PF3: Ave Caesar

Cyril Wagner

13. März 2025

1 Einleitung

Im Rahmen der Übungsaufgabe „Ave Caesar“ sollte ein verteiltes System realisiert werden, das mehrere Segmente einer Rennstrecke (im Sinne von Brettspiel-Feldern) digital abbildet. Die Kommunikation zwischen diesen Segmenten erfolgt asynchron über einen *horizontalskalierbaren* Streaming-Server (Apache Kafka). Ziel war es, mehrere virtuelle Streitwagen (entsprechend Tokens) durch die Segmente zu schicken und dabei verschiedene Erweiterungen des originalen Brettspiels (beispielsweise Engpässe, Caesar-Segmente, Aufspaltungen) zu berücksichtigen.

2 Problemstellung

Die Hauptanforderung bestand darin, eine Rennstrecke als JSON-Datei einzulesen und diese in ein *verteiltes System* zu überführen, in dem:

- Jedes Segment der Rennstrecke durch einen *eigenständigen Prozess* (Microservice) repräsentiert wird.
- Die Kommunikation zwischen den Segment-Prozessen ausschließlich über Publish/Subscribe-Themen (Topics) in Apache Kafka (bzw. einem Streaming-Framework) erfolgt.
- Start- und Ziel-Segmente zusätzliche Steuerkommandos per CLI entgegennehmen können.
- Eine *Containerisierung* mit Docker und eine Cluster-Fähigkeit sichergestellt wird, damit bei steigender Teilnehmerzahl (oder Segmentanzahl) horizontal skaliert werden kann.

In einer zweiten Ausbaustufe sollte das zentrale Kafka-System durch mindestens drei Broker-Knoten zu einem verteilten und ausfallsicheren Cluster ausgebaut werden. Darüber hinaus waren spezielle Segmenttypen wie „Caesar“ und „Bottleneck“ einzuführen, die das originale Brettspiel *Ave Caesar* virtuell weiter simulieren.

3 Implementierung

3.1 Architektur und Aufbau

- **Segment-Dienste:** Jeder Streckenabschnitt wurde als separater Python-Prozess realisiert, der beim Start sein jeweiliges Topic(s) abonniert. Eingehende Nachrichten (z. B. Positionen der Streitwagen) werden ausgewertet und anschließend an das nächste Segment weitergesendet.
- **Topic-Konfiguration:** Für jedes Segment wird in Kafka ein Topic nach dem Schema `<segment-name>` erstellt, sodass `Segment A` Nachrichten an `Segment B` publizieren kann, ohne direkten Netzwerkzugriff auf `B` zu benötigen.
- **CLI / Starter-Segment:** Das Startsegment (`start-and-goal`) akzeptiert ein CLI-Kommando, mit dem ein neues Token in den Datenstrom eingeführt wird. Hierbei wird eine Initialnachricht in das zugeordnete Kafka-Topic gesendet.
- **Erweiterungen (Bottleneck, Caesar):** Engpässe (Bottlenecks) verzögern die *Weiterleitung* eines Tokens zufällig. Das Caesar-Segment protokolliert zusätzlich eine „Grußnachricht“, bevor das Token weitergeleitet wird.

3.2 Kafka-Cluster und Docker

- **Kafka-Einbindung:** Für das Messaging wurde `apache/kafka` in einem `docker-compose` Setup verwendet. Dazu gehören `zookeeper` (im Single-Broker-Szenario) oder mehrere Kafka-Broker-Container (für die Cluster-Variante).
- **Skalierung:** In der zweiten Ausbaustufe wurden mindestens drei Kafka-Broker in einem Docker-Compose-Netzwerk bereitgestellt, sodass bei Ausfall eines Containers das System weiterhin lauffähig war. Zookeeper (bzw. in neueren Kafka-Versionen die interne Quorum-Implementierung) koordiniert den Controller und die Leader-Elections der Partitionen.
- **Containerisierung der Segmente:** Jedes Segment (z. B. `segment-1-1`, `segment-2-1` etc.) wird in einem eigenen Docker-Container gestartet. Für jedes Segment existiert ein Docker-Image (bzw. wird dynamisch erstellt), das die Python-Skripte und nötigen Abhängigkeiten enthält. Über Environment-Variablen werden die Namen der abonnierten Topics und die Adresse des Kafka-Brokers konfiguriert.
- **Fehlersuche:** Im Rahmen der Entwicklung traten typische Fehler auf, z. B. *Connection refused* beim Starten des Orchestrators, wenn

Kafka-Container noch nicht komplett `UP` war. Dies ließ sich durch Abhängigkeiten (`depends_on`) im `docker-compose.yml` sowie angepasste `restart: always`-Strategien minimieren.

