Software Architectures for Enterprises – Ergebnisbericht Übungsblatt 3

Hakan Bayindir, Matrikelnummer: 1496384

Der zugrunde liegende Code befindet vollständig in einem Git-Repository:

https://github.com/Nooctis/SA4e_Caesar_Bayindir/tree/main

Zur technischen Installation und Ausführung des Programms siehe bitte Readme.md-Datei im Repository

1. Einleitung

Im Rahmen dieser Projektabgabe wurde eine Rennsimulation entwickelt, die sich an dem klassischen Spiel "Ave Caesar" orientiert. Ziel war es, zunächst eine einfache Implementierung zu realisieren, bei der ein Token (stellvertretend für einen Streitwagen) von Segment zu Segment weitergegeben wird. Darauf aufbauend wurde die Kommunikation über einen zentralen Messaging-Server durch den Einsatz eines skalierbaren Clusters ersetzt, und schließlich erfolgte eine Erweiterung der Simulation, die neue Segmenttypen wie "Caesar" und "Bottleneck" einführte. Der folgende Bericht beschreibt die Umsetzung der drei Aufgaben, die dabei verwendeten Technologien (insbesondere Python und Kafka) sowie die auftretenden Herausforderungen.

2. Umsetzung der Aufgaben

Aufgabe 1 „ Nur der Schnellste gewinnt“

Zu Beginn wurde ein Rundkurs konzipiert, der aus mehreren kreisförmigen Strecken besteht. Jede Strecke ist in Segmente unterteilt. Zur Generierung der Streckenbeschreibung kam ein Python-Skript (circular-course.py) zum Einsatz, das dynamisch ein JSON-Dokument erstellt. Dieses Skript wurde uns zur Verfügung gestellt.

In dieser Phase wurden folgende Komponenten realisiert:

- **Streckengenerator:**
  Das Skript generiert eine JSON-Struktur, in der für jeden Track ein "start-goal" Segment definiert ist, dem mehrere "normal" Segmente folgen. Die Logik sorgte dafür, dass das letzte Segment wieder auf das Start-/Ziel-Segment verweist. Diese Basisimplementierung ermöglichte es, die Strecke zu konfigurieren und als Input für die weitere Verarbeitung zu verwenden.

- **Segment-Implementierung:**
  Ein weiteres Python-Skript (segment.py) simulierte die Funktionalität der Segmente. Hier wurden alle Segmente als eigenständige Prozesse gestartet, die ihren Token – symbolisch durch einfache Druckausgaben und kurze Verzögerungen – an das nachfolgende Segment weiterleiten.
  Die Kommunikation erfolgte ursprünglich in einer simulierten Form, sodass das Token lediglich über Print-Befehle weitergegeben wurde.

- **Konfigurationsprozess:**
  Mit dem Skript config.py wurde die zuvor generierte JSON-Konfiguration eingelesen, und es wurden für jedes Segment separate Prozesse gestartet. Damit konnte der Ablauf einer kompletten Strecke simuliert werden.
  Diese erste Implementierung erfüllte die Grundanforderungen, indem sie einen einfachen Tokenfluss und die Basisstruktur eines Rennens darstellte. Herausforderungen bestanden vor allem darin, die Prozesse parallel laufen zu lassen und den richtigen Ablauf in der Weitergabe des Tokens zu steuern. Auch die Synchronisation des manuellen Starts per Eingabe (Enter) stellte sich als schwierig heraus, da die Prozesse parallel arbeiten.

Aufgabe 2: Cluster

Im zweiten Schritt sollte der zentrale Messaging-Server durch ein Cluster ersetzt werden, um Skalierbarkeit und Zuverlässigkeit zu erhöhen. Hier kam ein Kafka-Cluster zum Einsatz, das mithilfe von Docker Compose realisiert wurde.
Wichtige Aspekte dieser Umsetzung waren:

- **Kafka-Cluster-Setup:**
  Mithilfe einer docker-compose.yml wurden drei Kafka-Broker und ein Zookeeper in Containern gestartet. Dabei mussten Anpassungen vorgenommen werden, damit interne und externe Listener korrekt konfiguriert wurden.
  Ein Problem war beispielsweise, dass zunächst Konflikte bei den Ports und

falsche Advertised Listeners konfiguriert waren. Durch die Einführung von zwei Listenern pro Broker – einem internen (für die Kommunikation innerhalb des Clusters) und einem externen (für Clients von außerhalb) – konnte dieses Problem gelöst werden.

