Bachelorarbeit

Architektur und Implementierung eines modularen, skalierbaren Backends für Sensordaten verschiedener Nutzer

Architecture and Implementation of a Modular, Scalable Backend for Sensor Data from Multiple Users

Marcel Garus

Hasso-Plattner-Institut an der Universität Potsdam

30. Juni 2021

Bachelorarbeit

Architektur und Implementierung eines modularen, skalierbaren Backends für Sensordaten verschiedener Nutzer

Architecture and Implementation of a Modular, Scalable Backend for Sensor Data from Multiple Users

von
Marcel Garus

Betreuung
Prof. Dr. Andreas Polze, Lukas Pirl, Robert Schmid
Professur für Betriebssysteme und Middleware
Philippe Fuchs, Henry Hübler, Jenne Krey,
Gwyneth Mettendorf, Alexander Al Schmitt, Ingo Schwarzer
DB Systel GmbH

Hasso-Plattner-Institut an der Universität Potsdam

30. Juni 2021

Zusammenfassung

Sensoren in Zügen oder in deren Frachtgut können für mehr Kundenkomfort, eine zielgerichtete Instandhaltung und bessere Planbarkeit sorgen. Eine Sensorplattform sollte wiederverwendbar sein und einen „Live"-Datenzugriff ermöglichen. Vorhandene Frameworks wie *AWS IoT Greengrass*, *Google Cloud IoT Core*, *Microsoft Azure RTOS* oder *Eclipse IoT* binden die Plattform an eine spezifische Cloud oder lösen nur kleine Teilprobleme.

In unserem Bachelorprojekt entwickeln wir deshalb eine eigene Plattform. In Wagen kann dazu Infrastruktur eingebaut werden, zu der sich Sensoren drahtlos verbinden können. Ein zentraler Server übernimmt die Verwaltung von Sensoren. Die Zuginfrastruktur sendet Messwerte direkt an Server, die von den Nutzern mithilfe einer bereitgestellten Bibliothek implementiert und selbst betrieben werden. Vorteil dieser Architektur gegenüber anderen betrachteten ist vor allem eine Unabhängigkeit der Nutzer vom Plattformbetreiber bezüglich Datenspeicherung und -verarbeitung.

Die Implementierung der Backend-Komponenten verwendet Rust, asynchrone Programmierung, das Kommunikationsformat *Cap'n Proto* und die Datenbank *MongoDB*. Abhängig von der Anzahl der Verbindungen und versendeten Messwerte wird die CPU-, RAM- und Netzwerkauslastung der bereitgestellten Bibliothek betrachtet.

Die Netzwerklast lässt sich durch Datenkompression während der Übertragung reduzieren; dabei werden ein Cap'n-Proto-spezifisches Packing sowie die Kompressionsverfahren zlib und LZ4 verglichen. Das Ausführen von Code direkt in Zügen kann mit einer Vorverarbeitung und Filterung der Messwerte ebenfalls den Netzwerkverkehr verringern.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit ist Teil eines Bachelorprojekts des Hasso-Plattner-Insituts (HPI) mit dem Ziel, eine Sensorplattform für Sensoren in Zügen zu entwickeln. Dazu kooperiert das Projektteam des HPI mit der DB Systel GmbH,[1] dem IT-Dienstleister der Deutschen Bahn.[2]

In vielen Anwendungsszenarien müssen Messwerte von Zügen gesammelt und analysiert werden. Beispiele sind das Messen der Temperaturen in Personenwagen, das Erkennen von Defekten anhand von Audiosignalen um zustandsbasierte Instandhaltung zu ermöglichen oder das kontinuerliche Überwachen von bestimmten Umgebungsparametern wie Erschütterungen oder Luftfeuchtigkeit in Frachtgut.

Sollen heutzutage Sensormesswerte in Zügen aufgenommen und unmittelbar online zur Verfügung gestellt werden, müssen Sensoren sich entweder im Zug-WLAN anmelden oder eine Langstrecken-Funktechnologie wie LoRaWAN [13] oder Mobilfunk nutzen. In beiden Fällen kümmern sich die Sensoren komplett um die Kommunikation mit den Backend-Servern und das Zwischenspeichern von Messwerten bei Verbindungsabbruch.

Unser Projekt entwickelt eine Software- und Hardware-Plattform für das Sammeln von Sensormesswerten aus Zügen. Dafür kann Infrastruktur in Züge eingebaut werden, bei der sich sowohl Sensoren aus Frachtgut als auch Sensoren direkt aus den Wagen lokal anmelden können. Weil die Anmeldung über ein angepasstes Funkprotokoll abläuft und die Infrastruktur das Zwischenspeichern und die tatsächliche Übertragung der Messwerte ins Internet übernimmt, ermöglicht dies die Installation von leistungsschwachen und damit kostengünstigen Sensoren.

Eine etablierte Infrastruktur für Sensormesswerte fördert außerdem ein schnelles Reagieren auf neue Herausforderungen. Beispielsweise hat während der Corona-Pandemie zur Bewahrung des Mindestabstands die gleichmäßige Belegung von Personenwagen eine höhere Priorität. Stünde eine existierende Software- und Hardware-Plattform für das Sammeln von Sensormesswerten bereit, könnten technische Lösungen dieser Aufgabe mit geringem Aufwand angegangen werden.

Diese Arbeit beschäftigt sich mit der Architektur und Implementierung des Systemteils, der außerhalb des Zuges liegt. Zunächst werden beispielhafte Anwendungsszenarien beschrieben und daraus Anforderungen an die Sensorplattform hergeleitet. Anhand dieser werden existierende Plattformen bewertet. Weil keine auf die konkrete Anwendung von Sensoren im Zug zugeschnitten ist, entscheiden wir uns dazu, eine eigene Plattform zu bauen. Deshalb betrachtet diese Arbeit mehrere Möglichkeiten, wie das System architektiert sein könnte, und bewertet sie anhand der zuvor ermittelten Kriterien. Nach der Wahl einer Architektur wird auf die Implementierung der Backend-Komponenten eingegangen und der Teil des Backends mit der größten Datenlast evaluiert. Schließlich wird ein Ausblick auf mögliche Weiterentwicklungen gegeben.

[1] DB Systel GmbH, `https://www.dbsystel.de` (besucht am 27. Juni 2021)
[2] DB Konzern – Deutsche Bahn AG, `https://www.db.de` (besucht am 27. Juni 2021)

2 Beispielhafte Anwendungsszenarien

Um Kriteren für die Sensorplattform aufzustellen, werden hypothetische Anwendungsszenarien vorgestellt, in denen die Plattform helfen können soll.

Belegung von Personenwagen Um den am Bahnsteig wartenden Fahrgästen stets aktuell mitteilen zu können, in welchen Wagen sie die größte Chance auf freie Sitzplätze haben, möchte eine Arbeitsgruppe der DB messen, wie viele Personen sich in den einzelnen Wagen von Zügen ungefähr aufhalten. Smartphones der Passagiere suchen nach WLAN, indem sie öffentliche WLAN-Probe-Pakete verschicken. Mithilfe von Mikrocontrollern möchte die Arbeitsgruppe diese WLAN-Probes zählen und die Anzahl als Metrik für die Belegung von Wagen verwenden. Die Anzahl der WLAN-Probes soll gesammelt, eine Belegung geschätzt und aufbereitete Belegungsinformationen den Kunden über Webseite oder Lautsprecherdurchsagen vor Ankunft der Züge zur Verfügung gestellt werden. Dadurch können sich Bahnkunden bereits an Positionen des Bahnsteigs stellen, bei denen die haltenden Wagen möglichst leer sind. Diese Anwendung erhöht den Kundenkomfort und verringert das Risiko einer Verzögerung des Betriebsablaufs aufgrund langer Umstiegszeiten. Wanke hat dieses Anwendungsszenario als Teil des Bachelorprojekts in [24] beispielhaft umgesetzt.

Zustandsbasierte Instandhaltung von Bremsen Die Bremsen eines Zuges nutzen sich über die Zeit ab. Ein Wartungsteam der DB möchte nahe der Bremsen Mikrofone installieren, in deren Aufnahmen Anomalien mittels stochastischer Methoden wie Clustering oder neuronaler Netze erkannt werden. So kann eine Bremse ständig überwacht und bei Bedarf ausgetauscht werden, bevor sie ausfällt. Das resultiert in einer Kostenersparnis für die Deutsche Bahn. Matrisch hat in [15] das automatisierte Erkennen von Defekten anhand von Sensordaten als Teil des Bachelorprojekts exploriert.

Medikamentenlieferung Die Impfstoffhersteller Pfizer und BioNTech möchten ihren Impfstoff Comirnaty[3] von einem pharmazeutischen Betrieb an einen anderen Ort in Deutschland bringen. Weil der Impfstoff durchgängig nicht wärmer als -70 °C gelagert und transportiert werden muss, sind entsprechende Kühleinrichtungen in den Cargo-Containern vorhanden. [8] [5] Pfizer und BioNTech möchten sicherstellen, dass dieser Temperaturbereich durchgängig eingehalten wird. Fällt die Kühlung aus, möchte Pfizer unmittelbar benachrichtigt werden und bereits eine erneute Lieferung organisieren können, bevor der erste Container ankommt.

[3] Comirnaty Covid-19 mRNA Vaccine, `https://www.comirnatyglobal.com` (besucht am 27. Juni 2021)

3 Anforderungen

Um mehrere Architekturen vergleichen zu können, werden die wichtigsten Anforderungen an unsere Plattform aus den zuvor beschriebenen Anwendungsszenarien abgeleitet.

3.1 Essentielle Anforderungen

Folgende Anforderungen müssen erfüllt sein, damit das System überhaupt einen Nutzen für die obigen Szenarien erbringt:

- Sowohl DB-interne als auch -externe Nutzer (z. B. DB-Cargo-Kunden, die Fracht verschicken) sollen die Sensorplattform benutzen können. Im Folgenden benutze ich den Begriff *Team* synonym für eine Nutzergruppe, die Sensoren verbaut. Dieses Teilen einer gemeinsamen Infrastruktur wird auch als *Multi-Tenancy* bezeichnet (deutsch etwa *mehrere Bewohner*). [1]
- Teams sollen nur auf die Messwerte der eigenen Sensoren Zugriff haben. Messwerte sollen weder von dritten eingesehen noch auf dem Weg zum Team verändert werden können.
- Autorisierte Mikrocontroller mit Sensoren sollen sich automatisch mit der Plattform verbinden können. Das ist insbesondere für Messungen in Cargo wichtig (Szenario „Medikamentenlieferung“). Sobald Container auf Züge geladen werden, sollen sich darin enthaltene Sensoren mit der Plattform verbinden.
- Es sollen beliebige Messwerte übertragen werden können. Für einige Messwerte können vordefinierte Verarbeitungsmodule zu Verfügung gestellt werden, aber Teams sollen ohne Absprache mit dem Plattformbetreiber neue Arten von Sensoren benutzen können. Die Übertragungsgeschwindigkeit soll so groß sein, dass z. B. Audiodaten mit 44 kHz und einer Bittiefe von 16 übertragen werden können.
- Messwerte sollen „live“ verfügbar sein. Dabei ist explizit keine Echtzeit-Lösung mit Deadline-Garantien nötig, weil die Internet-Verbindung von Zügen zeitweise schlecht oder z. B. in Tunneln ganz unterbrochen sein kann. Trotzdem sollen die Messwerte zeitnah übermittelt werden, damit z. B. das Szenario „Belegung von Personenwagen“ umgesetzt werden kann. Um zwischengespeicherte von aktuellen Messwerten unterscheiden zu können, soll jeder Messwert einen Zeitstempel besitzen.
- Komplexe Analysen der Messwerte sollen durchführbar sein. Das heißt, Teams sollen nicht nur vordefinierte Parameter (z. B. Grenzwerte für Benachrichtigungen) einstellen, sondern die Messwerte mit beliebigem Programmiercode weiterverarbeiten können (Szenario „Zustandsbasierte Instandhaltung von Bremsen“).

3.2 Nicht-essentielle Anforderungen

Folgende Anforderungen erleichtern das Nutzen oder Betreiben der Plattform. Sie sind nicht-binär, können also nur graduell erfüllt werden:

- Für viele häufig verbaute Sensoren (Audio, Temperatur etc.) sollen vordefinierte Analyseprogramme existieren. Im Rahmen des Bachelorprojekts haben wir dies für Audiodaten [15] und Belegungszustände von Personenwagen [24] beispielhaft implementiert.
- Teams sollten möglichst wenig Server-Wartung durchführen müssen. Das ermöglicht auch Teams mit wenig Ressourcen, auf unserer Plattform aufzubauen.
- Teams sollten möglichst viel Kontrolle über die Messwerte haben. Durch ihr fachspezifisches Domänenwissen können sie am besten beurteilen, wie lange welche Messwerte gespeichert werden müssen und wann eine Verarbeitung der Messwerte sinnvoll ist. Außerdem ermöglicht ihnen die komplette Verfügungsmacht über die Daten, unabhängiger vom Plattformbetreiber zu sein.
- Es soll möglichst wenig Infrastrukturaufwand für den Plattformbetreiber, die Deutsche Bahn, entstehen.

4 Bewertung existierender Sensorplattformen

In diesem Kapitel betrachte ich einige bestehende Plattformen für IoT-Geräte. Dabei stelle ich dar, wie sie sich auf das Anwendungsszenario von Sensoren in Zügen spezialisieren ließen und welche Vor- und Nachteile sich dadurch ergeben.

4.1 AWS IoT Greengrass

Im Rahmen von Amazons Cloud-Angebot *Amazon Web Services*[4] (AWS) bietet Amazon das Framework *AWS IoT Greengrass*[5] an. Dieses stellt die Softwarebibliothek *AWS IoT Greengrass Core*[6] zur Verfügung, die auf Mikrocontrollern installiert werden kann. Über diese Bibliothek lassen sich *AWS Lambda*-Funktionen[7], Docker-Container[8] und native Betriebssystemprozesse aus der Ferne verteilen, aktualisieren und ausführen. [2] Native Prozesse ermöglichen direkten Zugriff auf Peripheriegeräte, sodass Sensordaten ausgelesen werden können. Weiter bietet das Framework die Möglichkeit, mit anderen IoT-Geräten auf direktem Weg Nachrichten auszutauschen. Schließlich können gesammelte Daten von IoT-Geräten zur AWS-Cloud gestreamt werden. Dort können sie mit Diensten wie *AWS IoT Analytics*[9] verarbeitet werden. [2] Innerhalb von AWS lässt sich mit Amazons Benutzerverwaltungs- und Authentifizierungsdienst *Amazon Cognito*[10] eine Multi-Tenancy-Lösung aufbauen, sodass Daten nur den jeweiligen Teams zur Verfügung gestellt werden.

