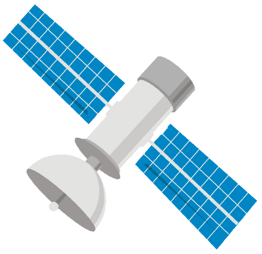
HOCHSCHULE HEILBRONN

Integrated Sensors - WS 20/21

REALTIME GPS-DASHBOARD

Realtime Vergleich des NEO-M8U Moduls und dem eingebauten Modul eines Smartphones

Professor: Dr.-Ing. Jürgen Doneit
Dozent: Ulrich Strauß



Mitglieder:
Timo Volkmann, 199267
Frank Herkommer, 194788

1. Kurzfassung

Ursprüngliches Ziel dieses Projekts war ein Realtime Dashboard für den Vergleich der 3D-Orientierung eines Smartphones und eines Ublox M8U, nach der Aufgabenstellung im Integrated Sensors Wiki.
Nach der Implementierung eines Software-Prototypen, dem Studium des Referenzgerätes M8U und der Smartphone Sensorik, kristallisierte sich jedoch heraus, dass wir dieses Unterfangen nicht mit unseren Ansprüchen und der Aufgabenstellung, zufriedenstellend in Einklang bringen konnten.
Nach einer Neuorientierung entschieden wir, die Grundzüge der Aufgabe – das Realtime Dashboard – mit einem neuen Use Case auszustatten und mit anderer Zielsetzung daran weiter zu arbeiten.

Das Ergebnis davon ist ein Realtime Dashboard, das Positionsdaten eines Referenzgerätes (Ublox M8U) mit denen eines Smartphones simultan und in annähernder Echtzeit darstellt. Es ist damit möglich, die Genauigkeit der Smartphone Position zu beurteilen, Aufnahmen von Fahrten zu erstellen und sogar Geräte übers Internet zu tracken. Die Trackings können aufgezeichnet werden und zu einem späteren Zeitpunkt zu Analysezwecken noch einmal in Echtzeit abgespielt werden. Alternativ können die Daten der verschiedenen Geräte eines Trackings in einer Analyseansicht komplett und übersichtlich verglichen werden.

Eine solche Technologie lässt sich unter anderem für folgende Use Cases einsetzen:

- Auswertung von Rennen,
- Drohnenentwicklung,
- Fahrzeugflotten-Tracking
- Evaluation und Analyse neuer Sensoren oder visueller Vergleich verschiedener IMU-Algorithmen

2. Einleitung

2.1. Ursprüngliche Zielsetzung

Erste Idee zu der Aufgabenstellung “M8U als digitales Gyroskop mit Smartphone Sensorik in Realtime Multiplex Ansicht vergleichen”, war die reinen Gyroskopdaten des M8U und eines Smartphones zu erfassen und zu vergleichen. Leider können die reinen Daten eines Gyroskops nur dann verglichen werden, wenn wir sicherstellen können, dass die wir mit beiden Geräten die absolut gleichen Bewegungen ausführen und sicherstellen, dass die Nullpunkte beider Gyroskope und deren Referenzrahmen bei der Messung exakt übereinstimmen. Zudem ist eine sinnvolle Nutzung der gewonnenen Daten hier sehr eingeschränkt.

Nach Rücksprache stellte sich hier heraus, dass mit dem “digitalen Gyroskop” die gesamte IMU-Einheit gemeint ist, mit der sich theoretisch Orientierung und Position im Raum bezogen auf einen Referenzpunkt ermitteln lässt. Damit ließe sich der Use Case umsetzen, die Genauigkeiten der Trägheitsnavigation beider Geräte gegenüberzustellen und zu vergleichen. Jedoch sind wir dafür auf neue Einschränkungen gestoßen, die uns dazu gezwungen haben unsere Zielsetzung noch einmal zu überdenken.

2.2. Anpassung der Zielsetzung

Unsere eigentliches Ziel war die absolute Orientierung zweier Geräte (Smartphone / M8U) im Raum als zentrales Datum und außerhalb eines Fahrzeugs über ein Realtime Dashboard sinnvoll und gewinnbringend zu vergleichen.

Eine Reihe von Punkten ließen am Mehrwert und der Machbarkeit unseres Vorhabens in dieser Form Zweifel aufkommen. Die nachfolgenden Punkte begründen, was uns zum Überdenken unseres Projekts und damit der Anpassung unserer Zielsetzung gebracht hat.

1. **Kopplung der IMU an das GNSS-System und Notwendigkeit der Kalibrierung des Ublox M8U**
   Der M8U wurde für den Einsatz zur Positionsbestimmung in Fahrzeugen konzipiert. Daher ist dieses IMU-System nicht ohne weiteres zur Bestimmung der Orientierung und Position ohne Rücksicht auf das GNSS möglich. Zudem muss das Inertial Navigation System (INS) des M8U kalibriert werden, um fusionierte Daten über Orientierung erhalten zu können, was eine Montage in einem Fahrzeug und eine Fahrt zur Kalibrierung erfordert. Wird das Gerät dann von seinem Montage-Platz im Fahrzeug entfernt, dauert es nicht lange bis die Kalibrierung ihre Gültigkeit verliert und keine fusionierten IMU-Daten und damit keine absoluten Orientierungsdaten mehr zur Verfügung stehen.

   Es wäre durchaus interessant zu evaluieren, wie sich Geräte, die für unterschiedliche Einsatzzwecke konzipiert sind, unter anderen Bedingungen eignen. Doch die

Einschränkung der Verfügbarkeit der Daten, insbesondere des M8U, lassen hier keine Verwendung des INS in anderem Kontext zu.

UPDATE 06.01.2021: Der Betriebsmodus ist auf Fußgängermodus oder einen generischen Modus umstellbar, was die Kalibrierungsfahrt hinfällig macht. Durch diese und nachfolgende Erkenntnis, ließe sich die ursprüngliche Idee wie geplant umsetzen. Dies haben wir jedoch nicht mehr mit einfließen lassen.

2. **Frequenz der Orientierungsdaten des M8U**
   Selbst mit Kalibrierung sind die Daten zur absoluten Orientierung des Fahrzeugs nur mit der Maximalfrequenz des GNSS-Receivers (2Hz) abrufbar. Um diese Daten sinnvoll vergleichen zu können und der Echtzeit-Anforderung gerecht zu werden, sind höhere Raten notwendig.
   Der Ublox M8U soll außerdem als Referenzgerät dienen. Dieser Aufgabe kann er jedoch nicht gerecht werden, wenn ein einfaches Smartphone eine höhere Auflösung an Messpunkten über die Zeit bietet.

   UPDATE 03.01.2021: Mit dem Update von U-Center von v20.06 auf v20.10 wurden der Message-Typ HNR-ATT in U-Center hinzugefügt. Damit lassen sich die Orientierungsdaten mit bis zu 30 Hz abrufen. Dieser Message-Typ erfordert ein Firmware-Update auf Protocol Version v19.20 (momentan v19.00). Dieses Hindernis würde daher nach einem Firmware-Update entfallen.

3. **Relative Daten**
   Um die beiden obigen Probleme zu umgehen, hätten wir von beiden Geräten relative und von Schwerkraft bereinigte Winkel- und Linear-Beschleunigungswerte in zufriedenstellender Frequenz bekommen. Dies hätte aber bedeutet, einen eigenen Sensor-Fusions-Algorithmus zur Bestimmung der absoluten Orientierung zu implementieren (z.B. Kalman-Filter oder Mahony-Madgwick-Filter). Außerdem hätten wir die geräte-internen Algorithmen umgangen, welche durch das Realtime Dashboard mit begutachtet werden sollten. Diese Tatsache und der geschätzte Aufwand dieses Weges, ließ uns diesen Weg verwerfen.

