Kurzfassung

Im Rahmen dieser Arbeit wurde ein Akka.NET-basiertes verteiltes System zur Verarbeitung von Formel-1-Telemetriedaten konzipiert, implementiert und evaluiert. Ziel des Use-Cases war es, kontinuierlich eintreffende Renndaten abzufangen, zu verarbeiten und in eine geeignete Datenstruktur zu transformieren, sodass diese resilient und fehlertolerant bereitgestellt werden können.

Ein zentraler Fokus lag dabei auf der Entwicklung einer Architektur, die eine robuste Verarbeitung auch bei auftretenden Fehlern oder Knotenausfällen gewährleistet. Durch den Einsatz des Aktorenmodells, asynchroner Nachrichtenkommunikation sowie Cluster-Sharding konnte ein System entworfen werden, das eingehende Datenströme zuverlässig verarbeitet und Zustandsänderungen konsistent verwaltet. Die implementierte Struktur erwies sich insbesondere im Hinblick auf Fehlertoleranz und Wiederherstellbarkeit nach Systemausfällen als geeignet.

Während der Umsetzung zeigte sich, dass die Konfiguration und Einrichtung eines verteilten Akka.NET-Clusters mit erheblichem Aufwand verbunden ist. Insbesondere die korrekte Abstimmung der Cluster-, Sharding- und Persistenzkomponenten erforderte eine intensive Einarbeitung, was teilweise auf die vergleichsweise geringere Verbreitung von Akka.NET im Vergleich zu anderen Streaming- oder Microservice-Technologien zurückzuführen ist.

Die Evaluation verdeutlicht jedoch, dass sich Akka.NET besonders für Szenarien eignet, in denen häufige Zustandsänderungen in Echtzeit verarbeitet werden müssen. Die Kombination aus asynchroner Aktorenkommunikation, integrierter Fehlertoleranz und automatischer Wiederherstellung von Zuständen ermöglicht eine stabile und resiliente Datenverarbeitung auch unter hoher Last oder bei partiellen Systemausfällen.

Ein weiterer Vorteil besteht in der Unterstützung verteilter Ausführung über Remote-Kommunikation, wodurch Ressourcen mehrerer Rechner effizient genutzt werden können. Dies eröffnet zusätzliche Möglichkeiten zur horizontalen Skalierung und zur besseren Auslastung verfügbarer Systemressourcen.

Abstract

In this thesis, Akka.NET was evaluated based on a concrete use case involving the processing of Formula 1 telemetry data. The goal of the use case was to capture continuously incoming race data, process it, and transform it into a suitable data structure that can be provided in a resilient and fault-tolerant manner.

A central focus was the design of an architecture capable of maintaining reliable data processing even in the presence of errors or node failures. By leveraging the actor model, asynchronous message passing, and cluster sharding, a distributed system was implemented that consistently handles streaming data and manages state changes in a robust way. The resulting structure proved to be particularly effective with respect to fault tolerance and state recovery after system crashes.

During the implementation, it became evident that configuring and operating a distributed Akka.NET cluster requires considerable effort. In particular, the correct setup and tuning of cluster, sharding, and persistence components demanded extensive configuration work. This complexity can partly be attributed to the comparatively lower popularity of Akka.NET compared to other streaming or microservice technologies, which results in fewer practical resources and examples.

However, the evaluation shows that Akka.NET is especially well suited for scenarios in which frequent state updates must be processed in real time. The combination of asynchronous actor communication, built-in fault tolerance, and automatic state recovery enables stable and resilient data processing even under high load or partial system failures.

Another important advantage is the support for remote and distributed execution. This allows the system to utilize resources across multiple machines, enabling horizontal scalability and a more efficient use of available computational resources.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Moderne Softwaresysteme stehen zunehmend vor der Herausforderung, kontinuierlich anfallende Datenströme performant, skalierbar und fehlertolerant zu verarbeiten (Bonér et al., 2014a; Kuhn, Hanafee & Allen, 2017a). Insbesondere in datenintensiven Anwendungsdomänen wie der Analyse von Telemetriedaten entstehen hochfrequente Datenströme, die in Echtzeit erfasst, verarbeitet und bereitgestellt werden müssen. Klassische monolithische Architekturen stoßen hierbei schnell an ihre Grenzen, da sie nur begrenzt skalierbar sind und eine geringe Fehlertoleranz aufweisen (Kuhn, Hanafee & Allen, 2017a).

Reaktive Architekturen auf Basis des Aktorenmodells bieten einen vielversprechenden Ansatz zur Bewältigung dieser Anforderungen (Bonér et al., 2014a). Durch lose Kopplung, natürliche Parallelisierung sowie eingebaute Fehlertoleranz eignen sie sich besonders für die Verarbeitung kontinuierlicher Datenströme. Frameworks wie Akka.NET ermöglichen die Entwicklung verteilter Aktorensysteme, während Akka.Streams reaktive Datenpipelines mit integrierten Backpressure-Mechanismen bereitstellt (Kuhn, Hanafee & Allen, 2017a). In Kombination mit Cluster Sharding können zustandsbehaftete Entitäten über mehrere Knoten verteilt und dynamisch skaliert werden (Kuhn, Hanafee & Allen, 2017a).

1.1 Motivation

Die Motivation dieser Arbeit ergibt sich aus der Notwendigkeit, kontinuierliche Telemetriedaten effizient und skalierbar zu verarbeiten. Im Kontext des Motorsports fallen während eines Rennens große Mengen an Daten an, beispielsweise Positions-, Geschwindigkeits- und Zustandsinformationen einzelner Fahrzeuge. Diese Daten müssen in Echtzeit verarbeitet werden, um aktuelle Rennsituationen analysieren und visualisieren zu können.

Gleichzeitig müssen solche Systeme auch unter hoher Last stabil bleiben und Ausfälle einzelner Komponenten kompensieren können. Verteilte Aktorensysteme in Kombination mit streambasierter Datenverarbeitung stellen hierfür einen geeigneten Ansatz dar.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist die Konzeption und Implementierung eines verteilten, aktorenbasierten Stream-Verarbeitungssystems zur Verarbeitung kontinuierlicher Telemetriedaten. Dabei soll ein System entwickelt werden, das eingehende Datenströme zuverlässig empfängt, verarbeitet und innerhalb eines Clusterverbunds verteilt bereitstellt.

Die Architektur basiert auf dem Akka.NET-Framework und integriert zentrale Konzepte wie Cluster Sharding, reaktive Stream-Verarbeitung mit Backpressure sowie eine koordinierte Kommunikation zwischen den Systemkomponenten.

1.3 Ziel der Arbeit

Das Ziel der Arbeit besteht in der Entwicklung und prototypischen Umsetzung einer skalierbaren und fehlertoleranten Architektur zur Echtzeitverarbeitung von Telemetriedaten. Es soll gezeigt werden, wie durch den Einsatz von Akka.NET, Akka.Streams und Cluster Sharding eine verteilte Verarbeitung kontinuierlicher Datenströme realisiert werden kann.

Darüber hinaus soll die entwickelte Architektur hinsichtlich ihrer Eignung für reaktive Systeme bewertet und ihre Erweiterbarkeit für zukünftige Anwendungen aufgezeigt werden.

1.4 Vorgehensweise

Zu Beginn werden die theoretischen Grundlagen zu reaktiven Systemen, dem Aktorenmodell sowie der Stream-Verarbeitung erläutert. Darauf aufbauend erfolgt die Konzeption der Systemarchitektur, in der die Rollen der Cluster-Komponenten definiert und deren Zusammenspiel beschrieben werden. Darauf folgt die detaillierte Implementierung des Systems. Abschließend werden Tests und Analysen durchgeführt, um das Verhalten des Systems hinsichtlich Skalierbarkeit, Fehlertoleranz und Leistungsfähigkeit zu evaluieren.

1.5 Statement zur Nutzung von AI-Tools

Im Rahmen dieser Arbeit wurden KI-gestützte Werkzeuge unterstützend eingesetzt. Insbesondere wurden textgenerierende Systeme zur sprachlichen Überarbeitung, Formulierungshilfe sowie zur Strukturierung einzelner Textpassagen verwendet. Die inhaltliche Konzeption, Implementierung, Analyse der Ergebnisse sowie die wissenschaftliche Bewertung wurden eigenständig vom Verfasser durchgeführt.

Alle fachlichen Inhalte, architektonischen Entscheidungen und Implementierungen basieren auf eigener Ausarbeitung und wurden kritisch geprüft. Die Nutzung der KI-Werkzeuge diente ausschließlich der Unterstützung bei der sprachlichen Ausarbeitung

und hatte keinen Einfluss auf die wissenschaftlichen Ergebnisse oder Schlussfolgerungen der Arbeit.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die theoretischen und technologischen Grundlagen vor, die für das Verständnis der in dieser Arbeit entwickelten Systemarchitektur erforderlich sind. Zunächst werden reaktive Systeme und das Aktorenmodell als konzeptionelle Basis verteilter, nebenläufiger Anwendungen eingeführt. Anschließend wird das Framework Akka.NET vorgestellt, dessen zentrale Mechanismen für Skalierbarkeit, Fehlertoleranz und zustandsbehaftete Echtzeitverarbeitung erläutert werden. Die dargestellten Konzepte bilden die Grundlage für die spätere Konzeption und Implementierung des verteilten Systems zur Verarbeitung von Telemetriedaten.

2.1 Einführung in Reactive Systems und Aktorenmodell

Dieser Abschnitt führt in die grundlegenden Konzepte reaktiver Systeme sowie in das Aktorenmodell ein, die die konzeptionelle Basis für moderne verteilte Softwaresysteme bilden. Zunächst wird die Motivation für reaktive Architekturen im Kontext wachsender Systemkomplexität und Verteilung erläutert. Anschließend werden die zentralen Eigenschaften reaktiver Systeme dargestellt, bevor das Aktorenmodell als geeignetes Paradigma zur Realisierung nebenläufiger, skalierbarer und fehlertoleranter Anwendungen vorgestellt wird.

2.1.1 Motivation reaktiver Systeme

Da sich bei der Entwicklung moderner Softwaresysteme immer wieder ähnliche Anforderungsmuster zeigen, wird deutlich, dass klassische Architekturen den heutigen Bedingungen nur noch eingeschränkt gerecht werden. Der rasante technische Fortschritt sowie die zunehmende gesellschaftliche Nutzung digitaler Systeme führen dazu, dass Anwendungen deutlich komplexer, verteilter und leistungsintensiver geworden sind. Während große Anwendungen früher aus wenigen miteinander kommunizierenden Servern bestanden, arbeiten heutige Systeme auf einer Vielzahl verteilter Knoten, häufig in Cloud-Umgebungen, und müssen große Datenmengen bei gleichzeitig hohen Verfügbarkeitsanforderungen verarbeiten (Kuhn, Allen & Hanafee, 2017; Bonér et al., 2014b).

Ein zentrales Problem moderner Softwaresysteme besteht darin, dass sie unter allen Umständen reaktionsfähig bleiben sollen, obwohl einzelne Komponenten jederzeit ausfallen können. Die zunehmende Verteilung von Anwendungen über mehrere Rechner macht deutlich, dass die Nebenläufigkeit und Fehlermöglichkeiten verteilter Systeme nicht länger verborgen oder abstrahiert werden können. Frühere Ansätze versuchten, die Illusion einer lokal und sequenziell arbeitenden Anwendung aufrechtzuerhalten, obwohl diese tatsächlich auf mehreren Kernen oder Netzwerkknoten ausgeführt wurde. Die wachsende Diskrepanz zwischen dieser Abstraktion und der tatsächlichen Systemrealität führte jedoch zu erhöhter Komplexität und Fehlanfälligkeit. Moderne Architekturen müssen daher die verteilte und nebenläufige Natur von Anwendungen explizit im Programmiermodell berücksichtigen, anstatt sie zu verstecken. Nur so kann gewährleistet werden, dass Systeme auch bei Teilausfällen, Lastschwankungen oder Programmfehlern weiterhin antwortbereit bleiben (Kuhn, Allen & Hanafee, 2017, Kap. 1).

Um diesen neuen Anforderungen gerecht zu werden, wurden vier zentrale Qualitätsmerkmale identifiziert, die zuvor meist isoliert betrachtet wurden. Reaktive Systeme zeichnen sich durch vier zentrale Eigenschaften aus: eine zeitnahe Reaktionsfähigkeit, eine hohe Fehlertoleranz, die Fähigkeit zur elastischen Skalierung sowie eine konsequent nachrichtenbasierte Kommunikation. Diese Merkmale bilden die Grundlage für den Entwurf verteilter, lastadaptiver Softwaresysteme. Erst die Kombination dieser Eigenschaften definiert ein sogenanntes reaktives System (Kuhn, Allen & Hanafee, 2017; Bonér et al., 2014b).

2.1.2 Eigenschaften reaktiver Systeme

Nach dem *Reactive Manifesto* zeichnen sich reaktive Systeme durch vier zentrale Eigenschaften aus (Bonér et al., 2014b):

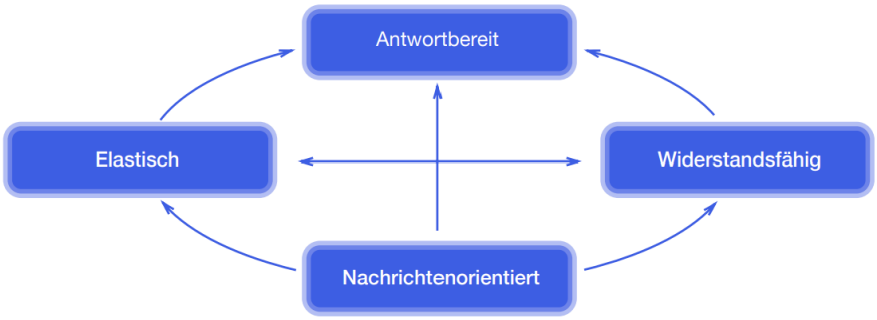

Abbildung 2.1: Eigenschaften reaktiver Systeme (Bonér et al., 2014b)

- **Antwortbereit:** Das System reagiert zeitgerecht auf Anfragen. Vorhersagbare Antwortzeiten sind essenziell für Benutzerfreundlichkeit und Fehlererkennung.
- **Widerstandsfähig:** Das System bleibt trotz Ausfällen funktionsfähig. Fehler werden isoliert behandelt und nicht global propagiert.
- **Elastisch:** Das System kann sich dynamisch an wechselnde Lastbedingungen anpassen, indem Ressourcen horizontal skaliert werden.

- **Nachrichtenorientiert:** Komponenten kommunizieren über asynchrone Nachrichten, wodurch lose Kopplung, Isolation und Lastverteilung ermöglicht werden.

Diese Eigenschaften bilden die konzeptionelle Grundlage für verteilte, fehlertolerante Softwaresysteme (Bonér et al., 2014b).

2.1.3 Das Aktorenmodell

Das Aktorenmodell ist ein Nebenläufigkeits- und Verteilungsmodell zur Strukturierung komplexer Softwaresysteme. Es wurde ursprünglich von Carl Hewitt in den 1970er-Jahren eingeführt und beschreibt ein System als Menge unabhängiger, parallel ausführbarer Einheiten, sogenannter Aktoren (Hewitt et al., 1973).

Ein Aktor stellt im Aktorenmodell eine isolierte Verarbeitungseinheit dar, deren Interaktion mit anderen Komponenten ausschließlich über asynchrone Nachrichten erfolgt. Durch diese strikte Entkopplung wird der Zugriff auf gemeinsamen Zustand vermieden, wodurch typische Nebenläufigkeitsprobleme wie Race Conditions reduziert werden. Beim Empfang einer Nachricht kann ein Aktor drei grundlegende Aktionen ausführen: (1) seinen internen Zustand verändern, (2) neue Nachrichten an andere Aktoren senden und (3) neue Aktoren erzeugen (Hewitt et al., 1973).

Ein wesentlicher Vorteil des Aktorenmodells besteht in der strikten Kapselung von Zustand und Verhalten. Jeder Aktor verwaltet seinen Zustand privat und interagiert mit anderen Aktoren ausschließlich über asynchrone Nachrichten. Beim Empfang einer Nachricht verarbeitet der Aktor diese gemäß seinem aktuellen Verhalten und kann daraufhin seinen Zustand ändern, weitere Nachrichten versenden oder neue Aktorinstanzen erzeugen. Durch die Isolation des Zustands und die ausschließliche Kommunikation über Nachrichten werden typische Probleme nebenläufiger Programmierung, wie Race Conditions oder Deadlocks, weitgehend vermieden (Hewitt et al., 1973; Kuhn, Hanafee & Allen, 2017b, pp. 12–14).

2.2 Akka.NET als Framework für verteilte reaktive Systeme

Aufbauend auf den zuvor erläuterten Konzepten reaktiver Systeme und des Aktorenmodells wird in diesem Abschnitt das Framework Akka.NET vorgestellt. Akka.NET bietet eine konkrete Implementierung des Aktorenmodells zur Entwicklung verteilter, fehlertoleranter und skalierbarer Anwendungen. Im Folgenden werden die zentralen Mechanismen erläutert, die für die Umsetzung reaktiver Systeme mit Akka.NET relevant sind. Dazu zählen insbesondere das asynchrone Nachrichtenmodell, Cluster-Funktionalitäten, Cluster Sharding, persistente Aktoren, Cluster-Singletons sowie Mechanismen zur verteilten Kommunikation und Datenstromverarbeitung.

2.2.1 Aktoren und asynchrones Nachrichtenmodell

Akka.NET stellt ein Framework zur Umsetzung verteilter und fehlertoleranter Softwaresysteme auf Basis des Aktorenmodells bereit. Es abstrahiert Nebenläufigkeit und

Verteilung durch ein einheitliches Nachrichtenmodell und ermöglicht damit die Entwicklung hochgradig skalierbarer Anwendungen ohne explizite Synchronisationsmechanismen (Roestenburg et al., 2016; Project, 2025e).

Die Kommunikation zwischen Aktoren erfolgt vollständig asynchron über Nachrichten, die in Mailboxen zwischengespeichert und sequenziell verarbeitet werden. Dieses nichtblockierende Nachrichtenmodell führt zu einer klaren Entkopplung der Komponenten und reduziert typische Nebenläufigkeitsprobleme wie Race Conditions oder Deadlocks (Kuhn, Hanafee & Allen, 2017b, pp. 12–18). Gleichzeitig erlaubt es eine hohe Parallelität und flexible Skalierung über mehrere Knoten hinweg.