Weil die gesamte Kommunikation sowohl mit anderen Geräten als auch der Cloud über eine direkte Internetanbindung der Geräte geschieht, ist ein leistungsstarker WLAN- oder Mobilfunkchip nötig. Auf das Anwendungsszenario von vielen kostengünstigen Sensoren in Wagen oder Fracht ist das Angebot also nicht optimal zugeschnitten.

Zusätzlich würde der Einsatz von *AWS IoT Greengrass* die Plattform an AWS binden und damit wirtschaftlich und politisch abhängig von Amazon machen. Änderungen an der

[4] Amazon Web Services (AWS) – Cloud Computing Services, `https://aws.amazon.com` (besucht am 27. Juni 2021)

[5] AWS IoT Greengrass – Amazon Web Services, `https://aws.amazon.com/greengrass/features` (besucht am 19. Juni 2021)

[6] Install the AWS IoT Greengrass Core software, `https://docs.aws.amazon.com/greengrass/v2/developerguide/install-greengrass-core-v2.html` (besucht am 27. Juni 2021)

[7] AWS Lambda – Serverless Compute – Amazon Web Services, `https://aws.amazon.com/lambda` (besucht am 27. Juni 2021)

[8] Empowering App Development for Developers – Docker, `https://www.docker.com` (besucht am 27. Juni 2021)

[9] AWS IoT Analytics Overview – Amazon Web Services, `https://aws.amazon.com/iot-analytics` (besucht am 27. Juni 2021)

[10] Amazon Cognito – Developer Guide, `https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-dg.pdf` (besucht am 27. Juni 2021)

Bepreisung oder politische Konflikte können das Betreiben von Sensoren unwirtschaftlich machen oder ganz verhindern.

4.2 Google Cloud IoT Core

Googles Plattform *Google Cloud IoT*[11] stellt eine Softwarebibliothek namens *Cloud IoT Core*[12] für IoT-Geräte zur Verfügung. Sie übernimmt das Ausrollen und Aktualisieren von Software sowie die Registrierung, Konfiguration, Authentifikation und das Monitoring von Geräten.

Anders als bei Amazons Angebot benötigen die Mikrocontroller keine direkte HTTP-Verbindung zu Servern. Stattdessen kann eine nahe der IoT-Geräte aufgestellte sogenannte *MQTT-Bridge*[13] Verbindungen zu Geräten über das MQTT-Protokoll[14] aufbauen und deren Nachrichten an die *Google Cloud* weiterleiten. [9]

MQTT funktioniert nach dem Publish/Subscribe-Prinzip und ist im Gegensatz zu HTTP explizit für IoT-Geräte konzipiert worden. Deshalb ist kein leistungsstarker WLAN- oder Mobilfunkchip nötig und MQTT funktioniert auch verlässlich bei geringen Bandbreiten. Pro verschickter Nachricht kann ausgewählt werden, ob diese mindestens, höchstens oder genau ein Mal beim Backend eintreffen soll. [3]

Wird solch eine MQTT-Bridge in Zügen der Deutschen Bahn eingebaut, könnten Sensoren aus Wagen und Frachtgut sich mit dieser energieeffizient verbinden. Weil die Verbindung zur MQTT-Bridge auch ohne Internet bestehen bleibt, werden die Mikrocontroller mit Sensoren entlastet. Sie müssen z. B. während einer Tunnelfahrt keine Daten selbst zwischenspeichern und danach keine erneute Verbindungsaufnahme und Authentifizierung durchführen. Sobald sich die MQTT-Bridge wieder mit dem Internet verbindet, werden die dort zwischengespeicherten Nachrichten an die *Google Cloud* geschickt.

Dort können Daten mit *BigQuery*[15] und *Dataflow*[16] analyisiert werden oder per *Pub/Sub*[17] anderen Google-Cloud-Diensten für weitere Analysen zur Verfügung gestellt werden. Mit Googles Benutzerverwaltung *Identity Platform*[18] kann außerdem eine Multi-Tenancy-Lösung implementiert werden. [10]

Der Nachteil dieser Lösung ist, dass die MQTT-Bridge Anfragen automatisch an Googles Cloud weiterleitet. Ähnlich wie bei Amazons Lösung würde das Einsetzen von Googles *Cloud IoT Core* die Deutsche Bahn also an Googles Cloud-Plattform binden und eine finanzielle und politische Abhängigkeit darstellen.

[11] Google Cloud IoT – Fully Managed IoT Services, `https://cloud.google.com/solutions/iot`, (besucht am 27. Juni 2021)
[12] Cloud IoT Core – Google Cloud, `https://cloud.google.com/iot-core` (besucht am 27. Juni 2021)
[13] Using gateways with the MQTT bridge – Cloud IoT Core Documentation, `https://cloud.google.com/iot/docs/how-tos/gateways/mqtt-bridge` (besucht am 27. Juni 2021)
[14] MQTT – The Standard for IoT Messaging, `https://mqtt.org` (besucht am 27. Juni 2021)
[15] BigQuery: Cloud Data Warehouse – Google Cloud, `https://cloud.google.com/bigquery` (besucht am 27. Juni 2021)
[16] Dataflow – Google Cloud, `https://cloud.google.com/dataflow` (besucht am 27. Juni 2021)
[17] Cloud Pub/Sub – Google Cloud, `https://cloud.google.com/pubsub` (besucht am 27. Juni 2021)
[18] Identity Platform – Google Cloud, `https://cloud.google.com/identity-platform` (besucht am 27. Juni 2021)

4.3 Microsoft Azure RTOS

Microsofts *Azure RTOS*[19] ist ein zertifiziertes Echtzeit-Betriebssystem samt Entwicklungsframework für ressourcenbeschränkte Geräte. Es stellt Betriebssystem-Primitive zur Verfügung, die auf energiesparende Echtzeitanwendungen zugeschnitten sind – unter anderem eine Echtzeit-Implementierung von Threads, Dateisystemen, Netzwerkstacks etc. Der gesamte Code ist quelloffen[20] und es können nicht nur Microsofts eigene *Azure Cloud Services* verwendet werden, sondern auch andere Clouds oder selbst gehostete Server. [16] Mittels *Azure IoT Hub*[21] können die Geräte überwacht und aktualisiert werden. Datenanalysen können mit *Azure Purview*[22] und *Azure Stream Analytics*[23] in der Azure-Cloud oder mittels *Azure IoT Edge*[24] direkt auf den IoT-Geräten durchgeführt werden.

Der Einsatz des *Azure RTOS* im Kontext von Zugsensoren ist durchaus denkbar. Allerdings sind die Sensoren rein passive Komponenten, sodass Echtzeit-Anforderungen im Mikrosekundenbereich selten zustande kommen; für das Sammeln der Daten gibt es explizit keine Echtzeit-Anforderung. Außer Microsofts Azure-Cloud stellt das *Azure RTOS* außerdem keine einheitliche Möglichkeit zur Verfügung, Sensormesswerte zu verschicken und online zu sammeln. Deshalb ist es wenn überhaupt nur Teil einer möglichen Lösung.

Zusätzlich benötigen nicht alle Mikrocontroller mit Sensoren ein komplettes Betriebssystem samt Threads und Dateisystem. Das von unserem Bachelorprojekt implementierte Anwendungsszenario „Belegung von Personenwagen“ kommt beispielsweise mit einem einfachen Embedded-Programm aus. [20]

4.4 Eclipse IoT

Die *Eclipse Foundation*[25] verwaltet unter dem Namen *Eclipse IoT*[26] über hundert Open-Source-Projekte, die zu IoT-Lösungen beitragen können. Die meisten Projekte sind Java-basiert und befassen sich mit Lösungen des IoT-Stacks, von Geräten über Gateways zu Clouds. Grundlegende Tools, Authentifizierungsbibliotheken und Kommunikationsstandards werden ebenfalls bereitgestellt. [7]

[19] Real Time Operating System (RTOS) – Microsoft Azure, `https://azure.microsoft.com/en-us/services/rtos` (besucht am 27. Juni 2021)
[20] Azure RTOS, `https://github.com/azure-rtos` (besucht am 27. Juni 2021)
[21] IoT Hub – Microsoft Azure, `https://azure.microsoft.com/en-us/services/iot-hub` (besucht am 27. Juni 2021)
[22] Azure Purview for Unified Data Governance – Microsoft Azure, `https://azure.microsoft.com/en-us/services/purview` (besucht am 27. Juni 2021)
[23] Azure Stream Analytics – Microsoft Azure, `https://azure.microsoft.com/en-us/services/stream-analytics` (besucht am 27. Juni 2021)
[24] IoT Edge – Microsoft Azure, `https://azure.microsoft.com/en-us/services/iot-edge` (besucht am 27. Juni 2021)
[25] Enabling Open Innovation & Collaboration – The Eclipse Foundation, `https://www.eclipse.org` (besucht am 27. Juni 2021)
[26] Eclipse IoT – Leading open source community for IoT innovation, `https://iot.eclipse.org/` (besucht am 27. Juni 2021)

Beispielsweise ist *Eclipse Concierge*[27] eine IoT-Implementierung des dynamischen Java-Modulsystems *OSGi Core*.[28] Im Rahmen des Projekts *Eclipse Mita*[29] wird eine komplett neue Programmiersprache für IoT-Sensorgeräte entwickelt. Mit dem *Eclipse Agail*-Framework[30] kann ein *Raspberry Pi*-Mikrocontroller[31] zu einem Gateway zwischen beliebigen Protokollen umfunktioniert werden. Serverseiting lässt sich z. B. *Eclipse hawkBit*[32] einsetzen, um Software-Aktualisierungen für IoT-Geräte zur Verfügung zu stellen. Das größte Projekt *Eclipse ioFog*[33] ist dafür gedacht, Microservices auf viele Geräte verteilen zu können. Grundlegendere Projekte sind *Eclipse Mosquitto*,[34] eine MQTT-Implementierung in Java, oder *Eclipse tinydtls*,[35] eine Implementierung des DTLS-Protokolls.[36]

Eclipse IoT bietet somit kein abgestimmtes Gesamtpaket, sondern viele Bausteine, die sich für den Anwendungsfall einer Sensorplattform auf Zügen individuell anpassen lassen. Eine Nutzung von Eclipse-IoT-Komponenten bedingt deshalb eine Einarbeitung, Anpassung und Integration in unser Projekt. Außerdem wird ein Großteil der Projekte nicht mehr aktiv weiterentwickelt – beispielsweise gab es bei *Eclipse Concierge* und *Eclipse Agail* im vergangenen Jahr nur eine Code-Änderung.[37]

Lösungen zu den durch die Komponenten angegangenen Standardaufgaben wie Authentifizierung und Software-Aktualisierungen werden benötigt, um die Plattform als fertiges Produkt anbieten zu können. Weil unserer Bachelorprojekt die generelle Machbarkeit einer Sensorplattform auf Zügen exploriert, rücken diese Themen zunächst nicht in den Fokus.

4.5 Fazit

Viele Unternehmen bieten IoT-Lösungen an. Die Lösungen von Amazon und Google machen die Plattform abhängig von deren weiteren Cloud-Produkten. Microsofts *Azure RTOS* kann für komplexere Mikrocontroller von Teams eine sinnvolle Lösung darstellen, Echtzeit ist jedoch keine Anforderung an unsere Plattform. Eclipse-IoT-Komponenten könnte man integrieren, wenn die Plattform als fertiges Produkt angeboten werden soll und IoT-Standardprobleme wie Authentifizierung gelöst werden müssen.

Indem wir neuartige Programmiersprachen, Funkprotokolle und Frameworks nutzen, erreichen wir eine potenziell besser integrierte und auf Sensoren in Zügen zugeschnittene Lösung.

[27] Concierge – A small-footprint implementation of the OSGi Core R5 Specification, `https://eclipse.org/concierge` (besucht am 27. Juni 2021)
[28] OSGi Working Group – The Eclipse Foundation, `https://www.osgi.org` (besucht am 27. Juni 2021)
[29] Eclipse Mita, `https://projects.eclipse.org/projects/iot.mita` (besucht am 27. Juni 2021)
[30] AGILE IoT Community Website, `http://agile-iot.eu/` (besucht am 27. Juni 2021)
[31] Teach, Learn, Make with Raspberry Pi, `https://www.raspberrypi.org/` (besucht am 27. Juni 2021)
[32] Eclipse hawkBit, `https://projects.eclipse.org/projects/iot.hawkbit` (besucht am 27. Juni 2021)
[33] Eclipse ioFog, `https://iofog.org` (besucht am 27. Juni 2021)
[34] Eclipse Mosquitto, `https://projects.eclipse.org/projects/iot.mosquitto` (besucht am 27. Juni 2021)
[35] Eclipse tinydtls, `https://projects.eclipse.org/projects/iot.tinydtls` (besucht am 27. Juni 2021)
[36] Datagram Transport Layer Security Version 1.2, `https://datatracker.ietf.org/doc/html/rfc6347` (besucht am 27. Juni 2021)
[37] Laut der jeweiligen Commit-Historie auf `https://projects.eclipse.org` (besucht am 27. Juni 2021)

5 Architektur

Die Architektur innerhalb des Zuges ist aus praktikablen Gründen festgesetzt und in Abbildung 5.1 gezeigt. Sensoren werden an einem Mikrocontroller angebracht, der dann im Wagen oder in der Fracht installiert werden kann. Ein Mikrocontroller mit potenziell mehreren Sensoren wird als *Sensorknoten* (engl. *Sensor Node*) bezeichnet. Damit Sensorknoten wenig kosten und lange ohne Batterieaustausch halten, dürfen sie nur wenig Strom verbrauchen. Zur Kommunikation nutzen wir die Funkmodule der nRF24-Serie?[38] In [21] erläutert Tiedt als weiteren Teil des Bachelorprojekts die genaue Auswahl dieser Funktechnologie. Genau wie WLAN operiert sie im 2.4-GHz-Funkspektrum. Durch geringere Datenraten und kleinere Reichweiten ist sie allerdings deutlich energiesparender.

Unterstützt ein Zug die Benutzung unserer Sensorplattform, befindet sich in jedem Wagen ein fest eingebauter Mikrocontroller, zu dem sich mehrere Sensorknoten aus dem Zug verbinden können. Wie dieser Verbindungsaufbau ablaufen kann und Messwerte übertragen werden können, legt Tiedt ebenfalls in [21] dar. Die fest eingebauten Mikrocontroller heißen *Infrastrukturknoten* (engl. *Edge Nodes*) und bilden zusammen die *Zuginfrastruktur*. Ein Infrastrukturknoten ist über das Zug-WLAN mit dem Internet verbunden und kann somit Messwerte weiterleiten. Seibold hat in [20] als weiteren Teil dieses Bachelorprojekts die auf der Infrastruktur laufende Software implementiert und ein Framework zum Programmieren von Sensorknoten entwickelt.