4. **Nur den Einsatzbereich auf Fahrzeuge beschränken**
   Nur den Scope auf Benutzung in einem Fahrzeug (Auto, Motorrad, Drohne, Flugzeug o.Ä.) zu beschränken, wäre möglich gewesen. Dies hätte aber immer noch nicht unseren Anspruch an die höheren Datenraten des Referenzgerätes erfüllt.
   Die Daten zur Orientierung im dreidimensionalen Raum in Echtzeit zu vergleichen, ist hier weniger interessant, zumindest wenn man von einem Auto ausgeht, unserem einzigen verfügbaren Testfahrzeug.

2.3. Neuer Scope und Ziel

Um möglichst nah an der Aufgabenstellung zu bleiben und den Prototypen nicht komplett verwerfen zu müssen, haben wir uns daher entschlossen, lediglich den Fokus der zu vergleichenden Daten zu ändern, ohne den Bezug zu den IMU-Daten komplett zu verlieren.

Statt also den Fokus auf das “digitale Gyroskop” zu legen, fokussieren wir uns nun auf den Vergleich der Positionsbestimmung beider Geräte und beschränken den Anwendungsbereich auf den Betrieb in einem Fahrzeug. So können wir allen Ansprüchen gerecht werden.

Insgesamt gelten für unser Realtime Dashboard nun folgende Ziele und Rahmenbedingungen:

- Vergleich von Positionsdaten eines Referenzgerätes (Ublox M8U) mit denen eines Smartphones in Echtzeit
- Schnittstelle für die Verarbeitung von Positionsdaten von Android und iOS
- Beurteilung der Genauigkeit der Position des Smartphones zu einem bestimmten Zeitpunkt, evtl. Warnung bei zu großer Ungenauigkeit
- Möglichkeit Aufnahmen anzufertigen und zu einem späteren Zeitpunkt wiederzugeben
- Anzeige einer Auswertung eines aufgenommenen Trackings
- Scope begrenzt auf den Einsatz im Fahrzeug (um Vorteile des INS des M8U nutzbar zu machen, Testfahrten nur mit Auto möglich)
- Vermeiden von Neuimplementierung einer eigenen Smartphone-Applikation zur Datensammlung und -weiterleitung aus Zeitgründen
- Mind. Nutzung der HNR (High Navigation Rate) Funktion des M8U, wenn verfügbar Messung auch auf dem Smartphone in hoher Frequenz

UPDATE 3.01.2020: Nachdem Ublox nun die Orientierungsdaten als HNR (High Navigation Rate) Messages zu Verfügung stellt, können wir unter Beibehaltung des Scopes Einsatzgebiet Fahrzeug auch den ursprünglichen Plan zukünftig wieder besser mit einfließen lassen.

3. Umsetzung

3.1. Hard- und Software Komponenten

Hardware

- Adafruit Ublox NEO-M8U mit Antenne (Linx ANT-LTE-RPC-UFL)
- Windows-Rechner oder VM zur Konfiguration des M8U über U-Center
- mobiler Rechner (Windows/Mac/Linux, mit WiFi & USB/Serial)
- Smartphone mit GPS, IMU und Hotspot/WiFi (iOS oder Android)

Software

- U-Center (Windows, Version >20.06)
- SensorLog (iOS, kostenpflichtig), HyperIMU (Android)
- Browser mit HTML5, WebGL und JS Support
- *“gyrogpsc”*

Entwicklungs Tools

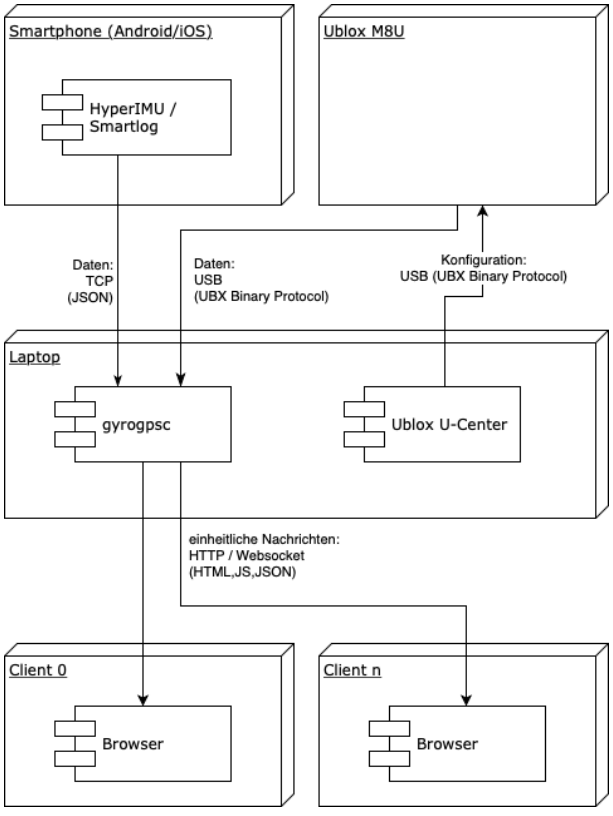
- Insomnia (Testen der HTTP-API) (optional)
- GoLand/IntelliJ (Jetbrains IDE für Go) (optional)

Wir haben uns für Go im Backend entschieden, da dies eine moderne, relativ leicht erlernbare Sprache ist, die First-Class-Support für nebenläufige Programmierung bietet, dabei statisch typisiert, kompiliert und sehr leicht auf verschiedenen Plattformen einzusetzen ist. Die einfache Handhabung der Nebenläufigkeit ist Hauptgrund für die Wahl dieses Werkzeugs. Später noch mehr dazu.

3.2. Systemarchitektur

Überblick

Grundsätzlich ist das System so aufgebaut, dass die Daten der Sensoren per TCP (Smartphones) und USB (M8U) an den Server gesendet werden. Das bedeutet, Smartphone und M8U müssen so konfiguriert werden, dass diese Ihre Daten selbstständig über die entsprechenden Kanäle pushen.
Der *gyrogpsc* Server lauscht auf den entsprechenden Schnittstellen[1] und leitet die Daten weiter an die entsprechenden Parser[2], die die Daten in eine einheitliche, interne Repräsentation überführen.
Hier können später noch beliebige Stream-Processing-Schritte eingebaut werden, unter anderem dachten wir hier über eine Out-Of-Order-Event-Synchronisation nach. Dazu aber später noch mehr.
Die Daten werden dann an einen Dispatcher übergeben, der die Daten an alle verbundenen Clients verteilt (Multiplexing). Somit können beliebig viele User ein und demselben Stream lauschen.
Wird eine Aufnahme über die HTTP-API gestartet, werden die Nachrichten bis zur Beendigung der Aufnahme In-Memory gesammelt und nach Beendigung auf einmal auf der Festplatte in einer Key/Value-Datenstruktur persistiert.



Überblick Gesamtsystem

[1] Siehe Source-Code: web/http und core/collectors.go
[2] Siehe Source-Code: core/datamodel.go:190 und ublox/decode.go:81
https://github.com/daedaleanai/ublox

3.3. Datenquellen – Einrichten der GPS-Module

Um die Datenquellen für den Betrieb im System nutzen zu können, müssen ein paar Vorbereitungsschritte durchgeführt werden.

