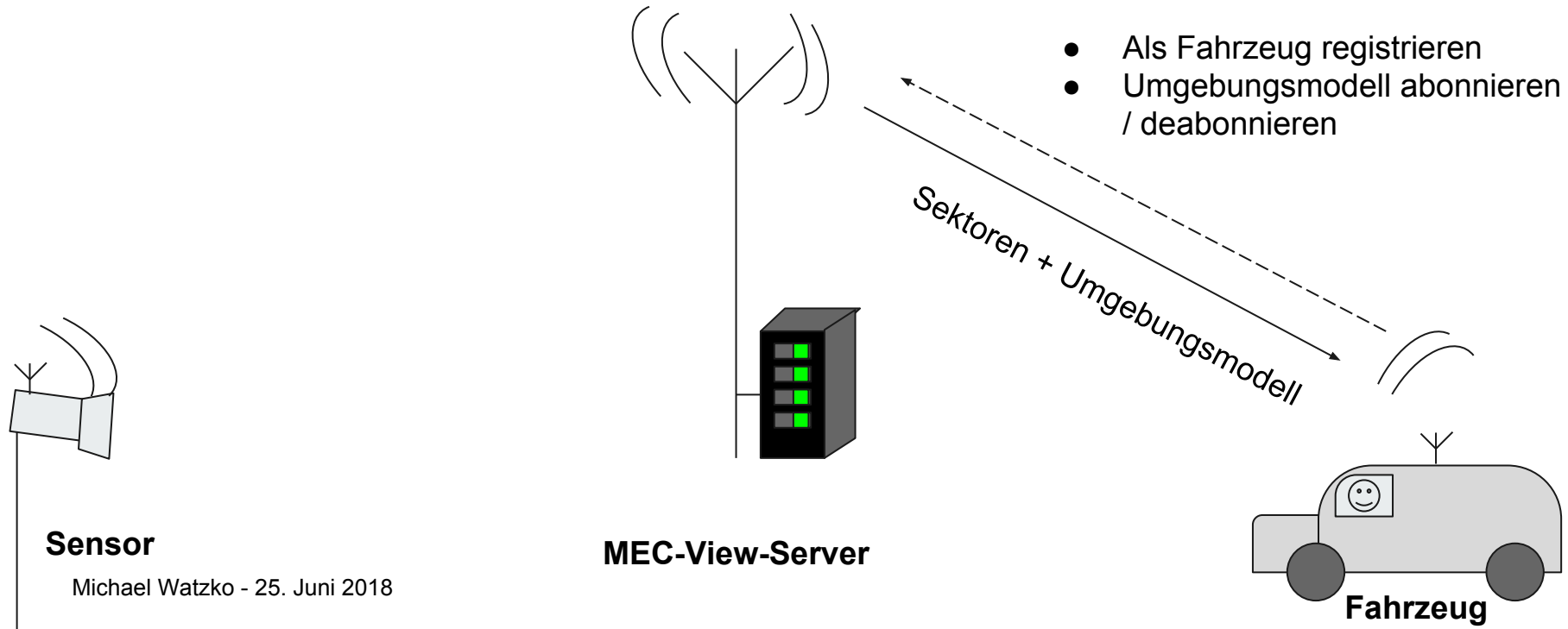


MEC-View-Server

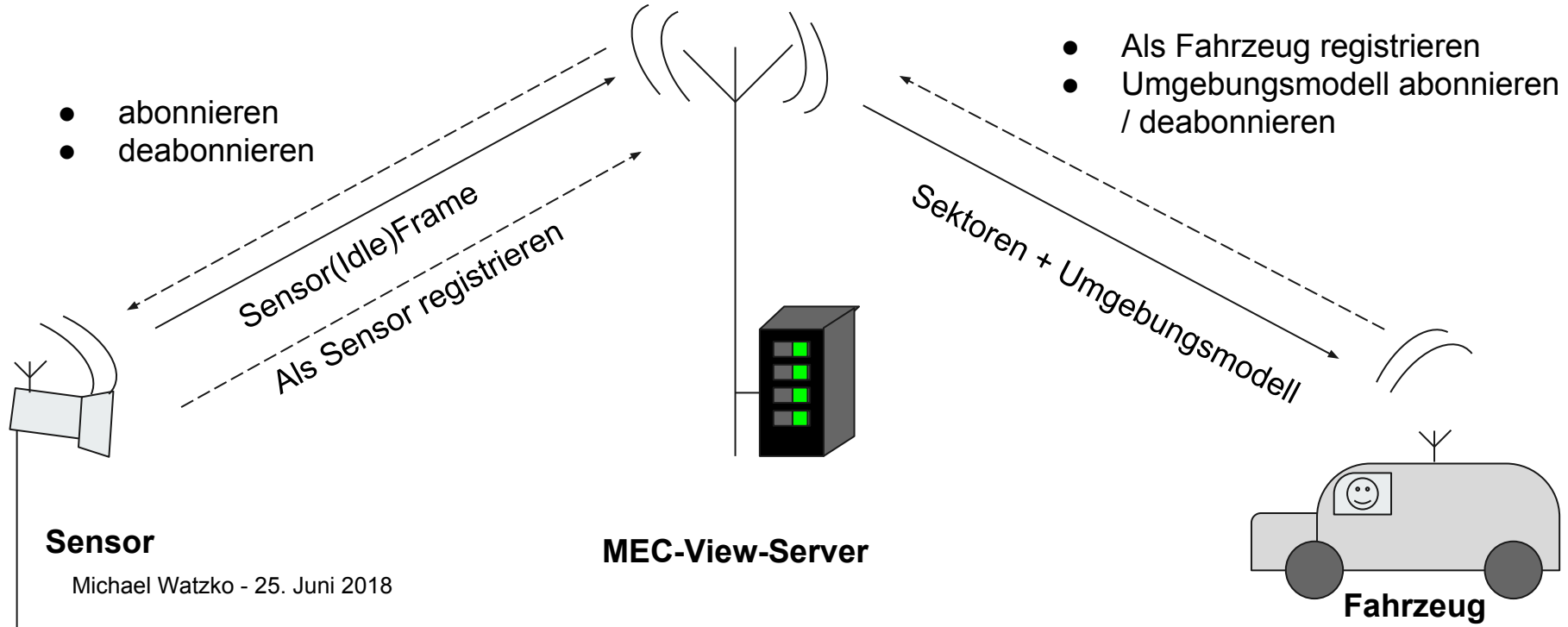
Systemanalyse



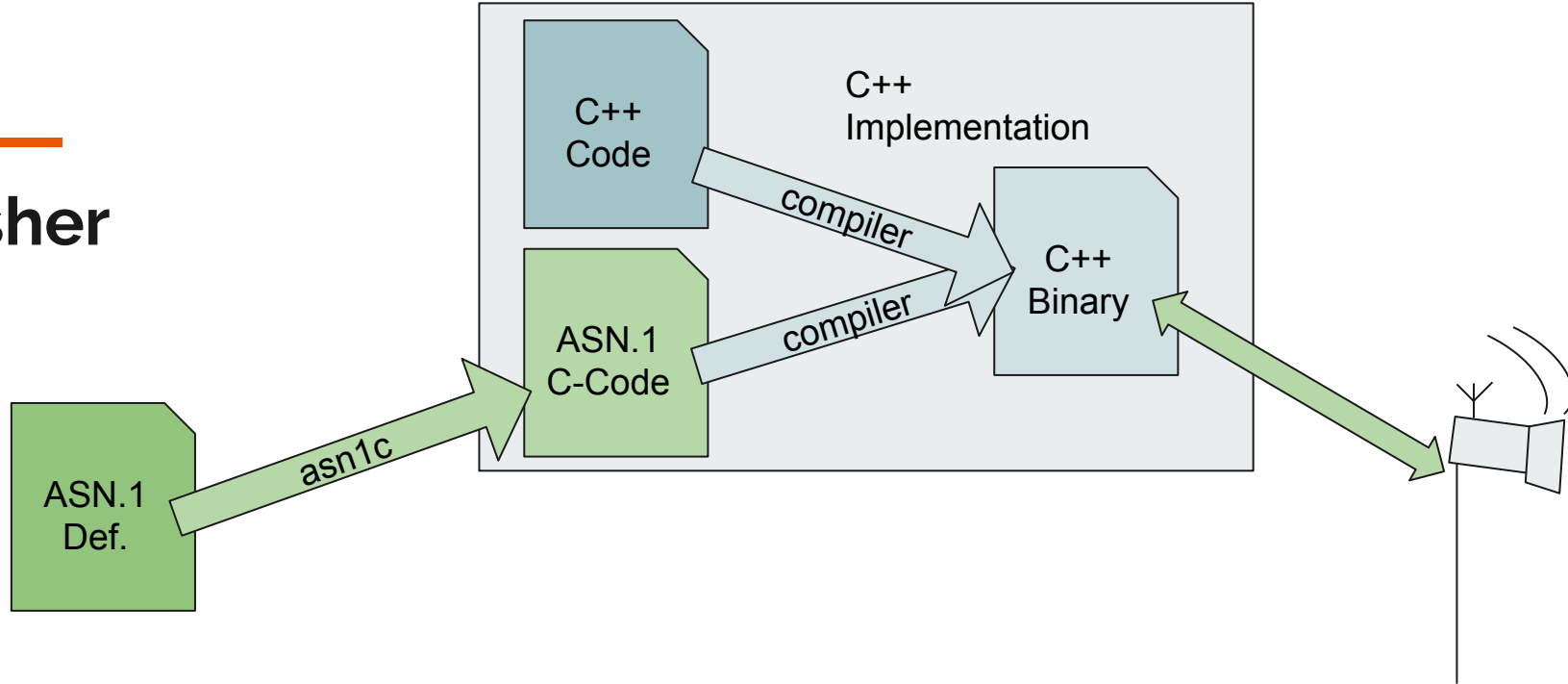
Systemanalyse



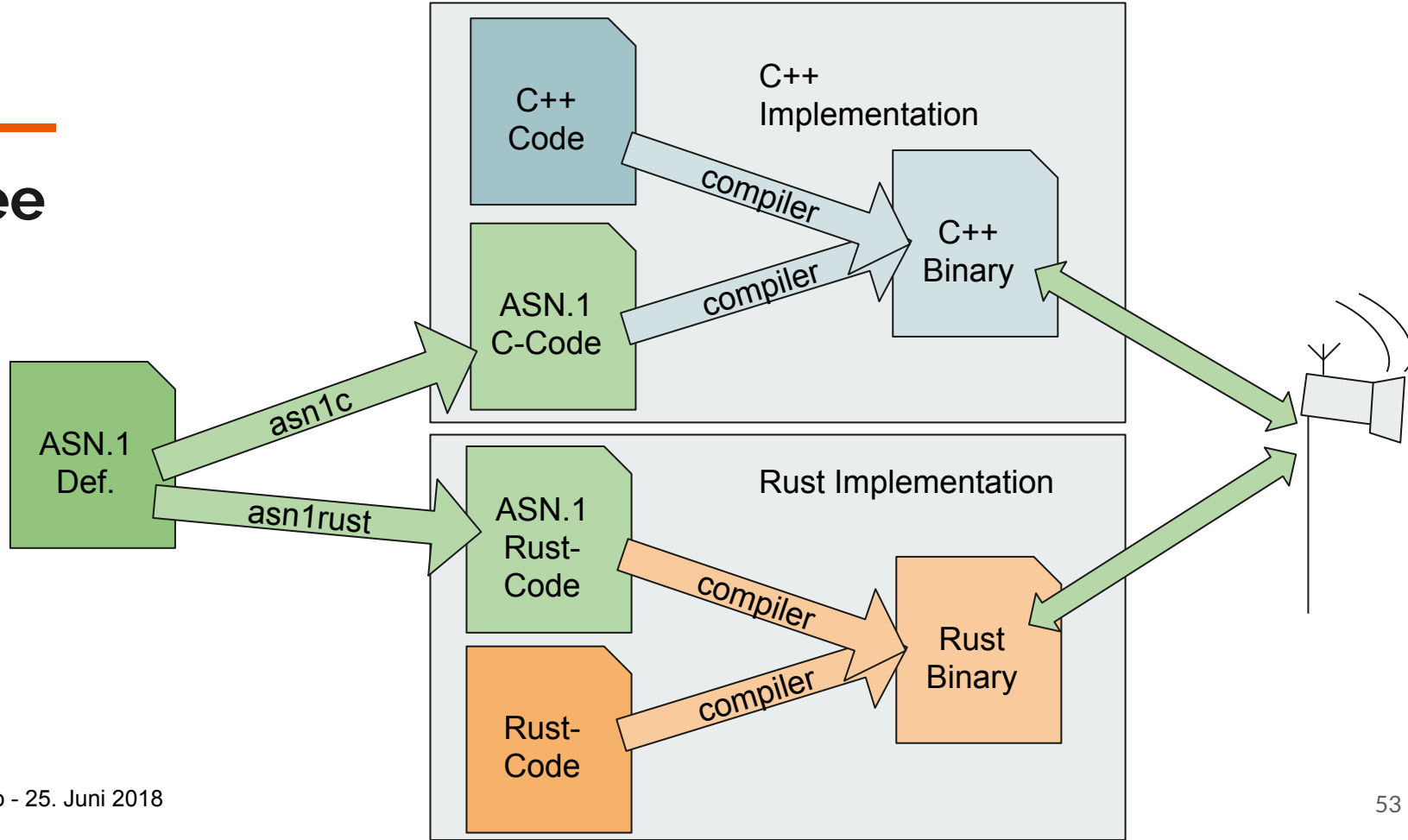
Systemanalyse



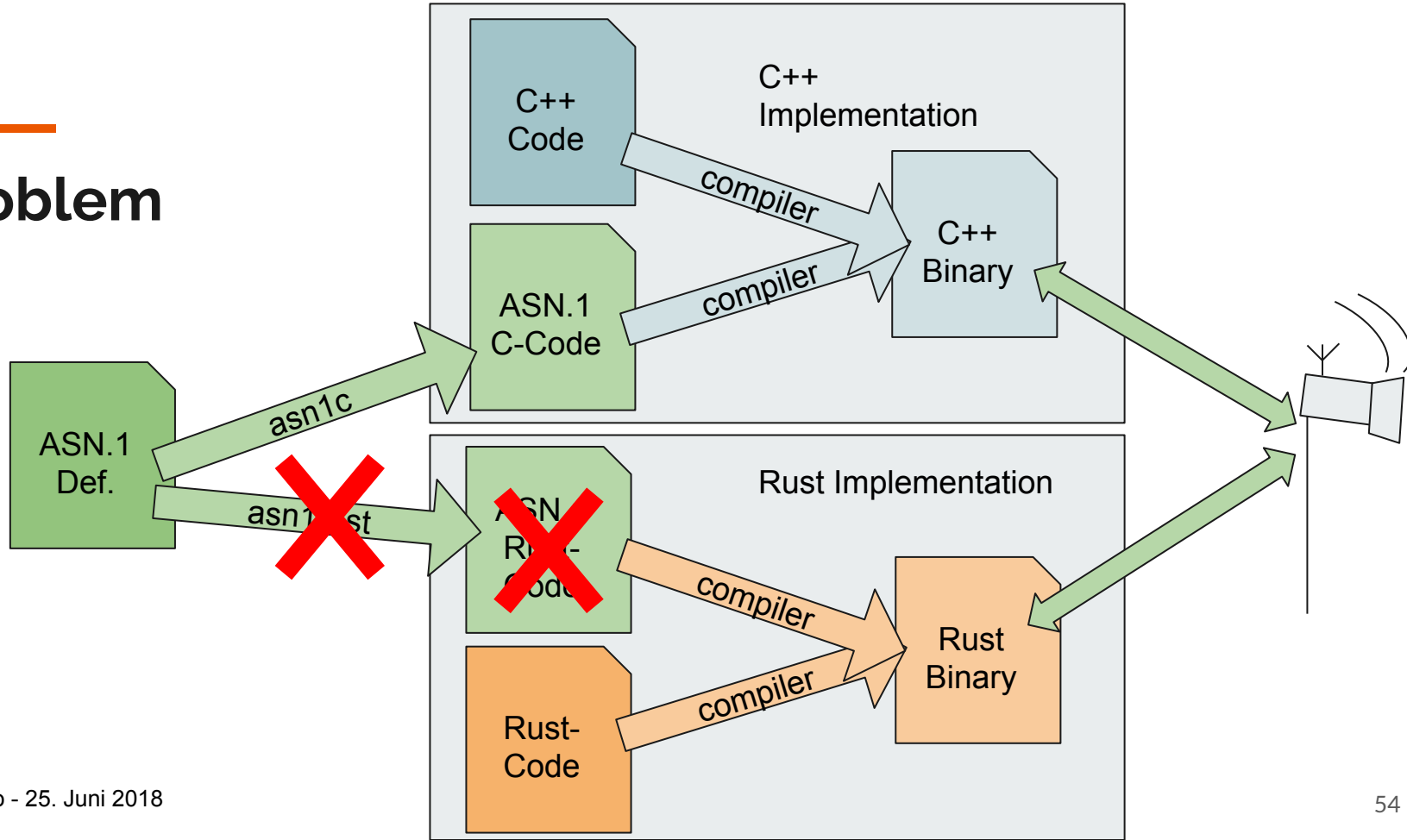