Dieses Kapitel beschäftigt sich damit, wie die Komponenten des gesamten Systems interagieren und wo welche Verantwortlichkeiten liegen. Dabei werden die folgenden Komponenten betrachtet:

- Züge mit Sensoren
- das Backend des Plattformbetreibers
- Teams

Abbildung 5.2 zeigt diese Komponenten – dabei wird das gesamte System aus Abbildung 5.1 als *Zug* abstrahiert.

Auf Basis der Anforderungen aus Kapitel 3 vergleiche ich mehrere in Erwägung gezogene Architekturen. Alle untersuchten Architekturen erfüllen die essentiellen Anforderungen, bezüglich der nicht-essentiellen haben sie unterschiedliche Kompromisse.

[38] nRF24 Series – Nordic Semiconductor, `https://www.nordicsemi.com/products/nrf24-series` (besucht am 27. Juni 2021)

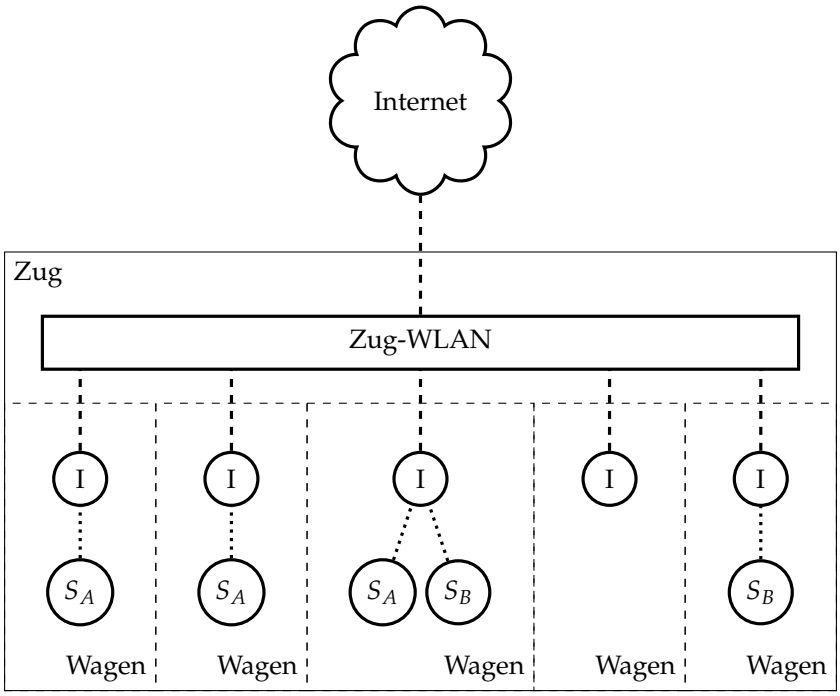

Abbildung 5.1: Übersicht über die Architektur innerhalb eines Zuges. Sensorknoten (S) werden von verschiedenen Teams (A und B) in Zug oder Frachtgut eingebaut. Sie verbinden sich mit dem nächstgelegenen Infrastrukturknoten (I) über ein lokales Funknetzwerk (gepunktete Linie). In jedem Wagen ist ein Infrastrukturknoten installiert und mit dem bestehenden Zug-WLAN verbunden (gestrichelte Linie). Die Infrastruktur kann somit Messwerte der Sensorknoten ins Internet weiterleiten (gestrichelte Linie).

Abbildung 5.2: Übersicht über die Komponenten des Systems. Wie Züge, Backend und Teams interagieren, wird in diesem Kapitel geklärt.

5.1 Zentralisierend

Ein Ansatz mit zentraler Backend-Komponente ist in Abbildung 5.3 dargestellt. Dabei empfängt und speichert das Backend alle Messwerte. Teams können Analyseanfragen an das Backend stellen, indem sie z. B. Software-Module hochladen, die dann auf den Messwerten ausgeführt werden. Das Backend übermittelt die Ergebnisse der Analysen zurück an die Teams.

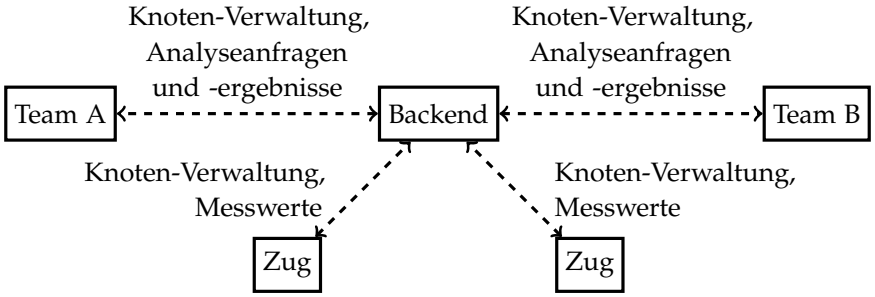

Abbildung 5.3: Züge schicken Messwerte zum Backend, das sie speichert. Teams können Analyseanfragen an das Backend stellen und bekommen Ergebnisse.

Weil die Teams keine eigene Server-Infrastruktur betreiben müssen, ist der Ansatz unter dem Namen *Serverless* bekannt. Laut [11] sind für Teams dies die größten Vorteile einer *Serverless*-Architektur gegenüber einem selbst gehosteten Server:

- Die Wartungskosten sind geringer, weil z. B. Ausfälle von Hardwarekomponenten oder Software-Updates vom Plattformbetreiber übernommen werden.
- Die benötigten Ressourcen müssen nicht vorher bestimmt werden. Teams zahlen für genau so viele Ressourcen, wie sie tatsächlich nutzen.
- Teams müssen sich nicht darum kümmern, bei mehr Nutzung zu skalieren. Weil Teams jedoch die Anzahl der Sensoren selbst bestimmen, können sie die resultierende Datenmenge besser abschätzen als beispielsweise Startups, deren Server an die Öffentlichkeit gerichtet sind. Deshalb ist dieser Vorteil weniger relevant für den Anwendungsfall einer Sensorplattform.

Weil die Teams jederzeit über die Messwerte verfügen können und das Backend außerdem vertraulich und integer ist, besitzen Teams *Datenhoheit* über die Messwerte, obwohl sie diese nicht physisch besitzen. [4]

Die Bindung der Datenspeicherung und -verarbeitung an das Backend des Plattformbetreibers ist gleichzeitig ein Nachteil: Durch ihr Domänenwissen können Teams besser mit den Messwerte umgehen; sie wissen, welche der Messwerte wie lange gespeichert und wie sie weiterverarbeitet werden sollen und müssen dieses Wissen dem Plattformbetreiber durch Codemodule kommunizieren. Weil der Plattformbetreiber Speicher und Rechenzeit bereitstellen muss, wird das Backend zu einer kompletten Compute-Cloud á la *Amazon Web Services*[39] oder *Google Cloud*,[40] von der die Teams abhängig sind.

[39] Amazon Web Services (AWS) – Cloud Computing Services, `https://aws.amazon.com` (besucht am 27. Juni 2021)

[40] Cloud Computing Services – Google Cloud, `https://cloud.google.com` (besucht am 27. Juni 2021)

5.2 Vermittelnd

Bei diesem in Abbildung 5.4 gezeigten Ansatz wurde das Backend in mehrere kleinere Komponenten aufgeteilt: Der Plattformbetreiber stellt immer noch ein Backend zur Verfügung (*Plattform-Backend*), das zur Verwaltung der Knoten genutzt wird und die Messwerte der Züge annimmt. Anstatt diese jedoch selbst zu speichern, leitet es die Messwerte an Backends weiter, die von Teams gehostet werden (*Team-Backends*). Die Teams kümmern sich somit selbst um das Speichern und Verarbeiten der Messwerte.

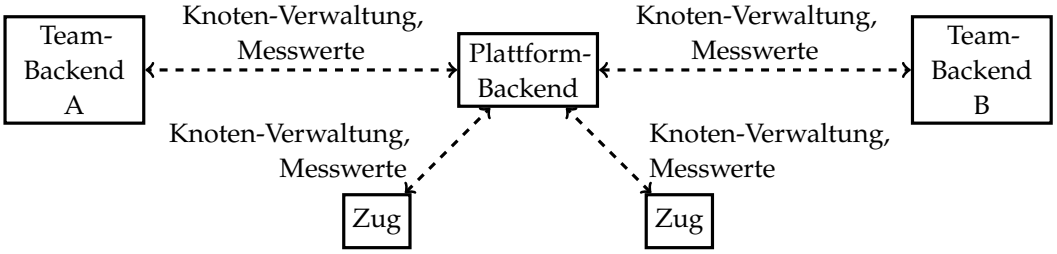

Abbildung 5.4: Das Backend ist auf ein Plattform-Backend und mehrere Team-Backends aufgeteilt. Züge schicken Messwerte zum Plattform-Backend, welches sie an die Team-Backends weiterleitet.

Während die Teams beim zentralisierenden Ansatz als *Eigentümer der Messwerte*[41] gelten, sind sie bei dieser vermittelnden Architektur zusätzlich *Besitzer der Messwerte*,[42] verfügen also physisch über die Daten. Dies bietet mehrere Vorteile:

- Teams besitzen die volle Flexibilität darüber, wie lange sie Messwerte speichern. Dabei können komplexe Speichersysteme umgesetzt werden. Ohne dass der Plattformbetreiber dafür Vorkehrungen schaffen muss, können Teams z. B. weiter zurückliegende Messwerte in geringerer zeitlicher Auflösung speichern oder nur bestimmte herausstechende Zeitabschnitte.
- Teams können komplexe Analysen durchführen, ohne auf die Rechenkapazitäten des Plattformbetreibers angewiesen zu sein. Teams haben außerdem Kontrolle über die eingesetzte Hardware. Sie können dadurch z. B. rechenintensive Machine-Learning-Algorithmen auf dafür optimierter Hardware wie *Tensor Processing Units* [19] durchführen, ohne dies mit dem Plattformbetreiber abzusprechen.
- Neben einer Verschlüsselung der Funkverbindungen kann bei sensiblen Messwerten auch eine Ende-zu-Ende-Verschlüsselung eingesetzt werden. Bei dieser Architektur haben Teams die Kontrolle über beide Endpunkte der Kommunikation – die Sensoren und ihren Server –, sodass der Plattformbetreiber bei einer Verschlüsselung keine Einsicht in die Messwerte bekommt.
- Zwischenzeitliche Ausfälle oder gar eine Abschaltung der gesamten Plattform (z. B. wegen Unwirtschaftlichkeit oder dem Aufkommen einer besseren Alternative) beeinflussen nicht bereits angefallene Messwerte.

Die vermittelnde Architektur verlagert Verantwortlichkeit vom Plattform-Backend zu den Teams. Deshalb scheint es, als sei die nicht-essentielle Anforderung, möglichst wenig

[41] BGB § 903, Definition Eigentümer
[42] BGB § 854, Definition Besitzer

Infrastrukturaufwand für die Teams zu erzeugen, nicht so gut erfüllt wie beim zentralisierenden Ansatz.

Das ist aber nicht so: Möchte ein Team keinen eigenen Server betreiben, könnte der Plattformbetreiber dies als weiteren Service anbieten. Dass das Team ihren Server sowohl selbst, auf DB-Servern oder bei anderen Cloud-Anbietern hosten kann, ist also *zusätzliche* Flexibilität im Vergleich zum zentralisierenden Ansatz. Fraglich ist allerdings, ob die Deutsche Bahn als Plattformbetreiber den Speicher und die Rechenleistung kostengünstiger anbieten kann als andere Cloud-Plattformen, die sich auf Datenspeicherung und -verarbeitung spezialisieren. Keine Anbieterbindung sorgt dafür, dass die Analyse für Teams insgesamt günstiger ist. [18]

Ein weiterer Nachteil dieser Architektur ist, dass das Plattform-Backend nun doppelt so viel Netzwerkverkehr wie beim zentralisierenden Ansatz hat, weil es Messwerte von den Zügen empfangen und diese an die Teams weiterleiten muss.

5.3 Verwaltend

Genau wie im vermittelnden Ansatz ist das Backend hier auf ein Plattform-Backend und mehrere Team-Backends aufgeteilt. Bei diesem in Abbildung 5.5 gezeigten Ansatz kommunizieren die Teams mit der Plattform allerdings nur für das Konfigurieren von Sensoren und einer öffentlichen Internetadresse ihres eigenen Servers. Das Plattform-Backend teilt diese Informationen der Zuginfrastruktur mit, welche dann die Messwerte der jeweiligen Sensoren an die Team-Backends auf direktem Weg weiterleitet. Weil das Plattform-Backend keine Messdaten mehr empfängt, sondern nur noch zur Verwaltung genutzt wird, wird es folglich deskriptiver als *Management-Backend* bezeichnet.

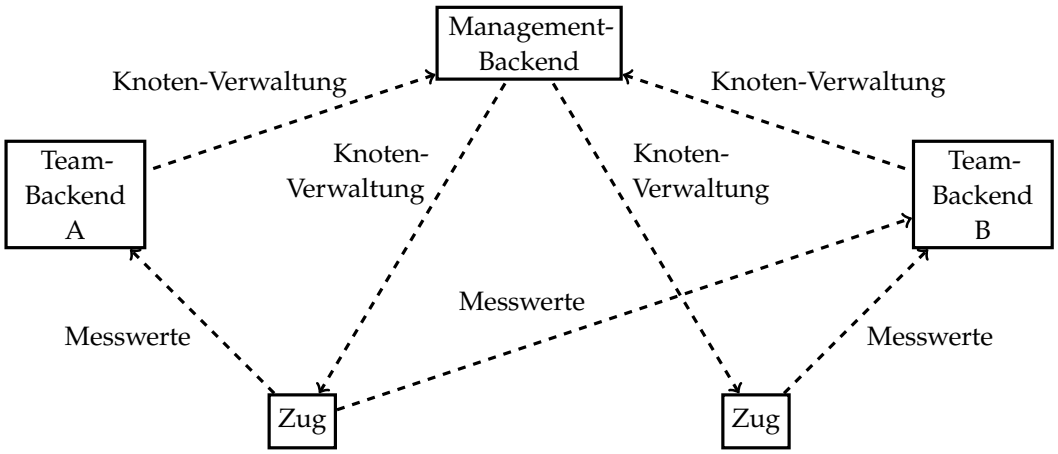

Abbildung 5.5: Das Backend teilt sich auf in ein Management-Backend und mehrere Team-Backends. Das Management-Backend übernimmt nur die Verwaltung von Knoten und einer öffentlichen Backend-Internetadresse pro Team. Züge schicken Messwerte direkt zu den entsprechenden Team-Backends. Dabei verbinden sich Züge nur mit den nötigen Backends; in der Grafik sind auf dem rechten Zug keine Sensorknoten von Team A, sodass der Zug auch keine Verbindung zu dessen Backend hat.