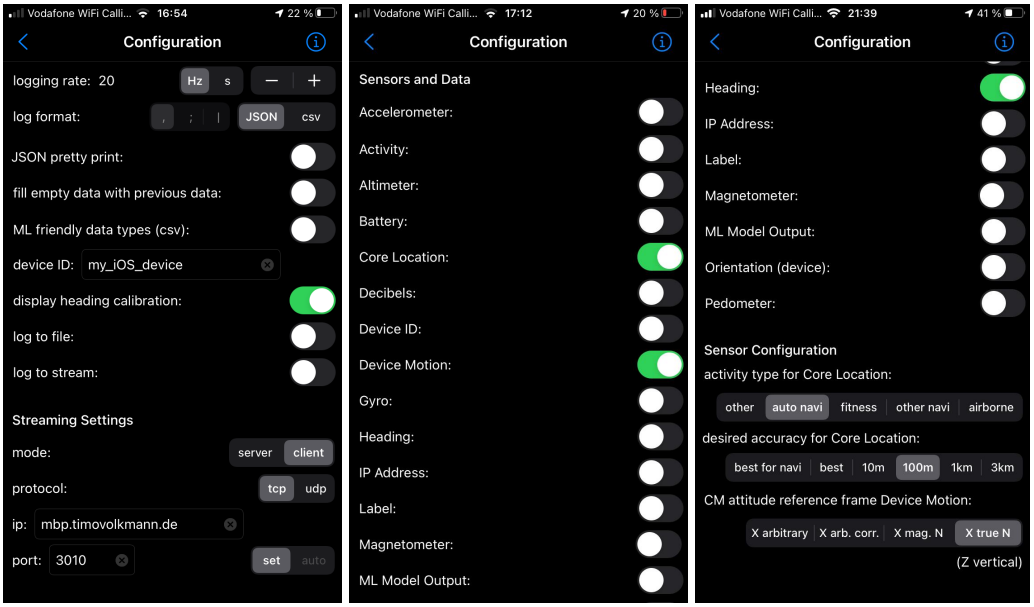
Smartphone

Je nach Modell (iOS oder Android) haben wir uns hier bereits existierender Apps bedient, um die Daten der Sensoren auszulesen und an unsere Anwendung zu senden.

Wir haben uns hier für den drahtlosen Übertragungsweg entschieden, da eigentlich jedes Smartphone und jeder Rechner ein WiFi-Modul haben. Tests haben ergeben, dass sich die Latenz hier im <10ms-Bereich für ein typisches Home-WiFi-Netz bewegt. Für das Transportprotokoll haben wir uns für reines TCP entschieden, da aufgrund der hohen Datenrate eine stehende Verbindung weniger Overhead mit sich bringt als z.B. HTTP, das nicht für hochfrequentierte kurze Nachrichten ausgelegt ist, da für jede Anfrage eine neue Verbindung aufgebaut wird. Weiter bietet TCP die Sicherheit, dass keine Nachrichten verloren gehen, was beispielsweise mit dem verbindungslosen UDP-Protokoll nicht gegeben wäre.

Apps hierfür gibt es einige. Unser Programm unterstützt JSON-Messages der Software *HyperIMU*[3] für Android und *SensorLog*[4] für iOS.

Konfiguration SensorLog:

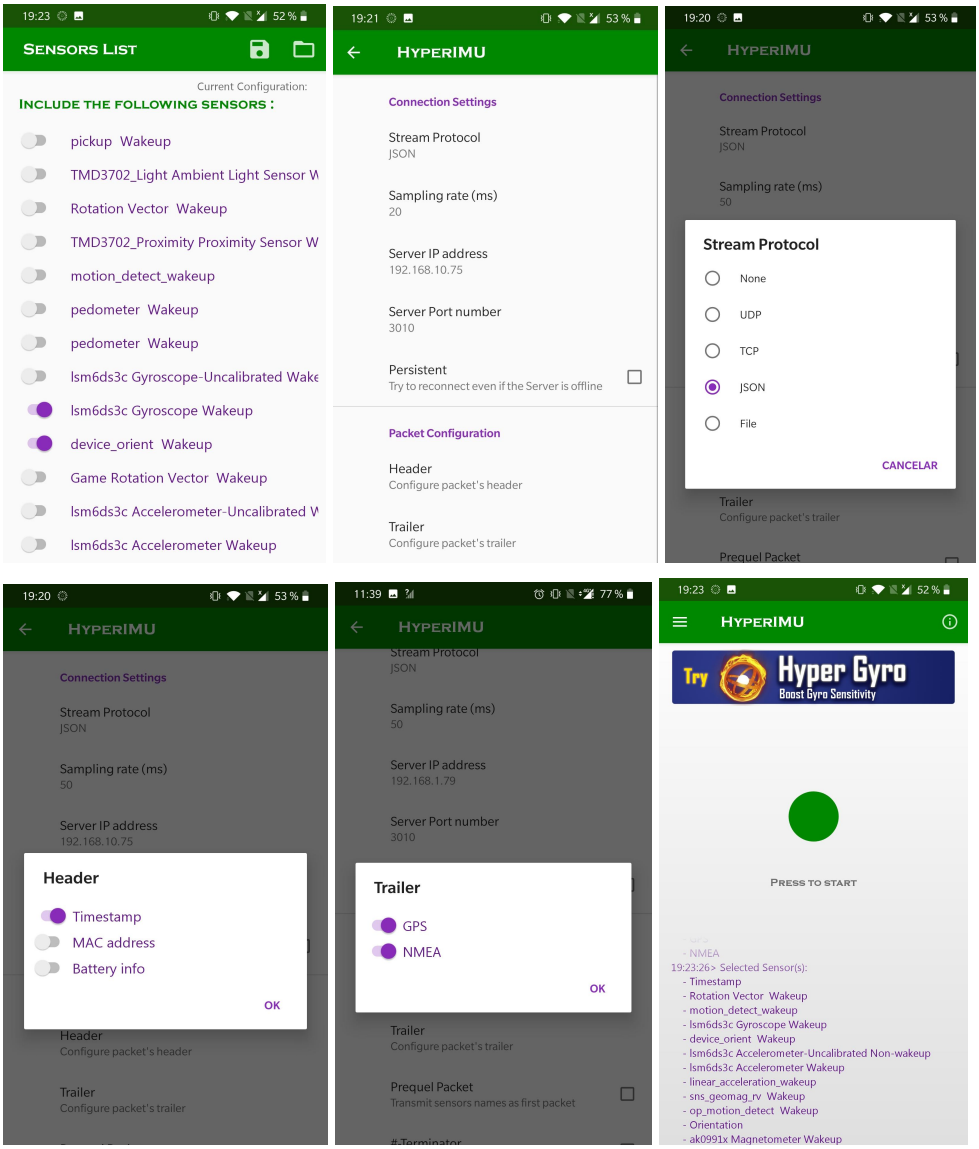


Dies ist die Grundkonfiguration, die wir auch für unsere Test verwendet haben. Der Host bzw. die IP-Adresse muss selbstverständlich auf den Host angepasst werden, auf dem der

[3] https://ianovir.com/works/mobile/hyperimu/hyperimu-help/
[4] http://sensorlog.berndthomas.net

gyrogpsc Server läuft. Wenn der *gyrogpsc* Server gestartet ist, kann mit dem Toggle “log to stream”, die TCP-Verbindung geöffnet werden. Die Messung muss danach seperat vom Startbildschirm gestartet werden.
Achtung: Die Verbindung muss nach einem Neustart des *gyrogpsc* Servers erneut hergestellt werden, auch wenn sie laut dieser App noch aktiv ist.

Konfiguration HyperIMU:

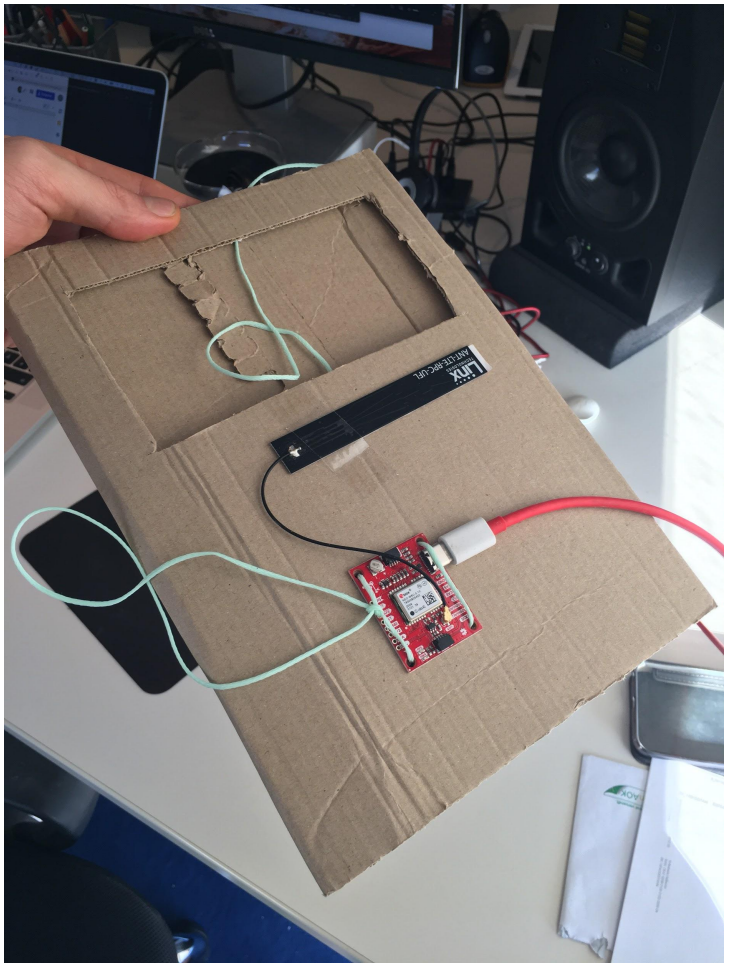


Dies ist die Konfiguration in der HyperIMU App. Hier muss natürlich auch die IP-Adresse auf den Host angepasst werden. Header und Trailer des Pakets müssen auch eingestellt werden: Im Header wollen wir einen Zeitstempel übergeben und im Trailer die GPS und NMEA Daten. Wenn der *gyrogpsc* Server gestartet ist, kann im Startbildschirm mit dem Button “Press to start”, die TCP-Verbindung geöffnet werden.
Wenn der *gyrogpsc* Server gestoppt ist, wird auch in der App die Verbindung unterbrochen und als inaktiv angezeigt.

Achtung: Da die HyperIMU App keine Genauigkeits-, Richtungswinkel (Heading/Kompass) und Geschwindigkeitsdaten liefert, kann kein vollständiger Vergleich mit den Daten des Ublox Moduls gemacht werden. Daher ist für diese App die Funktionalität des Dashboards momentan noch eingeschränkt.

Ublox M8U

Das M8U Modul wird per USB am Rechner angeschlossen und wird auch über diesen mit Strom versorgt. Voraussetzung für einwandfrei Funktion ist eine angeschlossene Antenne und die Montage im Fahrzeug, damit das INS sich kalibrieren lässt. In unserem Fall hat eine “mobile Montage” auf einem Stück Pappe genügt, das wir zu Testfahrten einfach zwischen Windschutzscheibe und Armaturen geklemmt haben.



Die Konfiguration erfolgt über U-Center. Wichtig ist, dass das UBX Protokoll aktiviert ist und folgende Messages für die USB-Schnittstelle aktiviert werden:

- UBX-NAV-ATT
- UBX-NAV-PVT
- UBX-HNR-PVT

Zusätzlich muss die Frequenz für die HNR (High Navigation Rate) Navigation gesetzt werden. Wir haben mit einer Frequenz von 20Hz gearbeitet.
Ist die Konfiguration erfolgt, muss sie auf dem M8U persistiert werden, um einen Neustart zu überdauern. Dies kann ebenfalls über die UBX-Message CFG-CFG erfolgen. Details zur Verwendung von U-Center und zur Konfiguration des Ublox-Moduls finden sich in folgenden Dokumenten:

- M8 Receiver Protocol Description[5]
- U-Center Userguide[6]

Ist die Konfiguration abgeschlossen, kann der M8U direkt verwendet werden. Da jedoch die Kalibrierung jedes Mal einige Zeit dauern kann, ist es ratsam vom SOS-Feature (Save-on-Shutdown) des Geräts gebrauch zu machen:

- Kalibrierungsfahrt mit U-Center und laufendem M8U durchführen.
- Sobald “DR-Fix” verfügbar ist, bzw. der Fusion Filter Mode (UBX-ESF-STATUS) aktiv ist, per UBX-UPD-SOS den Befehl zum speichern absenden.
- Danach ist die Kalibrierung auch auf dem Flash Speicher des Gerätes persistiert.

Achtung: Wenn die Konfiguration danach nochmals geändert werden soll, muss das SOS Backup gelöscht werden, da dies jegliche neue Konfiguration wieder überschreibt.

[5] https://www.u-blox.com/en/docs/UBX-13003221
[6] https://www.u-blox.com/sites/default/files/u-center_Userguide_(UBX-13005250).pdf

3.4. “gyrogpsc” – Unsere Streaming Zentrale

gyrogpsc ist unsere Server-Software, die folgende Aufgaben übernimmt:

- Sammeln und Parsen von JSON-formatierten Positions- und Orientierungsdaten von HyperIMU oder SmartLog über TCP[7]
- Sammeln und Parsen von UBX-formatierten Positions- und Orientierungsdaten vom Ublox Modul über USB/Serial[7]
- Aufnahme und Wiedergabe von Streams von TCP/Serial bzw. M8U/Smartphone, auch Aufnahme von nur einem Sensor möglich
- Persistierung der Daten bei Aufnahme
- Ausspielen der Daten in visuell aufbereiteter Form an beliebig viele Clients über HTTP und Websockets

Producer-Consumer-Modell[8]

Wir haben mehrere Erzeuger (die GPS-Module) und mehrere Verbraucher (Clients). Für beide Gruppen müssen separate Verbindungen (TCP, Serial, HTTP, Websockets) unterhalten werden. Die Daten der GPS-Module müssen in mehreren Zwischenschritten weiterverarbeitet werden (Parsen, Timestamping, Persistierung, …), bevor sie an die Clients ausgespielt werden können. Die Erzeuger können mit sehr unterschiedlichen Nachrichten-Frequenzen betrieben werden und während des Processings der Daten können bereits neue Daten eintreffen.

Um das alles sauber zu managen, ist es sinnvoll die Aufgaben auf mehrere Prozesse zu verteilen und dem Betriebssystem bzw. der Runtime das Scheduling zu überlassen (*Goroutinen*). Das Producer-Consumer-Modell ermöglicht es uns, diese Aufgaben effizient in mehreren Prozessen abzuarbeiten. Sobald Daten eintreffen, werden diese über *Channels* (native Go-Datenstrukturen für Prozess-Synchronisation) an die weiterverarbeitenden Prozesse übergeben. Somit ist der für den Empfang der Daten verantwortliche Prozess sofort wieder bereit die nächsten Daten entgegenzunehmen und jede Nachricht wird sofort weiterverarbeitet, was die Latenz des System niedrig hält. Gleichzeitig können wir so bei hohen Raten, wenn neue Nachrichten schneller eintreffen als das Processing dauert, das Processing parallelisieren indem wir es auf mehrere CPU-Kerne verteilen. Außerdem bleibt bei niedriger Last der CPU-Verbrauch niedrig, da die Prozesse blockieren, wenn es keine Daten zu verarbeiten gibt. Ein weiterer Vorteil dieses Modells ist die Modularisierung und Möglichkeit später beliebig viele Erzeuger und Verbraucher hinzuzufügen.

Vorsicht ist jedoch beim parallelen Processing geboten. Dies kann die Reihenfolge der Nachrichten durcheinander bringen. Auch hierfür gibt es Lösungen, welche im folgenden Abschnitt näher behandelt werden. Tests haben außerdem ergeben, dass mit unseren

---

[7] siehe Source-Code: core/collectors.go

[8] https://www.ni.com/de-de/support/documentation/supplemental/21/producer-consumer-architecture-in-labview.html

Parametern (ca. 20-60 Nachrichten pro Sekunde und Erzeuger) ein paralleles Processing nicht notwendig ist und keine Daten durch volle Buffer verloren gehen.

Um das Processing in Pipelines zu organisieren und die Parallelität des Processings zu steuern haben wir uns einer Stream-Library[9] bedient.

Out-of-Order Nachrichten

Durch unsere Systemarchitektur und unterschiedlichen Datenquellen können wir aus nachfolgenden Gründen zunächst nicht für eine korrekte Reihenfolge der ausgespielten Nachrichten garantieren.