Bisher



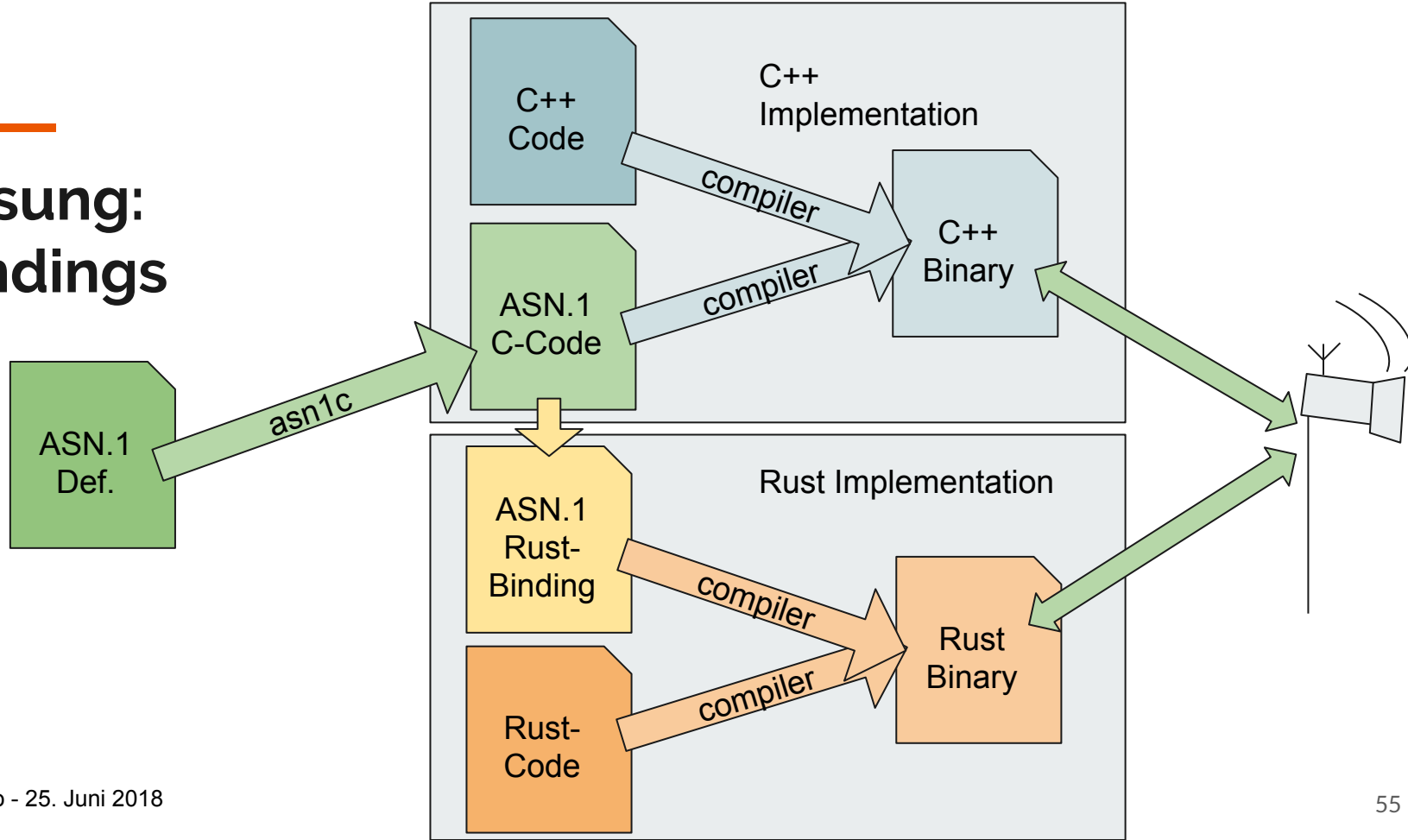
Idee



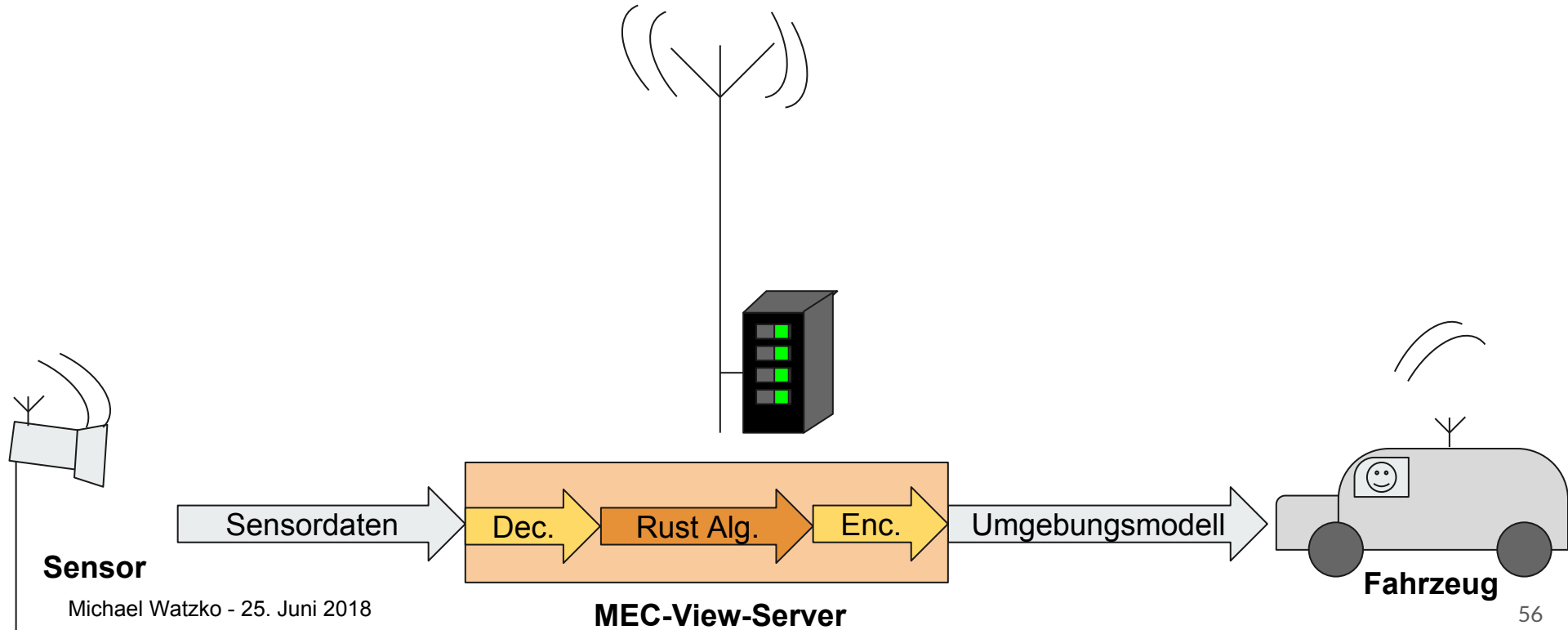
Problem



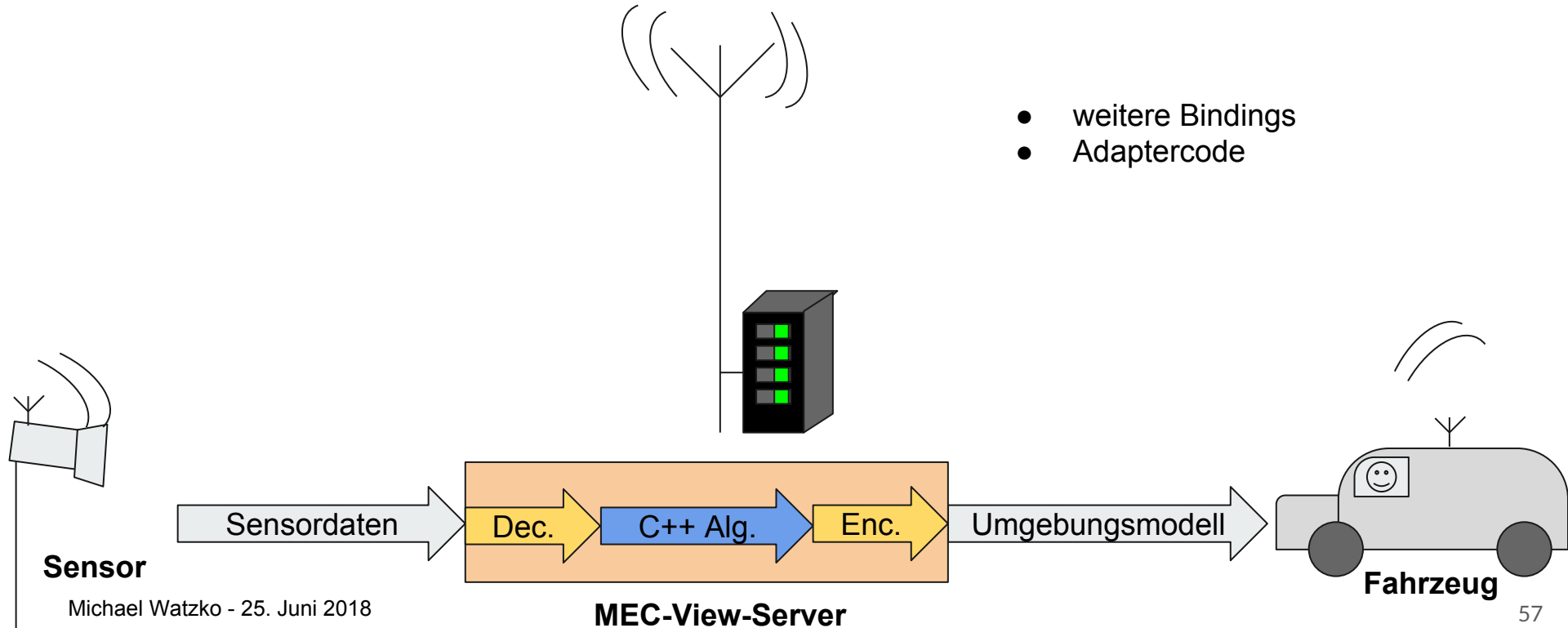
Lösung: Bindings



Machbarkeit? Dummy - Algorithmus in Rust



Machbarkeit? Dummy - Algorithmus in C++



Asynchrone Kommunikation?



Asynchrone Kommunikation: Channels



```
1 fn main () {  
2   let (sender, receiver) = mpsc::channel();  
3  
4   let thread = thread::spawn (move || {  
5       let command: String = receiver.recv ().unwrap() ;  
6       println!("Empfangen: {}", command);  
7   });  
8  
9   sender.send("Hallo, Kanal!".into()).unwrap();  
10  thread.join().unwrap();  
11 }
```

Asynchrone Kommunikation: Channels + Command Pattern

```
1 enum Command {  
2     Say(String) ,  
3 }  
4  
5 impl Command {  
6     fn execute(self) {  
7         match self {  
8             Command::Say(text) => println!("Say: {}", text),  
9         }  
10    }  
11 }
```

Asynchrone Kommunikation: Channels + Command Pattern

```
1 fn main () {  
2   let (sender, receiver) = mpsc::channel();  
3  
4   let thread = thread::spawn (move || {  
5       let command: Command = receiver.recv ().unwrap() ;  
6       command.execute () ;  
7   });  
8  
9   sender.send(Command::Say("Hallo, Kanal!".into())).unwrap();  
10  thread.join().unwrap();  
11 }
```