Darüber hinaus unterstützt Akka.NET transparente Kommunikation zwischen lokal und verteilt ausgeführten Aktoren, wodurch verteilte Systeme entwickelt werden können, ohne die Anwendungslogik an Netzwerkgrenzen anzupassen (Roestenburg et al., 2016, pp. 7–10). Durch die nachrichtengetriebene Interaktion folgt Akka.NET den Prinzipien reaktiver Systeme wie Reaktionsfähigkeit, Fehlertoleranz und Elastizität (Bonér et al., 2014a).

2.2.2 Cluster und Fehlertoleranz

Akka.NET stellt mit dem Modul *Akka.Cluster* Mechanismen zur Realisierung fehlertoleranter und skalierbarer verteilter Systeme bereit. Ein Cluster bildet dabei ein dezentrales Peer-to-Peer-Netzwerk mehrerer Akka.NET-Knoten ohne zentralen Koordinator oder Single Point of Failure (Project, 2025a). Neue Knoten können dem Cluster dynamisch beitreten, während ausgefallene Knoten automatisch erkannt und aus dem Verbund entfernt werden, wodurch eine hohe Verfügbarkeit des Gesamtsystems gewährleistet wird.

Die Clusterfunktionalität basiert auf dem Remoting-Modul von Akka.NET, erweitert dieses jedoch um Funktionen zur automatischen Knotenerkennung, Rollenverwaltung sowie zur Lastverteilung über clusterfähige Router (Project, 2025a). Dadurch lassen sich Anwendungen elastisch skalieren und zustandsbehaftete Komponenten replizieren, ohne die Anwendungslogik anpassen zu müssen.

Ein zentrales Ziel von Akka.Cluster ist die Fehlertoleranz verteilter Systeme. Durch die dezentrale Organisation können Ausfälle einzelner Knoten kompensiert werden, während andere Instanzen weiterhin Anfragen verarbeiten. Dieses Prinzip entspricht den Anforderungen reaktiver Systeme, die auf Resilience und Elasticity ausgelegt sind (Bonér et al., 2014a). Die Kombination aus Aktorenmodell und Clustering ermöglicht es somit, robuste, hochverfügbare und horizontal skalierbare Anwendungen zu entwickeln (Roestenburg et al., 2016, Kap.8).

2.2.3 Cluster Sharding

Cluster Sharding ist ein Mechanismus in Akka.NET zur verteilten Verwaltung zustandsbehafteter Aktoren (Entities) über mehrere Clusterknoten hinweg. Aktoren werden dabei über ihre logische Identität adressiert, ohne dass ihre physische Platzierung im Cluster bekannt sein muss. Erstellung, Lokalisierung und Migration der Entitäten erfolgen

automatisch durch das Framework, wodurch skalierbare verteilte Systeme vereinfacht umgesetzt werden können (Project, 2025b).

Zur Lastverteilung werden Entities in sogenannte Shards gruppiert, die als Verteilungseinheiten auf die Clusterknoten verteilt werden. Beim Beitritt oder Ausfall von Knoten können Shards automatisch neu zugewiesen werden (Rebalancing), sodass das System elastisch auf Veränderungen der Clustergröße reagiert (Roestenburg et al., 2016; Project, 2025b).

Je nach Konfiguration kann die Platzierung der Shards entweder über verteilte Datenstrukturen (CRDT-basierte Replikation) oder über persistente Koordinatoren verwaltet werden. Cluster Sharding ist dabei nur auf Knoten im Status *Up* aktiv, wodurch ausschließlich vollständig integrierte Knoten an der Verteilung beteiligt sind (Project, 2025b).

Insgesamt ermöglicht Cluster Sharding eine transparente Adressierung verteilter Aktoren sowie eine automatische Lastverteilung und stellt damit einen zentralen Baustein für skalierbare und fehlertolerante reaktive Systeme dar (Bonér et al., 2014a; Kuhn, Hanafee & Allen, 2017a, pp. 316–319).

2.2.4 Persistente Aktoren

Für die dauerhafte Speicherung zustandsbehafteter Aktoren wird in der Architektur Akka.Persistence eingesetzt. Dieses Modul ermöglicht es, interne Zustandsänderungen von Aktoren zu persistieren, sodass ihr Zustand nach einem Neustart, Ausfall oder einer Migration im Cluster wiederhergestellt werden kann (Project, 2025f). Dabei wird nicht der aktuelle Zustand direkt gespeichert, sondern ausschließlich die Änderungen am Zustand in Form von Ereignissen protokolliert.

Dieses Vorgehen folgt dem Event-Sourcing-Paradigma, bei dem alle Zustandsänderungen als unveränderliche Ereignisse in einem Journal abgelegt werden. Das Journal stellt somit die alleinige Quelle der Wahrheit dar, aus der der aktuelle Zustand eines Aktors durch erneutes Abspielen der Ereignisse rekonstruiert werden kann (Roestenburg et al., 2016, Kap. 9). Zur Beschleunigung der Wiederherstellung können zusätzlich Snapshots des aktuellen Zustands gespeichert werden, sodass nicht immer die vollständige Ereignishistorie repliziert werden muss (Project, 2025f).

Die Verwendung von Event Sourcing bietet mehrere Vorteile für verteilte Systeme. Da nur Zustandsänderungen persistiert und niemals überschrieben werden, entsteht eine nicht veränderliche Historie, die eine effiziente Replikation und hohe Transaktionsraten ermöglicht (Project, 2025f). Gleichzeitig lassen sich Persistenz, Änderungsnachverfolgung und Wiederherstellung konsistent vereinen, was insbesondere in skalierbaren und fehlertoleranten Architekturen von zentraler Bedeutung ist.

Im Kontext reaktiver Systeme stellt Event Sourcing zudem einen wichtigen Baustein zur konsistenten Zustandsverwaltung in elastischen Clustern dar, da replizierte Ereignisströme als Grundlage für verteilte Zustandsrekonstruktion und Event-basierte Kommunikation dienen (Kuhn, Hanafee & Allen, 2017a, pp. 312–315).

2.2.5 Cluster Singleton

Ein Cluster Singleton stellt sicher, dass innerhalb eines verteilten Clusters genau eine Instanz eines bestimmten Aktors aktiv ist. Dieses Konzept wird eingesetzt, wenn clusterweite Koordination oder konsistente Entscheidungen zentral getroffen werden müssen, beispielsweise als zentraler Einstiegspunkt für externe Systeme oder zur Koordination verteilter Arbeitsknoten (Project, 2025c).

Ein Cluster Singleton wird typischerweise auf einem geeigneten Knoten im Cluster ausgeführt und bei Ausfall automatisch auf einen anderen Knoten verlagert, sodass die Verfügbarkeit erhalten bleibt. Damit wird verhindert, dass mehrere konkurrierende Instanzen gleichzeitig existieren, was insbesondere für konsistente Zustandsverwaltung oder globale Koordinationsaufgaben erforderlich ist (Kuhn, Hanafee & Allen, 2017a, pp. 186–187).

Der Einsatz eines Singletons bringt jedoch auch Nachteile mit sich. Da alle Anfragen an eine zentrale Instanz gerichtet werden, kann dieser Aktor zu einem Engpass werden und potenziell eine Engstelle im System darstellen. Obwohl ein neuer Singleton nach einem Ausfall automatisch gestartet wird, kann es während der Migration zu kurzzeitigen Unterbrechungen kommen (Roestenburg et al., 2016, Kap. 13.4).

Cluster Singletons eignen sich daher insbesondere für Aufgaben mit zentraler Verantwortung, Koordinatoren oder zentrale Routinglogik, während für hochgradig skalierende und stark parallelisierte Verarbeitung eher verteilte Aktoren ohne zentrale Instanz bevorzugt werden sollten (Project, 2025c).

2.2.6 Verteiltes Publish/Subscribe

Akka.NET bietet mit dem Modul *Distributed Publish/Subscribe* einen Mechanismus zur losen Kopplung verteilter Aktoren über Topics. Subscriber registrieren sich beim lokalen Mediator, während Publisher Nachrichten an ein Topic senden, ohne die konkreten Empfänger kennen zu müssen. Die Zustellung erfolgt asynchron und wird clusterweit repliziert, wodurch eine skalierbare und entkoppelte Ereignisverteilung innerhalb verteilter Aktorensysteme ermöglicht wird (Project, 2025d).

2.2.7 Reaktive Streams

Unter einem Datenstrom wird eine kontinuierliche Folge von Ereignissen verstanden, die inkrementell verarbeitet wird, anstatt als vollständige Datenmenge vorzuliegen. Dieses Verarbeitungsmodell ist insbesondere für Echtzeitanwendungen relevant, in denen Daten fortlaufend eintreffen und ohne Verzögerung ausgewertet werden müssen (Roestenburg et al., 2016, Kap. 10).

Ein zentrales Problem bei der Verarbeitung solcher Datenströme entsteht, wenn Produzenten Daten schneller erzeugen, als Konsumenten diese verarbeiten können. Ohne geeignete Flusskontrolle würden sich Nachrichten in Puffern ansammeln und letztlich zu Ressourcenerschöpfung oder Systemausfällen führen (Roestenburg et al., 2016, Kap. 10). Zur Lösung dieses Problems wurde die Initiative Reactive Streams ins Le-

ben gerufen, auf deren Spezifikation Akka.Streams basiert. Sie definiert ein asynchrones, nicht-blockierendes Kommunikationsmodell mit integriertem Backpressure, bei dem Konsumenten die Datenrate steuern, indem sie aktiv Nachfrage signalisieren (Project, 2025g).

Dieses Prinzip entspricht dem Pull-Pattern, bei dem Verbraucher Arbeitseinheiten explizit anfordern und so die Verarbeitungskapazität kontrollieren. Dadurch kann sich das System dynamisch an unterschiedliche Lastsituationen anpassen und verhindert eine Überlastung einzelner Komponenten (Kuhn, Hanafee & Allen, 2017a, pp. 295–298). Backpressure bildet somit ein zentrales Konzept für resiliente und skalierbare Datenstromverarbeitung in reaktiven Systemen.

Kapitel 3

Konzept und Architektur

In diesem Kapitel wird das konzeptionelle und technische Fundament des entwickelten Systems dargestellt. Ziel ist der Entwurf einer skalierbaren, fehlertoleranten und reaktiven Architektur, die sowohl Live-Daten als auch historische Daten effizient verarbeiten kann. Die erfassten Daten werden in das System eingespielt, dort strukturiert verwaltet und anschließend so aufbereitet, dass nachgelagerte Komponenten sie ohne zusätzliche Verarbeitungsschritte nutzen können. Im weiteren Verlauf werden zunächst die Grundlagen der Skalierbarkeit erläutert, bevor die Systemarchitektur, die Rollen der Cluster-Komponenten sowie die Kommunikations- und Datenflüsse innerhalb des Systems vorgestellt werden.

3.1 Skalierbarkeit und Motivation

Ein zentraler Aspekt des Projekts ist die effiziente Verarbeitung großer Datenmengen in Echtzeit. Im Kontext der Formel-1-Datenanalyse entstehen pro Rennen mehrere Hunderttausend Datensätze, die eine Vielzahl unterschiedlicher Informationen umfassen, von Telemetriedaten über Zwischenzeiten, Positionswechsel und Boxenstopps bis hin zu Sektorzeiten. Diese Datenströme sollen parallel verarbeitet und in aggregierter Form zur Darstellung der aktuellen Rennsituation der einzelnen Fahrer bereitgestellt werden.

Um sicherzustellen, dass das System auch bei steigender Datenrate zuverlässig arbeitet, muss es skalierbar ausgelegt sein. Skalierbarkeit beschreibt die Fähigkeit eines Softwaresystems, seine Leistungsfähigkeit durch das Hinzufügen von Ressourcen zu erhöhen, ohne Änderungen an der Anwendung selbst vornehmen zu müssen (Bass et al., 2021). Ein skalierbares System kann somit wachsende Datenmengen oder Benutzeranforderungen verarbeiten, ohne dass ein vollständiges Redesign erforderlich wird.

Im Rahmen dieses Projekts wird horizontale Skalierung realisiert. Unter horizontaler Skalierung versteht man die Erweiterung eines Systems durch das Hinzufügen zusätzlicher Knoten, die gemeinsam dieselbe Aufgabe ausführen können (Bass et al., 2021). Die Anwendung kann als verteiltes System mit mehreren Knoten betrieben werden, wobei die einzelnen Rollen des Systems entweder gemeinsam in einem Prozess oder

getrennt auf unterschiedliche Knoten verteilt ausgeführt werden können. Diese flexible Ausführungsstrategie ermöglicht sowohl den Betrieb als monolithische Anwendung als auch eine skalierte Ausführung über mehrere Prozesse oder physische Maschinen. Selbst im monolithischen Modus kann das System dynamisch erweitert werden, indem zusätzliche Prozesse in den Cluster eingebunden werden. Durch das Hinzufügen weiterer Knoten wird die Arbeitslast automatisch auf mehrere Instanzen verteilt, was sowohl die Gesamtleistung als auch die Fehlertoleranz erhöht. Fällt ein Knoten aus, übernehmen verbleibende Instanzen mit derselben Rolle automatisch deren Aufgaben.

Durch den Einsatz von Akka.NET und dem Cluster Sharding Mechanismus können neue Knoten nach ihrer Registrierung am Hauptknoten dynamisch in das System integriert werden. Nach dem Beitritt zum Cluster übernehmen sie automatisch Teile der Verarbeitung, ohne dass Konfigurationsänderungen oder Neustarts erforderlich sind. Dieses Skalierungsprinzip führt zu einer flexiblen und anpassungsfähigen Architektur, bei der neu hinzukommende Prozesse automatisch in den Cluster integriert werden, ohne dass zusätzliche Konfigurationsschritte erforderlich sind.

3.2 Gesamtarchitektur des Systems

Die Architektur des entwickelten Systems basiert auf dem Akka.NET-Framework und nutzt dessen Cluster-Mechanismen zur Realisierung einer verteilten, fehlertoleranten und skalierbaren Anwendung. Abbildung 3.1 zeigt die zentralen Komponenten des Systems und deren Interaktion.

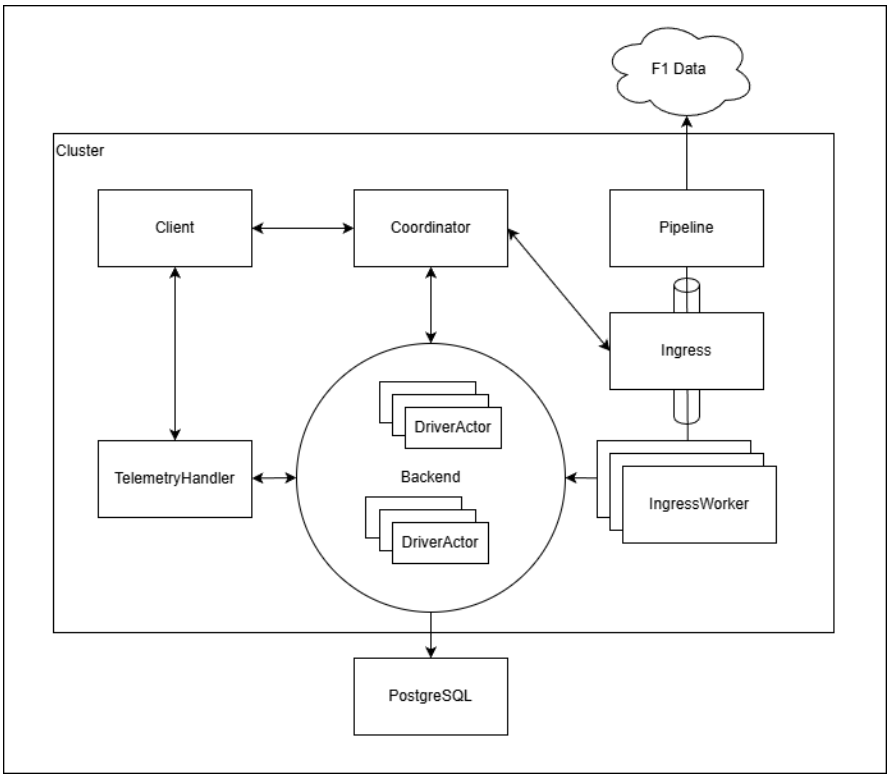

Abbildung 3.1: Architektur des Prototyps

Das System besteht aus mehreren Knoten, die unterschiedliche Rollen übernehmen. Für den Betrieb des Clusters sind grundsätzlich zwei Ausführungsmodi vorgesehen:

- **Monolithischer Modus:** In diesem Modus werden alle Rollen innerhalb eines einzelnen Prozesses ausgeführt. Diese Variante eignet sich insbesondere für Entwicklungs- und Testzwecke, da sie die Systemkomplexität reduziert und die Einrichtung vereinfacht.
- **Verteilte Ausführung:** Hierbei werden die verschiedenen Rollen auf separate Prozesse oder physische Maschinen verteilt. Dadurch wird eine effizientere Ressourcennutzung ermöglicht und das System kann bei steigender Last horizontal skaliert werden, da einzelne Komponenten unabhängig voneinander erweitert werden können.

Jeder Knoten im Cluster kann eine der folgenden Rollen übernehmen: Ingress, Backend und Coordinator. Diese Rollen sind für die verschiedenen Aufgaben innerhalb des Systems verantwortlich und arbeiten zusammen, um die Verarbeitung und Bereitstellung der Daten sicherzustellen. Die drei Rollen werden in Abschnitt 3.3 im Detail erläutert.

Zur Verwaltung und strukturierten Weiterleitung der Daten wird das Cluster-Sharding-Konzept von Akka.NET eingesetzt. Damit Nachrichten effizient an die jeweilige Shard-Region gesendet und von dieser empfangen werden können, kommen Proxies zum Einsatz, die als Vermittler zwischen den Knoten und den Shards fungieren.