- Unterschiedliche und teilweise dynamische Latenzen bei der Anbindung der GPS-Module (WiFi+TCP und USB+Serial)
- Paralleles Processing der Daten

Die Lösung dieses Problems wäre, nach dem Processing und vor dem Ausspielen der Daten an die Clients einen Mechanismus einzubauen, der den Stream für eine kurze Zeit buffert und alle Nachrichten im Buffer nach den GPS-Zeitstempeln der Geräte sortiert.
Dabei muss der Bufferzeitraum so gewählt werden, dass möglichst die maximale Verzögerung einer Nachricht mit berücksichtigt wird, jedoch gleichzeitig so kurz wie möglich zu buffern, um die Latenz des Gesamtsystems nicht zu hoch werden zu lassen.

Im Frontend ergaben sich bei unseren Realtime-Tests teilweise Unterschiede ca 100ms, vereinzelt auch mal bis zu 200ms, zwischen den Daten vom Smartphone und den vom M8U. Um diesen Versatz auszugleichen, müsste also mindestens eine Latenz dieser Größe eingeführt werden.

Anmerkung: Da die GPS-Zeit bereits eine sehr hohe Genauigkeit besitzt (< 1 µs), ist es nicht notwendig, die Zeit der GPS-Module separat nochmal zu synchronisieren.
Unser naiver Ansatz hierbei war der Einsatz einer Priority Queue (Min Heap) mit einer fest eingestellten Verzögerung bzw. einer festen Größe an Elementen. Dies hat auch super funktioniert, jedoch war die dadurch eingeführte Latenz im Verhältnis zum Nutzen nicht tragbar, weswegen wir uns letztendlich **gegen** den Einsatz entschieden haben und die Implementierung wieder verworfen haben.
Die Praxis hat gezeigt, dass die Ungenauigkeiten der GPS-Module und die Schätzungen der Position durch die Sensor-Fusions-Algorithmen, in unserem Use-Case deutlich stärker ins Gewicht fallen als Out-of-Order Nachrichten und diese somit kaum bis gar nicht auffallen. Zudem sind zu keinem Zeitpunkt Nachrichten eines einzelnen Gerätes Out-of-Order (gilt nur für nicht paralleles Processing).

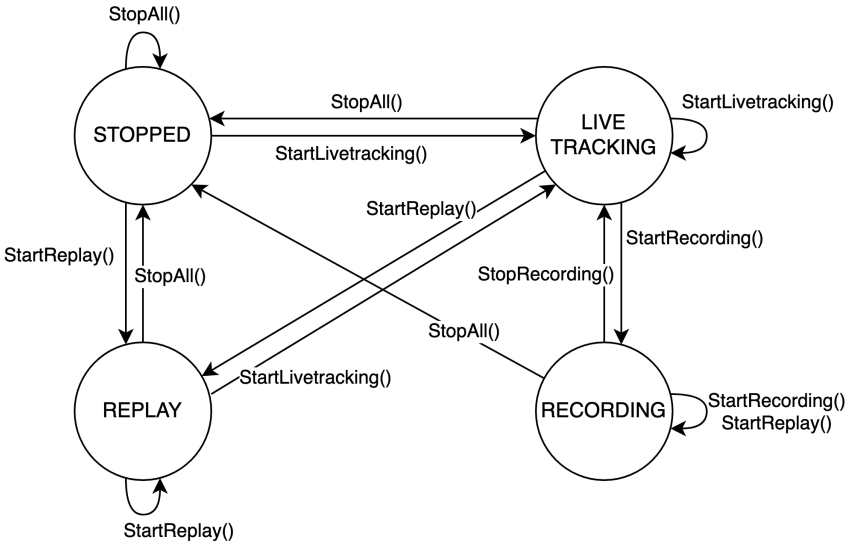
[9] https://github.com/reugn/go-streams und
Source-Code: core/pipeline_record.go, core/pipeline_replay.go

Websockets[10]

Da unsere Clients mit HTML und JavaScript im Browser laufen und HTTP standardmäßig dem Pull-Modell folgt, mussten wir einen zusätzlichen Kanal integrieren, der uns möglich macht den Nachrichten-Flow von der Quelle bis zum Endverbraucher einheitlich mit dem Push-Modell umzusetzen. Websockets sind die State-of-the-Art Lösung für diesen Anwendungsfall. Glücklicherweise bieten moderne Browser bereits eine Websocket Schnittstelle, die sich sehr einfach per JavaScript integrieren lässt. Somit können Nachrichten direkt nach dem Processing über einen dauerhaft offenen Websocket an den Client gestreamt werden ohne dass der Client erst danach fragen muss und es entsteht kein Overhead durch ständig sich ständig neu auf- und abbauende Verbindungen. Sobald ein Client die Seite im Browser lädt, wird eine Websocket Verbindung aufgebaut und diese beim Dispatcher[11] (Multiplexing) registriert.
Sollte die Verbindung verloren gehen, wird ein Timer gestartet der in regelmäßigen Abständen versucht, erneut eine Verbindung herzustellen.

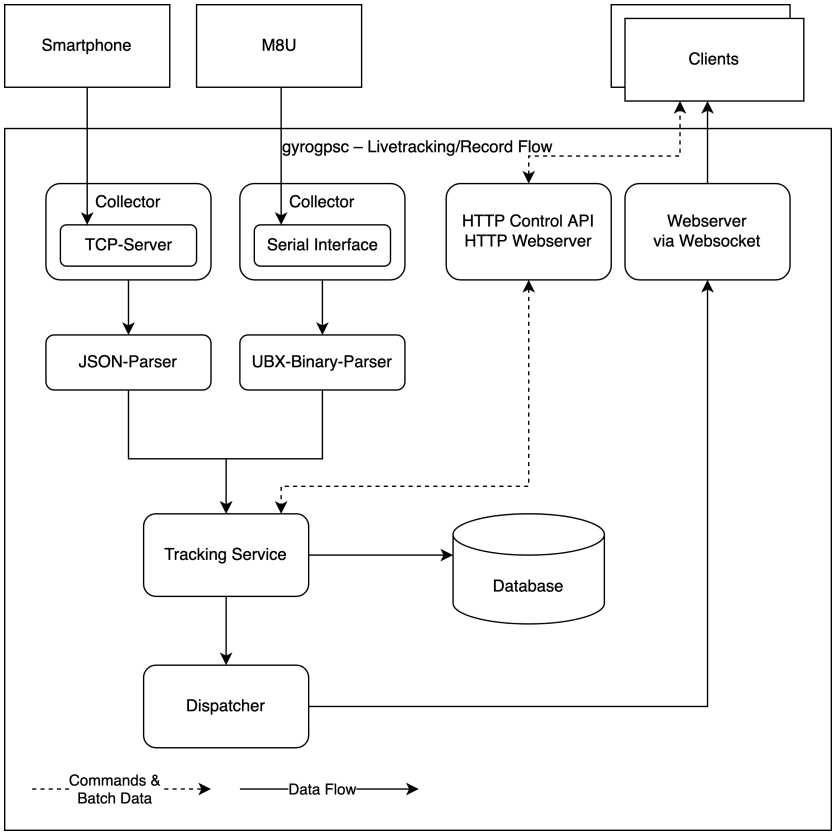
Betriebsmodi[12]

