Aufgabe 1 - Basis-Architektur

Erste Überlegungen zur Architektur

Zuerst habe ich mir überlegt, wie ich das Spiel *Ave Cäsar* auf eine Architektur aus Microservices und einer skalierbaren Datenverwaltung mit Apache Kafka abbilden kann. Dabei liegt es nahe, in Kafka zunächst für jedes Segment ein eigenes Topic anzulegen, für das dann im Folgenden verschiedene Producer und Consumer erstellt werden können.

Die Streitwagen werden dann jeweils durch eine Message repräsentiert, optimalerweise in Form eines JSON-Strings um Statusinformationen zu transportieren.

Jedes Segment wird dann von einem eigenen Microservice repräsentiert, der jeweils als Consumer auf dem ihm zugeordneten Kafka Topic agiert. So werden ankommende Streitwagen vom Service gefunden und können an das jeweils nächste Segment (bzw. eines der möglichen nächsten Segmente) durch einen Producer weitergeleitet werden, wo bereits der nächste Segment-Service bereitsteht. Eine Mechanik wie das Ziehen und Ausspielen von Karten wird nicht implementiert, da die Geschwindigkeit der Streitwagen hier allein von der Latenz der Weitergabe abhängt - Ein rundenbasiertes Spiel halte ich hier für wenig sinnvoll.

Für das Management des Systems muss außerdem ein zentrales Verwaltungsprogramm geschrieben werden, das den Streckenverlauf aus dem zuvor generierten JSON einliest und dann folgende Schritte unternimmt:

1. *Setup des Kafka-Servers* - Für jedes Streckensegment und für das Time-Tracking muss ein Kafka-Topic erstellt werden.
2. *Instantiierung der Microservices* - Für jedes Segment muss eine Instanz des Segment-Service erstellt werden, wobei die nötigen Informationen an das Programm weitergegeben werden müssen.
3. *Start-and-Goal Segmente* - Mit den Start-And-Goal-Segmenten muss in besonderer Weise kommuniziert werden, damit eine Ausgabe der Gesamtlaufzeit am Ende möglich ist. Es ist auch denkbar, die entsprechenden Programme erst bei Start des Rennens zu instantiieren. Ein zusätzliches Kafka-Topic sollte verwendet werden, um die Umlaufzeiten zu messen (es ist ja möglich, dass ein Wagen auf einem anderen Segment ankommt als jenes, von dem er gestartet ist).
4. *Start des Rennens* - Die Start-and-Goal Segmente müssen auf Kommando informiert/instantiiert werden, um die Streitwagen entsprechend ins Rennen zu schicken.
5. *Ausgabe der Ergebnisse* - Am Ende müssen die Umlaufzeiten der einzelnen Wagen ausgegeben werden.
6. *Aufräumen* - Die erstellten Prozesse müssen terminiert und die Kafka Topics gelöscht werden.

Implementierung

Bezüglich der Kafka-Konfiguration habe ich mich an obige Architektur gehalten: Für jedes Segment wird ein Kafka-Topic erstellt und es gibt ein weiteres Kafka-Topic `timetable`, über das die Start-Goal-Segmente über den Start des Rennens informiert werden und in die die Finish-Zeiten der einzelnen Streitwagen geschrieben werden.

Meine Implementierung besteht aus drei Komponenten:

1. Einer Bibliothek aus Daten-Klassen, die für den Umgang mit JSON-Objekten in der Kurs-Definition und der Messages in den Kafka-Topics zuständig sind.
2. Einem Micro-Service, der für jedes Segment der Strecke je einmal instantiiert wird und das Weiterleiten an die nächsten Segmente übernimmt.
3. Ein zentrales Programm, dass die Architektur in Kafka vorbereitet, die Micro-Services startet, das Rennen initiiert und überwacht, das Ergebnis ausgibt und schlussendlich die Micro-Services herunterfährt und die Architektur in Kafka wieder zurücksetzt.

Bibliothek aus Daten-Klassen

Hier habe ich zunächst die Klassen `CourseDefinition`, `Track` und `Segment` erstellt, die zusammen die JSON-Struktur repräsentieren, die von `circular_course.py` erstellt wird. Dazu kommt noch die Klasse `Chariot`, die einen Streitwagen repräsentiert und als Nachricht in die Kafka-Topics geschrieben wird. Bisher enthält die Klasse nur eine Streitwagen-ID und einen Rundenzähler, wird aber vermutlich in Aufgabe 3 noch erweitert werden. Zuletzt gibt es noch eine Klasse `TimetableEntry`, die die Nachrichten im `timetable`-Topic repräsentieren. Wo es erforderlich ist enthalten die Klassen je eine Methode `toJson()` und eine statische Methode `fromJson()`, die mithilfe von *Gson* die entsprechenden Konversionen von und nach JSON realisieren.