4 Ablauf und Tests

1. **Start des Clusters:** Mit `docker-compose up -d` wird Zookeeper und Kafka (bzw. das Kafka-Cluster) gestartet.
2. **Start der Segmente:** Anschließend werden alle Segment-Container hochgefahren. Sie verbinden sich mit dem Kafka-Broker und lauschen auf ihren jeweiligen Topics.
3. **Einführen eines Tokens:** Über `docker exec` oder einen CLI-Aufruf beim Startsegment wird ein Startkommando abgesetzt, das eine Nachricht in `start-and-goal` publiziert. Daraufhin *wandert* das Token entsprechend der JSON-Definition durch alle Segmente.
4. **Logging und Timing:** Jeder Container protokolliert die Nachrichteneingänge, ggfs. Verzögerungen (Bottlenecks) und den Zeitpunkt des Weiterleitens. Am Ende wird die *Gesamtzeit* für das Token ausgegeben.

Bei möglichen Verbindungsfehler, manuell den Container des Orchestrators herunterfahren und neu starten.

5 Nutzung von KI

Bei der Durchführung dieses Projekts wurde auch die KI-basierte Text- und Code-Unterstützung durch ChatGPT (GPT-4) hinzugezogen. ChatGPT kam in mehreren Bereichen zum Einsatz:

- **Problemanalyse:** Die KI unterstützte bei der Aufbereitung der Aufgabenstellung sowie der Diskussion möglicher technischer Realisierungsansätze.
- **Beispiele und Code-Snippets:** An Stellen, an denen bestimmte Sprachkonstrukte oder Libraries in Python, Java oder einer anderen Sprache eingesetzt werden sollten, lieferte ChatGPT Codebeispiele und Kurzbeschreibungen. Diese Beispiele dienten als Inspiration und wurden anschließend auf die projekt- und umgebungsspezifischen Anforderungen angepasst.
- **Fehlersuche und Debugging:** Bei der Fehlersuche in Konfigurationsdateien (z. B. Docker-Konfigurationen für die Streaming-Cluster)

sowie bei der Deutung von Fehlermeldungen konnte ChatGPT generelle Hilfestellungen anbieten und mögliche Ursachen aufzeigen.

- **Dokumentation und Textkorrektur:** ChatGPT war außerdem nützlich bei der sprachlichen Ausformulierung einzelner Passagen für dieses Dokument. Dies beschleunigte den Prozess der Berichtserstellung und sorgte für eine einheitlichere Ausdrucksweise.

Die Verantwortung für den finalen Code und die Texte lag jedoch weiterhin bei uns Menschen. Alle Vorschläge von ChatGPT wurden manuell überprüft, überarbeitet und in das bestehende System eingebunden, um sicherzustellen, dass die Lösungen robust und passgenau sind sowie den Anforderungen des Projekts entsprechen.

6 Fazit

Das Projekt *Ave Caesar* zeigt anschaulich, wie ein verteilter, asynchroner Datenstrom mithilfe von Kafka und Docker-Mikroarchitekturen realisiert werden kann. Besonders die Einführung von Segmenttypen wie `Bottleneck` und `Caesar` demonstriert flexibel erweiterbare Logik. Durch die horizontale Skalierung im Kafka-Cluster lassen sich sowohl mehr Teilnehmer (Streitwagen) als auch mehr Segmente verarbeiten, ohne das Grundsystem ändern zu müssen.

Die Hauptherausforderungen waren:

- **Synchronisation von Docker-Containern:** Kafka und Zookeeper müssen *vollständig* initialisiert sein, bevor Segment-Services zuverlässig Nachrichten publizieren und konsumieren können.
- **Fehlerbehandlung:** „Connection refused"-Fehler sind häufig und erfordern Retries in den Python-Skripten.
- **Nachrichtenformate und Topics:** Eindeutige Identifikatoren pro Segment sind wichtig, um Überschneidungen oder Falschrouting zu vermeiden.

Insgesamt konnte die Aufgabe erfolgreich gelöst werden: Alle Segmente starten als Docker-Container, empfangen Token über Kafka und leiten diese an die nachfolgenden Segmente weiter. Mit Bottlenecks und Caesar-Segmenten wurde das *Originalspiel* in grundlegender Form abgebildet.

Links:
`https://github.com/mein-repo/ave-caesar` (Beispiel-Link zum Code-Repository)