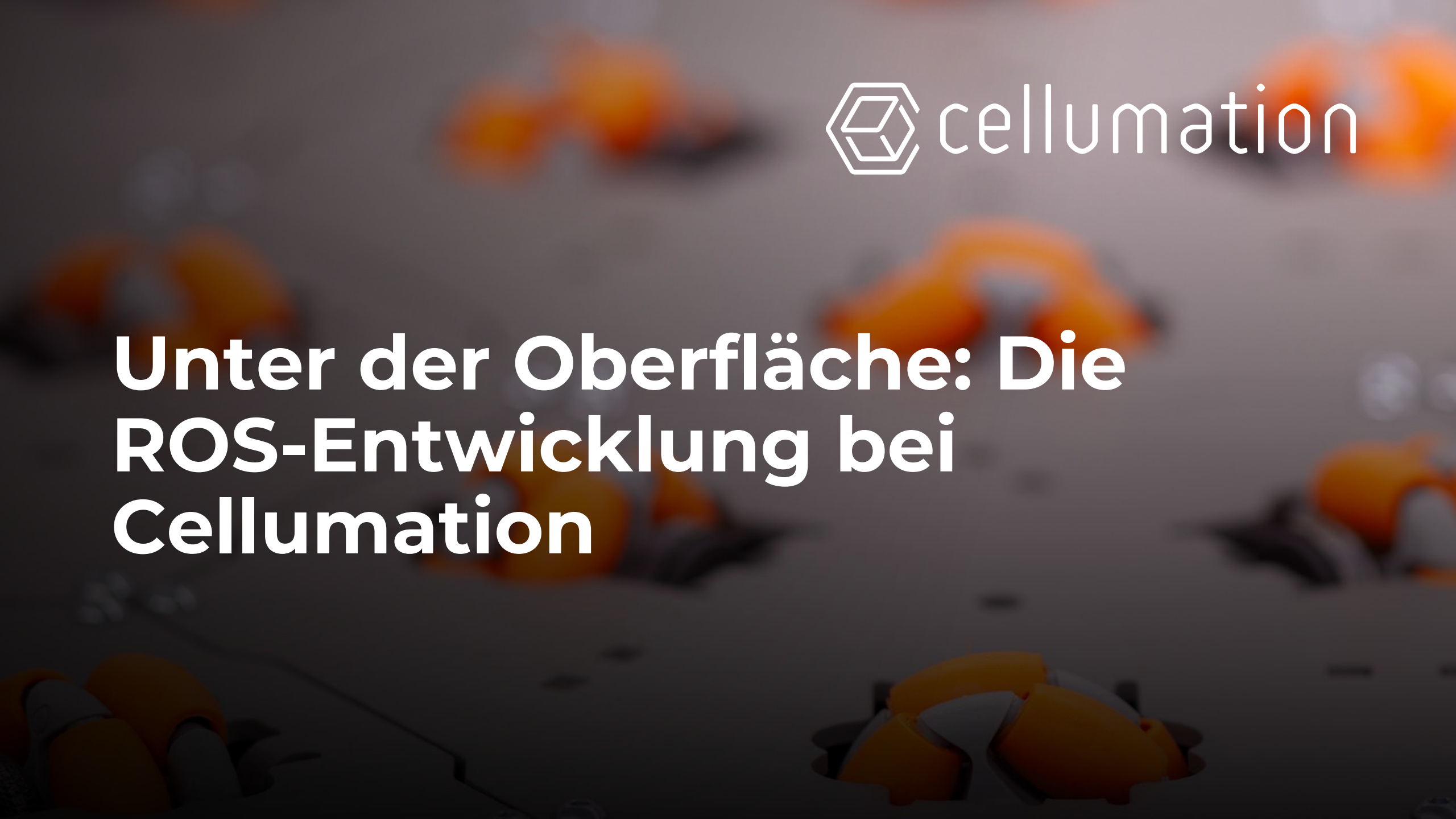


Unter der Oberfläche: Die ROS-Entwicklung bei Cellumation

Inhalt

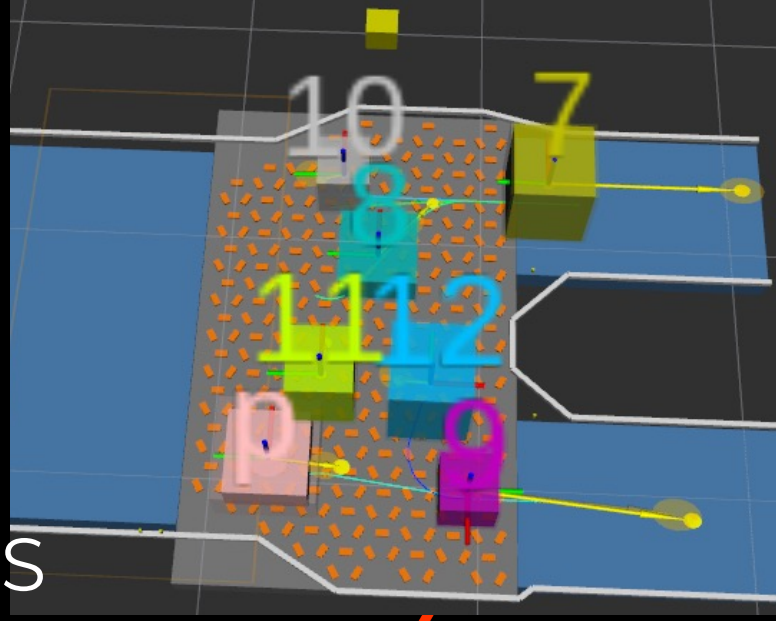
- Wer sind wir uns was machen wir
- Unser Softwarestack
- Port zu ROS2
  - Motivation
  - Vorgehen
  - Parameter
  - Kosten
  - The Bad
  - The Ugly
  - The good
- Fazit

Celluveyor

- System zum Transport von Fördergut in der Intralogistik
- Produktiv im Einsatz
- > 4000 Objekte pro Stunde
- Software defined

Software Stack

Allgemein



- Entwicklungsrichtlinien:
  - Basisbibliotheken unabhängig von ROS
  - 'Dünne' ROS Wrapper
- Nutzen ROS als Middleware und fast nur Core Pakete
- Ungewöhnliches Scenario : Eine ROS-Stack kontrolliert mehrere 'Roboter'
    - Standard ROS Pakete erzeugen in diesem Szenario zu viel Overhead (msg Frequenz zu hoch)