Asynchrone Kommunikation: Channels

- + Command Pattern
- + Proxy Pattern

```
1 trait EndPoint {  
2     fn say(&self, text: &str);  
3 }  
4  
5 impl EndPoint for mpsc::Sender<Command> {  
6     fn say(&self, text: &str) {  
7         self.send(Command::Say(text.into())).unwrap();  
8     }  
9 }
```

- Stichwort: Loose Coupling
- Fassade auf Methodenebene
- Proxy Pattern auf Klassenebene

Asynchrone Kommunikation: Channels

- + Command Pattern
- + Proxy Pattern

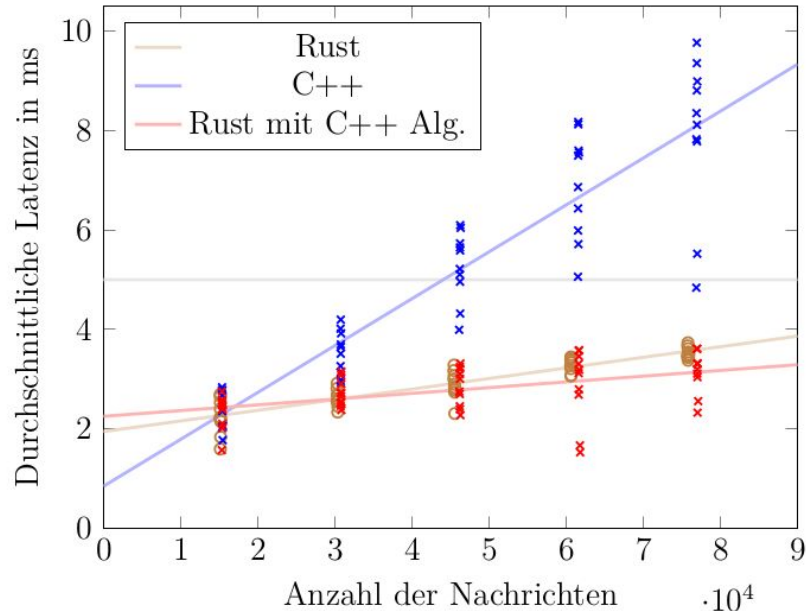
```
1 fn main () {  
2   let (sender, receiver) = mpsc::channel();  
3   let endpoint = &sender as &EndPoint ;  
4  
5   let thread = thread::spawn (move || {  
6       let command : Command = receiver.recv ().unwrap() ;  
7       command.execute () ;  
8   });  
9  
10  endpoint.say("Hallo, Proxy!");  
11  thread.join().unwrap();  
12 }
```

Auswertung

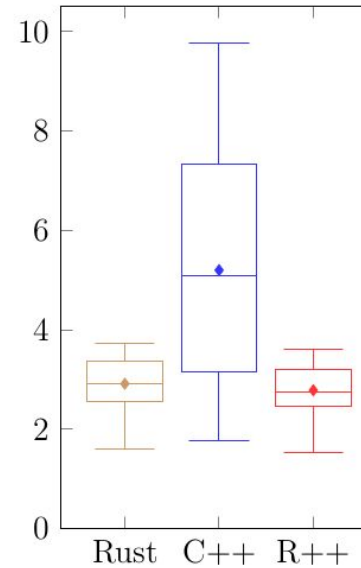


- Testdurchläufe mit
 - C++ Implementation
 - purer Rust-Implementation und
 - Rust-Implementation mit C++ Algorithmus
 - 25 Sensoren
 - 1-5 Fahrzeugen
- Testsystem
 - Ubuntu 16.04.4 LTS Server
 - 2x Intel Xeon CPUs, E5-2620 v4 @ 2.10GHz
 - CPU Governor: "performance"
- Durch Skript gesteuert
 - 10 Einzeltests

Ergebnisse



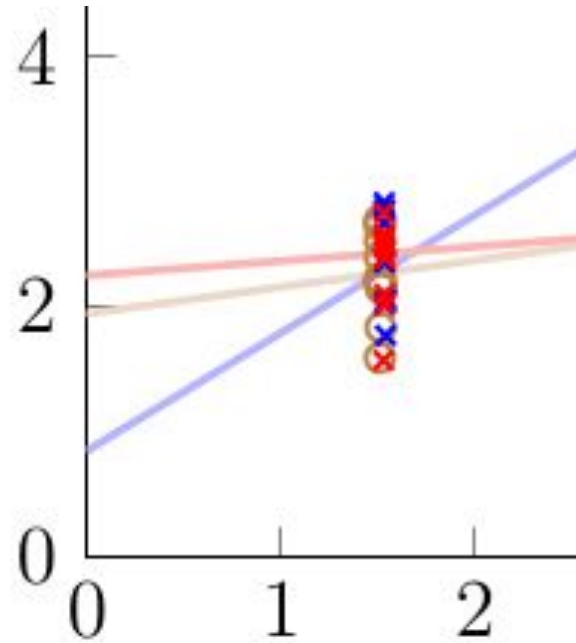
(a) Latenz-Nachrichtenanzahl Diagramm



(b) Boxplot für die Latenz

- Kein deutlicher Nachteil von Rust erkennbar
- C++ Implementation skaliert falsch / proportional zu der Anzahl an Fahrzeugen