- **Anpassung der Kommunikation:**
  Die Kommunikation zwischen den Segmenten wurde von Redis auf Kafka umgestellt. Der ursprüngliche Code in segment.py wurde dahingehend angepasst, dass KafkaProducer und KafkaConsumer verwendet werden, um Token in den Kafka-Topics zu versenden und zu empfangen.
  Zu den Herausforderungen zählte hier insbesondere die richtige Konfiguration der Consumer-Gruppe, damit nur neue Nachrichten verarbeitet werden, sowie die Behandlung von Kafka-Offsets. Hier wurden Einstellungen wie auto_offset_reset='latest' und dynamisch generierte Gruppen-IDs verwendet, um unerwünschte Alt-Nachrichten zu vermeiden.

Aufgabe 3: "Ave Caesar"

Die dritte Aufgabe stellte die Erweiterung der Anwendung dar. Hier sollte die Simulation an das Originalspiel angelehnt werden, indem zusätzliche Segmenttypen eingeführt wurden:

- **Neue Segmenttypen:**
  Neben den bereits existierenden "start-goal" und "normal" Segmenten wurden zwei neue Typen eingeführt:
  - **Bottleneck:** Diese Segmente führen zu einer zufälligen Verzögerung (z. B. 2–5 Sekunden), um einen Engpass zu simulieren.
  - **Caesar:** In diesen Segmenten wird zusätzlich "Ave Caesar!" ausgegeben, und sie verzögern den Tokenfluss für eine längere, zufällige Zeit (z. B. 3–7 Sekunden).
- **Anpassung des Streckengenerators:**
  In circular-course.py wurde die Logik erweitert, sodass in jedem Track – sofern die Länge dies zulässt – zufällig ein "caesar"-Segment und mit einer bestimmten Wahrscheinlichkeit auch "bottleneck"-Segmente integriert werden.
  Diese Erweiterung ermöglichte es, abwechslungsreiche Strecken zu simulieren, bei denen nicht nur der Tokenfluss, sondern auch zusätzliche "Schikanen" berücksichtigt werden.
- **Erweiterte Segment-Logik:**
  Das Skript segment.py wurde weiter angepasst, um die neuen Typen zu

verarbeiten. Hier erfolgt je nach Segmenttyp eine entsprechende Verzögerung, bevor der Token an das nächste Segment weitergeleitet wird.
Ein Problem bestand darin, dass alle Prozesse parallel gestartet werden. Dadurch wurde der Tokenfluss bereits ausgelöst, bevor alle Startsegmente manuell aktiviert wurden. Dies führte zu einer zusätzlichen Herausforderung in der Synchronisation.
Um diesen Effekt zu mildern, wurde eine Methode implementiert, die die Kafka-Consumer so konfiguriert, dass sie nur auf neue Nachrichten (mittels auto_offset_reset='latest') reagieren, und in den start-goal Segmenten wird eine kurze Verzögerung eingebaut, bevor der Token gesendet wird.

3. Schwierigkeiten und Lösungsansätze

Kafka-Konfiguration:

Die Einrichtung des Kafka-Clusters war herausfordernd, da Fehler wie "NoBrokersAvailable" und Probleme mit den Listenern (z. B. Konflikte bei den Ports oder falsche Advertised Listeners) auftraten.

Lösung:
Durch die Trennung von internen und externen Listenern sowie durch das dynamische Generieren von Consumer-Gruppen wurden diese Probleme behoben.
Zudem musste auf die richtige Reihenfolge beim Hochfahren des Clusters geachtet werden.

Token-Weitergabe und Offset-Problematik:
Es traten Probleme auf, bei denen alte Nachrichten (Token) verarbeitet wurden oder der Tokenfluss ungewollt gestartet wurde, wenn die Consumer bereits alte Offsets hatten.

Lösung:
Die Verwendung von auto_offset_reset='latest' und einer dynamischen Gruppen-ID stellte sicher, dass nur neue Nachrichten verarbeitet werden.