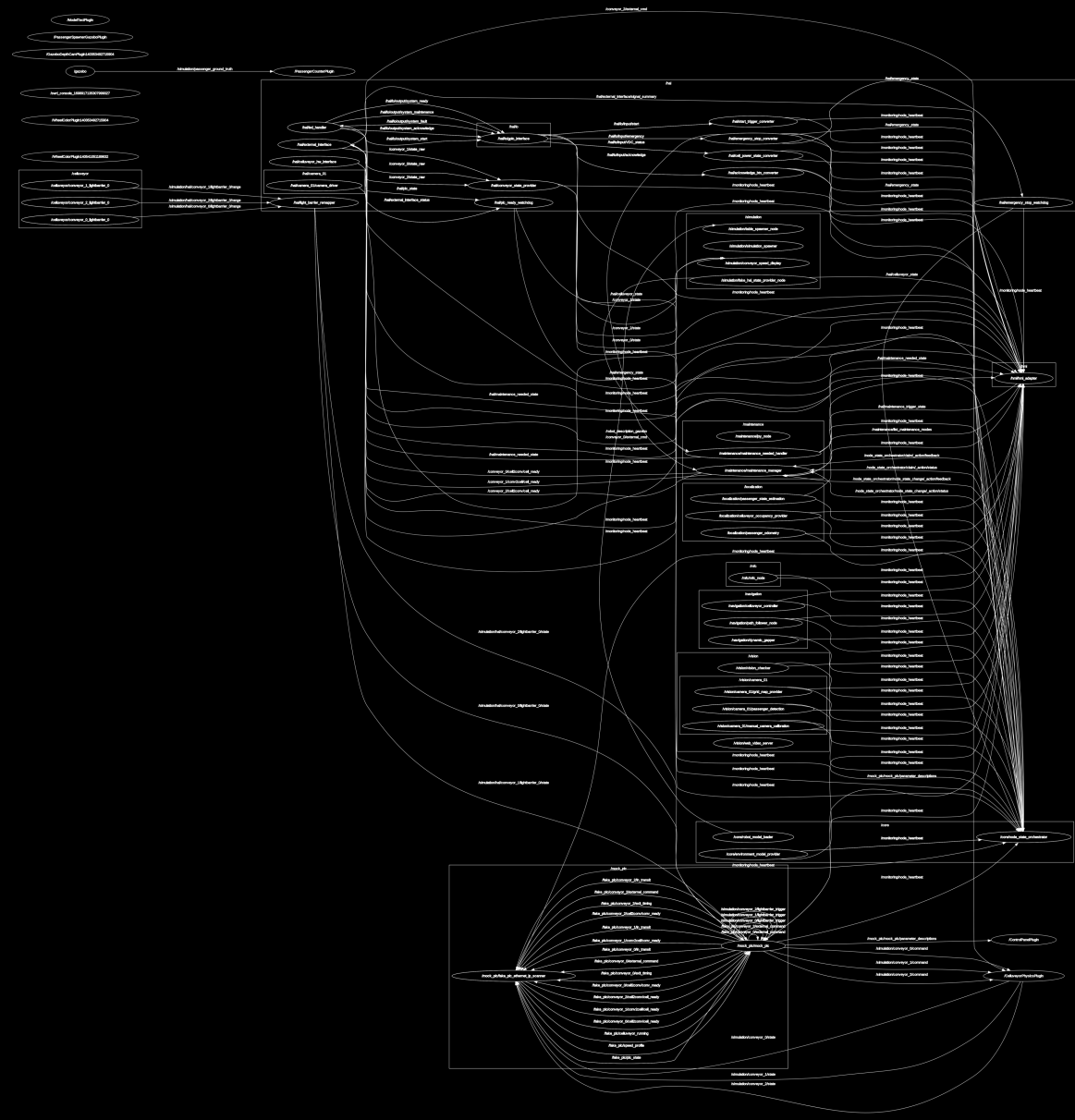
Software Stack

Node lifecycle Management

- Angelehnt an ROS2 managed nodes
- Features:
  - Deadlock / while (true) Detektion für default callback group
  - Absturz / Alive Detektion pro Node
  - Spezialisierte Fehlerzustände pro Node
    - Wichtig für Fehlermeldungen zum Kunden
- Tooling für:
  - Überwachung/Start/Stop von Nodes
  - Aktivieren von Teilsystemen, z.B. für Kamerakalibrierung

Software Stack

- 79 CMake Pakete
  - ~90K LoC
    - 95% Cpp
    - 5% Python
- 55 Nodes zur Laufzeit



Port zu ROS 2

Motivation

- ROS 1 abgekündigt
- Wir wollen neuere Tools (gcc, libs, c++20, etc.)
- Support für neue Hardware fehlt in 20.04
- Wollen ROS 1 nicht selber bauen / warten für 22.04

Port zu ROS 2

Vorgehen

- Aufsetzen auf Iron
- Wir sind erstmal bei Gazebo Classic geblieben
  - Gradlinige Portierung, ohne größere Probleme
- Port unseres Lifecycle Mangements zu ROS2
  - ROS2 Managed Nodes fehlen Features die wir brauchen
  - Weitere Nutzung unserer Lib war für UNS der bessere weg, da wir bereits eine Abstraktion hatten.
  - Relativ wenig Anpassungen in den Nodes selber wenn keine Services / Actions genutzt werden.

Port zu ROS 2

Parameter

- Parameter unterstützen keine Strukturen mehr

ROS 1: Arrays of Structs (Cool)

```yaml
cameras:
  - camera1:
      serial: 1234
      pose:
        x: 1.0
        y: 1.0
  - camera2:
      serial: 4321
      pose:
        x: 42.0
        y: 42.0
```

ROS 2: Arrays (uncool)

```yaml