Im Vergleich zur vorherigen Architektur kommt als einzige neue Anforderung für die Team-Backends hinzu, dass sie aus dem öffentlichen Internet erreichbar sein müssen. Sowohl bei selbst gehosteten Servern als auch bei extern gehosteten Cloud-Angeboten ist das in der Regel leicht umzusetzen.

Dadurch, dass die Messwerte nicht mehr über einen zentralen Server übertragen werden müssen, macht dieser Ansatz die Plattform zu einem *verteilten System*. Vorteile einer solchen Architektur beschreibt Wu in [25]:

- **Performance und Kosten.** Es gibt keinen zentralen Engpass für Messwerte. Ihr Weg von den Sensoren zu den Teams ist kürzer als beim vermittelnden Ansatz und die Belastung des Plattformbetreibers auf das Nötigste beschränkt.
- **Verfügbarkeit und Ausfallsicherheit.** Ein Ausfall des Management-Backends ist immer noch fatal, weil Infrastrukturknoten dann nicht mehr abfragen könnten, wohin die Messwerte neu verbundener Sensorknoten geschickt werden sollen. Allerdings ist es weniger fatal als im vermittelnden Ansatz, weil bereits bestehende Verbindungen zu Team-Backends aufrecht erhalten werden.
- **Skalierbarkeit.** Die tatsächlichen Messwerte der Sensoren kommen gar nicht mit dem Management-Backend in Kontakt. Die Verwaltung und Authentifizierung von Sensorknoten geschieht im Vergleich zur Erstellung von Messwerten selten. Deshalb benötigt der Plattformbetreiber bei diesem Ansatz am wenigsten Ressourcen und kann die Plattform leichter auf mehr Teams und Züge erweitern. Für die Deutsche Bahn ist dies besonders wichtig, weil im tatsächlichen Bahnbetrieb täglich mehrere tausend Züge fahren.
- **Flexibilität und Erweiterbarkeit.** Soll neue Software oder Hardware ausprobiert werden, muss dies nicht an einer zentralen Stelle geschehen. Durch offene Schnittstellendefinitionen können Teile des Systems unabhängig verändert werden. Beispielsweise könnte ein Team auf ihrem Backend und ihren Sensoren ein neues Betriebssystem ausprobieren.

5.4 Architektur-Wahl

Die drei Architekturoptionen erfüllen alle die essentiellen Anforderungen. In Tabelle 5.1 werden sie anhand der nicht-essentiellen Anforderungen verglichen.

Wir wählten die verwaltende Architektur, weil sie die Anforderungen am besten erfüllt. Insbesondere den Teams die volle Kontrolle über ihre Messwerte zu geben, ist ein Grund für diese Wahl. Die Wartung können sie selbst übernehmen oder an einen anderen Cloud-Anbieter auslagern. Der Infrastrukturaufwand für den Plattformbetreiber beschränkt sich auf die Zuginfrastruktur und ein zentrales Backend zur Konfiguration von Knoten.

Anforderung	Zentralisierend	Vermittelnd	Verwaltend
vorgegebene Analysemodule	können als Softwaremodule angeboten werden		
wenig Wartung für Teams	kaum Wartung	Teams können Wartung übernehmen oder an eine Cloud auslagern	
Teams kontrollieren Messwerte	nur indirekt; Plattform speichert Messwerte	Teams kontrollieren Messwerte samt Soft- und Hardware	
wenig Infrastrukturaufwand des Plattformbetreibers	Netzwerklast & Speicherlast & Rechenlast	doppelte Netzwerklast	kaum Serverlast

Tabelle 5.1: Die drei vorgestellten Architekturen werden tabellarisch gegenübergestellt.

6 Implementierung

Um das Erstellen von Team-Backends zu vereinfachen, entwickelten wir eine Softwarebibliothek (*Team-Bibliothek*). Im Folgenden möchte ich die Implementierung sowohl vom Management-Backend als auch dieser Bibliothek genauer erläutern. Dabei stelle ich zunächst die geplante Nutzung der Plattform aus Sicht eines neuen Teams vor und gehe dann von sehr grundlegenden zu konkreteren Implementierungsentscheidungen.

6.1 Benutzung der Plattform

Team erstellen Möchte ein Team Mikrocontroller mit Sensoren in Züge oder Fracht einbauen, kontaktiert es zunächst den Plattformbetreiber, die Deutsche Bahn. Dabei erhält es eine global eindeutige Identifikationsnummer (*Team-ID*) und einen Authentifizierungsschlüssel (*Auth-Key*), mit dem es sich in Zukunft gegenüber dem Management-Backend authentifizieren kann. Außerdem bekommt es einen weiteren Auth-Key, mit dem sich Züge später gegenüber dem Team authentifizieren können.

Die Vergabe von Team-IDs ist zunächst außerhalb des Projektrahmens, wird aber bei der Einführung der Plattform relevant. Insbesondere müssten automatisierte Vergabemechanismen für Team-IDs implementiert werden. Weil die Benutzung der Sensorplattform Netzwerklast auf den Zügen erzeugt, sollte eine zentrale Stelle der Deutschen Bahn Team-IDs vergeben.

Die Vergabe und Rücknahme von Auth-Keys wurde im Rahmen dieser Arbeit nicht implementiert. Ansätze für das Zurücknehmen von Auth-Keys werden in Abschnitt 6.3 dargestellt.

Team-Backend erstellen Als nächstes entwickelt das Team mit der vom Plattformbetreiber bereitgestellten Team-Bibliothek ein eigenes Team-Backend, das Knoten verwalten und Sensormesswerte von Zügen annehmen kann. Dieses lässt es wie in Kapitel 5 diskutiert entweder auf eigenen Servern oder in Clouds laufen. Das Team-Backend meldet sich mit Team-ID und Auth-Key beim Management-Backend an und teilt diesem die eigene öffentliche Internetadresse mit.

Sensorknoten erstellen Das Team kann über das Management-Backend außerdem eigene Sensorknoten erstellen und erhält dabei global eindeutige *Node-IDs*. Dabei handelt es sich um Ganzzahlen mit einer Bitlänge von 32; insgesamt können also über 4 Mrd. Knoten erstellt und vergeben werden. In jeden Mikrocontroller programmiert das Team eine Node-ID fest ein.

An diesen Sensorknoten können mehrere Sensoren angebunden sein, die gemeinsam das Funkmodul nutzen. Damit bei Messwerten nicht nur die Node-ID des Sensorknotens, sondern auch der konkrete Sensor als Quelle gespeichert werden kann, hat jeder

angeschlossene Sensor eine Sensor-ID. Diese ist nicht global eindeutig, nur innerhalb eines Sensorknotens. Deshalb ist sie kleiner als die Node-ID und besteht nur aus einer natürlichen Zahl mit einer Bitlänge von 8 – an einem Sensorknoten können also höchstens 256 Sensoren angeschlossen sein.

Seibold hat die Software auf den Sensoren entwickelt und in [20] Feinheiten der Implementierung beschrieben. Die Sensoren eines Sensorknotens müssen nicht unbedingt physischen Sensoren entsprechen. Beispielsweise könnte auch die Versionsnummer eines Sensorknotens als eigener Sensor-Messwert bereitgestellt werden.

Schließlich baut das Team die Sensorknoten in Zug oder Fracht ein.

Messwerte empfangen Sobald Sensorknoten in einen Zug eingeladen werden, melden sie sich automatisch mit der Node-ID bei der Zuginfrastruktur an. Diese ist über das Zug-WLAN mit dem Management-Backend verbunden und fragt dort bei Anmeldung neuer Sensorknoten an, wohin die jeweiligen Messwerte geschickt werden sollen. Als Antwort bekommt es die öffentliche Internetadresse des dazugehörigen Team-Backends sowie den Auth-Key für dieses. Die Zuginfrastruktur verbindet sich mit diesen Informationen mit dem Team-Backend.

Sensorknoten können Messwertpakete (*Records*) verschicken. Diese Pakete beinhalten neben den als Binärdaten gespeicherten Messwerten zusätzlich den Erstellzeitpunkt und die Node- und Sensor-ID, um die Quelle eindeutig zu identifizieren.

Die Zuginfrastruktur bestätigt den Empfang und speichert sie zwischen, falls keine aktive Internetverbindung besteht. Ansonsten leitet es sie auf direktem Weg zum Team-Backend weiter.

Die Messwertpakete eignen sich auch als Einheit für die Inspektion von Datenfluss auf der Meta-Ebene. Beispielsweise ist für Nutzer der Plattform eine genaue Abrechnung anhand der Anzahl und Größe der gesendeten Pakete denkbar.

6.2 Grundlagen

Diese grundlegenden Entscheidungen gelten sowohl für das Management-Backend als auch für die Team-Bibliothek.

6.2.1 Auswahl der Programmiersprache

Um auf potenziell tausende Sensoren pro Team und hunderte Teams skalieren zu können, sollten das Management-Backend und die Team-Backends ressourcensparsam sein. Das Vermeiden von Abstürzen der Backends ist ebenfalls wichtig, weil ansonsten im schlimmsten Fall Messwerte verloren gehen.

Rust ist eine Programmiersprache, die mit dem Motto „A language empowering everyone to build reliable and efficient software“[43] genau diese Werte teilt. Außerdem verwenden wir auf Mikrocontrollern wie den Sensor- und Infrastrukturknoten bereits Rust. [21] Eine einheitliche Sprache auf mehreren Teilmodulen des Systems zu nutzen sorgt dafür, dass wir als Entwickler uns einfacher in anderen Komponenten zurechtfinden, weil wir

[43] Rust Programming Language, `https://www.rust-lang.org` (besucht am 27. Juni 2021)

mit der Sprache und deren Paket-Ökosystem bereits vertraut sind. Deshalb habe ich das Management-Backend und die Team-Bibliothek in Rust implementiert.

Um das komfortable Erstellen von Team-Backends zu ermöglichen, bei denen die Geschwindigkeit nicht kritisch ist, kann es sinnvoll sein, eine Bibliothek für Sprachen mit höherem Abstraktionsniveau anbzubieten. Mithilfe von *Foreign Function Interfaces*[44] könnten diese Softwarebibliotheken die Rust-Bibliothek wiederverwenden.

6.2.2 Benutzung von asynchroner Programmierung

Beim Management-Backend und bei der Team-Bibliothek gibt es viele gleichzeitig offene Netzwerkverbindungen – jeweils zu Team-Backends bzw. Zügen. Diese Verbindungen zu verwalten ist eine Aufgabe, bei der es nicht nur einen Handlungsstrang im Programm gibt, sondern bei der jede Verbindung einen eigenen Handlungsstrang samt Zustand benötigt. Traditionell werden dafür Threads genutzt; diese verbrauchen jedoch verhältnismäßig viel Speicher.

Asynchrone Programmierung (auch *Async/Await* genannt) ist ein von Rust unterstütztes Konzept, das etwas Rechenzeit und Speicher gegen *User-Mode-Threads* eintauscht, also nebenläufige Handlungsstränge ohne vom Betriebssystem gestellte Threads. Ist die Hauptaufgabe eines Programms, auf Daten vom Netzwerk oder Speicher zuzugreifen, lohnt sich der Einsatz von asynchroner Programmierung meist. Im Projekt nutze ich deshalb das im Rust-Ökosystem häufig verwendete Async/Await-Framework `tokio`.[45] Für eine generelle Einführung in asynchrone Programmierung verweise ich auf Rusts *Async Book*.[46]

6.3 Kommunikation zwischen den Komponenten

Um viele Sensormesswerte (z. B. von Audioaufnahmen) effizient zu übertragen, eignet sich ein kompaktes Binärformat eher als textbasierte Formate wie JSON[47] oder CSV.[48] Die entwickelte Plattform verwendet *Cap'n Proto*[49] – ein speichereffizientes und versionierbares Format.

Das Protokoll zwischen den Komponenten muss kompatibel versionierbar sein: Wird die Plattform weiterentwickelt, können auf Zügen nach einiger Zeit unterschiedliche Softwareversionen laufen. Folglich muss das Team-Backend mit allen Versionen korrekt umgehen können. Cap'n Proto beinhaltet eine eigene Sprache für die Definition von Schnittstellen. Dabei hat jede Datenklasse und jede Methode eine eindeutige Identifikationsnummer, sodass das nachträgliche Hinzufügen oder Umbenennen von Methoden oder Feldern möglich ist. [23] Diese zentrale, formale Definition der Kommunikationsschnittstelle kann dann zu Implementierungsvorlagen in vielen Programmiersprachen[50] kompiliert werden. Um das Serialisieren zu und Deserialisieren von Binärdaten müssen

[44] FFI – The Rustonomicon, `https://doc.rust-lang.org/nomicon/ffi.html` (besucht am 27. Juni 2021)
[45] Tokio, `https://tokio.rs` (besucht am 27. Juni 2021)
[46] Getting Started – Asynchronous Programming in Rust, `https://rust-lang.github.io/async-book` (besucht am 27. Juni 2021)
[47] JSON, `https://json.org` (besucht am 27. Juni 2021)
[48] Common Format and MIME Type for Comma-Separated Values (CSV) Files, `https://datatracker.ietf.org/doc/html/rfc4180` (besucht am 27. Juni 2021)
[49] Cap'n Proto: Introduction, `https://capnproto.org/`
[50] Cap'n Proto: Other Languages, `https://capnproto.org/otherlang.html` (besucht am 27. Juni 2021)

sich Menschen nicht mehr kümmern; stattdessen übernimmt die Cap'n-Proto-Bibliothek dies und stellt typsichere Datenstrukturen bereit. Weil das Generieren dieser Vorlagen für die meisten bekannten Programmiersprachen möglich ist, ist die Kommunikation unabhängig von den Sprachen in den Komponenten, sodass sich diese auch nachträglich noch wechseln lassen.

In Cap'n Proto können neben übertragbaren Datenstrukturen auch Objektsignaturen mit Methoden definiert werden (*Interfaces*). Bei Methodenaufrufen wird dann eine Netzwerkanfrage durchgeführt – das Protokoll ist somit ein *Remote-Procedure-Call-Framework*[51] (RPC-Framework, deutsch etwa „Fern-Funktionsaufruf-Framework").

Im Gegensatz zu beispielsweise REST-APIs[52] wissen beim Cap'n-Proto-Protokoll beide Parteien, welche Objekte verschickt wurden. Das ermöglicht eine vereinfachte Handhabung von Zugriffsverwaltung: Während bei REST bei jeder privilegierten Anfrage ein Beweis für eine vorherige Authentifizierung mitgeschickt werden muss, kann eine Partei bei Cap'n Proto auf einem von der anderen Partei verschickten Objekt Methoden aufrufen. Weil die verschickten Objekte eine Identität haben, können sie als sogenannte *Capabilities* (deutsch etwa „Fähigkeiten") genutzt werden: Das Besitzen eines Objekts kann für eine vorher erbrachte Leistung oder Authentifizierung einstehen. Abbildung 6.1 vergleicht das Ausführen einer privilegierten Aktion mit vorheriger Authentifizierung an einer REST-API und einer Capability-basierten API. Die Komponenten unserer Plattform nutzen bei der Kommunikation Capability-basierte Authentifizierung.