Ein integrierter Load Balancer verteilt eingehende Datenströme aus externen Quellen gleichmäßig auf mehrere Arbeitsknoten.Diese Knoten leiten die empfangenen Daten anschließend an die zuständige ShardRegion weiter, wo sie verarbeitet und persistiert werden.

Um die Kommunikation zwischen den verteilten Komponenten sicherzustellen, wird zudem ein verteiltes Publish/Subscribe-System eingesetzt.Dieses ermöglicht es den verschiedenen Rollen, Nachrichten asynchron auszutauschen und auf Ereignisse zu reagieren, ohne dass direkte Verbindungen erforderlich sind.

3.3 Rollen im Cluster

Dieses Kapitel beschreibt die drei Hauptrollen innerhalb des Clusters: Ingress, Backend und Coordinator. Jede dieser Rollen erfüllt eine klar abgegrenzte Funktionalität und trägt zur Gesamtfunktionalität des Systems bei.

3.3.1 Ingress

Der Ingress-Knoten ist für die Datenbeschaffung und -weiterleitung an die ShardRegion zuständig. Die Kommunikation erfolgt über Proxies, die als Vermittler zwischen den eingehenden Datenströmen und den zuständigen Shards fungieren.

Zur Steuerung des Datenflusses wird Akka.Streams verwendet, wodurch eingehende Datenströme effizient verarbeitet werden können. Bei hohen Datenmengen sorgen integrierte Backpressure-Mechanismen dafür, dass keine Überlastung der Arbeitsknoten entsteht. Um Datenverluste zu vermeiden, sendet der Ingress-Knoten neue Daten erst dann an die Arbeitsknoten weiter, wenn eine Bestätigung der ShardRegion vorliegt. Dies gewährleistet eine kontrollierte und zuverlässige Datenaufnahme innerhalb des Clusters.

3.3.2 Backend

Der Backend-Knoten ist für die eigentliche Verarbeitung und Speicherung der eingehenden Daten verantwortlich. Er empfängt die Daten vom Ingress-Knoten und speichert sie in einer Datenstruktur, die als Modell für die Formel-1-Daten dient. Nach erfolgreicher Verarbeitung und Speicherung werden die aufbereiteten Daten über einen sogenannten Händler-Knoten an die entsprechenden Konsumenten weitergeleitet.

Zur Sicherstellung der Datenpersistenz werden die Informationen entweder in einer In-Memory-Datenbank oder einer PostgreSQL-Datenbank abgelegt.Um Datenverluste bei Ausfällen oder Neuzuweisungen von Backends zu verhindern, kommt die Akka.Persistence-Funktionalität zum Einsatz. Dabei werden regelmäßig sogenannte Snapshots erstellt, anstatt jede einzelne Änderung sofort zu speichern. Dies reduziert die Datenbanklast und gewährleistet gleichzeitig eine schnelle Wiederherstellung.

Nach einem Neustart oder einem Failover kann der aktuelle Zustand durch das Wiederherstellen der gespeicherten Snapshots und das erneute Abspielen der nachfolgenden

Events rekonstruiert werden.

3.3.3 Coordinator

Der Coordinator-Knoten übernimmt die Verwaltung und Überwachung des Clusters.Er stellt sicher, dass die verschiedenen Komponenten reibungslos zusammenarbeiten und auf neu beitretende oder ausgefallene Knoten korrekt reagieren.

Eine zentrale Aufgabe des Coordinators besteht darin, den Zustand der ShardRegion zu überwachen. Falls keine funktionsfähige Region verfügbar ist, beispielsweise nach einem Node-Ausfall oder während einer Rebalancing-Phase, verhindert der Coordinator, dass der Ingress-Knoten weiterhin Daten an die ShardRegion sendet. Er fungiert somit als Kontrollinstanz, die sicherstellt, dass Daten nur dann in den Cluster eingespeist werden, wenn eine stabile und empfangsbereite Verarbeitungseinheit vorhanden ist.

Darüber hinaus übernimmt der Coordinator allgemeine Verwaltungs- und Überwachungsaufgaben, wie das Erfassen von Clusterstatusmeldungen oder das Weiterleiten relevanter Ereignisse an andere Komponenten.

3.4 Cluster Sharding

Cluster Sharding ist ein Mechanismus, der es Cluster-Akteuren ermöglicht, über ihre logischen IDs zu kommunizieren, ohne die physischen Standorte der Zielakteure kennen zu müssen (Project, 2025b). Es handelt sich um ein Werkzeug, das die verteilte Verwaltung zustandsbehafteter Akteure über mehrere Knoten innerhalb eines Clusters ermöglicht (Petabridge, 2025a).

Cluster Sharding weist den in Abbildung 3.2 dargestellten strukturellen Aufbau auf.

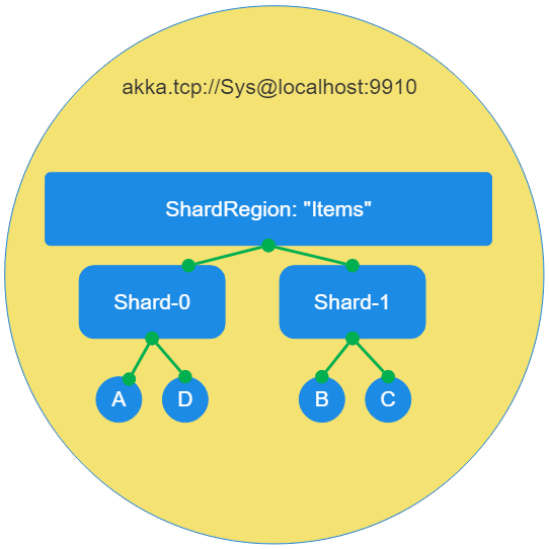

Abbildung 3.2: Cluster-Sharding-Architektur (Petabridge, 2025a)

- **ShardRegion:** Die ShardRegion ist eine Instanz, die als zentraler Einstiegspunkt für alle Nachrichten an die im Cluster verteilten Entitäten dient. Sie übernimmt das Routing eingehender Nachrichten, indem sie anhand der Entity-ID ermittelt, welche Shard und welcher Knoten für die Verarbeitung zuständig ist. Neben der reinen Nachrichtenverteilung steuert die ShardRegion auch den Lebenszyklus der Entitäten. Sie erstellt Entitäten bei Bedarf, kann sie passivieren, wenn sie längere Zeit nicht genutzt werden, und organisiert ihre Neuzuweisung, falls sich die Struktur des Clusters ändert (Petabridge, 2025a).
- **Shard:** Die Shard stellt die eigentliche Verteilungseinheit im Cluster dar. Sie definiert, auf welchem Knoten sich eine Gruppe von Entitäten befindet und übernimmt die Verwaltung dieser Entitäten. Intern agiert sie als übergeordneter Akteur und organisiert jede Entität gemäß dem Child-per-Entity-Muster (Petabridge, 2025a).

  Das Child-per-Entity-Muster beschreibt ein Kompositionsprinzip, bei dem ein Elternaktor jede Entität als eigenen Kindaktor repräsentiert. Der Elternaktor verwaltet dabei eine Zuordnung zwischen Domänenobjekten und den korrespondierenden Kindaktoren und kann diese bei Bedarf erstellen, wiederverwenden oder terminieren. Dieses Muster wird in Akka.NET sowohl im Cluster Sharding als auch in anderen verteilten Komponenten eingesetzt (Skotzko, 2015).
- **Entity:** Die Entität ist die kleinste Verarbeitungseinheit im Cluster Sharding. Sie repräsentiert einen konkreten, zustandsbehafteten Akteur, der über eine eindeutige ID identifiziert wird und Nachrichten eigenständig empfangen und verarbeiten kann (Petabridge, 2025a).

Das Ziel des Cluster Shardings besteht darin, sicherzustellen, dass im gesamten Cluster stets genau eine Instanz jeder Entität existiert. Darüber hinaus wird eine gleichmäßige Verteilung der Entitäten über alle verfügbaren Knoten angestrebt, um eine optimale Ressourcenauslastung und Lastverteilung zu gewährleisten. Dies wird durch den Einsatz von Hashing-Algorithmen erreicht, die bestimmen, auf welchem Knoten eine bestimmte Entität platziert wird. Durch diese gleichmäßige Verteilung wird verhindert, dass einzelne Knoten überlastet werden, was zu einer insgesamt besseren und gleichmäßigeren Auslastung der verfügbaren Ressourcen.

Die ShardRegion ist zudem für das Rebalancing verantwortlich, also die Umverteilung von Shards, wenn Knoten dem Cluster beitreten oder diesen verlassen. Sie koordiniert außerdem den gesamten Lebenszyklus der Entitäten, indem sie diese bei Bedarf erstellt, passiviert oder entfernt (Petabridge, 2025a).

3.4.1 Neuausgleich der Shards

Ein zentraler Aspekt des Cluster-Shardings ist der Neuausgleich (Rebalancing) der Shards. Dieser Prozess wird vom Shard-Koordinator gesteuert, der alle beteiligten Akteure im Cluster darüber informiert, dass die Übergabe einer Shard begonnen hat. Während der Migration werden eingehende Nachrichten für die betroffene Shard zunächst zwischengespeichert. Sobald die Übertragung abgeschlossen ist, werden die bisherigen Shard-Instanzen beendet und die gespeicherten Nachrichten an die neuen Shard-

Instanzen weitergeleitet.

Während dieses Vorgangs wird der Zustand der Entitäten nicht automatisch migriert. Um Datenverlust zu vermeiden, sollten Entitäten daher so implementiert sein, dass ihr Zustand persistent gespeichert und bei Bedarf wiederhergestellt werden kann (Petabridge, 2025a).

Die Logik des Rebalancing kann bei Bedarf durch eine benutzerdefinierte Zuweisungsstrategie angepasst werden. Standardmäßig verwendet Akka.NET eine Hash-basierte Verteilung, um eine gleichmäßige Auslastung der verfügbaren Knoten sicherzustellen. (Project, 2025b)

3.4.2 ShardRegion-Proxy

Ein ShardRegion-Proxy wird benötigt, wenn eine Kommunikation mit einer ShardRegion stattfindet, die sich nicht auf demselben Knoten befindet (Project, 2025b).

In der vorliegenden Architektur befindet sich die ShardRegion ausschließlich auf Knoten mit der Rolle *Backend*. Daher muss ein Proxy verwendet werden, wenn Komponenten außerhalb dieser Rolle Nachrichten an die ShardRegion senden sollen.

3.5 Singleton

Der Cluster-Singleton gewährleistet, dass eine bestimmte Aktor-Instanz innerhalb des gesamten Clusters exakt einmal existiert. Dieses Konzept wird eingesetzt, wenn Aufgaben zentral ausgeführt werden müssen und eine Mehrfachausführung zu Inkonsistenzen oder erhöhtem Synchronisationsaufwand führen würde. Typische Einsatzbereiche umfassen globale Verwaltungsfunktionen, zentrale Namensauflösung oder Routinglogik.

Ein Vorteil dieser Architektur ist, dass wichtige Entscheidungen und Aufgaben an einer Stelle gebündelt werden. Dadurch entsteht keine Verwirrung darüber, welcher Aktor zuständig ist. Der Nachteil liegt jedoch darin, dass der Singleton ein einzelner Engpass sein kann. Wenn er ausfällt oder überlastet wird, kann das Auswirkungen auf das gesamte System haben.

Die Bereitstellung der Singleton-Instanz erfolgt auf Knoten, die über eine vordefinierte Rolle verfügen. Existieren mehrere Knoten mit dieser Rolle, wird die Instanz üblicherweise auf dem Knoten mit der längsten Cluster-Mitgliedschaft ausgeführt. Im Falle eines Ausfalls übernimmt automatisch ein anderer geeigneter Knoten die Ausführung des Singletons, wodurch eine hohe Verfügbarkeit gewährleistet bleibt (Project, 2025c).

3.5.1 Singleton-Proxy

Um auf den Cluster-Singleton zugreifen zu können, wird ein Singleton-Proxy verwendet. Dieser ermöglicht es allen Komponenten innerhalb des Clusters, Nachrichten an den Singleton zu senden, ohne dessen tatsächlichen Ausführungsort kennen zu müssen. Die

Identifikation erfolgt über einen definierten Marker, sodass keine explizite Aktor- oder Netzwerkadresse notwendig ist.

Falls der Singleton vorübergehend nicht erreichbar ist, werden eingehende Nachrichten im Proxy gepuffert und nach Wiederherstellung der Verbindung automatisch weitergeleitet. Wird die maximale Puffergöße überschritten, verwirft der Proxy die ältesten Nachrichten, um eine Überlastung zu verhindern (Project, 2025c).

3.6 Kommunikation über verteiltes Publish/Subscribe

Der Einsatz verteilter Kommunikation in Akka.NET wird relevant, sobald Nachrichten nicht mehr nur zwischen Aktoren auf demselben Knoten ausgetauscht werden, sondern zwischen Aktoren eines verteilten Aktorensystems, das über mehrere Knoten hinweg betrieben wird. Dabei spielt es keine Rolle, auf welchen physischen oder virtuellen Maschinen diese Knoten ausgeführt werden. In Abbildung 3.3 ist eine solche verteilte Publish Subscribe Architektur veranschaulicht.

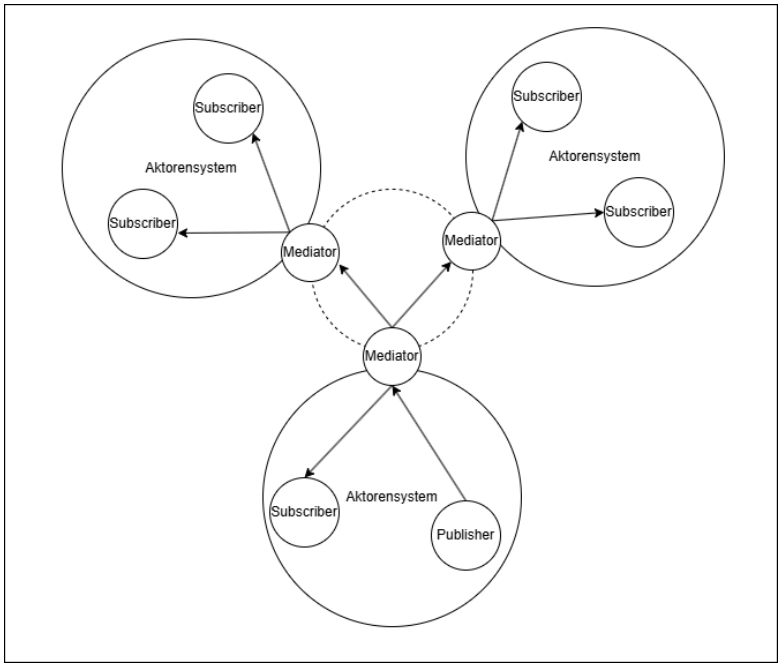

Abbildung 3.3: Verteilte Publish-Subscribe-Architektur (Petabridge, 2025b)

Für die Kommunikation registriert sich ein Aktor, der Ereignisse von anderen Aktoren empfangen möchte, bei seinem lokalen Mediator unter Angabe eines Topics oder eines Aktorenpfads. Nach der Registrierung erhält der abonnierende Aktor eine Bestätigungsmeldung, womit sichergestellt wird, dass er nun Nachrichten über das entsprechende Topic empfangen kann.

Wenn ein Aktor eine Nachricht an alle Abonnenten eines Topics senden möchte, informiert er ebenfalls seinen lokalen Mediator durch eine Publikationsnachricht und gibt

dabei das betreffende Topic oder den Zielpfad an. In Szenarien, in denen eine Nachricht nur an einen einzelnen Abonnenten gesendet werden soll, ist dies ebenfalls möglich. Standardmäßig erfolgt die Zustellung dabei zufällig an einen der Teilnehmer der Abonnentengruppe (Petabridge, 2025b).

3.7 Streams

Akka.Streams bietet ein modellbasiertes und fehlertolerantes Konzept zur Verarbeitung kontinuierlicher Datenströme innerhalb eines Aktorensystems. Dabei ermöglicht das Framework die Übertragung von Daten zwischen Produzenten und Konsumenten, ohne dass interne Pufferspeicher oder Mailboxen überlaufen können. Dies wird durch einen kontrollierten und ressourcenschonenden Datenfluss erreicht, der sicherstellt, dass nur so viele Elemente produziert werden, wie tatsächlich verarbeitet werden können.

Ein zentrales Prinzip ist dabei der integrierte Rückstau-Mechanismus (Backpressure). Dieser Mechanismus sorgt dafür, dass Produzenten automatisch verlangsamt werden, sobald nachgelagerte Verarbeitungsschritte oder Konsumenten die eingehenden Daten nicht in ausreichender Geschwindigkeit verarbeiten können. Backpressure ist ein Kernelement der Reactive-Streams-Spezifikation, auf der Akka.Streams basiert, und stellt sicher, dass das System auch unter hoher Last stabil und vorhersehbar arbeitet (Project, 2025g).

3.7.1 Overflowstrategien

Akka.Streams stellt verschiedene Strategien zur Verfügung, um Situationen zu bewältigen, in denen ein Produzent schneller Daten erzeugt, als ein Konsument diese verarbeiten kann. Sobald ein Puffer seine maximale Kapazität erreicht, legt die gewählte Overflowstrategie fest, wie der Stream auf diesen Zustand reagiert (Project, 2025h).

Backpressure: Bei aktivem Backpressure wird der Datenfluss reguliert, indem ein Rückstau-Signal an den Produzenten gesendet wird. Dieser verlangsamt seine Erzeugungsrate, bis im Puffer wieder Kapazität frei ist. Dadurch bleibt der Stream stabil, ohne Elemente zu verwerfen (Project, 2025h).

DropBuffer: Ist der Puffer voll und trifft ein neues Element ein, werden alle aktuell im Puffer enthaltenen Elemente verworfen. Anschließend wird das neu ankommende Element in den nun leeren Puffer eingefügt. Diese Strategie priorisiert stets die neuesten Daten (Project, 2025h).

DropHead: Bei einem vollen Puffer wird das Element verworfen, das sich am längsten im Puffer befindet. Auf diese Weise wird Platz für das neu eintreffende Element geschaffen, wodurch jüngere Daten bevorzugt werden (Project, 2025h).