cameras:
  - "camera1"
  - "camera2"
serials:
  - 1234
  - 4321
poses_x:
  - 1.0
  - 42.0
poses_y:
  - 1.0
  - 42.0
```

ROS 2: Alternative (auch uncool)

```yaml
cameras:
  - "camera1"
  - "camera2"

camera1_serial: 1234
camera1_pose_x: 1.0
camera1_pose_y: 1.0

camera2_serial: 4321
camera2_pose_x: 42.0
camera2_pose_y: 42.0
```

Port zu ROS 2

Parameter

- Erster Versuch Upstream Erweiterung
  - Merkbarer Widerstand, Merge unwahrscheinlich
- Schwenk auf eigene Lösung
  - Lädt Parameter aus config/'namespace/NodeName'.yaml
  - Support für Includes
  - Basis Parameter werden automatisch über ROS deklariert
- Nachteile:
  - Keine Dynamic Reconfigure Unterstützung für Strukturierte Paramater

Port zu ROS 2

Kosten

- Was hat der Spaß gekostet?
  - Ca. 6PM
- Fun Facts:
  - 1 PM nur Bugs finden und fixen in rcl, rclcpp, rclcpp_action
  - 0.5 PM FastDDS Probleme suchen

Unerwartete Probleme (The Bad)

Allgemein

- Kompilierzeit um ~30% gestiegen
- Keine Gazebo Classic Updates für Ubuntu 22.04
- Steigerung Basis-CPU-Last aller Nodes ~5-10%
- Massive CPU Last bei Python Nodes
  - Überfordert mit Empfangen von Daten mit >200hz

  -> Wir nutzen keine Python Nodes mehr, wenn möglich
- Dokumentation schlechter als in ROS 1
  - APIs unvollständig beschrieben / oder schlecht zu finden
  - Dokumentation hat vor 3 Jahren URL gewechselt. Google findet neue URL nicht. Forwards fehlen.
  - Benutzbarkeit von http://docs.ros.org Rückschritt zu Doxygen

Unerwartete Probleme (The Bad)