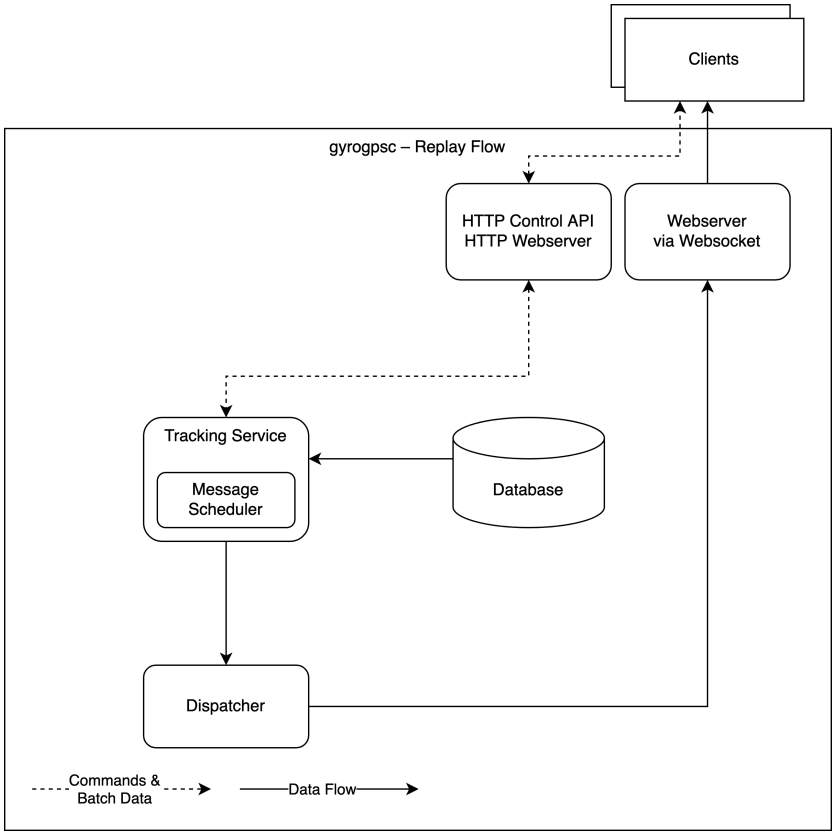
Um sicherzustellen, dass unsere Anwendung keine Funktion mehrfach zur selben Zeit ausführt, zu wissen in welchem Zustand sie sich zu jedem Zeitpunkt befindet und festzulegen welche Aktion wann ausgeführt werden darf, haben wir eine State Machine in unserer Anwendung implementiert:



[10] Siehe Source-Code: web/http.go:194
[11] Siehe Source-Code: core/dispatcher.go
[12] Siehe Source-Code: core/service.go

Nachfolgende Diagramme skizzieren den Fluss der Daten in der Applikation während des Live Trackings bzw. der Aufnahme und während der Wiedergabe eines Trackings.





Persistenz

Zur Speicherung unserer Aufnahmen, haben wir uns aus folgenden Gründen für eine eingebettete Key-Value-Datenbank[13] entschieden:

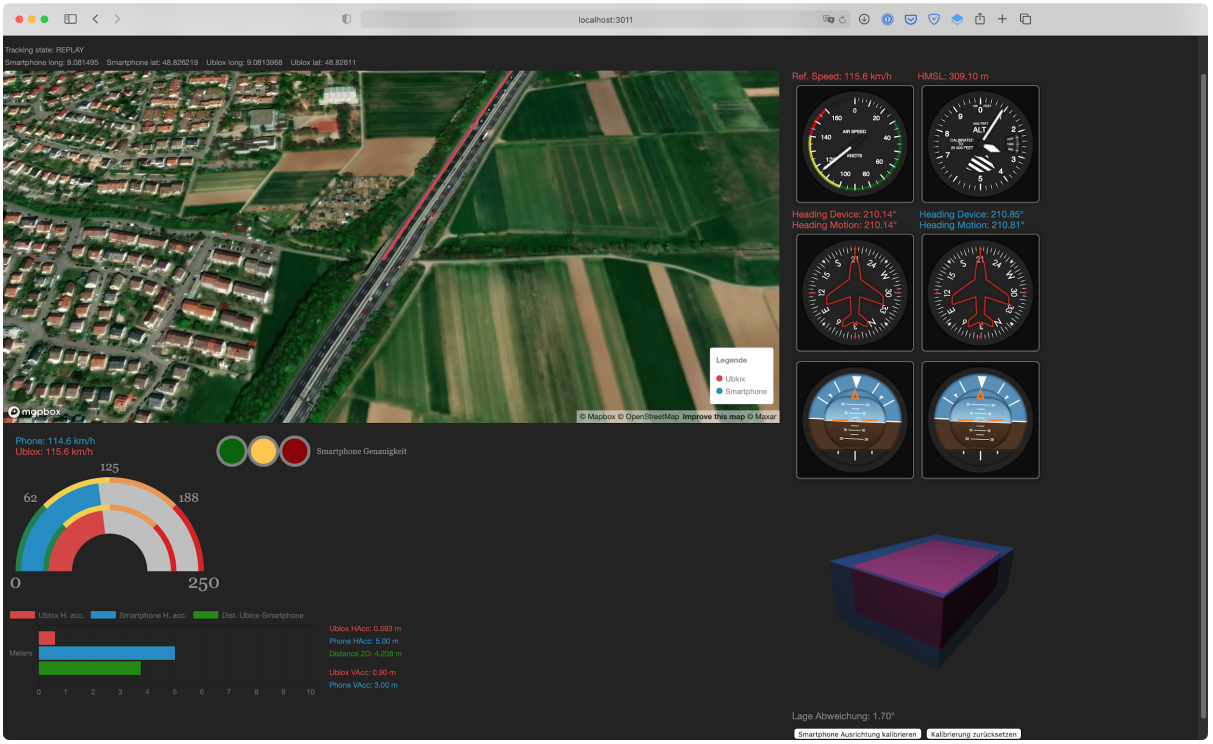
- keine Schemas wie bei SQL, da relativ simple Datenstrukturen
- keine externen Abhängigkeiten
- sehr performant, evtl. wichtig für sehr lange Aufnahmen

Wir haben zudem ein Interface im Quellcode bereitgestellt, das es ermöglicht, die Implementierung der Datenbank auszutauschen bzw. zu erweitern.

Die Benutzeroberfläche

Zum Realtime Vergleich der beiden Geräte haben wir uns für eine Weboberfläche entschieden, welche mit Javascript und HTML läuft und unter http://localhost:3011 aufgerufen werden kann.

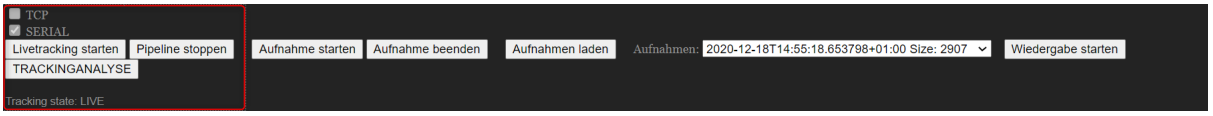
Live Tracking, Tracking aufnehmen und Aufnahmen abspielen



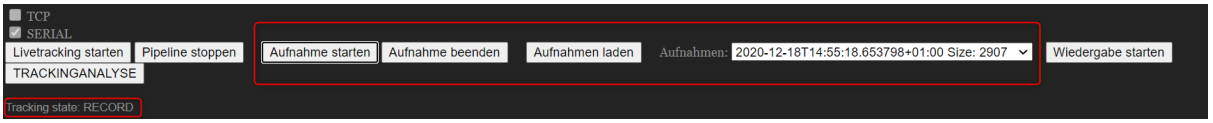
[13] https://github.com/dgraph-io/badger/ und Source-Code: storage/kvstore.go

Menüleiste:

Auf der Live Tracking Seite unserer Weboberfläche hat der User die Möglichkeit in Echtzeit, unter anderem, GPS-, Geschwindigkeits- und Orientierungsdaten eines oder beider Geräte einzusehen indem mit hilfe der Checkboxen die Verbindungsart gewählt wird und auf den “Livetracking starten” Button geklickt wird. Mit dem Button “Pipeline stoppen” wird der Datenfluss gestoppt.
Das darunter stehende Label “Tracking state” gibt aus, welcher Betriebsmodus momentan ausgeführt wird und zeigt uns in diesem Fall “LIVE”.



Des Weiteren kann eine Aufnahme des Trackings mit den Buttons “Aufnahme starten” und “Aufnahme beenden” gemacht werden, welche in einer Datenbank gespeichert wird und durch den Button “Aufnahme laden” auch wieder abrufbar ist, um sie erneut abzuspielen. Auch hier zeigt uns das Label “Tracking state” bei Aufnahme “RECORD” und bei Wiedergabe des Trackings “REPLAY” an.