DropNew: Ist der Puffer voll, wird das neu eintreffende Element verworfen, während der aktuelle Pufferinhalt unverändert bleibt. Diese Strategie schützt die bereits wartenden Elemente und verhindert zusätzlichen Druck auf nachgelagerte Komponenten (Project, 2025h).

DropTail: Wenn der Puffer voll ist, wird das zuletzt hinzugefügte (jüngste) Element verworfen, um Platz für ein neu eintreffendes Element zu schaffen. Im Gegensatz zu DropHead bleiben ältere Elemente erhalten (Project, 2025h).

Fail: Bei einem Pufferüberlauf wird der Stream sofort mit einem Fehlerzustand beendet. Diese Strategie wird vor allem dann eingesetzt, wenn Datenverlust nicht tolerierbar ist und ein Überlauf einen kritischen Fehler darstellt (Project, 2025h).

3.8 Alternative Architekturvarianten

Im Rahmen des Systementwurfs wurden verschiedene alternative Architekturansätze konzeptionell analysiert und hinsichtlich ihrer Eignung für die Projektanforderungen bewertet. Die Bewertung erfolgte auf Basis etablierter architekturrelevanter Kriterien wie Skalierbarkeit, Fehlertoleranz und Echtzeitfähigkeit sowie unter Berücksichtigung der spezifischen Anforderungen des Use-Cases. Im Folgenden werden exemplarisch mehrere Alternativen vorgestellt und begründet, warum diese im Kontext des Projekts nicht weiterverfolgt wurden.

Single-Node-Aktorensystem

Eine Ausführung des Systems in einem einzelnen Prozess ist grundsätzlich möglich und wird auch durch die aktuelle Implementierung unterstützt. Eine rein monolithische Single-Node-Variante ohne verteilte Komponenten würde jedoch keine horizontale Skalierung zulassen. Zudem könnte die Anwendung nicht auf mehrere Maschinen verteilt werden, was sowohl die Skalierbarkeit als auch die Fehlertoleranz erheblich einschränken würde. Für den Einsatz unter hoher Datenlast wäre dieses Modell daher ungeeignet.

Microservices-Architektur

Eine Microservices-Architektur, bei der jede Funktionalität als eigenständiger Dienst implementiert wird, wurde ebenfalls in Betracht gezogen. Obwohl ein solcher Ansatz eine hohe Modularität und unabhängige Skalierbarkeit einzelner Komponenten ermöglicht, wäre der Infrastruktur- und Betriebsaufwand deutlich höher. Aspekte wie Service-Orchestrierung, Deployment, Monitoring und die Absicherung der Kommunikation zwischen Services würden die Komplexität des Gesamtsystems stark erhöhen, ohne einen entscheidenden Vorteil für die Verarbeitung der zeitkritischen Formel-1-Daten zu bieten.

Zentralisierte Datenbank

Auch der Ansatz, alle eingehenden Formel-1-Daten ausschließlich über eine zentralisierte Datenbank zu verwalten, wurde evaluiert. Obwohl eine zentrale Datenhaltung die Konsistenz der Daten vereinfachen könnte, entsteht dadurch ein Single Point of Failure. Zudem skaliert eine zentrale Datenbank nur begrenzt mit steigenden Datenmengen, und die damit verbundene Latenz könnte die Anforderungen an eine Echtzeitverarbeitung nicht erfüllen. Daher wurde dieser Ansatz als ungeeignet eingestuft.

Nachrichtenwarteschlangen

Der Einsatz von Nachrichtenwarteschlangen wie RabbitMQ oder Apache Kafka zur Entkopplung der Systemkomponenten stellt eine weitere mögliche Architekturvariante dar. Diese Technologien bieten eine zuverlässige und verteilte Nachrichtenverarbeitung. Allerdings würde die Integration eines externen Message Brokers einen zusätzlichen Infrastruktur-Layer einführen und damit sowohl die Komplexität als auch die Latenz erhöhen. Zudem würde die enge Kopplung an den zustandsbehafteten Aktor-Ansatz verloren gehen, weshalb dieser Architekturansatz verworfen wurde.

Bewertung

Zusammenfassend bietet die gewählte Architektur auf Basis von Akka.NET und Cluster Sharding die beste Balance zwischen Skalierbarkeit, Fehlertoleranz und Echtzeitverarbeitung. Sie unterstützt sowohl eine dynamische Verteilung der Rechenlast als auch eine robuste Verwaltung zustandsbehafteter Entitäten und erfüllt damit die Anforderungen des Projekts in optimaler Weise.

Kapitel 4

Implementierung

Dieses Kapitel beschreibt die praktische Umsetzung der in den vorherigen Kapiteln konzipierten Architektur. Zunächst wird die Projektstruktur vorgestellt und die Aufteilung in Module sowie Rollen innerhalb des Systems erläutert. Anschließend wird die Konfiguration und Initialisierung des Akka.NET-Clusters beschrieben, einschließlich der verwendeten HOCON- und Akka.Hosting-Ansätze. Darauf aufbauend werden die Implementierung des Eingangs-Services, die Datenverarbeitung mit Akka.Streams sowie die zustandsbehaftete Verarbeitung innerhalb der ShardRegion dargestellt. Abschließend werden die Mechanismen zur Nachrichtenverteilung, zur Cluster-Steuerung und Fehlertoleranz sowie die Client- und Visualisierungsschicht erläutert, bevor das Zusammenspiel aller Komponenten zusammengeführt wird.

4.1 Projektstruktur

Die Projektstruktur ist so gewählt, dass der zentrale Anwendungsteil den vollständigen Code für das Kernmodul enthält. Innerhalb dieses Moduls werden die verschiedenen Rollen des Systems klar voneinander getrennt umgesetzt. Komponenten, die nicht direkt zum Kernfunktionalität gehören, wie Infrastrukturkomponenten oder Tests, sind in eigenständige Projekte ausgelagert. Dadurch wird die Wartbarkeit verbessert und eine klare Trennung der Verantwortlichkeiten erreicht. Die Gesamtstruktur des Projekts ist in Abbildung 4.1 dargestellt.

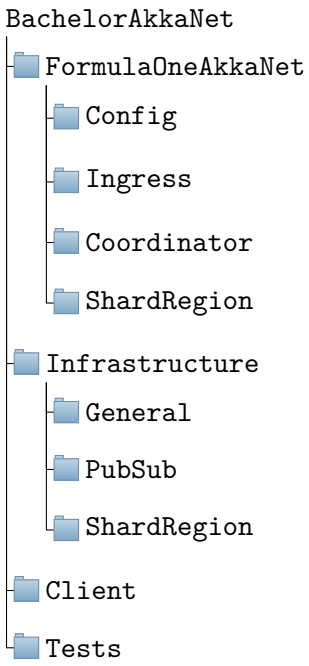

Abbildung 4.1: Gesamtstruktur des Projekts

Die modulare Struktur des Projekts stellt sicher, dass die implementierten Rollen des Systems klar voneinander abgegrenzt sind. Jede Modulkomponente enthält ausschließlich die für ihre Rolle relevanten Aktorimplementierungen und Logik. Die einzige gemeinsam genutzte Komponente ist die zentrale Konfiguration, die im Ordner *Config* abgelegt ist. Sie stellt sicher, dass alle Module konsistent in das .NET-Hostingmodell eingebunden werden können.

Das Projekt *Infrastructure* umfasst wiederverwendbare Funktionen und Hilfskomponenten, die sowohl von den Hauptmodulen als auch von den Systemtests und dem Client genutzt werden. Im *Client*-Projekt erfolgt die Darstellung und Auswertung der erfassten Daten, wodurch eine externe Visualisierung des Systemzustands möglich wird.

Für die Implementierung wurden zentrale Bibliotheken aus dem Akka.NET-Ökosystem verwendet. Dazu gehören Pakete für Clusterbildung, Sharding, Persistenz und Stream-Verarbeitung. Zusätzlich kommen Hosting-Erweiterungen zum Einsatz, welche die Konfiguration der Cluster- und Persistenzkomponenten über das .NET-Hostingmodell ermöglichen. Ergänzend werden Npgsql für den Datenbankzugriff und Serilog für das Logging eingesetzt. Tabelle 4.1 gibt einen Überblick über die relevanten Kernpakete.

Paket	Version
Akka	1.5.54
Akka.Cluster	1.5.54
Akka.Cluster.Sharding	1.5.54
Akka.Hosting	1.5.53
Akka.Cluster.Hosting	1.5.53
Akka.Persistence	1.5.54
Akka.Persistence.Sql	1.5.53
Akka.Persistence.Hosting	1.5.53
Akka.Persistence.Sql.Hosting	1.5.53
Akka.Streams	1.5.54
Akka.Serialization.Hyperion	1.5.54
Npgsql	9.0.4
Serilog	4.3.0

Tabelle 4.1: Verwendete Kernbibliotheken und Versionen

Die ausgewählten Pakete decken die zentralen technischen Anforderungen des Systems ab. Akka.Cluster und Akka.Cluster.Sharding ermöglichen eine verteilte und fehlertolerante Verarbeitung der eingehenden Daten. Akka.Persistence und die zugehörigen SQL-Erweiterungen stellen sicher, dass zustandsbehaftete Entitäten konsistent wiederhergestellt werden können. Akka.Streams wird für die kontrollierte Verarbeitung der Datenströme eingesetzt. Mit den Hosting-Erweiterungen wird eine einheitliche Konfiguration über das .NET-Hostingmodell ermöglicht, wodurch die Initialisierung des Clusters sowie der Persistence-Komponenten vereinfacht wird. Serilog und Npgsql ergänzen die Architektur durch Logging und Datenbankanbindung.

4.2 Konfiguration und Initialisierung des Clusters

In diesem Abschnitt wird die Konfiguration und Initialisierung des Akka.NET-Clusters beschrieben. Dabei werden zunächst die verwendeten Konfigurationsansätze vorgestellt, insbesondere die klassische HOCON-basierte Konfiguration sowie die programmgesteuerte Einrichtung über Akka.Hosting. Darauf aufbauend werden die Initialisierung der verschiedenen Rollen im Cluster sowie die Konfiguration von Remoting-, Clustering- und Persistenzkomponenten erläutert.

4.2.1 HOCON-Konfiguration

Die Konfiguration eines Akka.NET-Systems kann auf zwei unterschiedliche Arten erfolgen. Zum einen kann die HOCON-Konfiguration aus einer externen Datei eingelesen werden, die den vollständigen Konfigurationstext enthält. Zum anderen ermöglicht Akka.Hosting die Erstellung der Konfiguration direkt über Erweiterungsmethoden innerhalb des .NET-Hostingmodells.

Beide Ansätze wurden im Rahmen dieses Projekts eingesetzt. Für Konsolenanwendungen ist das Laden einer separaten HOCON-Datei zweckmäßig, da der Overhead des Hostingmodells dort nicht erforderlich ist. In Anwendungen, die auf dem .NET-Hostingmodell basieren, ist die programmgesteuerte Konfiguration über Akka.Hosting dagegen besonders vorteilhaft, da sie eine engere Integration in den Lebenszyklus des Hosts und eine klar strukturierte Initialisierung des Clusters ermöglicht.

HOCON-Konfiguration

In klassischen Konsolenanwendungen ist es üblich, die Akka.NET-Konfiguration in einer separaten HOCON-Datei zu hinterlegen und beim Start der Anwendung einzulesen.

Programm 4.1: Auszug aus der HOCON-Konfiguration des Clients

```
akka {
    loglevel = "INFO"
    stdout-loglevel = "OFF"
    loggers = ["Akka.Logger.Serilog.SerilogLogger, Akka.Logger.Serilog"]

    actor {
      provider = cluster
      serializers {
        hyperion = "Akka.Serialization.HyperionSerializer,
                    Akka.Serialization.Hyperion"
      }
      serialization-bindings {
        "System.Object" = hyperion
      }
    }

    remote.dot-netty.tcp {
        hostname = "localhost"
        port = 0
        maximum-frame-size  = 2MiB
        send-buffer-size    = 2MiB
        receive-buffer-size = 2MiB

    }

    cluster {
      roles = ["api"]
      seed-nodes = [
        "akka.tcp://cluster-system@localhost:5000"
      ]
    }
}
```

Wie in Programm 4.1 dargestellt, folgt die Konfigurationsdatei der HOCON-Syntax, die strukturell an JSON erinnert, jedoch flexibler und für Akka.NET optimiert ist.

Jede Akka-Konfiguration beginnt mit dem Hauptelement `akka`, unter dem die zentralen Systemparameter definiert werden.

Die initialen Einträge betreffen die Konfiguration des Loggings. Anschließend wird über `actor.provider = cluster` festgelegt, dass dieser Prozess Teil eines verteilten Clusters ist. Die Serialisierung der Nachrichten wird mithilfe des Hyperion-Serialisierers konfiguriert, der eine effiziente binäre Datenrepräsentation ermöglicht und sich dadurch besonders für Clusterkommunikation eignet.

Der Block `remote.dot-netty.tcp` enthält die Netzwerkkonfiguration. Der Port ist auf `0` gesetzt, wodurch zur Laufzeit ein freier Port gewählt wird. Weitere Parameter wie `maximum-frame-size` oder die Puffergrößen steuern die maximale Nachrichtengröße und die Netzwerkeffizienz der Punkt-zu-Punkt-Kommunikation.

Im Abschnitt `cluster` werden die Rollen des Knotens sowie die *seed nodes* festgelegt. Seed-Knoten dienen als initiale Kontaktpunkte beim Beitritt eines neuen Knotens zum Cluster. Nach der ersten Verbindung erfolgt die weitere Clusterentdeckung über das Gossip-Protokoll. Die Rollendefinition `api` legt fest, welche Aufgaben und Aktoren diesem Knoten im Clusterverbund zugeordnet sind.

Akka.Hosting

Anstatt die Konfiguration ausschließlich über externe HOCON-Dateien zu definieren, die beim manuellen Bearbeiten anfällig für Fehler sein können, bietet Akka.Hosting die Möglichkeit, die Akka.NET-Konfiguration direkt im Anwendungscode aufzubauen. Dadurch lässt sich die Konfiguration eng in die Dependency-Injection-Pipeline des .NET-Hostingmodells integrieren und gemeinsam mit anderen Diensten der Anwendung verwalten.

In diesem Projekt kann die Anwendung wahlweise als monolithischer Prozess oder als verteiltes System mit mehreren Clusterknoten ausgeführt werden. Die Konfiguration wurde daher so gestaltet, dass auf Basis der zur Laufzeit vergebenen Rollen die jeweils benötigten Komponenten initialisiert werden. Ein entsprechender Ausschnitt ist in Programm 4.2 dargestellt. Die Rollen werden über Parameter der Konsolenanwendung festgelegt und anschließend in der Klasse `AkkaConfig` ausgewertet.

Um die Konfiguration übersichtlich und wartbar zu halten, wurden zentrale Initialisierungsschritte in Erweiterungsmethoden gekapselt. Diese Methoden übernehmen unter anderem die Einrichtung der Clusteroptionen, der Remote-Kommunikation sowie der Serialisierungs- und Loggingkonfiguration und binden die jeweiligen Rollen wie Ingress, Backend oder Coordinator in das Hostingmodell ein.

Programm 4.2: Auszug aus der Akka.Hosting-Konfiguration

```
1 public static IServiceCollection UseAkka(this IServiceCollection sp, AkkaConfig
      akkaHc, ConfigurationManager manager)
2 {
3     string connectionString = manager.GetConnectionString("PostgreSql")!;
4
5     sp.AddAkka(akkaHc.ClusterName, akka =>
6     {
7
8         akka.UseAkkaLogger();
9         akka.UseRemoteCluster(akkaHc);
10        akka.UseHyperion();
11
12        if (akkaHc.Roles.Contains(ClusterMemberRoles.Controller.ToStr()))
13            akka.RegisterCoordinator(akkaHc);
14
15        if (akkaHc.Roles.Contains(ClusterMemberRoles.Backend.ToStr()))
16        {
17            akka.RegisterShardRegion(akkaHc);
18            akka.RegisterBackendJournal(connectionString);
19        }
20
21
22        if (akkaHc.Roles.Contains(ClusterMemberRoles.Ingress.ToStr()))
23            akka.RegisterIngress(akkaHc);
24
25        if (!akkaHc.Roles.Contains(ClusterMemberRoles.Controller.ToStr()))
26            akka.RegisterControllerProxy();
27    });
28    return sp;
29 }
```

Akka.Hosting Koordinator-Initialisierung

Um exemplarisch zu zeigen, wie die Initialisierung der verschiedenen Rollen erfolgt, ist in Programm 4.3 ein Auszug der Konfiguration für die *Coordinator*-Rolle dargestellt.

Zunächst wird ein Singleton-Aktor des Typs `ClusterCoordinator` registriert, der zentrale Steuerungsaufgaben innerhalb des Clusters übernimmt. Damit andere Komponenten diesen Aktor über die Dependency-Injection-Pipeline referenzieren können, wird ein sogenannter *Marker* (`ClusterCoordinatorMarker`) verwendet. Ein solcher Marker dient als eindeutiger Schlüssel, über den sich ein bestimmter Aktor später typsicher abrufen lässt.

Im Anschluss werden weitere Aktoren registriert, die für das Monitoring von Clusterereignissen sowie für die Verarbeitung von Ingress- und Shard-bezogenen Ereignissen zuständig sind. Auf diese Weise wird die Beobachtung und Verwaltung relevanter Clusterzustände dezentral über spezialisierte Listener-Aktoren umgesetzt.

Die Methode `resolver.Props<T>()` ist eine Erweiterung von Akka.Hosting und er-

möglicht die Erstellung von Aktoren mit Hilfe von Dependency Injection. Mit `system.ActorOf(...)` wird der jeweilige Aktor schließlich instanziert und dem Aktorensystem hinzugefügt.

Über `registry.Register<T>()` wird der erzeugte Aktor anschließend im internen Registry-System von Akka.Hosting hinterlegt, damit er später von anderen Diensten oder Modulen abgerufen werden kann.