Colcon

- Usability schlechter als Catkin
  - Sehr verbose/umständlich
  - Shortcuts fehlen
  - Muss zwingend im Basisverzeichnis aufgerufen
- Build Job scheduling fehlt
  - Default wird mit min(Packete, CPU Cores) * CPU Cores Threads gebaut -> Speicher läuft schnell voll

- Lösung: Colcon Wrapper implementiert

Unerwartete Probleme (The Bad)

ROS_DOMAIN_ID

- ROS 1: Nodes kommunizieren by default mit localhost rosmaster.
  - Multi-Computer muss extra konfiguriert werden.
  - Guter Default
- ROS 2: Nodes kommunizieren by default mit allen Nodes im selben Netzwerk
  - Erzeugt nicht deterministische Probleme
  - Schlechter Default

Unerwartete Probleme (The Ugly)

FastDDS Discovery Probleme

- FastDDS Discovery funktioniert nicht zuverlässig wenn viele Nodes laufen
  - Services wurden teilweise nicht gefunden.
  - Vermutlich nur bei Nutzung von MultiThreadedExecutor
- Lösung: Umstieg auf CyclonDDS
- ROS2 Discovery Daemon erfordert ab und zu Neustart

Unerwartete Probleme (The Ugly)

Internale Scheduling

- Kein FiFo Scheduling
  - Timer haben immer Priorität
  - Lang laufende Timer Callback sind schwer zu debuggen
    - Keine Warnung,
      wenn Timer Callback Zeit > Timer Periode
    - Effekt : Es werden keine anderen Callbacks mehr verarbeitet.

Unerwartete Probleme (The Ugly)

Async API

- Service Clients / Action Clients
  - API ist Kontext abhängig
    - Unterschiedliche API in Single / Multi-Threaded Fall
    - Deadlocks bei Nutzung 'falscher' APIs
  - Bei Nutzung in Bibliotheken, muss Kontext übergeben werden
    - Fehleranfällig
  - Erschwert Nutzung von Common Code in Konstruktoren
  - ROS Tutorial ist minimalistisch
    - Verstecktes Tutorial zum Multi Threaded Fall
  - Keine Benachrichtigung wenn Gegenseite 'Verstirbt'
  - Keine API um Callback Gruppen einzeln auszuführen

- Wahl zwischen Pest oder Cholera:
  - Multi Threaded Executor mit Callback Gruppen, mutex etc
  - Single Threaded Executor mit Callbacks und State Machines

Unerwartete Probleme (The Ugly)

Bugs

- Timer werden mit identischem Timestamp aufgerufen.
- Race zwischen node->now(), TimerCb
- Action Client : Deadlock bei handle->get_status() in cb
- Action Client / Server : Sporadische Exceptions ('Taking data but nothing is readyTaking ')
- Action Client : Keine Möglichkeit sicher ein handle abzumelden
- Action Client : Handle wird intern gehalten bis result Cb ausgeführt wurde
- Action Client : Speicherleck in bestimmten Fehlerfällen
- Gazebo : Division By Zero beim steppen (Warnung)

Unerwartete Probleme (The Ugly)

Bugfixes

- Zäher Prozess Bugfixes zu mergen
  - ROS2 wirkt nicht gewartet
  - Für Fortschritt muss man spezielle Entwickler explizit anschreiben

  -> Merge requests ziehen sich über Monate

- Wenig Hoffnung auf Merge vor nächster LTS

The good

- Bessere Message Latenz
  - +-(3-5) ms vs +-100us
  - Deutlich kleinere StdDev
- Topics sind schneller verbunden
  - ~1sec vs ~10ms
- Externe CMake Bibliotheken brauchen keinen Wrapper mehr
- Support für CMake Targets

Fazit

- Unerwartet grundlegende Bugs
  - Gefühlt Alpha

- Kommend von ROS 1:
  - Service/Action API weniger nutzerfreundlich,
    es ist einfach sich selber ins Bein zu schießen.
  - Parameter sind ein Vor und Rückschritt
    - Integriertes Dynamic Reconfigure
    - Keine Arrays of Structs
  - Keine Revolution im Vergleich zu ROS1

Fazit

- Kann man ROS2 in einem produktiven System nutzen ?
- Ja, aber
  - Benutzt, wen möglich, keine Multithreaded Executor
  - Hohe Wahrscheinlichkeit für Bugs

    -> eigene ROS2 Patches
  - In dem Moment wo ihr ein Bug habt

    -> Eigene Buildpipeline für ROS2

Fazit

- UNSERE Erfahrung war nicht positiv
  - Bugs die die Lauffähigkeit beeinflussen sind für uns nicht akzeptabel
  - Dies ist unserer speziellen Anforderung als Produktivsystem geschuldet
    - Der Kunde ist unzufrieden wenn das System nicht 24/7 durchläuft

Fazit

Wir wollten keine eigene ROS 1 Version pflegen,

jetzt pflegen wir eine eigene ROS 2 Version...