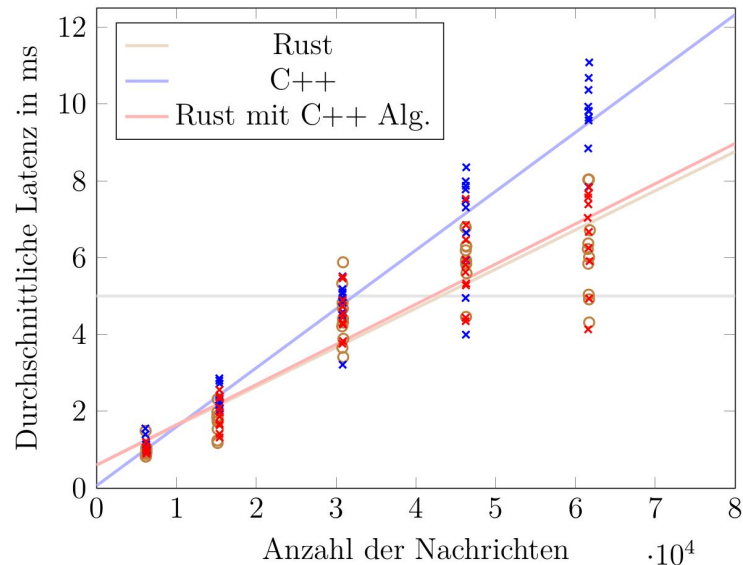
Ergebnisse



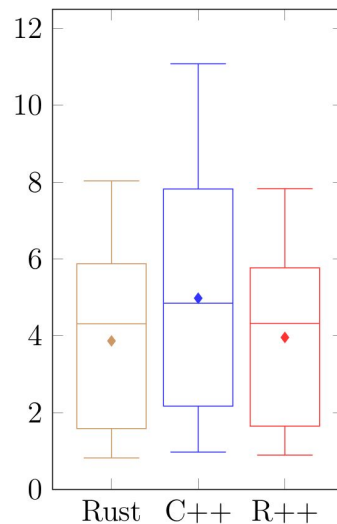
- Rust-Implementation bei einem Fahrzeug gleichauf

— Rust
— C++
— Rust mit C++ Alg.

Ergebnisse - 2. Test



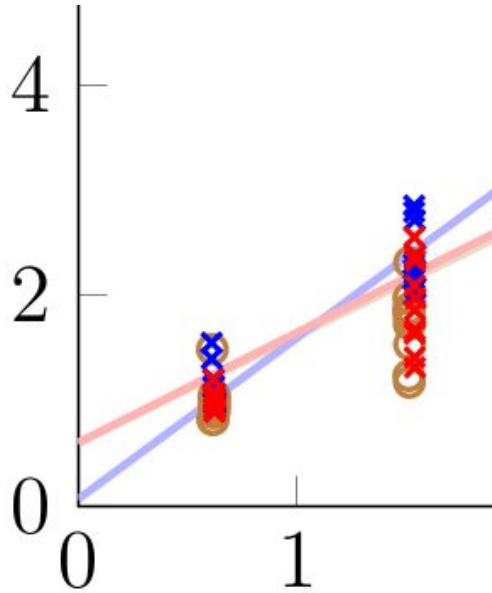
(a) Latenz-Nachrichtenanzahl Diagramm



(b) Boxplot für
die Latenz

- 1 Fahrzeug
- 10, 25, 50, 75, und 100 Sensoren
- Rust hat Nase vorn

Ergebnisse - 2. Test



- 1 Fahrzeug
- 10, 25, 50, 75, und 100 Sensoren
- Rust hat Nase vorn

Fazit



- kein Performancenachteil gegenüber C++
- Rust
 - verlangt sauberes programmieren
 - verhindert Speicherlecks
 - verhindert Segmentation Faults
 - verhindert RaceConditions

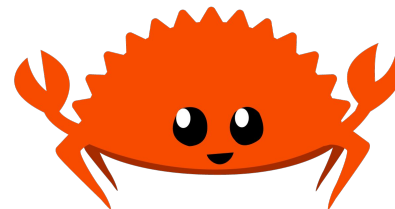
- steile Lernkurve: Eigentümerprinzip

Fazit

- kein Performancenachteil gegenüber C++
- Rust
 - verlangt sauberes programmieren
 - verhindert Speicherlecks
 - verhindert Segmentation Faults
 - verhindert RaceConditions

→ nutzt Rust

- steile Lernkurve: Eigentümerprinzip



**Vielen Dank für Ihre
Aufmerksamkeit!**