Programm 4.3: Auszug aus der Akka.Hosting-Koordinator-Konfiguration

```
1 private static void RegisterCoordinator(this AkkaConfigurationBuilder config,
      AkkaConfig akkaHc)
2 {
3     string role = ClusterMemberRoles.Controller.ToStr();
4
5     config.WithSingleton<ClusterCoordinatorMarker>(
6             singletonName: role,
7             propsFactory: (_, _, resolver) => resolver.Props<ClusterCoordinator>(),
8             options: new ClusterSingletonOptions { Role = akkaHc.Role })
9         .WithActors((system, registry, resolver) =>
10        {
11            var controller = registry.Get<ClusterCoordinatorMarker>();
12
13            registry.Register<ClusterEventListener>(
14                system.ActorOf(resolver.Props<ClusterEventListener>(),
15                    "cluster-event-listener"));
16
17            registry.Register<IngressListener>(
18                system.ActorOf(resolver.Props<IngressListener>(controller),
19                    "ingress-listener"));
20
21            registry.Register<ShardListener>(
22                system.ActorOf(resolver.Props<ShardListener>(controller),
23                    "shard-listener"));
24        });
25 }
```

Akka.Hosting Backend-Initialisierung

Die *Backend*-Rolle stellt das Herzstück der verteilten Datenverarbeitung dar, da sie die `ShardRegion` hostet, in der die einzelnen Fahrerdaten als zustandsbehaftete Entitäten repräsentiert und verarbeitet werden. Programm 4.4 zeigt einen ausgewählten Ausschnitt der Konfiguration dieser Rolle.

Für die ShardRegion wird die *DistributedData*-Variante des Shardings verwendet. Hierbei werden die internen Sharding-Metadaten, insbesondere die Zuordnung von Entitäten zu Shards sowie der aktuelle Aktivierungszustand der Entitäten, über das Modul `Akka.DistributedData` repliziert. Dies erhöht die Ausfallsicherheit, da bei einem Knotenverlust die übrigen Clusterknoten weiterhin einen konsistenten Überblick über die Shard-Verteilung behalten. Der eigentliche fachliche Zustand der Fahrerdaten wird hin-

gegen durch die persistenten Entitätsaktoren (`DriverActorPersistent`) verwaltet und über Akka.Persistence im Journal gespeichert.

Zu Beginn werden die Optionen für das verteilte Sharding konfiguriert. Parameter wie `MajorityMinimumCapacity` legen fest, wie viele Clusterknoten für eine Mehrheitsentscheidung innerhalb von DistributedData erforderlich sind, während `MaxDeltaElements` bestimmt, wie umfangreich die über Gossip propagierten inkrementellen Zustandsänderungen (Delta-States) sein dürfen. Diese Einstellungen beeinflussen das Verhalten der Replikation insbesondere bei wachsender Anzahl von Entitäten.

Im Anschluss wird die eigentliche ShardRegion registriert. Ein zugehöriger *Marker* (`DriverRegionMarker`) ermöglicht den späteren Zugriff über die Dependency-Injection-Pipeline. Die Entitäten innerhalb der ShardRegion werden als persistent implementierte Aktoren des Typs `DriverActorPersistent` ausgeführt. Ein `MessageExtractor` (`DriverMessageExtractor`) ordnet eingehende Nachrichten sowohl den Shards als auch den einzelnen Entitäten zu und übernimmt damit eine zentrale Rolle in der Lastverteilung.

Zusätzlich werden Optionen wie die automatische Passivierung inaktiver Entitäten sowie die Nutzung des DistributedData-Modus als Speichermechanismus für Sharding-Metadaten konfiguriert. Mit `RememberEntities = true` wird sichergestellt, dass zuvor bekannte Entitäten nach einem Neustart oder Rebalancing automatisch reaktiviert werden.

Programm 4.4: Auszug aus der Akka.Hosting-Backend-Konfiguration

```
1 private static void RegisterShardRegion(this AkkaConfigurationBuilder config,
      AkkaConfig akkaHc, IMessageExtractor? ex = null)
2 {
3     string role = ClusterMemberRoles.Backend.ToStr();
4
5     var extractor = ex ?? new DriverMessageExtractor();
6
7     config.WithShardingDistributedData(options =>
8         {
9             options.RecreateOnFailure = true;
10            options.MajorityMinimumCapacity = 2;
11            options.MaxDeltaElements = 2000;
12            options.Durable.Keys = [];
13        })
14        .WithShardRegion<DriverRegionMarker>(
15            typeName: akkaHc.ShardName,
16            entityPropsFactory: (_, _, resolver) =>
17                                _ => resolver.Props<DriverActorPersistent>(),
18            messageExtractor: extractor,
19            shardOptions: new ShardOptions
20            {
21                Role = role,
22                PassivateIdleEntityAfter = TimeSpan.FromMinutes(5),
23                StateStoreMode = StateStoreMode.DData,
24                RememberEntities = true
25            })
26        .WithActors((system, registry, resolver) =>
27            registry.Register<TelemetryRegionHandler>(
28                system.ActorOf(resolver.Props<TelemetryRegionHandler>(),
29                    "telemetry-region-handler")));
30 }
```

Akka.Hosting Ingress-Initialisierung

Die *Ingress*-Rolle ist für den Empfang und die Vorverarbeitung der eingehenden Fahrerdaten zuständig. Programm 4.5 zeigt einen Ausschnitt der Konfiguration dieser Rolle.

Zunächst wird ein Singleton-Aktor des Typs `IngressHttpActor` registriert, der als Einstiegspunkt für die eingehenden Telemetriedaten dient. Die Daten werden von einem vorgelagerten externen Dienst bereitgestellt, der die Rohdaten aus der OpenF1-Schnittstelle bezieht und als kontinuierlichen Datenstrom zur Verfügung stellt. Der `IngressHttpActor` übernimmt den Empfang dieses Datenstroms und leitet die empfangenen Nachrichten an die nachgelagerte Stream-Pipeline zur weiteren Verarbeitung im Aktorensystem weiter.

Anschließend wird ein ShardRegion-Proxy für die Kommunikation mit der `ShardRegion` erstellt. Hierzu wird ein Marker (`DriverRegionProxyMarker`) verwendet, über den der Proxy später über die Dependency-Injection-Pipeline referenziert werden kann. Der

Proxy ermöglicht es den Ingress-Komponenten, Nachrichten an die tatsächliche Shard-Region zu senden, ohne direkt an deren physische Instanz gebunden zu sein. Dies fördert eine lose Kopplung zwischen Ingress und Backend.

Zusätzlich wird ein `IngressControllerActor` registriert, der als zentrale Steuerungskomponente für die Ingress-Logik dient und die weitere Verarbeitung der eingehenden Daten innerhalb des Aktorensystems koordiniert.

Programm 4.5: Auszug aus der Akka.Hosting-Ingress-Konfiguration

```
1 private static void RegisterIngress(this AkkaConfigurationBuilder config, AkkaConfig
      akkaHc)
2 {
3     string role = ClusterMemberRoles.Ingress.ToStr();
4     config
5         .WithSingleton<HttpWrapperClientMarker>(
6             singletonName: role,
7             propsFactory: (_, _, resolver) => resolver.Props<IngressHttpActor>(),
8             options: new ClusterSingletonOptions {Role = akkaHc.Role})
9         .WithShardRegionProxy<DriverRegionProxyMarker>(
10            typeName: akkaHc.ShardName,
11            roleName: ClusterMemberRoles.Backend.ToStr(),
12            messageExtractor: new DriverMessageExtractor())
13        .WithActors((system, registry, resolver) =>
14        {
15            registry.Register<IngressControllerActor>(
16                system.ActorOf(resolver.Props<IngressControllerActor>(),
17                    "controller-handler"));
18        });
19 }
```

Akka.Hosting Generelle Remote und Cluster Konfiguration

Damit ein Aktorensystem als Clusterknoten mit anderen Systemen kommunizieren kann, müssen Remoting und Clustering konfiguriert werden. Programm 4.6 zeigt eine eigene Erweiterungsmethode, die diese Konfiguration bündelt und von allen Rollen gemeinsam genutzt wird.

Zunächst werden Hostname und Port für das Remote-Modul gesetzt, sodass der Knoten über TCP erreichbar ist. Anschließend werden die Clusteroptionen konfiguriert. Dazu gehören die Rollen des Knotens sowie die Seed Nodes, über die der Cluster initial aufgebaut wird. Abhängig vom gewählten Startmodus wird entweder eine vollständige Seed-Liste oder nur ein initialer Seed-Knoten verwendet.

Für die spätere Nachrichtenverteilung zwischen unabhängigen Aktoren wird außerdem der *DistributedPubSub*-Mediator aktiviert. Dieser ermöglicht eine publish/subscribe-basierte Kommunikation, ohne dass Sender und Empfänger direkt voneinander wissen müssen. Dies fördert eine lose Kopplung innerhalb des Systems.

Abschließend werden Parameter wie die maximale Nachrichtengröße und die Puffergrö-

ßen für die TCP-Kommunikation definiert. Diese Einstellungen sind insbesondere bei der Übertragung größerer Datenmengen relevant und tragen zur Stabilität und Effizienz der Clusterkommunikation bei.

Programm 4.6: Auszug aus der Akka.Hosting-Remote-Cluster-Konfiguration

```
public static AkkaConfigurationBuilder UseRemoteCluster(this
    AkkaConfigurationBuilder builder, AkkaConfig config)
{
    builder
        .WithRemoting(
            port: config.Port,
            hostname: config.Hostname)
        .WithClustering(
            new ClusterOptions
            {
                SeedNodes = config.Roles.Count == 1
                            ? config.SeedNodes
                            : [config.SeedNodes.First()],
                Roles = config.Roles.ToArray()
            })
        .WithDistributedPubSub(role: null!)
        .AddHocon("""
                        akka.remote.dot-netty.tcp {
                          maximum-frame-size  = 2 MiB
                          send-buffer-size    = 2 MiB
                          receive-buffer-size = 2 MiB
                        }
                        """, HoconAddMode.Append);

    return builder;
}
```

Akka.Hosting Persistenzkonfiguration

Die Persistenz ist ein zentraler Bestandteil der Backend-Rolle, da der Zustand der Fahrerdaten dauerhaft gesichert werden muss. Programm 4.7 zeigt die Einbindung der Persistence-Komponenten in das Aktorensystem mithilfe von Akka.Hosting.

Die Implementierung unterstützt zwei Betriebsmodi. Zum einen wird eine In-Memory-Persistenz für Testläufe verwendet, die ein schnelles und reproduzierbares Verhalten ohne externe Abhängigkeiten ermöglicht. Zum anderen kommt im produktiven Betrieb eine PostgreSQL-basierte Persistenz zum Einsatz. Dadurch kann der Zustand der Fahrerdaten auch über Neustarts hinweg konsistent wiederhergestellt werden.

Für den Datenbankzugriff wird Npgsql eingesetzt. Über `WithSqlPersistence` wird Akka.Persistence.Sql in das System integriert und automatisch mit den erforderlichen Journal- und Snapshot-Schemata initialisiert. Die In-Memory-Variante wird aktiviert, wenn kein Connection-String für die Datenbank konfiguriert wurde (leerer Connection String), wodurch die Backend-Rolle flexibel in unterschiedlichen Umgebungen betrieben

werden kann.

Programm 4.7: Auszug aus der Akka.Hosting-Journal-Konfiguration

```
1 private static void RegisterBackendJournal(this AkkaConfigurationBuilder config,
      string connectionString)
2 {
3     string dbName = "driverRegion";
4
5     if (connectionString is "")
6     {
7         config.WithInMemoryJournal(journalId: dbName, journalBuilder: _ => { });
8         config.WithInMemorySnapshotStore(dbName);
9         return;
10    }
11
12    var dataSource = new NpgsqlDataSourceBuilder(connectionString).Build();
13
14    var dataOptions = new DataOptions()
15        .UseDataProvider(
16            DataConnection.GetDataProvider(
17                ProviderName.PostgreSQL,
18                dataSource.ConnectionString)
19                ?? throw new Exception("Could not get data provider")
20            )
21        .UseProvider(ProviderName.PostgreSQL)
22        .UseConnectionFactory(_ => dataSource.CreateConnection());
23
24    config.WithSqlPersistence(
25                    dataOptions,
26                    autoInitialize: true,
27                    schemaName: "public");
28
29    config.AddStartup((sys, ct) =>
30    {
31        var c = sys.Settings.Config;
32        sys.Log.Info("Journal={0}, Snapshot={1}",
33            c.GetString("akka.persistence.journal.plugin"),
34            c.GetString("akka.persistence.snapshot-store.plugin"));
35
36        sys.RegisterOnTermination(() => dataSource.Dispose());
37        return Task.CompletedTask;
38    });
39 }
```

4.3 Eingangs-Service

Der Eingangs-Service bildet den Einstiegspunkt in die Datenverarbeitung und empfängt Telemetriedaten über eine dauerhafte Verbindung zu einem externen Datenprovider. Die Bereitstellung der Telemetriedaten erfolgt dabei über einen vorgelagerten Dienst, der die Rohdaten aus einer externen Quelle (z. B. OpenF1) bezieht und als kontinuierlichen

Datenstrom bereitstellt. Dieser vorgelagerte Dienst ist nicht Bestandteil dieser Arbeit, er dient ausschließlich als Datenquelle für die Evaluation der entwickelten Stream- und Clusterarchitektur.

Zur Einspeisung der eingehenden Daten wird eine Pipeline mit Akka.Streams aufgebaut. Nach der Materialisierung des Stream-Graphs steht eine `SourceQueue` als Einspeisepunkt zur Verfügung. Über diese Queue können neue Elemente kontrolliert in den Stream eingebracht werden. Durch die gewählte Overflow-Strategie Backpressure wird der Produzent automatisch verlangsamt, sobald nachgelagerte Verarbeitungsschritte die Daten nicht schnell genug verarbeiten können.

Programm 4.8: Stream-Einspeisung der Daten

```
1 await _queue.OfferAsync(dto);
```

4.4 Datenbearbeitung mit Akka.Streams

Für den Aufbau der Verarbeitungspipeline wird die GraphDSL-API von Akka.Streams verwendet, um den Stream-Graphen explizit zu definieren. Beim Start werden mehrere Worker-Aktoren instanziiert, welche die Weiterleitung der eingehenden Daten an die ShardRegion übernehmen. Zur Lastverteilung werden die Worker als parallele Senken in den Stream integriert und über eine `Balance`-Stufe angebunden.

Die `Balance`-Stufe verteilt eingehende Elemente demand-basiert auf die verfügbaren nachgelagerten Verarbeitungsstufen. Durch `waitForAllDownstreams = true` wird sichergestellt, dass die Verteilung erst beginnt, nachdem alle nachgelagerten Stufen initiale Nachfrage signalisiert haben. Die anschließende Verteilung erfolgt nicht strikt im Round-Robin-Verfahren, sondern abhängig von der jeweils anliegenden Nachfrage beziehungsweise dem aktuellen Backpressure-Zustand der einzelnen Worker.

Zur Einspeisung externer Daten in die Verarbeitungspipeline wird eine `Source.Queue` als Einstiegspunkt des Streams verwendet. Diese Queue bildet einen asynchronen Übergabepunkt zwischen der Socket-basierten Datenquelle und der internen Verarbeitung im Stream. Die gewählte Puffergröße begrenzt die Anzahl zwischengespeicherter Elemente, während die Overflow-Strategie `Backpressure` sicherstellt, dass der Produzent automatisch verlangsamt wird, sobald der Puffer ausgelastet ist und nachgelagerte Verarbeitungsschritte die Daten nicht schnell genug verarbeiten können.

Zur Unterstützung von Backpressure wird für die Worker-Aktoren eine Sink-Stufe mit Acknowledgement-Mechanismus eingesetzt. Mittels `Sink.ActorRefWithAck` wird jedes Element erst dann als verarbeitet betrachtet, wenn der jeweilige Worker eine Bestätigung (`ack`) zurückmeldet. Dadurch wird verhindert, dass schneller produziert wird als verarbeitet werden kann, und der Datenfluss bleibt auch unter Last stabil.

Nach der Definition des Graphen erfolgt die Materialisierung, wodurch die Stream-Komponenten initialisiert und die Verbindungen zwischen den einzelnen Stufen herge-

stellt werden. Anschließend können neue Elemente über die materialisierte `SourceQueue` in den Stream eingespeist werden (vgl. Programm 4.8).

Programm 4.9: Aufbau und Materialisierung des Stream-Graphs

```
var source = Source.Queue<IOpenF1Dto>(
    bufferSize: 8192,
    overflowStrategy: OverflowStrategy.Backpressure);

var kill = KillSwitches.Single<IOpenF1Dto>();

var graph = RunnableGraph.FromGraph(GraphDsl.Create(
    source, kill,
    (q, ks) => (q, ks),
    (builder, src, ks) =>
    {
        var balancer = builder.Add(
            new Balance<IOpenF1Dto>(workers.Count, waitForAllDownstreams: true));

        builder.From(src).Via(builder.Add(ks)).To(balancer.In);

        for (int i = 0; i < workers.Count; i++)
        {
            var sink = Sink.ActorRefWithAck<IOpenF1Dto>(
                workers[i],
                onInitMessage: StreamInit.Instance,
                ackMessage: StreamAck.Instance,
                onCompleteMessage: StreamCompleted.Instance,
                onFailureMessage: ex => new StreamFailed(ex));

            builder.From(balancer.Out(i)).To(sink);
        }

        return ClosedShape.Instance;
    }));

var (queue, ks) = graph.Run(_mat);
```

4.5 Verarbeitung in der ShardRegion

Der `DriverActorPersistent` bildet die zentrale Komponente zur konsistenten Verwaltung des Zustands eines einzelnen Fahrers innerhalb einer `ShardRegion`. Durch die Ableitung von `ReceivePersistentActor` besitzt er die Fähigkeit, Zustandsänderungen als Ereignisse (Events) dauerhaft zu persistieren. Diese Persistenzmechanismen ermöglichen es dem Aktor, seinen vollständigen Zustand bei einem Neustart, einem Ausfall einzelner Clusterknoten oder einem Rebalancing des Clusters zuverlässig wiederherzustellen.

4.5.1 Konstruktor und Wiederherstellung

Zu Beginn der Aktorinitialisierung werden mithilfe der Methode `RecoverState` alle zuvor gespeicherten Snapshots sowie die nach dem letzten Snapshot aufgetretenen Ereignisse geladen. Dies gewährleistet, dass sich der Zustand des Fahrers auch nach einem Ausfall oder einer Migration innerhalb des Clusters vollständig rekonstruieren lässt.