Segment-Service

Diese Komponente enthält das Interface `SegmentRoutine`, durch welches verschiedene Verhaltensweisen für die verschiedenen Segment-Typen erzielt werden. Für Aufgabe 1 stehen folgende Implementierungen zur Verfügung:

- `StandardSegment` repräsentiert das Segment vom Typ `normal`. Hier werden in der Hauptroutine mit einem Consumer die Streitwagen vom dem Segment zugeordneten Topic abgerufen und dann einfach zufällig an eines der nachfolgenden Segmente weitergegeben, indem sie per Producer in das entsprechende Topic geschrieben werden.
- `StartAndGoal` repräsentiert das Segment vom Typ `start-goal`. Es verhält sich weitestgehend wie das `StandardSegment` und erbt auch von diesem. Es wurden allerdings ein paar zusätzliche Features eingebaut:
  - Neben seinem eigenen Topic lauscht das Segment auch auf dem Topic `timetable`. Wird dort eine Nachricht gesendet, die den Start des Rennens markiert, wird ein neuer Streitwagen erstellt und auf eines der nachfolgenden Segmente befördert.
  - Wenn ein Streitwagen das Segment passiert, wird der Rundenzähler um eins erhöht
  - Hat der Streitwagen alle Runden absolviert, so wird er nicht weitergeleitet und stattdessen ein Eintrag in das Topic `timetable` geschrieben. Dieser Eintrag enthält die Systemzeit zum Zeitpunkt des Eintreffens, um eine Laufzeitmessung für die ganze Strecke zu realisieren.

Die `Main`-Klasse des Programms bekommt eine JSON-Repräsentation des Segments, für dass die Instanz verantwortlich ist, als Argument übergeben. Nachfolgend wird anhand des `type`-Attributs entschieden, welche Implementierung für die `SegmentRoutine` verwendet wird, und diese wird initialisiert und gestartet.

Admin-Tool

Das Admin-Tool bekommt als Argument einen Pfad zu einer JSON-Datei übergeben, in der sich die Kurs-Definition befindet. Diese wird aus dem JSON eingelesen und steht fortan als Objekt in Java zur Verfügung.

Dann werden über einen Kafka `AdminClient` alle notwendigen Topics erstellt: Je eines für jedes Segment und ein weiteres Topic `timetable` für die Zeiterfassung und Steuerung.

Als nächstes werden die Service-Prozesse gestartet und in einer Liste verwaltet, um diese später wieder beenden zu können.

Nachdem alle Prozesse erstellt wurden, kann das Rennen durch einen Druck der Enter-Taste gestartet werden. Dies geschieht, indem ein `TimetableEntry`-Objekt mit Typ `started` in das `timetable`-Topic geschrieben wird.

Nun wird außerdem ein Observer-Thread gestartet, der das `timetable`-Topic beobachtet und eine Meldung ausgibt, sobald ein Streitwagen (bzw. alle Streitwagen) im Ziel ist.

Nach einem weiteren Druck auf die Enter-Taste wird der Thread beendet, falls nicht schon geschehen, die Service-Prozesse terminiert und die Kafka-Topics wieder gelöscht.

Zeitmessung

Für den finalen Test habe ich einen Kurs mit zwei Spuren und je 10 Segmenten erstellt. Für diesen Kurs brauchten die Streitwagen jeweils etwa 500ms, wenn man vor Rennstart ein paar Sekunden wartet, bis Kafka das Erstellen der Topics vollständig verarbeitet hat. Wartet man nicht, so dauert es auch schon mal 7000ms.

Probleme bei der Implementierung

Ich bin bei der Implementierung auf einige Hürden gestoßen, von denen ich hier ein paar nennen möchte.