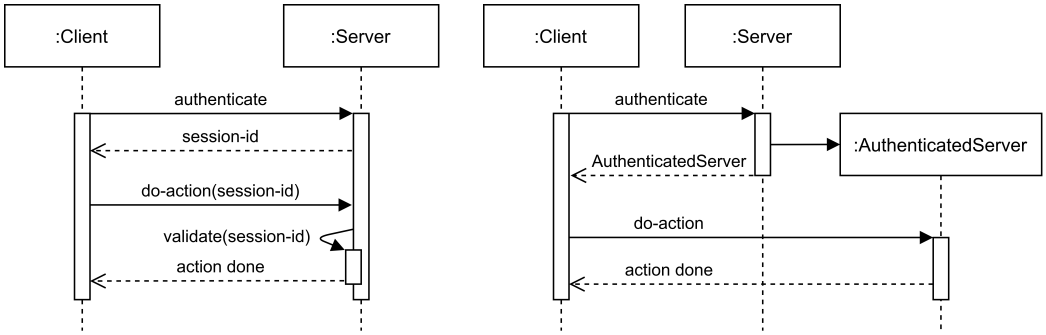

Abbildung 6.1: Bei einer typischen REST-API (links) gibt es keinen Zustand, der sich im Kontext der Verbindung ansammelt. Weil alle Anfragen an eine globale API gestellt werden, sind gesonderte Authentifizierungsbeweise wie z. B. *Session-IDs* erforderlich. Bei einer Capability-basierten API (rechts) können Objekte übertragen werden, mit denen die andere Seite interagieren kann. Diese können Zustand wie z. B. eine Authentifikation ausdrücken.

Im Folgenden erläutere ich kurz, wie die Kommunikation zwischen den Komponenten abläuft. Dabei verdeutlichen beispielhafte Sequenzdiagramme den Ablauf. In Anhang A befinden sich die formalen Cap'n-Proto-Schnittstellendefinitionen der einzelnen Verbindungen.

[51] Cap'n Proto: RPC Protocol, `https://capnproto.org/rpc.html` (besucht am 27. Juni 2021)
[52] What is REST, `https://restfulapi.net` (besucht am 27. Juni 2021)

Kommunikation zwischen Team- und Management-Backend Wie in Abbildung 6.2 gezeigt, startet die Kommunikation, indem das Management-Backend eine `Backend`-Schnittstelle zur Verfügung stellt. Authentifiziert sich das Team gegenüber dem Management-Backend mit der Team-ID und dem Auth-Key, erhält es ein `AuthenticatedBackend`. Dort kann es eine Internetadresse angeben, zu der die Zuginfrastruktur die Messwertpakete der Knoten schicken soll. Weil Knoten außer ihrer Node-ID keine weiteren Eigenschaften besitzen, bleiben für die Sensorknoten von den üblichen *CRUD-Operationen*[53] (*Create, Read, Update, Delete*) nur noch die Möglichkeiten, neue zu erstellen und existierende zu löschen.

Kommunikation zwischen Zug und Management-Backend Bei dieser in Abbildung 6.3 gezeigten Kommunikation stellt das Management-Backend eine `Backend`-Schnittstelle zur Verfügung. Die Zuginfrastruktur authentifiziert sich mit einem Auth-Key und erhält ein `AuthenticatedBackend`. Meldet sich ein Knoten über Funk im Zug an, kann die Zuginfrastruktur mithilfe eines `getAddressForNode`-Aufrufs die Internetadresse ermitteln, an die Messwerte des Knotens geschickt werden.

Kommunikation zwischen Zug und Team-Backend Bei dieser in Abbildung 6.4 gezeigten Kommunikation stellt das Team-Backend eine `Backend`-Schnittstelle zur Verfügung. Die Zuginfrastruktur authentifiziert sich mit einem Auth-Key und erhält ein `AuthenticatedBackend`. Mithilfe einer Node-ID und einer Sensor-ID kann es eine `Sensor`-Schnittstelle bekommen, über die es Messwerte mithilfe der Methode `publishRecords` veröffentlichen kann. Diese Methode nimmt mehrere Messwertpakete an, sodass nicht mehrere Anfragen durchgeführt werden müssen, um angesammelte Pakete zu übertragen. Wie sich das Versenden von mehreren Paketen in einer Anfrage auf die Menge des Netzwerkverkehrs auswirkt, wurde in Kapitel 7 evaluiert.

[53] What is CRUD? `https://www.codecademy.com/articles/what-is-crud` (besucht am 27. Juni 2021)

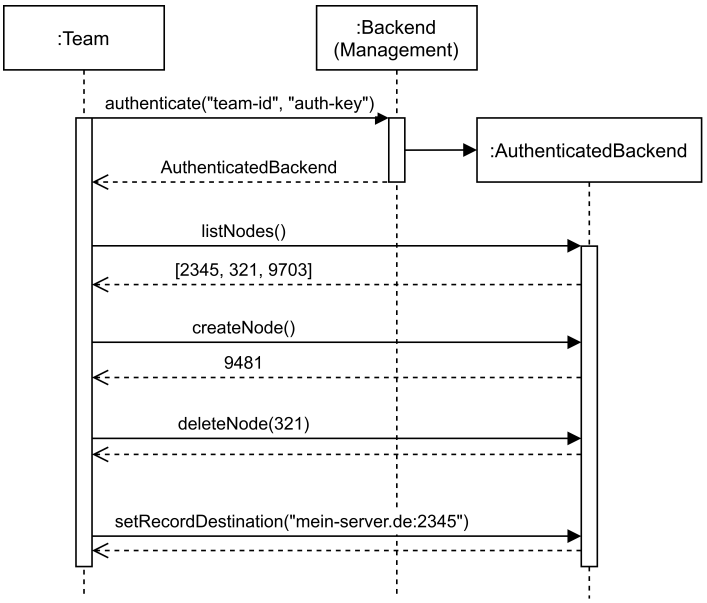

Abbildung 6.2: Ein möglicher Ablauf der Kommunikation zwischen Team- und Management-Backend.

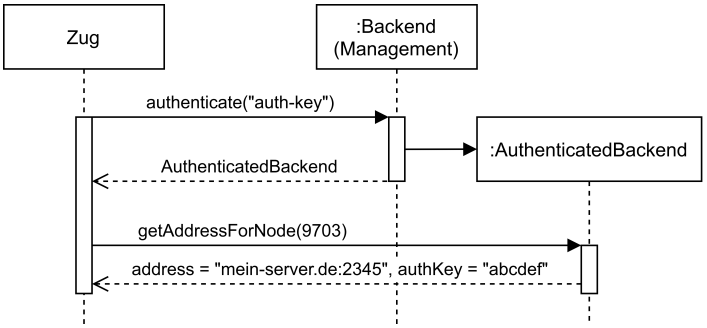

Abbildung 6.3: Ein möglicher Ablauf der Kommunikation zwischen Zug und Management-Backend.

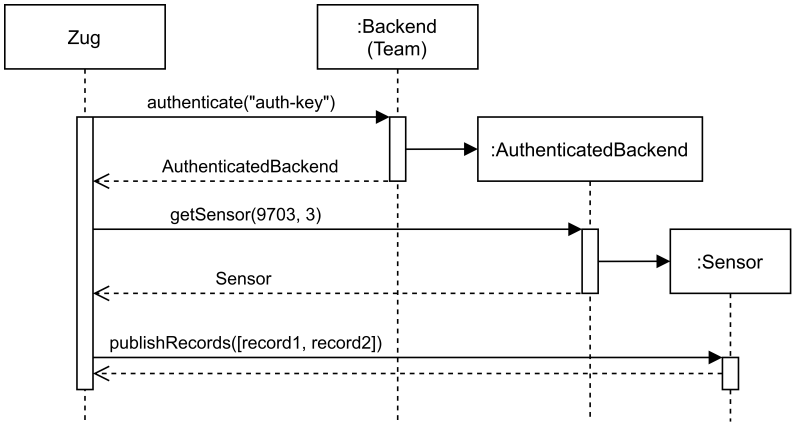

Abbildung 6.4: Ein möglicher Ablauf der Kommunikation zwischen Zug und Team-Backend.

6.4 Team-Bibliothek

Die Team-Bibliothek kann von Teams in eigene Projekte eingebunden werden und ermöglicht so, mit dem Management-Backend und den Zügen zu kommunizieren. Die Bibliothek könnte als wiederverwendbares Software-Paket veröffentlicht werden – in Rust heißen diese Pakete *Crates*.[54] Wie Abbildung 6.5 zeigt, ist die Bibliothek in zwei Teile aufgeteilt:

- Die *Management-API* ermöglicht eine Verwaltung der eigenen Knoten.
- Der *Sensor-Server* hört auf eingehende Verbindungen von Zügen und empfängt Messwerte.

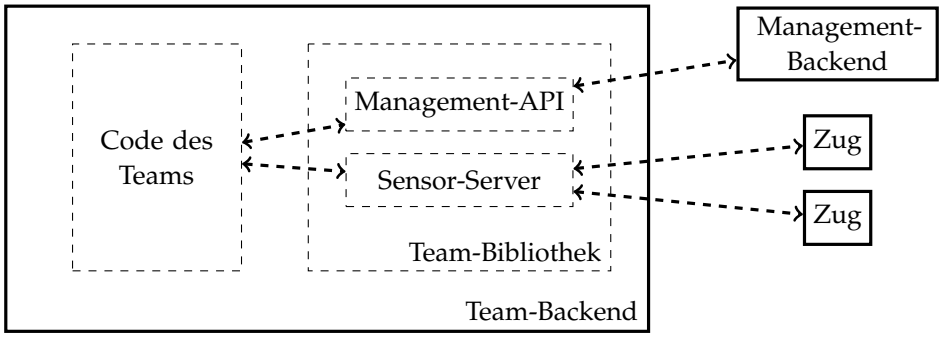

Abbildung 6.5: Die Team-Bibliothek besteht aus Management-API und Sensor-Server.

Die Trennung in diese beiden APIs ermöglicht eine unabhängige Benutzung. Soll ein Programm Knoten verwalten ohne Messwerte zu empfangen, benötigt es keine Verbindungen zu Zügen. Beide APIs und ihre Implementierung werden in diesem Abschnitt vorgestellt.

6.4.1 Management-API

Die `Management`-API ermöglicht die Kommunikation mit dem Management-Backend. Dazu meldet sie sich dort mit Team-ID und Auth-Key an. Dann lassen sich die eigenen Knoten auflisten, Knoten erstellen und löschen sowie die Internetadresse des Team-Backends festlegen. Eine beispielhafte Nutzung der API ist hier dargestellt:

```
let management = team::Management::connect(
    "60b4a23239ef2740f16f2944".into(),     // Team-ID
    "my-team-auth-key".into(),             // Auth-Key
    "betreiber.de:2444".parse().unwrap(),  // Management-Backend
)
.await
.unwrap();

let my_nodes: Vec<NodeId> = management.list_nodes().await.unwrap();
```

[54] Crates – Rust by Example, `https://doc.rust-lang.org/rust-by-example/crates.html` (besucht am 27. Juni 2021)

```
let node_id: NodeId = management.create_node().await.unwrap();
management.delete_node(node_id).await.unwrap();

management.set_record_destination("my-server.com:2345")
    .await
    .unwrap();
```

Weil bei Netzwerkanfragen die Prozessorleistung üblicherweise nicht der Engpass ist, sondern ein Großteil der Zeit mit dem Warten auf Antworten verbraucht wird, sind die Methoden der API alle asynchron. Dadurch können während einer aktiven Anfrage andere Operationen nebenläufig durchgeführt werden und mehrere Anfragen parallel laufen. Erst das Aufrufen von `await` stellt sicher, dass die Anfrage erfolgreich durchgeführt wurde und das Ergebnis verfügbar ist.

Die Bibliothek delegiert diese Aufrufe zu einem `AuthenticatedBackend`-Objekt einer Cap'n-Proto-Verbindung zum Management-Backend. Weil solche Verbindungen Teile des Verbindungszustands in nicht-atomaren *Reference-Countern* speichern,[55] können sie nicht von mehreren Threads aus benutzt werden; ansonsten könnten diese versuchen, den Verbindungszustand gleichzeitig zu modifizieren. Deshalb erlaubt Rusts Typsystem nicht, Cap'n-Proto-Verbindungen von mehreren Threads aus zu benutzen.[56]

Die Laufzeitumgebung der genutzten Async/Await-Runtime Tokio verwendet standardmäßig mehrere Betriebssystemthreads, sodass bei folgendem Code die Funktionen `foo` und `bar` auf verschiedenen Threads ausgeführt werden könnten:

```
1 foo().await;
2 bar();
```

Diese Eigenschaft ist erwünscht: Wenn `foo` eine Netzwerkanfrage durchführt und der ursprüngliche Thread mit anderen Aufgaben beschäftigt ist, wenn die Antwort kommt, kann ein anderer Thread „einspringen" und mit der Ausführung von `bar` in Zeile 2 weitermachen. [12]

Weil eine Cap'n-Proto-Verbindung nicht von mehreren Threads aus genutzt werden kann, ist eine naive Benutzung über ein `await` hinweg nicht möglich. Eine Möglichkeit, dieses Problem zu umgehen, ist die Verbindung in einem Betriebssystem-Lock (`Mutex`) und einem atomaren *Reference-Counter* (`Arc`) zu speichern (insgesamt also ein `Arc<Mutex<...>>`). Das Betriebssystem-Lock sorgt dafür, dass nie zwei Threads gleichzeitig die Verbindung nutzen. Sobald der letzte Thread die Referenz löscht, wird dank des atomaren *Reference-Counters* die Verbindung geschlossen und der Speicher freigegeben.