Ein vorhandener `SnapshotOffer` stellt dabei den zuletzt gespeicherten Zustandsstand bereit, wohingegen alle nachfolgenden Ereignisse sequentiell erneut angewendet werden, um den Zustand auf den aktuellen Stand zu bringen. Der Wiederherstellungsprozess endet mit dem Ereignis `RecoveryCompleted`. Anschließend wird geprüft, ob der Zustand bereits initialisiert wurde. In diesem Fall wechselt der Aktor in das Verhalten `Initialized`, andernfalls verbleibt er im Zustand `Uninitialized`.

Programm 4.10: Auszug aus dem DriverActorPersistent

```
1  public DriverActorPersistent(IRequiredActor<TelemetryRegionHandler> handler)
2  {
3      _handler = handler.ActorRef;
4      _logger.Info(\$"DriverActorPersistent constructor: {Self.Path.Name}");
5      RecoverState();
6
7      Become(Uninitialized);
8  }
9
10 private void RecoverState()
11 {
12     Recover<SnapshotOffer>(offer =>
13     {
14         _logger.Info(\$"SnapshotOffer for {offer.Snapshot}");
15         if (offer.Snapshot is DriverInfoState state)
16         {
17             _state.RestoreFromSnapshot(state);
18             _logger
19              .Info(\$"Recovered snapshot for driver {_state.ToDriverInfoString()}");
20         }
21     });
22
23     Recover<IHasDriverId>(evt =>
24     {
25         _state.Apply(evt);
26         _logger
27          .Info("Replayed {Event} for {Pid}", evt.GetType().Name, PersistenceId);
28     });
29
30     Recover<RecoveryCompleted>(_ =>
31     {
32         if (_state.IsInitialized) Become(Initialized);
33     });
34 }
```

4.5.2 Initialisierung des Aktors

Solange der Aktor noch nicht initialisiert wurde, befindet er sich im Verhalten *Uninitialized*. Dieser Zustand wird im Konstruktor über `Become(Uninitialized)` aktiviert. Die Methode `Become` ersetzt sämtliche bisher registrierten Nachrichtenhandler und ermöglicht damit ein dynamisches, zustandsabhängiges Verhalten des Aktors.

Erst wenn der Aktor eine `CreateModelDriverMessage` empfängt, wird diese persistiert, auf den internen Zustand angewendet und damit die Initialisierung abgeschlossen. Nach erfolgreicher Initialisierung wechselt der Aktor in das Verhalten `Initialized`, in dem er regulär auf weitere fachliche Nachrichten reagieren kann.

Empfängt der Aktor vor Abschluss der Initialisierung eine andere Nachricht, wird er passiviert, um unnötige Ressourcenbelegung zu vermeiden. Dies erfolgt über den generischen Nachrichtenhandler `CommandAny`, der für nicht explizit definierte Nachrichtentypen zuständig ist, die empfangenen Nachrichten protokolliert und gleichzeitig die Passivierung der zugehörigen Entität innerhalb der ShardRegion anstößt.

Programm 4.11: Auszug aus dem DriverActorPersistent

```
1 private void Uninitialized()
2 {
3     Command<CreateModelDriverMessage>(m =>
4     {
5         Persist(m, evt =>
6         {
7
8             try
9             {
10                _state.Apply(evt);
11                _logger.Info(\$"Initialized driver {_state.ToDriverInfoString()}");
12                Sender.Tell(
13                    new Status.Success(CreatedDriverMessage.Success(_state.Key)));
14                Become(Initialized);
15            }
16            catch (ArgumentNullException ex)
17            {
18                _logger.Error(ex,
19                        "Failed to initialize driver with message: {Message}", m);
20                Sender.Tell(new Status.Failure(ex));
21                Context.System.PubSub().Api.Publish(
22                        new NotifyStatusFailureMessage(ex.Message));
23            }
24        });
25    });
26
27    Command<StopEntity>(_ => Context.Stop(Self));
28
29    CommandAny(msg =>
30    {
31        var entityId = Self.Path.Name;
32        _logger.Warning(\$"Received {msg.GetType().Name} before initialization
33                            for entity {entityId}. Passivating.");
34
35        Sender.Tell(new NotInitializedMessage(entityId));
36        Context.Parent.Tell(new Passivate(new StopEntity()));
37        Context.System.PubSub().Api.Publish(
38                    new NotifyStatusFailureMessage("DriverActor was not init"));
39    });
40 }
```

4.5.3 Initialisierter DriverActorPersistent

Nach erfolgter Initialisierung wechselt der Aktor in das Verhalten `Initialized`. In diesem Zustand reagiert er auf verschiedene fachliche Nachrichten, insbesondere auf Änderungen im Fahrerzustand. Jede Änderung wird über die Methode `PersistAndApply` verarbeitet, die die empfangene Nachricht persistiert, auf den internen Zustand anwendet und anschließend die aktualisierten Informationen an den `TelemetryRegionHandler` weiterleitet. Dies stellt sicher, dass alle nachgelagerten Komponenten jederzeit über den

aktuellen Zustand des Fahrers verfügen.

Programm 4.12: Auszug aus dem DriverActorPersistent

```
1 private void Initialized()
2 {
3     Command<UpdateTelemetryMessage>(m =>
4     {
5         _logger.Debug("Telemetry: {Id} speed={Speed} t={Ts:o}",
6                         _state.Key,
7                         m.Speed,
8                         m.TimestampUtc);
9         PersistAndApply(m);
10    });
11
12    // weitere Handler ausgelassen
13 }
14
15 private void PersistAndApply(IHasDriverId element)
16 {
17    if (!_state.IsInitialized || !KeysMatchOrFail(element.Key))
18    {
19        Sender.Tell(
20            new Status.Failure(
21                new DriverInShardNotFoundException(
22                        element.Key,
23                        \$"Key is not {_state.Key} or initialized")));
24
25        Context.System.PubSub().Api.Publish(
26                    new NotifyStatusFailureMessage(
27                        \$"Key is not {_state.Key} or initialized"));
28        return;
29    }
30
31    Persist(element, ev =>
32    {
33        _state.Apply(ev);
34        SendToHandler();
35        CreateSnapshot();
36    });
37    Sender.Tell(new Status.Success(element));
38 }
```

4.5.4 Persistierung des Zustands

Damit der Wiederherstellungsprozess effizient bleibt, erzeugt der Aktor in regelmäßigen Abständen einen Snapshot seines aktuellen Zustands. Die Methode `CreateSnapshot` überprüft dazu, ob die aktuelle Sequenznummer ein Vielfaches von zehn ist. Nur in diesem Fall wird ein Snapshot erzeugt und gespeichert. Dieses Verfahren reduziert die Anzahl der beim Recovery zu verarbeitenden Ereignisse und führt damit zu einer schnelleren Wiederherstellung, ohne bei jeder einzelnen Zustandsänderung einen Snapshot zu

erzeugen.

Programm 4.13: Auszug aus dem DriverActorPersistent

```
1 private void CreateSnapshot()
2 {
3     if (LastSequenceNr % 10 == 0)
4     {
5         _logger.Debug("Creating snapshot for {Pid} at seqNr {Seq}",
6                       PersistenceId,
7                       LastSequenceNr);
8         SaveSnapshot(_state.CopyState());
9     }
10 }
```

4.5.5 Shard-Zuordnung und Rebalancing

Die Zuordnung eingehender Nachrichten zu den jeweiligen Shards und Entitäten erfolgt über den `DriverMessageExtractor`, der sowohl die Entitäts-ID als auch die Shard-ID aus einer Nachricht ableitet. Dies geschieht über den Standardmechanismus von `Akka.Cluster.Sharding`, der auf dem `HashCodeMessageExtractor` basiert. Die Shard-Zuordnung ergibt sich dabei aus einer Hashfunktion, die von Akka.NET bereitgestellt wird und eine gleichmäßige Verteilung der Entitäten über alle verfügbaren Clusterknoten anstrebt.

Eine eigene Rebalancing-Strategie wurde nicht implementiert, da die Standardstrategie von `Akka.Cluster.Sharding` bereits eine effiziente Lastverteilung sicherstellt. Sie verschiebt Shards automatisch, sobald neue Clusterknoten hinzukommen oder bestehende Knoten ausfallen, und ermöglicht somit eine robuste Wiederherstellung der Systembalance ohne zusätzlichen Implementierungsaufwand.

Zusammenfassung

Der `DriverActorPersistent` kombiniert persistente Ereignisverarbeitung mit einem klar strukturierten, zustandsabhängigen Aktormodell innerhalb einer `ShardRegion`. Im Recovery-Prozess werden sowohl Snapshots als auch nachfolgende Ereignisse rekonstruiert, wodurch ein konsistenter Fahrerzustand auch nach Ausfällen oder Rebalancing gewährleistet wird. Durch das Wechseln zwischen den Zuständen *Uninitialized* und *Initialized* wird sichergestellt, dass nur vollständig initialisierte Entitäten fachliche Nachrichten verarbeiten. Alle Zustandsänderungen werden über `PersistAndApply` als Ereignisse gespeichert und optional durch periodische Snapshots ergänzt. Damit erfüllt der Aktor die Anforderungen an ein fehlertolerantes, verteiltes und deterministisch rekonstruierbares Fahrermanagement im Cluster.

4.6 Nachrichtenverteilung im Cluster

Um bei einer losen Kopplung Nachrichten zwischen Aktoren auszutauschen, wird der `DistributedPubSub`-Mediator verwendet. Dadurch können Aktoren Nachrichten versenden und empfangen, ohne direkte Referenzen aufeinander zu besitzen.

4.6.1 Mediator: Nachrichtenempfänger

Damit ein Aktor Nachrichten über den Mediator empfangen kann, muss er sich mit einem bestimmten Topic beim `DistributedPubSub`-Mediator registrieren. Zur Vereinheitlichung wurde hierfür eine Basisklasse implementiert, die die Anmeldung beim Mediator übernimmt. Alle Aktoren, die Nachrichten über ein Topic empfangen sollen, erben von dieser Basisklasse.

Während der Anmeldephase werden eingehende Nachrichten zunächst zwischengespeichert, bis eine Bestätigung (`SubscribeAck`) des `DistributedPubSub`-Mediators eingetroffen ist. Erst danach wird das eigentliche Verhalten des Aktors aktiviert und die zwischengespeicherten Nachrichten werden verarbeitet. Um zusätzlich Gruppenkommunikation zu unterstützen, erfolgt neben der Anmeldung am Topic eine weitere Anmeldung mit einer generierten Gruppen-ID. Dadurch können mehrere Aktoren dieselben Nachrichten erhalten, wenn sie mit derselben Gruppen-ID registriert sind.

Programm 4.14: Auszug aus der Basisklasse für PubSub-Empfänger

```
1  private PubSubMember _member;
2
3  protected override void PreStart()
4  {
5      _pubSubActorRef = DistributedPubSub.Get(Context.System).Mediator;
6      if (_pubSubActorRef.IsNobody())
7          throw new ActorNotFoundException("Mediator not connected!");
8
9      _member = PubSubTypeMapping.ToMember(typeof(TTopic)) ?? PubSubMember.All;
10     _logger.Info(\$"Mediator is trying to connected to topic {_member.ToStr()}");
11
12     _pubSubActorRef.Tell(new Subscribe(_member.ToStr(), Self));
13     _pubSubActorRef.Tell(
14         new Subscribe(_member.ToStr(), Self, GenerateGroupId(_member)));
15 }
```

Nach der Anmeldung wird auf die Bestätigung durch den Mediator gewartet. Solange diese Bestätigung noch nicht eingetroffen ist, werden alle eingehenden Nachrichten zunächst zwischengespeichert. Sobald die erwartete Anzahl an `SubscribeAck`-Nachrichten empfangen wurde, werden die zwischengespeicherten Nachrichten verarbeitet und das eigentliche Verhalten des Aktors aktiviert.

Programm 4.15: Auszug aus der Basisklasse für PubSub-Empfänger

```
1 private void HandleAckSub()
2 {
3     Receive<SubscribeAck>(msg =>
4     {
5         _logger.Info(\$"Grab Ack for {msg.Subscribe.Topic} with
6                         group ({msg.Subscribe.Group})");
7
8         Become(() =>
9         {
10            Activated();
11            Stash.UnstashAll();
12        });
13    });
14
15    ReceiveAny(_ => Stash.Stash());
16 }
```

4.6.2 Mediator: Nachrichtensender

Zum Versenden von Nachrichten wird der `DistributedPubSub`-Mediator referenziert und über eine `Publish`-Nachricht ein bestimmtes Topic adressiert. Um sicherzustellen, dass Nachrichten stets mit einem konsistenten und korrekt definierten Topic gesendet werden, wurden mehrere typisierte Zugriffsmethoden definiert.

Diese Methoden (`Api`, `Backend`, `Ingress`, `Controller`) stellen jeweils einen vordefinierten logischen Kommunikationskanal bereit. Anstatt frei formulierte Topic-Strings zu verwenden, wird das jeweils zugehörige Topic intern über die Zuordnung des entsprechenden Mitglieds bestimmt und anschließend über den Mediator veröffentlicht.

Auf diese Weise entsteht eine string-freie und fehlerresistente Mechanik zur Nachrichtenverteilung, die sowohl eine lose Kopplung der Aktoren als auch eine einheitliche Struktur der verwendeten Topics im gesamten System sicherstellt.

Programm 4.16 zeigt exemplarisch die Verwendung dieser Methoden.

Programm 4.16: Auszug aus der Erweiterungsmethode für PubSub-Sender

```
1 Context.PubSub().Api.Publish(new NotifyDriverStateMessage(msg.Key, msg.State));
2 Context.PubSub().Backend.Publish(new NotifyDriverStateMessage(msg.Key, msg.State));
```

4.7 Cluster-Steuerung und Fehlertoleranz

Die Steuerung der Clusterverfügbarkeit sowie das Zusammenspiel zwischen `ShardRegion` und den Ingress-Komponenten werden durch eine explizite Koordinationsschicht realisiert. Zu dieser Schicht gehören insbesondere die Aktoren `ClusterCoordinator`, `ShardListener`, `IngressListener` und `ClusterEventListener`. Ziel ist es, den aktuellen Zustand der relevanten Clusterrollen (Backend- und Ingress-Knoten) zu überwachen und darauf aufbauend die Erreichbarkeit der `ShardRegion` gegenüber dem Ingress-Datenstrom zu steuern.

4.7.1 ClusterCoordinator

Der `ClusterCoordinator` übernimmt die zentrale Rolle der Zustands- und Verfügbarkeitskoordination zwischen der ShardRegion und den Ingress-Komponenten. Er empfängt Statusmeldungen sowohl vom `ShardListener` als auch vom `IngressListener` und verwaltet intern den aktuellen Erreichbarkeitsstatus der ShardRegion. Ändert sich dieser Zustand, benachrichtigt der `ClusterCoordinator` unmittelbar den `IngressListener`, damit die Ingress-Knoten ihren Datenfluss entsprechend anpassen können. Dadurch wird sichergestellt, dass Daten nur dann an die ShardRegion gesendet werden, wenn diese tatsächlich verfügbar ist.

Die Ermittlung des Shard-Status erfolgt entweder reaktiv durch eingehende Statusmeldungen oder proaktiv über explizite Statusabfragen. Der `ClusterCoordinator` selbst ist nicht an der Verarbeitung der Daten beteiligt, sondern fungiert als administrative Instanz, die den Ingress-Komponenten mitteilt, ob eine Weiterleitung von Nachrichten an die ShardRegion zulässig ist. Zusätzlich stellt er Informationen über die derzeit verfügbare Anzahl an Clusterknoten bereit.

Programm 4.17: Auszug aus dem ClusterCoordinator

```
Receive<ShardConnectionUpdateMessage>(msg =>
{
    _hasShardRegion = msg.IsShardOnline;
    var con = _hasShardRegion ? "connected" : "disconnected";
    _logger.Debug(\$"Shard connection changed. Shard {con}");
    _ingressListener.Tell(new IngressConnectionCanActivated(_hasShardRegion));
});

ReceiveAsync<IngressConnectivityRequest>(async _ =>
{
    if (!_hasShardRegion)
    {
        var res = await _shardListener
            .Ask<ShardConnectionUpdateMessage>(ShardConnectionRequest.Instance);
        _hasShardRegion = res.IsShardOnline;
    }

    _logger.Debug(\$"Ingress activate request. Shard active: {_hasShardRegion}");
    Sender.Tell(new IngressConnectivityResponse(_hasShardRegion));
});
```

4.7.2 ShardListener: Verfügbarkeit der ShardRegion

Der `ShardListener` ist für die kontinuierliche Überwachung der Backend-Knoten verantwortlich, auf denen die ShardRegion ausgeführt wird. Zu diesem Zweck verwaltet er eine interne Menge aller aktuell aktiven Backend-Adressen. Änderungen in der Zusammensetzung dieser Menge, etwa durch das Hinzukommen oder Wegfallen eines Knotens, werden über die Methode `SendCountUpdate` an den `ClusterCoordinator` gemeldet.

Um eine übermäßige Anzahl an Statusmeldungen zu vermeiden, insbesondere bei kurzzeitigen oder rasch aufeinanderfolgenden Clusterereignissen, wird ein Entlastungsmechanismus eingesetzt, der funktional einer Debounce-Strategie entspricht. Dabei werden Statusänderungen nicht sofort nach ihrem Auftreten weitergeleitet, sondern zunächst verzögert gesammelt. Erst wenn innerhalb eines kurzen Zeitfensters keine weiteren Änderungen auftreten, wird eine konsolidierte Aktualisierung an den `ClusterCoordinator` gesendet. Dies reduziert die Kommunikationslast erheblich und verhindert, dass der `ClusterCoordinator` mit einer großen Anzahl kurzlebiger Statusmeldungen überflutet wird.