- Zunächst wurden die Einträge zeitversetzt in die Topics geschrieben, wodurch die Geschwindigkeit der Streitwagen erheblich litt. Hier war einige zusätzliche Konfiguration der Producer (insbesondere das Setzen des Parameters `linger.ms` auf `0`) notwendig, um die Latenzen in den Griff zu bekommen.
- Die Consumer waren zunächst auch nicht ganz korrekt konfiguriert, sodass sie nur die Nachrichten erhielten, die NACH ihrer Instantiierung geschrieben worden waren. Insbesondere beim Observer im Admin-Tool führte das regelmäßig dazu, dass der Zieleinlauf der Streitwagen "verpasst"wurde. Auch hier konnte ich das Problem durch ein paar Konfigurations-Optionen lösen.
- Das Jonglieren mit Prozessen in Java ist immer etwas lästig, vor allem wenn die Prozesse nicht korrekt beendet werden. Es brauchte etwas Aufwand, bis ich eine relativ robuste Lösung gefunden habe, die eine Beendigung der Prozesse zu jeder Zeit möglich macht und mich nicht zu regelmäßigen Aufräummaßnahmen (`killall -9 java`) zwingt. (Lösung: Observer in separatem Thread statt im Main-Thread, sodass der Main-Thread weiter auf Eingaben lauschen kann.) Außerdem habe ich zuerst das Ausmaß der Beispielstrecke viel zu groß gewählt, sodass mein Rechner beim Starten der vielen Prozesse eingefroren ist und ich einen Hard-Reset machen musste. Wenn die Services aber auf vielen Rechnern verteilt laufen würden, wäre das kein Problem. Theoretisch wären mit Kafka sogar

mehrere Instanzen des selben Service möglich, auf die die eingehenden Messages verteilt werden.

Aufgabe 2 - Cluster

Ab Version 4.0 enthält Apache Kafka ein eigenes Framework für die Cluster-Koordination. Mithilfe einer Docker-Compose-Datei habe ich drei Instanzen von Kafka erstellt und untereinander koordiniert. Um das Cluster zu nutzen, sind innerhalb des Codes theoretisch keine weiteren Änderungen nötig. Für optimale Redundanz übergebe ich beim Erstellen der Kafka-Clients eine Liste von Kafka-Knoten, damit beim Ausfall eines Knotens auf die anderen ausgewichen werden kann. Um das Potenzial voll auszuschöpfen, habe ich zusätzlich noch den Replikations- bzw. Partitions-Faktor der Topics erhöht, um die verteilte Architektur optimal zu nutzen.

Aufgrund spärlicher Dokumentation war das erstellen einer funktionierenden Compose-File gar nicht so leicht, und auch ChatGPT lieferte zunächst keine funktionierenden Lösungen, sondern eher Ansatzpunkte. Letztendlich konnte ich mit einer Kombination von KI und eigener Recherche das Cluster aber zum Laufen bringen.

Aufgabe 3 - Erweiterte Spielregeln

Als Erweiterung zur Basisversion des Spiels habe ich zwei neue Segmenttypen erstellt, die beide auf dem Standard-Segment aufbauen und kleine Änderungen mitbringen:

- `Bottleneck` wartet eine zufällige Zeit zwischen 100 und 500 Millisekunden, bevor der Streitwagen weitergeschickt wird.
- `Caesar` setzt beim Streitwagen im Vorüberfahren eine Variable `hasGreetedCaesar` auf `True`

Dank der geerbten Eigenschaften der `StandardSegment`-Klasse sind hier nur minimale Änderungen nötig gewesen.

In der `StartAndGoal`-Implementierung habe ich einen Check eingebaut, ob der Wagen nach Absolvieren der zweiten Runde Cäsar gegrüßt hat. Falls nicht, wird sein Versagen in das Timetable-Topic geschrieben und letztendlich vom Admin-Client ausgegeben.

Geänderter Rundkurs

Das Skript zur Generierung der Rennstrecke habe ich so geändert, dass parallel zu den Start-Ziel-Segmenten Cäsar sitzt, in einer Spur aus Bottleneck-Segmenten, um es etwas spannender zu machen. Außerdem sind die Bahnen nicht mehr fest, sondern die Wagen können jederzeit auf eine benachbarte Bahn wechseln. So sind sie dann überhaupt erst in der Lage, Cäsar zu erreichen.

Beobachtung

Da die Wagen in meiner Implementierung bisher nur durch den Zufall von einer Bahn auf eine andere steuern, ist es recht unwahrscheinlich, dass die Wagen im Laufe des Rennens in die Kaisergasse einfahren. Ich musste das Experiment einige Male starten, bevor dies überhaupt einmal geschieht. Naja, die Gladiatoren in der Arena müssen ja auch irgendwo herkommen.