6.4.2 Sensor-Server

Der `SensorServer` kann mit der `run`-Methode gestartet werden; dazu benötigt er einen lokalen Port und eine Anzahl an maximal zwischengespeicherten Messwertpaketen. Züge können sich danach mit diesem Server verbinden und Messwertpakete melden. Hier ist eine beispielhafte Nutzung der API dargestellt:

[55] capnproto-rust/lib.rs at aa09220f26ed6df5af76cefc804879bdf6aacdcc, https://github.com/capnproto/capnproto-rust/blob/aa09220f/capnp-rpc/src/lib.rs#L169 (besucht am 27. Juni 2021)

[56] capnp_rpc::RpcSystem – Rust, https://docs.capnproto-rust.org/capnp_rpc/struct.RpcSystem.html#impl-Send

```
let records = team::run(
    "0.0.0.0:2345".parse().unwrap(),  // öffentlich erreichbar
    "my-auth-key",                    // für eingehende Verbindungen
    100,                              // max. gepufferte Pakete
);
for record in records.next().await {
    println!("{:?}", record);
}
```

Der Sensor-Server hört auf der angegebenen Adresse auf eingehende Verbindungen. Mehrere Verbindungen behandelt er mittels `StreamExt::for_each_concurrent`[57] gleichzeitig. Tokios Standard-Runtime nutzt so viele Betriebssystem-Threads, wie es CPU-Kerne gibt, sodass viele eingehende Verbindungen auf diesen aufgeteilt werden. [12]

Um die Messwerte zu sammeln, wird eine *Multiple-Producer-Single-Consumer-Queue* genutzt – eine Warteschlange mit mehreren Produzenten und einem Konsument (`tokio::sync::mpsc`;[58] Tokios Äquivalent zum Channel der Standardbibliothek[59]). Diese ermöglicht, einen `Receiver`[60] zu erstellen, der Elemente aus der Queue entfernen kann und mehrere `Sender`,[61] die neue Elemente an die Queue hinzufügen können. So können die Verbindungen zu Zügen nebenläufig Messwerte der Queue hinzufügen und der Nutzer der Bibliothek diese abrufen.

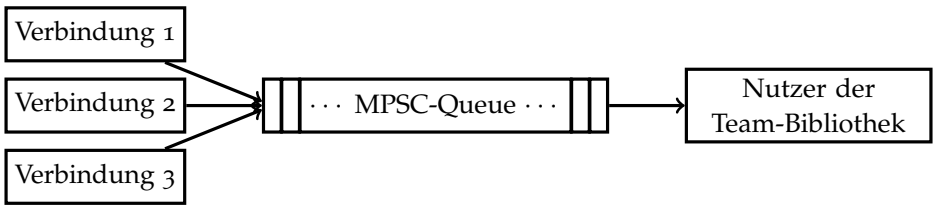

Abbildung 6.6: Die Multiple-Producer-Single-Consumer-Queue. Pro eingehender Cap'n-Proto-Verbindung gibt es einen Tokio-Task mit einem `Sender`. Alle Tasks fügen Messwerte mithilfe der `Sender` zur selben Queue hinzu. Dem Nutzer der Bibliothek wird die Queue mithilfe des Receivers als Stream von Messwerten angeboten.

Um jeder Verbindung einen eigenen `Sender` zur Verfügung zu stellen, wird wie in Abbildung 6.7 zu sehen das Prototype-Pattern[62] genutzt: Ein `Sender` wird in einem `Arc<Mutex<...>>` gespeichert (Zeile 3). Jede Verbindung sperrt bei ihrer Erstellung kurzzeitig diesen `Sender` für andere Threads (`lock`, Zeile 7), kopiert ihn mittels `clone` (Zeile 7) und gibt ihn wieder frei (automatisch in Zeile 7, weil die ursprünglichen `records` danach nicht mehr genutzt werden).

[57] `futures::stream::StreamExit` – Rust, `https://docs.rs/futures/0.3.4/futures/stream/trait.StreamExt.html#method.for_each_concurrent` (besucht am 27. Juni 2021)
[58] `tokio::sync::mpsc` – Rust, `https://docs.rs/tokio/0.1.16/tokio/sync/mpsc` (besucht am 27. Juni 2021)
[59] `std::sync::mpsc` – Rust, `https://doc.rust-lang.org/std/sync/mpsc` (besucht am 27. Juni 2021)
[60] `tokio::sync::mpsc::Receiver` – Rust, `https://docs.rs/tokio/0.1.16/tokio/sync/mpsc/struct.Receiver.html` (besucht am 27. Juni 2021)
[61] `tokio::sync::mpsc::Sender` – Rust, `https://docs.rs/tokio/0.1.16/tokio/sync/mpsc/struct.Sender.html` (besucht am 27. Juni 2021)
[62] Prototype – `https://refactoring.guru/design-patterns/prototype` (besucht am 27. Juni 2021)

```
1  let listener = TcpListener::bind(...).await?;
2  let (tx, rx) = mpsc::channel(records_buffer_length);
3  let records = Arc::new(Mutex::new(tx));
4  TcpListenerStream::new(listener).for_each_concurrent(None, |stream| {
5      let records = Arc::clone(&records);
6      async move {
7          let records = records.lock().await.clone();
8          ...
9          LocalSet::new().run_until(async move {
10             let backend = capnp_rpc::new_client(Backend { records });
11             run_rpc_system(stream, backend.client).await;
12         })
13         .await
14     }
15 });
```

Abbildung 6.7: Teil des Codes der Team-Bibliothek.

Das `Receiver`-Ende der Queue wird den Nutzern der Bibliothek als gebündelter `Stream` von Messwerten angeboten. Über diesen kann unendlich lange iteriert werden und mit Funktionen wie `StreamExt::for_each_concurrent` lassen sich mehrere eintreffende Messwerte parallel behandeln.

Dauert das Verarbeiten von Messwerten zu lange, kann der Stream die am Anfang festgelegte Anzahl an Paketen zwischenspeichern – im Code aus Abbildung 6.7 wurde diese Anzahl als `records_buffer_length` bei der Erstellung der Queue genutzt (Zeile 2). Kommt das gesamte Programm mit dem Abfragen von Messwertpaketen nicht mehr nach, werden sie kurz auf den Zügen zwischengespeichert. Nimmt ein Team-Backend über eine längere Zeit keine Messwertpakete mehr an, nimmt der Infrastrukturknoten zeitlich verzögert ebenfalls keine Messwerte mehr vom Sensorknoten an. Dieser nimmt dann in der Regel keine neuen Messwerte mehr auf.

Langfristig wäre es ein besseres Verhalten, dass die Zuginfrastruktur weiterhin Messwerte annimmt und die ältesten Messwerte des Sensors löscht, sobald nur noch wenig Speicher verfügbar ist. Diese Löschung von Messwerten eines Zeitraums sollte ebenfalls protokolliert werden.

6.5 Management-Backend

Die Aufgabe des Management-Backends ist die Verwaltung der Teams und Knoten.

6.5.1 Datenbank

Um Teams und Knoten zu verwalten, benötigt das Backend eine Datenbank. Meine Implementierung nutzt die weit verbreitete Datenbank *MongoDB*.[63] Diese ist horizontal skalierbar, lässt sich also auch auf mehrere Computer aufteilen.

Bei MongoDB sind Daten schemalose JSON-Objekte (auch *Dokumente* genannt) und werden in *Collections* angeordnet. Jedes Dokument hat außerdem eine von MongoDB zugewiesene 12 Byte große `ObjectID`,[64] die unter dem Feld `_id` gespeichert wird und das Dokument eindeutig kennzeichnet. Um die Geräte und Teams zu verwalten, nutze ich zwei Collections:

- Die `teams`-Collection speichert alle Teams. Dessen von MongoDB zugewiesene ID entspricht der Team-ID. Zusätzlich bestehen sie aus dem Auth-Key gegenüber dem Management-Backend, dem Auth-Key, den die Züge nutzen, und einer Internetadresse, an die die Messwerte geschickt werden. So könnte ein Team-Dokument beispielsweise aussehen:

```
{
    "_id": "60b4a23239ef2740f16f2944",
    "managementAuthKey": "my-team-auth-key",
    "trainAuthKey": "trains-use-this-key",
    "recordDestination": "my-server:2345"
}
```

- Die `nodes`-Collection speichert alle Knoten – sowohl Infrastruktur- als auch Sensorknoten. Knoten haben neben der MongoDB-ID eine Node-ID, die mit 4 statt 12 Bytes deutlich kürzer ist und dadurch besser für speichereffiziente Übertragung geeignet ist. Weil die Node-IDs auch innerhalb des Zuges bei der Funkübertragung genutzt werden, ist das eine wichtige Eigenschaft. Außerdem speichert jeder Knoten die ID des dazugehörigen Teams. Dies ist ein Beispiel für einen Knoten:

```
{
    "_id": "60be232300101e29007b8036",
    "id": 693878851,
    "teamId": "60b4a23239ef2740f16f2944"
}
```

Im Code des Management-Backends ist die Interaktion mit den Collections der Datenbank in eine eigene Abstraktion gekapselt, die die benötigten Aktionen typsicher unterstützt. Beispielsweise kann eine Interaktion mit der `nodes`-Collection wie folgt ablaufen:

```
let node_db = NodeDb::from_localhost_client();
let node_id = node_db.create(my_team_id).await.unwrap();
let node_ids = node_db.list(my_team_id).await.unwrap();
node_db.delete(node_ids[0]).await.unwrap();
```

[63] The most popular database for modern apps – MongoDB, `https://www.mongodb.com` (besucht am 27. Juni 2021)

[64] ObjectId, `https://docs.mongodb.com/manual/reference/method/ObjectId` (besucht am 27. Juni 2021)

6.5.2 Verbindungsmanagement

Genau wie beim Team-Backend werden eingehende Verbindungen gleichzeitig behandelt. Pro eingehender Team-Verbindung wird außerdem eine neue Verbindung zur Datenbank hergestellt. Weil Verbindungen zu MongoDB-Collections kopierbar sind,[65] könnten sich mehrere Team-Verbindungen auch eine Datenbankverbindung teilen, aber MongoDB kann mit vielen gleichzeitigen Verbindungen gut umgehen.

Sobald das Team gegenüber dem Backend authentifiziert ist, wird der Auth-Key nicht wiederholt übertragen. Das ist ähnlich zu Session-IDs bei Website-Logins, die ebenfalls selbst bei einer Änderung der Authentifizierungsmethode (z. B. Passwort) gültig bleiben. Um dieses Verhalten zu ändern, müsste das Backend vor jeder Aktion erneut die Authentifizierung überprüfen. Im Rahmen des Bachelorprojekts haben wir die Verwaltung von Teams und Auth-Keys noch nicht implementiert, deshalb ist diese Lösung bislang noch nicht implementiert.

[65] `mognodb::Collection – Rust`, `https://docs.rs/mongodb/1.1.1/mongodb/struct.Collection.html#impl-Clone` (besucht am 27. Juni 2021)

7 Evaluation

Die Architektur ist durch die Aufteilung des Backends auf ein Management- und mehrere Team-Backends und die Aufteilung der Datenquellen auf mehrere Züge intrinsisch auf viele Sensoren und Teams skalierbar. Um sicherzustellen, dass die Plattform auch für ein einzelnes Team eine gute Performance hat, untersuche ich, wie sich die Implementierung der Team-Bibliothek abhängig von den Verbindungen und versendeten Messwertpaketen verhält. Dabei untersuche ich drei Metriken: CPU-Nutzung, RAM-Nutzung und IO-Auslastung.

Für die Messungen verwende ich den Windows Resource Monitor,[66] der von Prozessen genutzte Ressourcen visualisiert.

7.1 CPU-Nutzung

In der Team-Bibliothek findet kaum aufwändige Datenverarbeitung statt; das Empfangen von Messwerten über Netzwerkverbindungen ist die zentrale Aufgabe. Die Verarbeitung der Messwerte übernehmen die Teams als Nutzer der Bibliothek selbst. Deshalb ist CPU-Nutzung bei den Tests nicht die limitierende Ressource. In allen weiter unten beschriebenen Tests lag die CPU-Ausnutzung des Team-Backend-Prozesses unter 12 %.

Als Testgerät wurde ein *Asus ZenBook 14*[67] genutzt. Weil dessen CPU des Typs *Intel Core i7-8565U*[68] symmetrisches Multithreading („hyperthreading“[69]) unterstützt, werden die 4 physischen CPU-Kerne als 8 logische Kerne für Prozesse bereitgestellt. Tokio erstellt für jeden Kern einen neuen Thread. [12] Neben dem ursprünglichen Thread existieren also 8 weitere Threads, die sich Aufgaben untereinander aufteilen.

7.2 RAM-Nutzung

Arbeitsspeicher (*Random Access Memory*, kurz RAM) ist in den meisten Fällen in ausreichenden Mengen vorhanden. Trotzdem ist es wichtig, sicherzustellen, dass eine einzelne Verbindung nicht allzu viel RAM benötigt – schließlich sollen Team-Backends Verbindungen zu tausenden Knoten verwalten können, z. B. wenn die Mikrocontroller des Szenarios

[66] Resource Monitor – Microsoft Docs, `https://docs.microsoft.com/en-us/previous-versions/windows/desktop/mscs/resource-monitor` (besucht am 27. Juni 2021)

[67] ZenBook 14 Q407, `https://www.asus.com/us/Laptops/For-Home/ZenBook/ZenBook-14-Q407` (besucht am 27. Juni 2021)

[68] Intel Core i7-8565U Processor (8M Cache, up to 4.60 GHz) Product Specifications, `https://ark.intel.com/content/www/us/en/ark/products/149091/intel-core-i7-8565u-processor-8m-cache-up-to-4-60-ghz.html` (besucht am 27. Juni 2021)

[69] Intel Hyper-Threading Technology, `https://intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html` (besucht am 27. Juni 2021)

„Belegung von Personenwagen“ in viele Wagen eingebaut werden. Fracht, die auf mehrere Züge oder Wagen aufgeteilt wird, sorgt auch für mehrere Verbindungen.

Verbindet sich ein Sensorknoten zu einem Infrastrukturknoten, baut dieser eine Verbindung zum jeweiligen Team-Backend auf. Die Anzahl der zu erwartenden eingehenden Verbindungen lässt sich deshalb nach oben durch die Anzahl der verbauten Sensorknoten abschätzen.

Abbildung 7.1 zeigt den Speicherbedarf abhängig von der Anzahl der eingehenden Verbindungen. Dazu wurden zu einem Team-Backend viele gleichzeitige Verbindungen erstellt, aber noch keine Messwerte verschickt. Dann wurde der vom Betriebssystem zur Verfügung gestellte (*committete*) Speicher des Team-Backend-Prozesses gemessen.

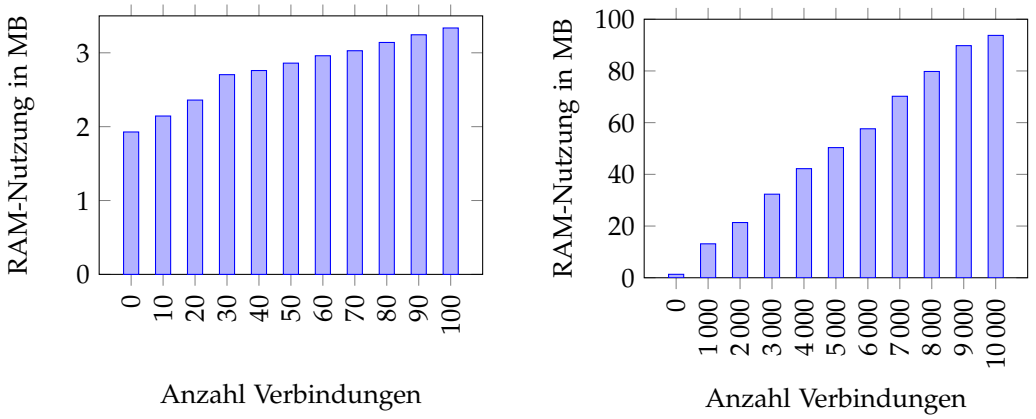

Abbildung 7.1: Vom Betriebssystem committeter RAM-Speicher des Team-Backends abhängig von der Anzahl der Verbindungen.