Programm 4.18: Auszug aus dem ShardListener

```
1 Receive<IncreaseClusterMember>(msg =>
2 {
3     Logger.Debug("Increase counter");
4     _activeBackends.Add(msg.ClusterMemberRef);
5     SendCountUpdate(new ShardConnectionUpdateMessage(_activeBackends.Count > 0));
6 });
7
8 Receive<DecreaseClusterMember>(msg =>
9 {
10    Logger.Debug("Decrease counter");
11    _activeBackends.Remove(msg.ClusterMemberRef);
12    SendCountUpdate(new ShardConnectionUpdateMessage(_activeBackends.Count > 0));
13 });
14
15 Receive<ShardCountRequest>(_ =>
16    Sender.Tell(new ShardCountResponse(_activeBackends.Count)));
```

4.7.3 IngressListener: Verwaltung und Benachrichtigung der Ingress-Knoten

Der `IngressListener` übernimmt analog zum `ShardListener` die Aufgabe, den aktuellen Zustand der Ingress-Knoten zu überwachen und eine konsistente Sicht auf deren Verfügbarkeit bereitzustellen. Zu diesem Zweck verwaltet er eine interne Menge aller aktiven Ingress-Instanzen und aktualisiert diese basierend auf den vom `ClusterEventListener` weitergeleiteten Cluster-Ereignissen.

Neben dieser Überwachungsfunktion fungiert der `IngressListener` zugleich als zentrale Anlaufstelle für Anfragen seitens der Ingress-Knoten. Er beantwortet insbesondere Rückfragen zur aktuellen Erreichbarkeit der `ShardRegion` und dient somit als Vermittler zwischen Ingress-Schicht und Backend-Struktur. Änderungen im Status der `ShardRegion` werden unmittelbar per Publish-Subscribe an alle registrierten Ingress-Knoten weitergegeben, um deren Sendeverhalten entsprechend zu steuern.

Programm 4.19: Auszug aus dem IngressListener

```
1 Receive<IngressConnectionCanActivated>(msg =>
2 {
3     var isOnline = msg.IsShardOnline ? "online" : "offline";
4     Logger.Debug(\$"Auto send to shard is {isOnline} and ingress will be notified");
5     Context.PubSub().Ingress.Publish(new NotifyIngressShardIsOnline(msg.
      IsShardOnline));
6 });
7
8 Receive<IncreaseClusterMember>(msg =>
9 {
10    Logger.Debug("Increase counter");
11    _activeIngress.Add(msg.ClusterMemberRef);
12    SendCountUpdate(new IngressConnectionUpdateMessage(_activeIngress.Count > 0));
13 });
```

4.7.4 ClusterEventListener: Verarbeitung und Weiterleitung von Cluster-Events

Der `ClusterEventListener` ist unmittelbar an das Cluster-Subsystem angebunden und abonniert zentrale Ereignisse des Cluster-Lifecycles. Hierzu zählen insbesondere `MemberUp` und `MemberRemoved`, die das Hinzufügen bzw. Entfernen eines Clusterknotens signalisieren, sowie Erreichbarkeitsänderungen wie `UnreachableMember` und `ReachableMember`.

Die eingehenden Ereignisse werden anhand der Rollen des betroffenen Clusterknotens analysiert. Auf dieser Grundlage entscheidet der `ClusterEventListener`, welcher spezialisierte Listener (z. B. `ShardListener`, `IngressListener`) informiert werden muss. Dadurch fungiert der `ClusterEventListener` als zentrale Verteilerinstanz, die Cluster-Statusänderungen effizient an die jeweils relevanten Systemkomponenten weiterleitet. Die Zuordnung erfolgt über die Rollen des jeweiligen Knotens, die zur Klassifizierung seiner Funktion im Cluster dienen.

Programm 4.20: Auszug aus dem ClusterEventListener

```
Receive<ClusterEvent.MemberUp>(u =>
{
    _logger.Info(\$"Cluster Member is Up {u}, {string.Join(',', u.Member.Roles)}");
    foreach (var s in u.Member.Roles)
        Broadcast(ClusterMemberExtension.Parse(s),
                  new IncreaseClusterMember(u.Member.Address));
});

Receive<ClusterEvent.MemberRemoved>(r =>
{
    _logger.Info(\$"Cluster Member is Removed {r},
                    {string.Join(',', r.Member.Roles)}");
    foreach (var s in r.Member.Roles)
        Broadcast(ClusterMemberExtension.Parse(s),
                  new DecreaseClusterMember(r.Member.Address));
});
```

4.7.5 Fehlertoleranzkonzept der Koordinationsschicht

Die Koordinationsschicht ist darauf ausgelegt, automatisch auf Änderungen im Clusterzustand zu reagieren und dadurch die Erreichbarkeit der ShardRegion zuverlässig zu bestimmen. Grundlage hierfür bilden die in Akka.NET integrierten Fehlertoleranzmechanismen, insbesondere das Gossip-Protokoll sowie die Erreichbarkeitsüberwachung mittels periodischer Herzschlagsignale.

Akka.NET verbreitet Clusterzustände kontinuierlich über Gossip-Nachrichten, die in regelmäßigen Intervallen zufällig zwischen den Knoten ausgetauscht werden. Auf diese Weise konvergiert der Wissensstand aller Clusterteilnehmer sukzessive zu einer gemeinsamen Sicht auf den Systemzustand, ohne dass ein zentraler Koordinator erforderlich ist. Parallel dazu überwacht das Cluster-Subsystem die Erreichbarkeit der einzelnen Knoten anhand von Heartbeats. Bleiben diese über einen bestimmten Zeitraum aus oder treten nur noch unregelmäßig auf, wird der entsprechende Knoten zunächst als `Unreachable` markiert. Persistiert dieser Zustand, stuft das Cluster den Knoten als ausgefallen ein und kennzeichnet ihn als `MemberRemoved` (Project, 2025a).

Die Listener-Komponenten der Koordinationsschicht werten die dadurch generierten Ereignisse aus und aktualisieren ihre interne Sicht auf die Clusterverfügbarkeit. Auf dieser Basis kann der `ClusterCoordinator` unmittelbar reagieren, indem er beispielsweise den Ingress-Datenstrom deaktiviert, sobald die ShardRegion nicht mehr erreichbar ist. Dieses Verhalten ermöglicht eine fehlertolerante Betriebsweise, bei der das Gesamtsystem trotz einzelner Knotenausfälle konsistent weiterarbeitet und verhindert wird, dass Nachrichten an eine nicht verfügbare Backend-Struktur gesendet werden.

4.8 Client- und Visualisierungsschicht

Zur Veranschaulichung der im System erzeugten und verarbeiteten Daten wurde eine konsolenbasierte Client-Anwendung entwickelt, die als leichtgewichtige Visualisierungsschicht dient. Der Client verbindet sich direkt mit dem Aktorensystem und ermöglicht die Interaktion mit verschiedenen Komponenten, etwa zur Abfrage von Systemmetriken oder zur Steuerung der Datenverarbeitung.

Eine separate API-Schicht wurde im Rahmen dieser Arbeit noch nicht umgesetzt, wird jedoch als sinnvolle Erweiterungsmöglichkeit betrachtet. Eine solche Schnittstelle ließe sich auf Basis der bestehenden Architektur vergleichsweise einfach realisieren, da sie über eine Anbindung an das Aktorensystem dieselben Nachrichten empfangen und versenden könnte wie der aktuelle Konsolen-Client. Dadurch wäre es möglich, externe Anwendungen oder Visualisierungstools einzubinden und die Daten des Systems auch außerhalb der Konsolenoberfläche verfügbar zu machen.

4.8.1 Konsolenbasierter Monitoring-Client

Die entwickelte Konsolenanwendung dient als Monitoring-Client und ermöglicht eine übersichtliche Darstellung der im System erzeugten Metriken und Statusinformationen. Für die strukturierte und visuell ansprechende Aufbereitung der Daten kommt das Drittanbieterpaket `Spectre.Console` zum Einsatz, das erweiterte Funktionalitäten für die Gestaltung interaktiver und formatiert strukturierter Konsolenoberflächen bereitstellt.

Abbildung 4.2 zeigt einen Screenshot der Anwendung während der Laufzeit und veranschaulicht die textbasierte Visualisierung der Systemzustände.

Abbildung 4.2: Konsolenbasierter Monitoring-Client („Bachelor Akka Client“)

Die Oberfläche ist in mehrere Bereiche gegliedert:

- **Throughput**: Anzeige der aktuell verarbeiteten Nachrichtenrate (Nachrichten pro Sekunde), der insgesamt verarbeiteten Nachrichten, der Laufzeit sowie einer einfachen Fehlerrate. Zusätzlich werden Latenz-Quantile (p50, p95, p99) visualisiert, sofern Messungen aktiv sind.
- **Latency**: Detailliertere Darstellung der gemessenen Antwortzeiten in Millisekunden, ebenfalls aufgeteilt nach p50, p95 und p99 sowie Minimal- und Maximalwert.
- **Cluster & Shards**: Übersicht über den aktuellen Clusterzustand, insbesondere die Anzahl aktiver Knoten, Shards und Entities sowie den aktuell verwendeten Pipeline-Modus.
- **Race-Informationen**: Anzeige kontextbezogener Domäneninformationen, etwa zur aktuell ausgewählten Rennstrecke (Location, Country).
- **Controls**: Liste der verfügbaren Tastaturbefehle (z. B. Verbindungsaufbau, Start und Stop von Messungen, Wechsel des Rennens oder der Pipeline-Konfiguration).

Der Client kommuniziert hierzu direkt mit dem Cluster und bezieht sowohl Systemmetriken (z. B. Cluster- und Shard-Zustand) als auch domänenspezifische Daten. Er ist nicht als Endanwenderoberfläche konzipiert, sondern als leichtgewichtige Monitoring- und Debugging-Komponente, die es ermöglicht, die Funktionsweise des verteilten Systems im Rahmen dieser Arbeit transparent und nachvollziehbar darzustellen.

4.9 Zusammenführung der Komponenten

In diesem Abschnitt wird das Zusammenspiel der zuvor beschriebenen Komponenten, `ClusterCoordinator`, `ShardRegion`, Ingress-Schicht sowie Monitoring-Client, im Gesamtsystem erläutert.

4.9.1 Ablauf aus Sicht des Clusters

Nach dem Start des Clusters initialisieren sich die beteiligten Knoten entsprechend ihrer konfigurierten Rollen. Die ShardRegion wird auf den Backend-Knoten gestartet und übernimmt die Verwaltung der persistierten Fahrerinstanzen (DriverActorPersistent). Parallel dazu verbinden sich die Ingress-Knoten mit dem ClusterCoordinator, um Informationen über die Verfügbarkeit der ShardRegion zu erhalten.

Sobald mindestens eine ShardRegion verfügbar und ein Ingress-Knoten aktiv ist, erteilt der ClusterCoordinator die Freigabe für die Weiterleitung von Daten vom Ingress an die ShardRegion. Wählt der Monitoring-Client anschließend ein Rennen aus, erhält der Ingress-Dienst die Anweisung, die entsprechenden Telemetriedaten über OpenF1 zu beziehen und in einen Akka.Streams-Graph einzuspeisen. Innerhalb des Stream-Graphs werden die Rohdaten aufbereitet und in ein einheitliches Nachrichtenformat konvertiert, das von der ShardRegion verarbeitet werden kann.

Die konvertierten Nachrichten werden anschließend über den ShardRegionProxy an die ShardRegion übermittelt. Diese prüft die Nachricht und ordnet sie anhand der Fahrer-ID

der zuständigen DriverActorPersistent-Instanz zu. Der Aktor verarbeitet die eingehende Nachricht, aktualisiert seinen Zustand, persistiert die Änderungen und sendet schließlich aktualisierte Fahrerdaten an den Monitoring-Client.

4.9.2 Ablauf aus Sicht der Datenübertragung

Der Datenfluss beginnt im Ingress-Service, der einen kontinuierlichen Telemetriedatenstrom vom vorgelagerten Datenprovider entgegennimmt (z. B. basierend auf OpenF1) und in einen Akka.Streams-Graph überführt.

Die Übergabe an die ShardRegion erfolgt über einen ShardRegionProxy. Der Proxy stellt einen transparenten Zugriff auf die ShardRegion bereit, ohne dass die Ingress-Komponenten an eine konkrete physische Instanz gebunden sind. Die Zuordnung der Nachrichten zu Shards und Entitäten erfolgt über den `MessageExtractor` auf Basis der Fahrer-ID. Dadurch werden Nachrichten deterministisch an die zuständige Entity (`DriverActorPersistent`) geroutet, die Platzierung der Shards auf Backend-Knoten wird durch Cluster Sharding verwaltet und kann sich durch Rebalancing dynamisch ändern.

4.9.3 Ablauf aus Sicht der Fehlertoleranz

Die Fehlertoleranz wird durch die in Akka.NET integrierten Mechanismen zur Clusterüberwachung sichergestellt. Fällt ein Clusterknoten aus oder reagiert nicht mehr, erkennt das Gossip-Protokoll den Zustand und markiert den betreffenden Knoten zunächst als `Unreachable`. Persistiert dieser Zustand, wird der Knoten als `MemberRemoved` eingestuft und aus dem Cluster entfernt.

Der `ClusterEventListener` empfängt diese Ereignisse und informiert `ShardListener` und `IngressListener` über die entsprechenden Änderungen. Der `ShardListener` aktualisiert daraufhin seine interne Sicht auf die Backend-Knoten und leitet diese Information an den `ClusterCoordinator` weiter. Dieser informiert anschließend den `IngressListener`, der wiederum die Ingress-Knoten über die neue Erreichbarkeitssituation der ShardRegion benachrichtigt. Solange keine funktionsfähige ShardRegion verfügbar ist, senden die Ingress-Knoten keine Daten an den Cluster, wodurch inkonsistente Zustände und Nachrichtenverluste verhindert werden.

Fällt eine einzelne `DriverActorPersistent`-Instanz aus, wird diese von der ShardRegion automatisch neu gestartet. Aufgrund der Persistierung ihres vorherigen Zustands kann die Aktor-Instanz ihren letzten konsistenten Zustand rekonstruieren und die Verarbeitung fortsetzen, ohne dass Daten verloren gehen oder eine manuelle Wiederherstellung erforderlich ist.

Kapitel 5

Tests und Analysen

In diesem Kapitel werden die durchgeführten Tests und Analysen beschrieben, um einen detaillierten Einblick in die Funktionsweise, Zuverlässigkeit und Leistungsfähigkeit des entwickelten Systems zu geben. Darüber hinaus werden die erzielten Ergebnisse diskutiert und bewertet. Als Grundlage für die anschließenden Auswertungen werden zunächst die verwendete Teststrategie sowie die Testumgebung erläutert.

5.1 Teststrategie und Testumgebung

Zur Überprüfung der Funktionsfähigkeit und Robustheit des Systems wurden unterschiedliche Testarten eingesetzt. Diese Tests dienen sowohl der Validierung der funktionalen Anforderungen als auch der Analyse des Verhaltens der einzelnen Komponenten unter verschiedenen Betriebsbedingungen. Da das Gesamtsystem im Wesentlichen aus zwei zentralen Technologiebereichen besteht – der Akka.NET `ShardRegion` und der Datenverarbeitung über Akka.NET Streams – wurden die Testfälle entsprechend auf diese Bereiche abgestimmt.

Die durchgeführten Tests lassen sich in drei Kategorien einteilen:

- **Funktionale Tests**: Überprüfung der korrekten Verarbeitung, Persistierung und Weiterleitung der Daten.
- **Failover- und Resilienztests**: Analyse des Systemverhaltens bei Node-Ausfällen, unerreichbaren Clusterknoten sowie Neustarts der persistierten Entitäten.
- **Performance- und Durchsatztests**: Bewertung der Effizienz der implementierten Streamverarbeitungspipelines sowie des Systemverhaltens unter Last.

Für die Tests auf Actor-Ebene wurde das Akka.NET `TestKit` eingesetzt. Dieses ermöglicht das Testen von Aktor-Interaktionen, Nachrichtenflüssen und Zustandsänderungen in einer kontrollierten, aber dennoch realitätsnahen Umgebung.

5.2 Funktionale Tests

Im Rahmen der funktionalen Tests wurde die korrekte Initialisierung und Zusammenarbeit der zentralen Cluster-Komponenten überprüft. Dabei wurde verifiziert, dass die ShardRegion erfolgreich erstellt wird und die Kommunikation zwischen mehreren Instanzen (Cluster-Knoten) erwartungsgemäß funktioniert.

Weiterhin wurde das Verhalten der Driver-Entities in verschiedenen Zuständen getestet. Insbesondere wurde geprüft, dass ein `DriverActorPersistent` vor der Initialisierung eingehende Nachrichten korrekt ablehnt und eine entsprechende Rückmeldung liefert. Nach erfolgreicher Initialisierung werden Telemetrie- und Zustandsupdates akzeptiert, persistiert und der interne Zustand konsistent aktualisiert.

Zusätzlich wurde validiert, dass verarbeitete Driver-Daten korrekt an nachgelagerte Komponenten weitergeleitet werden, insbesondere an die API sowie an die Monitoring-Komponente. Für Fehlerszenarien wurde geprüft, dass ungültige oder nicht zur jeweiligen Entität passende IDs zuverlässig erkannt werden und keine inkonsistenten Zustände entstehen.

5.3 Verhalten bei Node-Ausfällen und Neuausrichtung

Zur Bewertung der Fehlertoleranz des verteilten Systems wurden Szenarien mit ausfallenden und wieder hinzukommenden Cluster-Knoten untersucht. Dabei wurde insbesondere das Verhalten der ShardRegion sowie die Koordination durch den ClusterCoordinator analysiert.

Fällt ein Backend-Knoten aus, wird dies durch die Cluster-Mitgliedsüberwachung erkannt. Solange mindestens eine ShardRegion im Cluster verfügbar ist, wird die Verarbeitung weiterhin fortgesetzt und die betroffenen Shards werden auf verbleibende Knoten umverteilt (Rebalancing). Erst wenn keine ShardRegion mehr vorhanden ist, deaktiviert der ClusterCoordinator den Ingress-Dienst temporär, um eingehende Daten nicht an nicht verfügbare Backend-Komponenten weiterzuleiten.

Beim erneuten Hinzukommen eines Knotens werden die Shards neu gestartet und in den Cluster integriert. Die Persistenzmechanismen der `DriverActorPersistent`-Instanzen sorgen dabei dafür, dass der zuvor gespeicherte Zustand aus dem Journal beziehungsweise Snapshot wiederhergestellt wird. Dadurch bleibt der Zustand der Driver-Entities konsistent, und es gehen keine bereits verarbeiteten Telemetriedaten verloren.