Visualisierung der Daten:

Zur Visualisierung der Sensordaten haben wir in erster Linie mit der Javascript Bibliothek Mapbox GL JS[14] eine Karte implementiert, auf welcher aus einkommenden GPS Daten der Sensoren eine Linie gezeichnet und somit auch die aktuelle Position des Gerätes dargestellt wird. Eine rote Linie zeichnet die registrierten GPS Daten des Ublox moduls und eine blaue Linie die des Smartphones. Über der Karte werden die aktuellen Positionen der Geräte in Longitude und Latitude (Längengrad und Breitengrad) angezeigt.

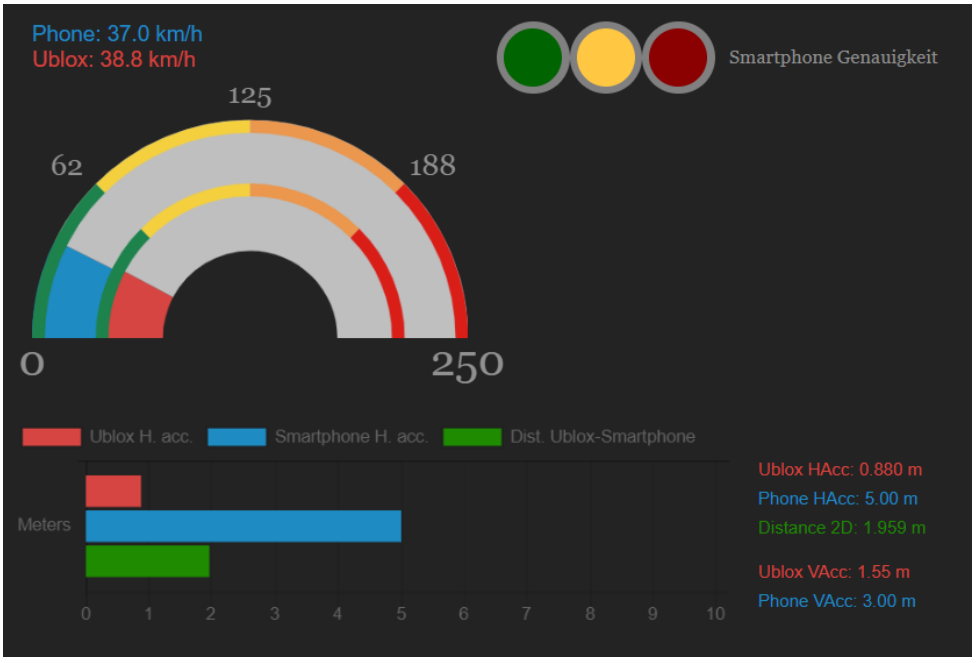
[14] https://docs.mapbox.com/mapbox-gl-js/api/

Zunächst haben wir mit der Javascript Bibliothek Chart.js[15] eine Art Tachometer implementiert, welcher die aktuelle Geschwindigkeit in km/h beider Geräte darstellt. Weiter unten wird auf einem Balkendiagramm die horizontale Genauigkeit (H. acc) in Meter beider Sensoren angezeigt. Zudem wird auch die Distanz in Meter zwischen beiden Geräten angezeigt, welche durch die *Vincenty Formel*[16] ermittelt wird. Dies machen wir zum Zeigen, ob die Position des Smartphones sich innerhalb des Toleranzbereiches des Referenzgeräts befindet (unter der horizontalen Genauigkeit des Ublox Moduls). Oben zu sehen auf der Abbildung unten sind drei Kreise die Lämpchen darstellen sollen. Das grüne Lämpchen leuchtet wenn sich die berechnete Distanz unter der horizontalen Genauigkeit des Ublox moduls befindet. Das gelbe Lämpchen leuchtet wenn sich die Distanz über der Genauigkeit des Ublox aber unter der Genauigkeit des Smartphones befindet. Das rote dagegen leuchtet nur wenn sich die Distanz über der Genauigkeit beider Geräte befindet.

Ein Problem dabei war, dass Positionsdaten beider Sensoren nicht zum exakt selben Zeitpunkt bei dem Server ankommen. Das heißt, die Distanzen werden mit Messpunkten die nicht zum exakt selben Zeitpunkt stattfanden berechnet.

Lösungsansatz für eine genauere Abschätzung der Distanz[17]:

1. Berechnen der Zeitdifferenz zwischen den aktuell zwischengespeicherten Messungen des Smartphones und Ublox Moduls.
2. Fehlerabschätzung: Berechnen der zurückgelegten Strecke in der Zeitdifferenz basierend auf der Geschwindigkeit des Ublox Moduls.
3. Subtrahieren der Fehlerabschätzung von der ursprünglich berechneten Distanz.

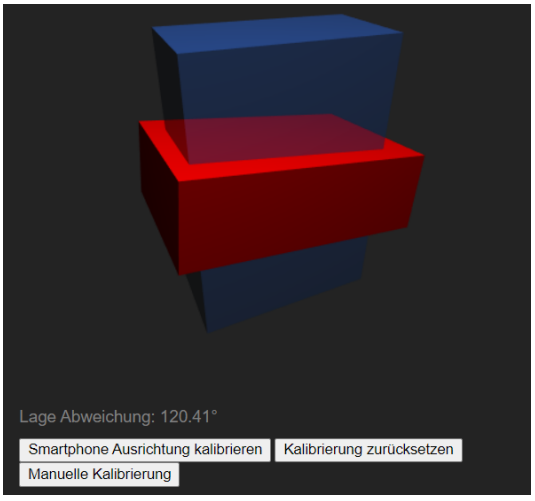


[15] https://www.chartjs.org/
[16] Siehe Source-Code: static/scripts/distanceCalc.js:37
[17] Siehe Source-Code: static/scripts/websocket.js:78-87

Rechts auf der Live Tracking Seite sind erst noch einmal in einem etwas schöneren Layout die Geschwindigkeit und Höhe in Meter des Referenzgeräts (Ublox) zu sehen.
Darunter werden auch die Richtungswinkel beider Geräte auf einem Kompass und deren Neigungen auf einem künstlichen Horizont angezeigt. Auf der linken Seite die Daten des Ublox Moduls und auf der rechten die Daten des Smartphones. Für diese Layouts haben wir das *jQuery-Flight-Indicators*[18] Plugin in unser Projekt importiert.

Im untersten Teil der Live Tracking Seite sind zwei Quader welche die Ausrichtung des Fahrzeugs repräsentieren. Die Ausrichtung des Smartphones kann kalibriert werden und zurückgesetzt werden. Hierfür laufen im Hintergrund Berechnungen mit Quaternionen. Siehe Code Dokumentation[19]. Dies bietet folgende Vorteile:

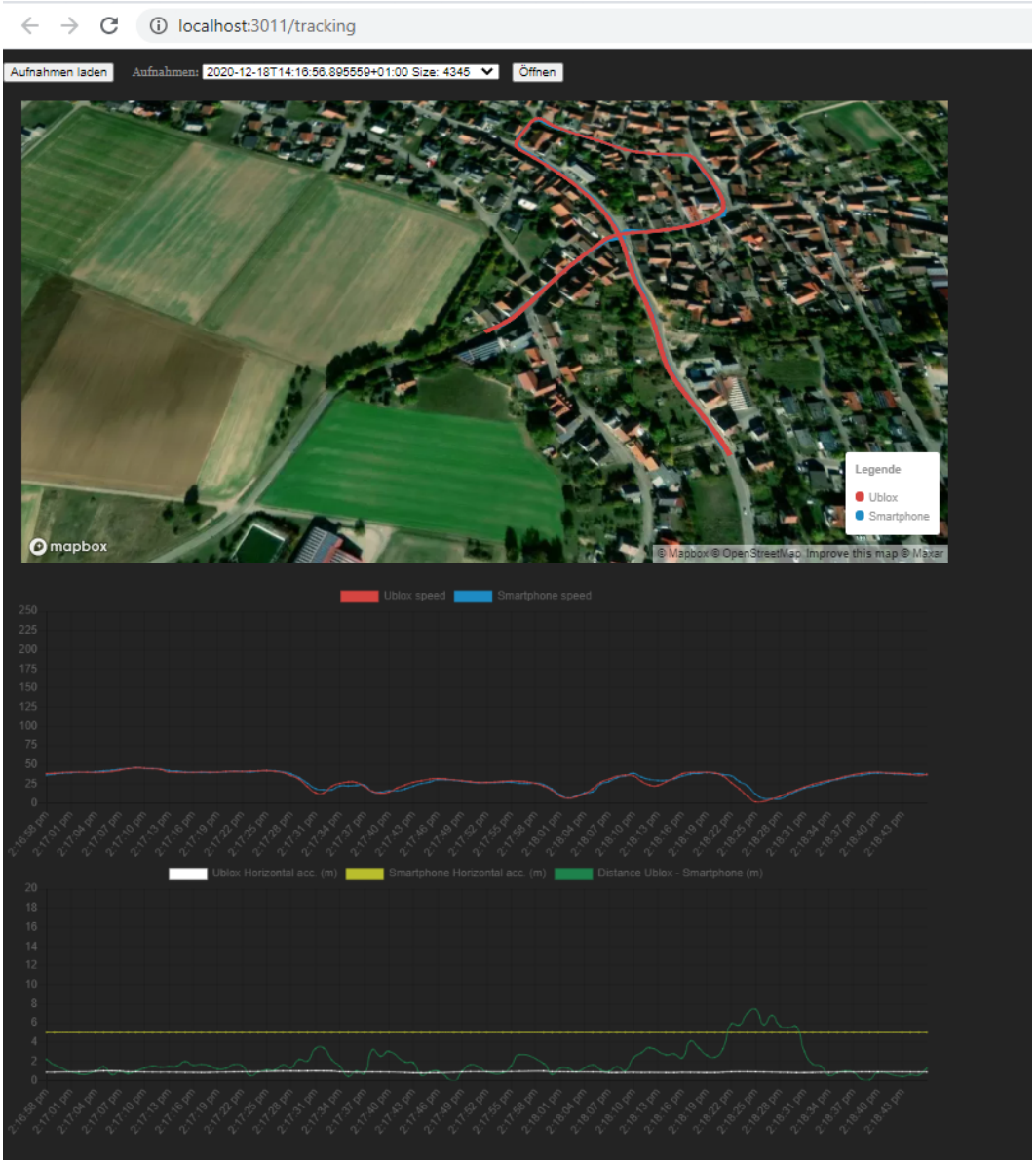
- Rechnen mit Drehungen
- Nullen: Angleichen der Ausgangsdrehung
- Kein Gimbal Lock
- Anzeige der Abweichung der Orientierung in 3D



[18] https://github.com/sebmatton/jQuery-Flight-Indicators
[19] Siehe Source-Code: static/scripts/indicators.js

Tracking Analyse

Auf der zweiten Seite unserer Weboberfläche, welche mittels dem Button “TRACKINGANALYSE” abgerufen werden kann, ist eine Analyse einer Aufnahme nach Auswahl einzusehen. Hier kann durch den Button “Aufnahmen laden” eine Liste aller Aufnahmen angezeigt werden. Mit dem Button “Öffnen” wird das ausgewählte Tracking abgerufen.
In erster Linie werden auf einer Karte die gefahrenen Routen beider Geräte abgebildet. Darunter werden zwei Graphen gezeichnet. Der erste Graph gibt uns eine Gesamtübersicht der Geschwindigkeiten beider Geräte über die Zeit der Aufnahme. Der zweite Graph gibt uns, falls vorhanden, eine Gesamtübersicht der Genauigkeiten beider Geräte und die Distanz zwischen dem Ublox Modul und dem Smartphone über die Zeit der Aufnahme. Somit können wir die Daten der beiden Geräte viel übersichtlicher vergleichen.



4. Ausblick & mögliche Einsatzgebiete

4.1. Sonstige Verbesserungen

Busy-Wait-Loop[20]

Der Replay-Mechanismus ist noch nicht optimal umgesetzt. Momentan wird hier ein CPU-Kern voll ausgelastet, da der Loop, der für das Scheduling der Nachrichten verantwortlich ist, permanent die Systemzeit abfragt um die Nachricht zum richtigen Zeitpunkt an die Streaming-Pipeline zu übergeben. Eine deutlich effizientere Variante wäre Timer (Sleep) des Betriebssystems bzw. der Runtime zu nutzen und den Prozess schlafen zu legen bis die Nachricht gesendet werden soll.

Stream Synchronisation / Out-of-Order Nachrichten

Falls es für bestimmte Anwendungen doch nötig sein sollte, die absolut korrekte Nachrichtenfolge einzuhalten, könnte man den Nutzer die Stream Synchronisation manuell über die GUI an- und abschalten lassen. Eine bessere Methode wie die von uns oben beschriebene, wäre z.B. eine Implementierung des *AQ-K-Slack-Algorithmus*[21]

Verbessertes Replay Handling

- Vor- / Zurückspulen
- Auswählen und Betrachten eines kleineren Zeitabschnitts innerhalb einer Aufnahme,
- Popups mit Werten bei Mouseover (Graph, Map)
- Playback Controls (Play, Pause, Stop, Springen zu beliebigen Zeitpunkt)

User Interface

- Deaktivieren von Display- und Steuerelemente, wenn zugehöriges Gerät nicht aktiv ist
- Zusätzliche Live-Ansicht ohne Steuerungselemente für simultane Read-Only Ansicht

Import/Export

Aufnahmen von Messfahrten könnten über folgende Formate leichter austauschbar sein:

- Excel
- JSON
- GPX

[20] https://en.wikipedia.org/wiki/Busy_waiting
[21] https://dl.acm.org/doi/10.1145/2675743.2771828 und https://medium.com/@Vithursa/buffer-based-events-handling-68c9e18cdb66

Verbesserte Android Unterstützung

Durch die späten Änderungen im Plan hat die gewählte Software HyperIMU für Android nicht mehr alle Ansprüche von Haus aus gedeckt, daher können wir zu diesem Zeitpunkt in dieser Konstellation leider keinen vollen Funktionsumfang bieten. Bei den fehlenden Parametern handelt es sich um Heading (Kompass), Speed und Location Accuracy.
Die fehlenden Daten für Heading und Speed können jedoch aus den Übertragenen abgeleitet bzw. berechnet werden. Dazu blieb jedoch keine Zeit. Alternativ könnte eine eigene App für das sammeln aller relevanter Daten direkt vom Smartphone entwickelt werden um alle relevanten Parameter des Smartphones zu erfassen.

5. Referenzen

Alle abgerufen am 15. Januar 2021.

HyperIMU:
https://ianovir.com/works/mobile/hyperimu/hyperimu-help/

SensorLog:
http://sensorlog.berndthomas.net

u-blox 8 / u-blox M8 Receiver description Including protocol specification:
https://www.u-blox.com/en/docs/UBX-13003221

u-center Userguide:
https://www.u-blox.com/sites/default/files/u-center_Userguide_(UBX-13005250).pdf

Go Streams Library:
https://github.com/reugn/go-streams

MapboxGL JS:
https://docs.mapbox.com/mapbox-gl-js/api/

Chart.js:
https://www.chartjs.org/

jQuery-Flight-Indicators:
https://github.com/sebmatton/jQuery-Flight-Indicators

Golang Getting Started:
https://golang.org/doc/tutorial/getting-started

Badger Embeddable Key-Value Datenbank:
https://github.com/dgraph-io/badger/

AQ-K-Slack-Algorithmus:
https://dl.acm.org/doi/10.1145/2675743.2771828
https://medium.com/@Vithursa/buffer-based-events-handling-68c9e18cdb66

UBX-Parsing Blueprint:
https://github.com/daedaleanai/ublox

Producer-Consumer-Pattern:
https://www.ni.com/de-de/support/documentation/supplemental/21/producer-consumer-architecture-in-labview.html

Busy Waiting:
https://en.wikipedia.org/wiki/Busy_waiting