Zunächst lässt sich ablesen, dass Team-Backends mit relativ geringer Speichernutzung viele eingehende Verbindungen managen können – mit knapp 100 MB können 10 000 Verbindungen verwaltet werden. Der Speicherbedarf steigt außerdem linear zu der Anzahl der eingehenden Verbindungen – mehrere Verbindungen sind weitestgehend unabhängig voneinander, sodass sich der Speicherbedarf mehrerer Verbindungen addiert.

7.3 IO-Auslastung

Weil CPU-Geschwindigkeiten und RAM-Kapazitäten in den letzten Jahren deutlich schneller gestiegen sind als die Performance von anderen Computer-Komponenten, ist die Geschwindigkeit der meisten Programme heutzutage überwiegend vom Laden und Schreiben von Daten zu Peripheriemedien wie dem Speicher oder Netzwerk dominiert. Die Performance dieser Ein- und Ausgabe (Input/Output, kurz IO) ist daher häufig die Operation, die es sich am meisten lohnt zu optimieren. Weil bei der Team-Bibliothek das Empfangen von Messwerten die Hauptaufgabe ist, ist die Performance gerade beim Team-Backend sehr IO-dominiert.

Eine bereits eingebaute Optimierung des Team-Backends ist die Benutzung asynchroner Programmierung. Ein CPU-Thread wartet dadurch nicht auf Pakete einer Verbindung,

bevor er sich anderen Verbindungen widmet, sondern kann gleichzeitig bei mehreren Verbindungen auf eingehende Netzwerkpakete hören.

Um tatsächlich Messwerte zu versenden, muss die `publishRecords`-Methode der Cap'n-Proto-Schnittstelle aus Anhang A auf einer `Sensor`-Schnittstelle aufgerufen werden. Dabei kann direkt eine Liste mehrerer Messwerte verschickt werden. Hier ist der relevante Teil der Schnittstellendefinition aus Anhang A:

```
interface Sensor {
    publishRecords @0 (records :List(Record));
}

struct Record {
    timestamp @0 :UInt64;
    payload @1 :Data;
}
```

Als nächstes untersuche ich, wie sich Aufrufe der `publishRecords`-Methode auf die benötigte Bandbreite auswirken. Es gibt verschiedene Möglichkeiten, wie die Datenrate erhöht werden kann:

- die `publishRecords`-Methode häufiger aufrufen
- der `publishRecords`-Methode mehr Messwerte als Argument geben
- mehr Nutzdaten bei Messwerten mitschicken

In Abbildung 7.2 ist die benötigte Netzwerklast dieser drei Varianten dargestellt. Es lässt sich ablesen, dass der Netwerkverkehr linear zu der Anzahl der Methodenaufrufe korreliert (1. Graph). Der Aufruf einer Methode sorgt für eine feste Menge an Netzwerkverkehr, mehr Aufrufe resultieren also in entsprechend mehr Netzwerkverkehr.

Neben den übergebenen Parametern wird bei einem Methodenaufruf noch eine feste Menge weiterer Daten übertragen[70] (*Overhead*). Diese beinhaltet unter anderem eine Aufrufnummer, damit eintreffende Ergebnisse von mehreren parallelel laufenden Aufrufen dem richtigen zugeordnet werden können. Außerdem gibt es Informationen darüber, auf welchem Objekt welche Methode aufgerufen wird. Die Bestätigung der erfolgreichen Durchführung des Methodenaufrufs benötigt 80 Bytes (unterer Graph).

Bytes werden bei Cap'n Proto direkt in die Nachricht eingebettet. Im unteren Graphen resultiert das Übertragen von 1000 Bytes gegenüber 0 Bytes in einer Netzwerklast, die um 1 kB/s größer ist, also nur die zusätzlichen Bytes beinhaltet. Dies bestätigt, dass bei mehr Nutzdaten außer diesen keine weiteren Daten dazu kommen.

Im zweiten Graphen ist eine Stufe zu sehen; 500 Messwerte zu verschicken benötigt weniger Daten als 510 zu verschicken. Eine Inspektion des Netzwerkverkehrs ergibt, dass bei 500 Messwerten ein zusätzliches Netzwerkpaket hinter der eigentlichen Liste an Messwerrten verschickt und ab einer bestimmten Nutzdatengröße weggelassen wird. Wahrscheinlich handelt es sich dabei um eine Eigenheit des Cap'n-Proto-Protokolls; wobei es sich dabei konkret handelt, konnte ich noch nicht herausfinden.

[70] capnproto/rpc.capnp at master – capnproto/capnproto, `https://github.com/capnproto/capnproto/blob/da823d6350fb81f6bf66390898e5307b7b8a872c/c%2B%2B/src/capnp/rpc.capnp#L389-L479` (besucht am 27. Juni 2021)

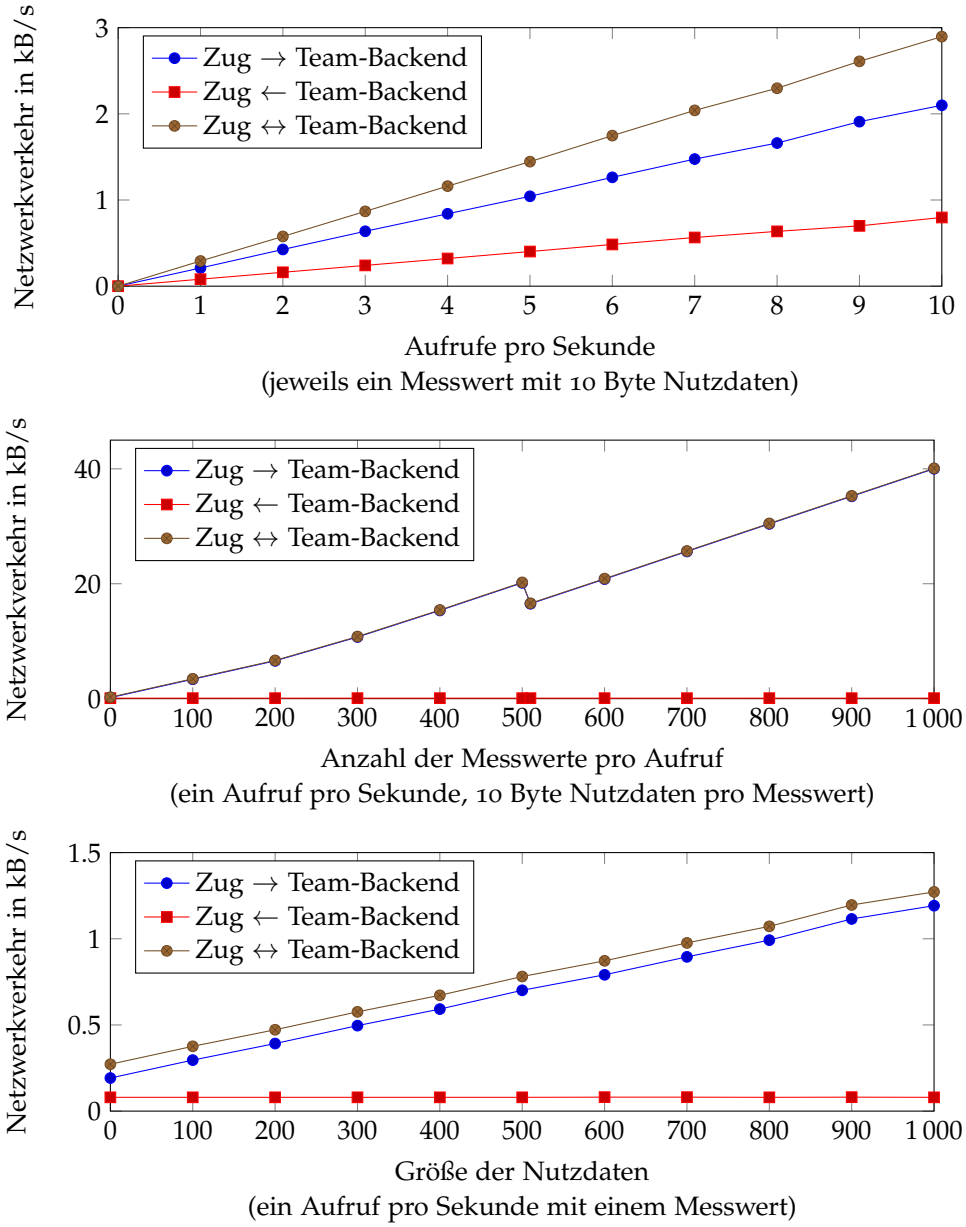

Abbildung 7.2: Über eine Verbindung werden in regelmäßigem Abstand Messwerte verschickt. Die untersuchten Variablen (von oben nach unten) sind: Anzahl der `publishRecords`-Aufrufe, Anzahl der Messwerte und Größe der Nutzdaten. Zwei der drei Variablen wurden dabei festgehalten, die andere variiert.

In Abbildung 7.3 werden mehrere Optionen verglichen, pro Sekunde 1000 Messwerte mit 10 Bytes an Nutzdaten über eine Verbindung zu transportieren. Offensichtlich handelt es sich hierbei um einen nichtlinearen Zusammenhang.

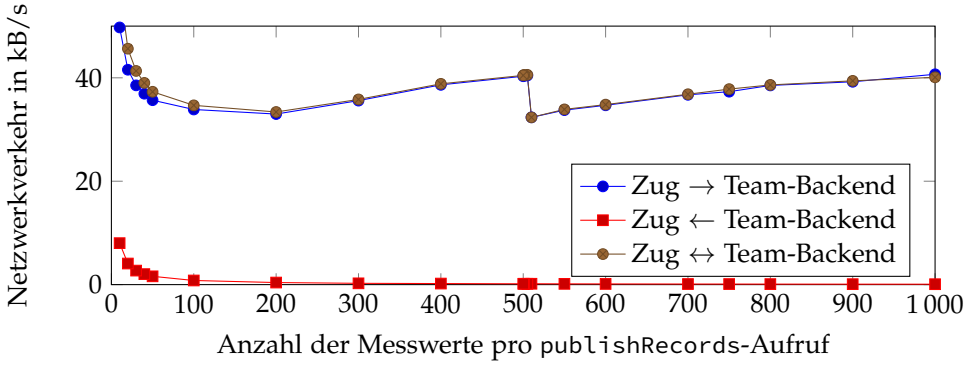

Abbildung 7.3: Über eine Verbindung werden 1000 Messwerte mit jeweils 10 zufälligen Bytes an Nutzdaten verschickt. Dabei werden unterschiedlich viele Messwerte pro `publishRecords`-Aufruf verschickt.

Werden nur wenige Messwerte pro Aufruf verschickt, müssen als Ausgleich viele Aufrufe durchgeführt werden, damit insgesamt 1000 Messwerte übertragen werden. Dann nimmt der Aufruf-Overhead einen großen Teil des Netzwerkverkehrs ein. Das resultiert im rapiden Anstieg des Graphen, wenn die Anzahl der Messwerte gegen 0 geht. Werden viele Messwerte pro Aufruf verschickt, dann verringert sich die Anzahl und erhöht sich die Größe der zu sendenden Netzwerkpakete. Ab ca. 200 Messwerten pro Aufruf hat die Größe der Netwerkpakete einen größeren Einfluss als deren Quantität. Auch die Stufe aus Abbildung 7.2 ist im Graphen zu sehen.

Dieses Wissen kann genutzt werden, um den Netzwerkverkehr zu optimieren: Ist eine geringe Latenz von Messwerten keine Priorität, kann die Zuginfrastruktur warten, bis sich 200 oder 510 Messwerte angesammelt haben und dann einen einzigen Aufruf durchführen.

8 Ausblick

Ausgehend von der Evaluation der Implementierung eröffnen sich weitere Forschungsfelder, die eine noch effizientere Implementierung erlauben würden. In diesem Kapitel stelle ich einen implementierungsspezifischen und einen architekturellen Ansatz für weitere Optimierungen vor.

8.1 Kompression von Daten

Das Cap'n-Proto-Format ist für schnelles Serialisieren und Deserialisieren optimiert. Insbesondere ermöglicht es *Random Access*, also Zugriff auf Teile von übertragenen Nachrichten, ohne dass die gesamte Nachricht in ein verarbeitbares Format deserialisiert werden muss.

Deshalb beinhaltet es Padding, also mit Nullen gefüllten Platz ohne Bedeutung, der dafür sorgt, dass auf gespeicherte Werte schneller zugegriffen werden kann. Beinhaltet eine Cap'n-Proto-Datenklasse beispielsweise eine 1-Byte-Zahl A und eine 4-Byte-Zahl B, dann gibt es mehrere Möglichkeiten, wie diese Werte im Speicher angeordnet werden können. Zwei mögliche Anordnungen sind hier abgebildet:

```
Byte:        0 1 2 3 4 5 6 7
Variante 1:  A B B B B
Variante 2:  A 0 0 0 B B B B
```

Variante 1 platziert alle Werte hintereinander und optimiert damit die Länge des Encodings. Variante 2 füllt Speicher zwischen den Werten mit Nullen und optimiert die Geschwindigkeit des Zugriffs: Die meisten CPUs ermöglichen nur dann, eine Zahl effizient vom RAM-Speicher in ein CPU-Register lesen, wenn sie an ihrer Speichergröße ausgerichtet (*aligned*) ist, also wenn die absolute Speicheradresse eine Vielfaches der Speichergröße ist. Ein 4-Byte-Zahl an den Speicheradressen 0, 4, 8, 12 etc. auszulesen (Variante 2), ist also effizienter als es an Adresse 1 auszulesen (Variante 1). [17]

Weil Cap'n Proto auf effiziente Deserialisierung optimiert ist, präferiert es deshalb Variante 2 über Variante 1, obwohl sie mehr Platz verbraucht. Der Autor von Cap'n Proto gesteht diesen Kompromiss ein und führt weiter aus, dass eine Kompression angewandt werden könne, wenn an Bandbreite gespart werden muss?[71]

[71] „Wont't [...] padding waste space on the wire? Yes. However, since all these extra bytes are zeros, when bandwidth matters, we can apply an extremely fast Cap'n-Proto-specific compression scheme to remove them. Cap'n Proto calls this ‚packing' the message; it achieves similar (better, even) message sizes to protobuf encoding, and it's still faster. When bandwidth really matters, you should apply general-purpose compression, like zlib or LZ4, regardless of your encoding format." [23]

In Tabelle 8.1 vergleiche ich mehrere Kompressionsverfahren anhand ihrer Kompressionsrate und Geschwindigkeit: ein Cap'n-Proto-spezifisches Packing,[72] zlib[73] und LZ4.[74]

Kompressionsverfahren	Größe	Kompressionsrate	Kompressionsdauer	Dekompressionsdauer
unkomprimiert	126 480 B	1,00	0,0 ms	0,0 ms
Packing	117 290 B	1,07	4,1 ms	4,5 ms
zlib	2 442 B	51,79	35,7 ms	24,3 ms
LZ4	2 517 B	50,25	0,7 ms	0,1 ms

Tabelle 8.1: Verschiedene Kompressionsverfahren wurden angewandt auf eine Cap'n-Proto-Liste mit 100 Messwerten je 100 zufälligen Bytes an Nutzdaten. Verglichen werden das Cap'n-Proto-eigene Packing, zlib und LZ4. Die Zeitwerte sind jeweils der Durchschnitt von 1000 Kompressionen bzw. Dekompressionen.