Die Tests zeigen, dass das System Ausfälle einzelner Knoten korrekt erkennt, ein Rebalancing der Shards durchführt und nach Wiederherstellung der Knoten den persistierten Zustand zuverlässig lädt. Damit wird eine hohe Fehlertoleranz und Zustandskonsistenz im Clusterbetrieb gewährleistet.

5.4 Leistungsanalyse und Bewertung der Stream-Strategien

In diesem Abschnitt wird die Leistungsfähigkeit der implementierten Verarbeitungspipeline sowie der Einfluss unterschiedlicher Overflow-Strategien systematisch untersucht. Ziel ist es, das Verhalten der Pipeline unter Last zu analysieren und die Auswirkungen auf Durchsatz, Latenz und Nachrichtenverluste zu bewerten. Die Ergebnisse dienen dazu, die Eignung der gewählten Strategien für die Verarbeitung zeitkritischer Telemetriedaten zu beurteilen.

5.4.1 Versuchsaufbau und Messgrößen

Zur Untersuchung der Leistungsfähigkeit der implementierten Stream-Pipeline wurden experimentelle Messungen unter kontrollierten Bedingungen durchgeführt. Hierzu wurde ein synthetischer Datenstrom mit einer festen Anzahl von 4000 Elementen erzeugt und durch eine Akka.Streams-Pipeline geleitet. Die Einspeisung des Datenstroms erfolgte ohne explizite Ratenbegrenzung, sodass die Elemente unmittelbar erzeugt und in die Pipeline eingespeist wurden (Burst-Einspeisung). Die effektive Eingangsrate ergab sich somit aus der Verarbeitungskapazität der Pipeline, dem Backpressure-Mechanismus sowie der konfigurierten Puffergröße. Die Pipeline enthielt einen konfigurierbaren Puffer mit einer Größe von 50 Elementen sowie eine simulierte Verarbeitungsverzögerung von 2 ms pro Element, um das Verhalten realer Telemetriedatenverarbeitung nachzubilden.

Als zentrale Messgrößen wurden der Durchsatz in Nachrichten pro Sekunde sowie die Latenzverteilung erfasst. Die Latenz wurde als Zeitdifferenz zwischen Erzeugung und Verarbeitung eines Elements bestimmt. Zur detaillierten Analyse wurden die Perzentile p50, p95 und p99 der Latenzverteilung berechnet. Zusätzlich wurde die Anzahl der tatsächlich verarbeiteten Elemente ermittelt, um mögliche Nachrichtenverluste in Abhängigkeit von der gewählten Overflow-Strategie zu quantifizieren.

5.4.2 Vergleich der Overflow-Strategien

Zur Bewertung der unterschiedlichen Overflow-Strategien wurden Durchsatz, Latenz und Nachrichtenverluste gemessen. Verglichen wurden die Strategien `Backpressure`, `DropNew`, `DropHead` und `DropTail`. Der Test verwendete einen Datenstrom von 4000 Elementen bei einer simulierten Verarbeitungszeit von 2 ms pro Element und einer Puffergröße von 50 Elementen.

Der gemessene Durchsatz lag bei allen Strategien mit etwa 64 Nachrichten pro Sekunde auf einem ähnlichen Niveau. Dies zeigt, dass der Durchsatz primär durch die Verarbeitungsgeschwindigkeit der Pipeline und weniger durch die gewählte Overflow-Strategie bestimmt wird.

Deutliche Unterschiede zeigen sich bei der Anzahl verarbeiteter Elemente. Während Backpressure alle 4000 Elemente verlustfrei verarbeiten konnte, wurden bei den Drop-Strategien jeweils nur etwa 51 Elemente verarbeitet, da neue Elemente bei ausgelastetem Puffer verworfen werden.

Auch die Latenz unterscheidet sich deutlich: Backpressure weist mit einem Median von etwa 322 ms und p95-Werten über 600 ms höhere Latenzen auf, bedingt durch Warteschlangen im Puffer. Die Drop-Strategien erreichen dagegen niedrige Latenzen zwischen 11 und 21 ms, da nur ein kleiner Teil der Elemente tatsächlich verarbeitet wird.

Insgesamt zeigt sich ein Zielkonflikt zwischen Vollständigkeit und Reaktionszeit. Backpressure ermöglicht eine verlustfreie Verarbeitung bei höheren Latenzen, während Drop-Strategien geringere Latenzen auf Kosten erheblicher Nachrichtenverluste bieten. Für die Verarbeitung kritischer Telemetriedaten erscheint daher Backpressure als die geeignetere Strategie.

5.5 Diskussion der Ergebnisse

Die durchgeführten Tests zeigen, dass die entwickelte Architektur die funktionalen und nicht-funktionalen Anforderungen im Wesentlichen erfüllt. Die funktionalen Tests bestätigen, dass die Interaktion zwischen Ingress-Service, Stream-Pipeline und Cluster-Sharding korrekt umgesetzt wurde und eingehende Telemetriedaten zuverlässig verarbeitet und weitergeleitet werden.

Die Failover-Tests verdeutlichen die hohe Fehlertoleranz des Systems. Durch die Verwendung von Akka.NET Cluster Sharding sowie persistenter Entities konnten Ausfälle einzelner Knoten erkannt und durch Rebalancing kompensiert werden. Gleichzeitig stellt die Wiederherstellung des Zustands aus Journal und Snapshots sicher, dass keine bereits verarbeiteten Daten verloren gehen. Dies bestätigt die Eignung der gewählten Architektur für verteilte, resiliente Stream-Verarbeitungssysteme.

Die Leistungsanalyse der Stream-Pipeline zeigt einen klaren Zielkonflikt zwischen Latenz und Vollständigkeit der Datenverarbeitung. Während Drop-basierte Overflow-Strategien niedrige Latenzen ermöglichen, führen sie zu erheblichen Nachrichtenverlusten. Die Backpressure-Strategie gewährleistet hingegen eine vollständige Verarbeitung aller Elemente, allerdings mit erhöhten Latenzen unter Last. Für den vorliegenden Anwendungsfall der Telemetriedatenverarbeitung ist diese Eigenschaft entscheidend, da Datenkonsistenz wichtiger ist als minimale Reaktionszeiten.

Es ist jedoch zu berücksichtigen, dass die Untersuchung der Overflow-Strategien in einem kontrollierten Testszenario mit synthetischen Daten, begrenzter Datenmenge und konstanter Verarbeitungszeit durchgeführt wurde. Die Ergebnisse liefern daher eine qualitative Einschätzung des Systemverhaltens unter Last, können jedoch nicht uneingeschränkt auf beliebig große Produktionsszenarien übertragen werden. Insbesondere können sich bei realen Telemetriedatenströmen mit variierender Last, Netzwerkverzögerungen und dynamischer Clustergröße abweichende Performancecharakteristiken ergeben.

Insgesamt bestätigen die Ergebnisse, dass die Kombination aus Akka.NET Cluster Sharding, persistierenden Entities und reaktiver Stream-Verarbeitung eine geeignete Grundlage für die skalierbare und fehlertolerante Verarbeitung kontinuierlicher Datenströme darstellt. Gleichzeitig zeigen die Tests, dass insbesondere unter hoher Last eine Abwägung zwischen Latenz und Datenvollständigkeit erforderlich ist, die je nach Anwen-

dungsszenario unterschiedlich gewichtet werden muss.

Kapitel 6

Zusammenfassung und Ausblick

Dieses Kapitel fasst die zentralen Ergebnisse der Arbeit zusammen, bewertet die Zielerreichung der entwickelten Architektur und gibt einen Ausblick auf mögliche Weiterentwicklungen des Systems. Dabei werden sowohl die erreichten Eigenschaften hinsichtlich Skalierbarkeit, Fehlertoleranz und reaktiver Datenverarbeitung als auch bestehende Einschränkungen und zukünftige Erweiterungspotenziale diskutiert.

6.1 Zusammenfassung der Ergebnisse

Im Rahmen dieser Arbeit wurde ein verteiltes, aktorenbasiertes System zur Verarbeitung kontinuierlicher Datenströme auf Basis von Akka.NET konzipiert und implementiert. Ziel war es, eingehende Telemetriedaten effizient, skalierbar und fehlertolerant zu verarbeiten sowie innerhalb eines Clusterverbunds zu speichern und weiterzuleiten.

Hierzu wurde eine Architektur entwickelt, die aus Ingress-, Backend- und Koordinator-Komponenten besteht. Der Eingangs-Service übernimmt den kontinuierlichen Empfang der Telemetriedaten über eine persistente Socket-Verbindung und speist diese in eine Akka.Streams-Pipeline ein. Durch die Verwendung einer `Source.Queue` mit Backpressure konnte ein kontrollierter Datenfluss zwischen externer Datenquelle und interner Verarbeitung realisiert werden.

Die Verarbeitung der Daten erfolgt innerhalb eines explizit definierten Stream-Graphs unter Verwendung der GraphDSL-API. Zur parallelen Verarbeitung wurden mehrere Worker-Aktoren eingesetzt, die über eine `Balance`-Stufe in den Stream integriert sind. Die Lastverteilung erfolgt dabei abhängig von der jeweils signalisierten Verarbeitungskapazität der Worker, wodurch eine dynamische Anpassung an die aktuelle Auslastung erreicht wird.

Das Backend bildet die zentrale Verarbeitungsschicht des Systems. Hier wird mittels Akka.NET Cluster Sharding eine `ShardRegion` betrieben, in der zustandsbehaftete Entitäten als persistente Aktoren (`DriverActorPersistent`) verwaltet werden. Jede Entität repräsentiert den Zustand eines einzelnen Fahrers und verarbeitet eingehende Telemetriedaten sequenziell. Zustandsänderungen werden als Ereignisse persistiert und

in regelmäßigen Abständen durch Snapshots ergänzt, wodurch eine konsistente Wiederherstellung nach Ausfällen oder Rebalancing des Clusters gewährleistet wird. Durch die Verteilung der Shards auf mehrere Backend-Knoten kann die Verarbeitung automatisch skaliert und bei Knotenausfällen transparent fortgeführt werden.

Ergänzend wurde ein Acknowledgement-Mechanismus mittels `Sink.ActorRefWithAck` implementiert, um Backpressure auch auf Ebene der Aktor-Kommunikation zu unterstützen. Die implementierte Architektur ermöglicht somit eine skalierbare, reaktive und fehlertolerante Verarbeitung kontinuierlicher Telemetriedatenströme innerhalb eines verteilten Aktorensystems.

Zur Überwachung der Kommunikation zwischen Ingress- und Backend-Komponenten wurde zusätzlich eine Koordinationsinstanz implementiert. Diese stellt sicher, dass der Ingress-Service eingehende Daten erst dann in das System einspeist, wenn mindestens eine empfangsbereite Backend-Komponente verfügbar ist. Hierzu werden Cluster-Ereignisse überwacht, insbesondere das Beitreten und Verlassen von Knoten sowie die Verfügbarkeit relevanter Aktoren. Auf diese Weise wird verhindert, dass Daten an nicht verfügbare Verarbeitungseinheiten gesendet werden, wodurch die Stabilität und Konsistenz des Systems erhöht wird.

6.2 Bewertung der Zielerreichung

Das primäre Ziel der Arbeit bestand darin, ein skalierbares und fehlertolerantes System zur Verarbeitung kontinuierlicher Telemetriedaten auf Basis von Akka.NET zu entwickeln. Dieses Ziel konnte weitgehend erreicht werden, da eine funktionale Architektur implementiert wurde, die eine verteilte Verarbeitung innerhalb eines Clusters ermöglicht.

Die Verwendung von Akka.Streams erwies sich als geeigneter Ansatz zur kontrollierten Verarbeitung der eingehenden Datenströme. Insbesondere der integrierte Backpressure-Mechanismus trug maßgeblich zur Stabilität des Systems bei, da eine Überlastung nachgelagerter Verarbeitungsschritte effektiv verhindert werden konnte. Durch den Einsatz mehrerer Worker-Aktoren in Kombination mit der `Balance`-Stufe konnte zudem eine dynamische Lastverteilung realisiert werden.

Auch die Integration von Cluster Sharding stellte einen wesentlichen Beitrag zur Zielerreichung dar, da dadurch zustandsbehaftete Entitäten automatisch auf mehrere Knoten verteilt und bei Bedarf neu ausbalanciert werden können. Dies erfüllt die Anforderungen an Skalierbarkeit und Fehlertoleranz verteilter Systeme.

Dennoch zeigten sich im Verlauf der Implementierung auch Einschränkungen. Die Komplexität der Konfiguration und des Debuggings verteilter Aktorensysteme erwies sich als hoch, insbesondere im Zusammenspiel von Streams, Clustering und Persistenz. Zudem hängt die tatsächliche Leistungsfähigkeit stark von der Anzahl der Worker-Aktoren sowie der verfügbaren Systemressourcen ab.

Insgesamt kann festgehalten werden, dass die gesetzten Ziele in Bezug auf Skalierbarkeit, Reaktivität und verteilte Verarbeitung erfolgreich umgesetzt wurden, wenngleich

die praktische Komplexität solcher Systeme eine sorgfältige Konfiguration und Überwachung erfordert.

6.3 Ausblick

Zur weiteren Verbesserung des Systems bietet sich insbesondere der Einsatz von Cloud-Infrastrukturen an. Durch den Betrieb der Anwendung in einer containerisierten Umgebung könnte die Skalierbarkeit und Verfügbarkeit weiter erhöht werden. Container-Orchestrierungsplattformen wie Kubernetes ermöglichen dabei eine dynamische Anpassung der verfügbaren Ressourcen an die aktuelle Last und unterstützen somit den elastischen Betrieb verteilter Clusteranwendungen.

Darüber hinaus stellt die Kombination von Echtzeit- und historischen Daten einen vielversprechenden Erweiterungsansatz dar. Durch die parallele Verarbeitung beider Datenquellen könnten umfassendere Analysen sowie detailliertere Visualisierungen der Telemetriedaten realisiert werden. Dies würde es ermöglichen, aktuelle Rennsituationen in den Kontext vergangener Ereignisse zu setzen und weiterführende Auswertungen durchzuführen.

Ein weiterer zukünftiger Entwicklungsschritt betrifft die Integration von Logging- und Monitoring-Lösungen zur kontinuierlichen Überwachung der Leistungsfähigkeit und Stabilität des Systems. Der Einsatz von Werkzeugen wie Prometheus zur Erfassung von Metriken sowie Grafana zur visuellen Aufbereitung dieser Daten könnte eine detaillierte Analyse des Systemverhaltens ermöglichen und die Wartbarkeit sowie den produktiven Betrieb des Systems nachhaltig verbessern.

Quellenverzeichnis

Literatur

Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4. Aufl.). Addison-Wesley. (Siehe S. 11).

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014a). The Reactive Manifesto [Version 2.0, veröffentlicht 2014]. (Siehe S. 1, 7, 8).

Hewitt, C., Bishop, P., & Steiger, R. (1973). A Universal Modular Actor Formalism for Artificial Intelligence. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, 235–245 (siehe S. 6).

Kuhn, R., Allen, J., & Hanafee, B. (2017). *Reactive Design Patterns: Designing Event-Driven Applications with Akka, Scala, and Play*. Manning Publications. (Siehe S. 4, 5).

Kuhn, R., Hanafee, B., & Allen, J. (2017a). *Reactive Design Patterns*. Manning Publications. (Siehe S. 1, 8–10).

Kuhn, R., Hanafee, B., & Allen, J. (2017b). *Reactive Design Patterns: Designing Event-Driven Applications with the Actor Model and Reactive Streams*. Manning Publications. (Siehe S. 6, 7).

Roestenburg, R., Bakker, R., & Williams, R. (2016). *Akka in Action* (2. Aufl.). Manning Publications. (Siehe S. 7–9).

Online-Quellen

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014b). *The Reactive Manifesto* [Abgerufen am 1. Februar 2026]. https://www.reactivemanifesto.org/ (siehe S. 4–6).

Petabridge. (2025a). *Akka.NET Cluster Sharding* [Abgerufen am 25. Oktober 2025]. https://petabridge.com/blog/distributing-state-with-cluster-sharding/ (siehe S. 15–17).

Petabridge. (2025b). *Akka.NET Distributed Publish Subscribe* [Abgerufen am 25. Oktober 2025]. https://petabridge.com/blog/distributed-pub-sub-intro-akkadotnet/ (siehe S. 18, 19).

Project, A. (2025a). *Akka.NET Cluster Overview* [Abgerufen am 30. November 2025]. https://getakka.net/articles/clustering/cluster-overview.html (siehe S. 7, 47).

Project, A. (2025b). *Akka.NET Cluster Sharding* [Abgerufen am 25. Oktober 2025]. https://getakka.net/articles/clustering/cluster-sharding.html (siehe S. 8, 15, 17).

Project, A. (2025c). *Akka.NET Cluster Singleton* [Abgerufen am 25. Oktober 2025]. https://getakka.net/articles/clustering/cluster-singleton.html (siehe S. 9, 17, 18).

Project, A. (2025d). *Akka.NET Distributed Publish Subscribe* [Abgerufen am 25. Oktober 2025]. https://getakka.net/articles/clustering/distributed-data.html (siehe S. 9).

Project, A. (2025e). *Akka.NET Official Documentation*. Verfügbar 20. Oktober 2025 unter https://getakka.net/articles/intro/what-problems-does-actor-model-solve.html (siehe S. 7).

Project, A. (2025f). *Akka.NET Persistence* [Abgerufen am 25. Oktober 2025]. https://getakka.net/articles/persistence/architecture.html (siehe S. 8).

Project, A. (2025g). *Akka.NET Streams - Relationship with Reactive Streams* [Abgerufen am 19. November 2025]. https://getakka.net/articles/streams/introduction.html (siehe S. 10, 19).

Project, A. (2025h). *Buffers and Overflow Strategies* [Abgerufen am 19. November 2025]. https://getakka.net/articles/streams/buffersandworkingwithrate.html (siehe S. 19, 20).

Skotzko, A. (2015). *Actor Composition Patterns* [Abgerufen am 19. November 2025]. https://petabridge.com/blog/top-akkadotnet-design-patterns/ (siehe S. 16).