Packing Das Cap'n-Proto-Packing macht sich zunutze, dass in Cap'n-Proto-Nachrichten viele Nullen hintereinander stehen. Eine Reihe von Null-Bytes wird durch eine Null gefolgt von einem Byte mit der Anzahl der weiteren Nullen ersetzt. Werden beispielhaft die Werte A = 42 und B = 100 in das obige Beispiel `A 0 0 0 B B B B` eingesetzt, wird die resultierende Byte-Folge `42 0 0 0 0 0 0 100` auf `42 0 5 100` komprimiert (zu Lesen als: eine 42, eine Null gefolgt von 5 weiteren Nullen, eine 100). [22]

Cap'n-Proto-Packing erreicht aufgrund des einfachen Ansatzes nur eine niedrige Kompressionsrate, ist dafür aber vergleichsweise schnell.

zlib Das Kompressionsverfahren zlib komprimiert die Messwertpakete deutlich stärker. Dies ist möglich, weil viele Messwertpakete ähnlich zueinander aufgebaut sind und dadurch wenig Entropie besitzen. Bei tatsächlichen Sensordaten ist außerdem abzusehen, dass zeitlich nah beieinanderliegende Messwerte sich ähneln und das Kompressionsniveau noch besser wird. Um Regelmäßigkeiten zu erkennen, wird CPU-Zeit aufgewendet, sodass das Komprimieren länger dauert als beim Cap'n-Proto-Packing.

LZ4 Das Kompressionsverfahren LZ4 ist auf Geschwindigkeit optimiert?[75] Deshalb komprimiert es nicht ganz so gut wie zlib, aber um Größenordnungen schneller. Der Kompromiss zwischen CPU-Zeit und Kompressionsniveau lässt sich dynamisch über einen „Beschleunigungsfaktor" einstellen. [14] Besonders die auf den Team-Backends durchgeführte Dekompression ist schnell. Weil die Implementierung über Jahre von vielen Programmierenden optimiert wurde, ist sie sogar schneller als das Cap'n-Proto-Packing.

[72] capnp – crates.io: Rust Package Registry, `https://crates.io/crates/capnp` (besucht am 27. Juni 2021)

[73] zlib Home Site, `https://zlib.net` (besucht am 27. Juni 2021); Implementierung von `https://crates.io/crates/flate2` (besucht am 27. Juni 2021)

[74] LZ4 – Extremely fast compression, `https://lz4.github.io/lz4` (besucht am 27. Juni 2021); Implementierung von `https://crates.io/crates/lz4` (besucht am 27. Juni 2021)

[75] „LZ4 is a very fast lossless compression algorithm, providing compression speed at 400 MB/s per core, with near-linear scalability for multi-threaded applications. It also features an extremely fast decoder, with speed in multiple GB/s per core, typically reaching RAM speed limits on multi-core systems." [14]

Wo ein Austausch von CPU-Zeit gegen weniger Netzwerkaktivität sinnvoll ist, lohnt sich der Einsatz eines Kompressionsverfahrens. Die LZ4-Kompression verbraucht wenig CPU-Zeit und komprimiert Cap'n-Proto-typische Datenpakete um ca. zwei Größenordnungen.

Laut der Evaluation in Kapitel 7 wird zumindest beim Team-Backend das Netzwerk deutlich stärker ausgelastet als die CPU. Weil die Zuginfrastruktur eine ständige Stromversorgung und deshalb das Nutzen von CPU-Ressourcen keine nachteiligen Seiteneffekte auf die Langlebigkeit hat, ist naheliegend, die Nachrichten zwischen Team-Backend und Zug mit LZ4 zu komprimieren.

8.2 Software-Module im Zug ausführen

Auf einigen Zügen sind bereits leistungsstarke Computer vorhanden. [6] Daher kann es sinnvoll sein, direkt auf den Zügen Messwerte zu verarbeiten. Diese verteilt ausgeführten Operationen können ein Großteil der Messwerte lokal filtern und so den Netzwerkverkehr drastisch reduzieren.

Beispielsweise könnten die Audiodaten des Szenarios „Zustandsbasierte Instandhaltung von Bremsen“ innerhalb des Zuges mithilfe eines Machine-Learning-Modells auf Anomalien untersucht werden. Nur wenn Anomalien auftreten, müssen überhaupt Daten an das jeweilige Team-Backend übermittelt werden.

Zum einen kann die Einführung lokaler Verarbeitungsmodule durch die Vermeidung von Netzwerklast Ressourcen sparen, zum anderen verringert sie die Angriffsfläche bei sensiblen Daten. Sollen beim Szenario „Belegung von Personenwagen“ z. B. Personen aus Nachbarwagen bei der Untersuchung der WLAN-Pakete herausgefiltert werden, könnte ein lokales Verarbeitungsmodul die MAC-Adressen der WLAN-Probes von verschiedenen Wagen abgleichen und sie nur in dem Wagen mit dem besten Empfang zählen. Die MAC-Adressen, welche als personenbezogene Daten gelten, verlassen dabei nicht den Zug.

Eine Implementierung dieses Ansatzes ist außerhalb des Umfangs dieses Bachelorprojekts. Das Ausführen von Code-Modulen direkt im Zug hat jedoch Potenzial für große Dateneinsparungen und eignet sich deshalb als Gegenstand zukünftiger Untersuchungen.

9 Zusammenfassung

Eine Sensorplattform für Züge kann sich dadurch auszeichnen, dass sie besonders kostengünstige Sensoren in Wagen und Fracht ermöglicht. Dazu ist eine Verlagerung von Aufgaben auf eine in den Zügen eingebaute Infrastruktur nötig. Eine offene Plattform dieser Art gibt es noch nicht.

Eine solche Sensorplattform intrinsisch verteilt zu bauen sorgt für bessere Modularität und Skalierbarkeit. Die Modularität zeichnet sich dadurch aus, dass Komponenten unabhängig voneinander betrieben und weiterentwickelt werden können. Trotz zentraler Vergabe der Sensorknoten ist die Plattform skalierbar, weil der maximale Datenfluss durch keine zentrale Stelle limitiert wird.

Die meisten Daten müssen potenziell von den Backends der Teams verarbeitet werden. Anforderungen an CPU- und RAM-Ressourcen sind überschaubar. Der Netzwerkverkehr kann durch das Senden mehrerer Messwerte in einer Nachricht und durch Kompression von Nachrichten reduziert werden. Eine Verarbeitung von Messwerten direkt auf dem Zug stellt eine weitere Möglichkeit dar, die Netzwerklast zu verringern.

Literaturverzeichnis

[1] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau und Jie Xu. „Multi-Tenancy in Cloud Computing“. In: *Proceedings of the 8th IEEE International Symposium on Service Oriented System Engineering*. Oxford, UK: IEEE, Apr. 2014, Seite 2. ISBN: 978-1-4799-2504-9. DOI: 10.1109/SOSE.2014.50. URL: https://eprints.whiterose.ac.uk/80819/1/sose.pdf (besucht am 21. Juni 2021).

[2] Amazon Web Services, Inc. *AWS IoT Greengrass Features – Amazon Web Services*. URL: https://aws.amazon.com/greengrass/features (besucht am 19. Juni 2021).

[3] Andrew Banks, Ed Briggs, Ken Borgendale und Rahul Gupta. *MQTT Version 5.0 – Specification*. Organization for the Advancement of Structured Information Standards. März 2019, 1 – 2. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (besucht am 27. Juni 2021).

[4] Pino Bosesky, Christian Hoffmann und Sönke E. Schulz. „Datenhoheit im Cloud-Umfeld“. In: *Datenschutz und Datensicherheit* 2 (2013). DOI: 10.1007/s11623-013-0025-1. URL: https://link.springer.com/content/pdf/10.1007/s11623-013-0025-1.pdf.

[5] Centers for Disease Control and Prevention. *Pfizer-BioNTech COVID-19 Vaccine: Storage and Handling Summary*. Mai 2021. URL: https://www.cdc.gov/vaccines/covid-19/info-by-product/pfizer/downloads/storage-summary.pdf (besucht am 16. Juni 2021).

[6] DB Systel GmbH. *Five examples of the smart use of IT on the ICE 4*. Feb. 2018. URL: https://digitalspirit.dbsystel.de/en/five-examples-of-the-smart-use-of-it-on-the-ice-4 (besucht am 27. Juni 2021).

[7] Eclipse Foundation. *Eclipse IoT*. URL: https://iot.eclipse.org/ (besucht am 21. Juni 2021).

[8] European Medicines Agency. *Comirnaty: EPAR – Medicine overview, Annex I: Summary of Product Characteristics*. Mai 2021. URL: https://www.ema.europa.eu/en/documents/product-information/comirnaty-epar-product-information_en.pdf (besucht am 5. Juni 2021).

[9] Google. *Cloud IoT Core*. URL: https://cloud.google.com/iot-core/ (besucht am 19. Juni 2021).

[10] Google. *Getting Started with Multi-Tenancy*. URL: https://cloud.google.com/identity-platform/docs/multi-tenancy-quickstart (besucht am 27. Juni 2021).

[11] Google. *Serverless Architecture*. Technischer Bericht. URL: https://cloud.google.com/serverless/whitepaper (besucht am 11. Juni 2021).

[12] Carl Lerche. *Making the Tokio scheduler 10x faster*. Technischer Bericht. Okt. 2019. URL: https://tokio.rs/blog/2019-10-scheduler (besucht am 21. Juni 2021).

[13] LoRa Alliance. *What is LoRaWAN – Specification*. URL: https://lora-alliance.org/about-lorawan (besucht am 27. Juni 2021).

[14] *LZ4 – Extremely fast compression*. URL: https://lz4.github.io/lz4 (besucht am 27. Juni 2021).

[15] Heinrich Matrisch. *Classification and Preprocessing of Sensor Data in Networks*. Teil dieses Bachelorprojekts. Juni 2021.

[16] Microsoft. *Azure RTOS*. URL: https://azure.microsoft.com/en-us/services/rtos/ (besucht am 19. Juni 2021).

[17] Sumedh Naik. *Coding for Performance: Data alignment and structures*. Technischer Bericht. Intel Corporation, Sep. 2013. URL: https://software.intel.com/content/www/us/en/develop/articles/coding-for-performance-data-alignment-and-structures.html (besucht am 23. Juni 2021).

[18] J. Opara-Martins, R. Sahandani und F. Tian. „Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective“. In: *Journal of Cloud Computing* 5 (Apr. 2016). DOI: 10.1186/s13677-016-0054-z. URL: https://link.springer.com/article/10.1186/s13677-016-0054-z (besucht am 27. Juni 2021).

[19] Sami Salamin, Georgios Zervakis, Florian Klemme, Hammam Kattan, Yogesh Singh Chauhan, Jorg Henkel und Hussam Amrouch. „Impact of NCFET Technology on Eliminating the Cooling Cost and Boosting the Efficiency of Google TPU“. In: *IEEE Transactions on Computers* (März 2021). DOI: 10.1109/TC.2021.3065454.

[20] Leonard Seibold. *A flexible Toolbox to Stream Sensor Data from Microcontrollers*. Teil dieses Bachelorprojekts. Juni 2021.

[21] Clemens Tiedt. *A Robust Wireless Network Protocol for On-train IoT Applications*. Teil dieses Bachelorprojekts. Juni 2021.

[22] Kenton Varda. *Cap'n Proto: EncodingSpec*. URL: https://capnproto.org/encoding.html (besucht am 27. Juni 2021).

[23] Kenton Varda. *Cap'n Proto: Introduction*. URL: https://capnproto.org (besucht am 14. Juni 2021).

[24] Jonas Wanke. *From Measurement to Visualization: Live Railroad Car Occupancy as an Example Use Case for a Sensor Middleware*. Teil dieses Bachelorprojekts. Juni 2021.

[25] Jie Wu. *Distributed System Design*. CRC Press, Juli 1998, Seite 2. ISBN: 0849331781. URL: https://books.google.de/books?id=fEq2_vq-RGwC (besucht am 14. Juni 2021).

A Anhang

Cap'n-Proto-Definition der Kommunikation zwischen Team- und Management-Backend

Zu Beginn der Kommunikation stellt das Management-Backend ein `Backend`-Objekt zur Verfügung.

```
interface Backend {
    authenticate @0 (team :Text, authKey :Text) ->
        (backend :AuthenticatedBackend);
}

interface AuthenticatedBackend {
    setRecordDestination @0 (destination :Text);
    createNode @1 () -> (node :Int32);
    listNodes @2 () -> (nodes :List(Int32));
    deleteNode @4 (id :Int32) -> ();
}
```

Cap'n-Proto-Definition der Kommunikation zwischen Zug und Team-Backend

Zu Beginn der Kommunikation stellt das Team-Backend ein `Backend`-Objekt zur Verfügung.

```
interface Backend {
    authenticate @0 (authKey :Text) -> (backend :AuthenticatedBackend);
}

interface AuthenticatedBackend {
    getSensor @0 (nodeId :Int32, sensorId :UInt8) -> (sensor :Sensor);
}

interface Sensor {
    publishRecords @0 (records :List(Record));
}

struct Record {
```

```
    timestamp @0 :UInt64;
    payload @1 :Data;
}
```

Cap'n-Proto-Definition der Kommunikation zwischen Zug und Management-Backend

Zu Beginn der Kommunikation stellt das Management-Backend ein `Backend`-Objekt zur Verfügung.

```
interface Backend {
    authenticate @0 (authKey :Text) -> (backend :AuthenticatedBackend);
}

interface AuthenticatedBackend {
    getAddressForNode @0 (nodeId :Int32) ->
        (address :Text, authKey :Text);
}
```

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Architektur und Implementierung eines modularen, skalierbaren Backends für Sensordaten verschiedener Nutzer“ („Architecture and Implementation of a Modular, Scalable Backend for Sensor Data from Multiple Users“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 30. Juni 2021,

_____

(Marcel Garus)