Ergebnisbericht Übungsblatt 3 SA4E

Simon Haebenbrock, s4sihaeb@uni-trier.de, dc: SimonH (simon_7672)

Für die Bearbeitung des Übungsblatts 3 verwendete ich IntelliJ IDEA, da diese Entwicklungsumgebung eine hervorragende Integration für Java- und Kafka-Projekte bietet und somit die Entwicklungsarbeit deutlich erleichtert. Zur Verwaltung der Abhängigkeiten und des Build-Managements habe ich Apache Maven verwendet. Darüber hinaus verwendete ich Docker Desktop in Kombination mit Docker Compose via CLI, um sowohl einfache als auch verteilte Kafka-Setups mit mehreren Brokern und Zookeeper-Instanzen einfach und effizient orchestrieren zu können. Docker erwies sich dabei als sehr hilfreich, da es mir ermöglichte, konsistente Testumgebungen lokal aufzusetzen.

Architektur der Lösung

In **Aufgabe 1 („Fastest One Wins“)** entwarf ich ein verteiltes System, bei dem mehrere Spieler (als Streitwagen bezeichnet) parallel auf jeweils separaten Tracks fahren. Jeder Track bestand aus einer Reihe von Segmenten, welche in Kafka jeweils durch ein eigenes Topic repräsentiert wurden. Die Logik der Segmente lief jeweils in separaten Threads, die auf eingehende Kafka-Nachrichten reagierten. Die Spieler durchliefen ihre jeweilige Strecke linear und vollkommen unabhängig voneinander.

Die Zielsetzung der **Aufgabe 2 („Cluster“)** bestand darin, die zuvor entwickelte Architektur clusterfähig zu machen. Dazu erweiterte ich die Kafka-Infrastruktur auf ein Cluster mit insgesamt drei Brokern. Die Kommunikation innerhalb des Rennsystems wurde über mehrere Kafka-Instanzen verteilt. Dafür definierte ich die Kafka-Topics mit mehreren Partitionen und aktivierte eine entsprechende Replikation. Die zusätzlichen Broker sorgten für eine erhöhte Redundanz und verbesserten die Robustheit des Systems.

In **Aufgabe 3 („Ave Caesar“)** habe ich schließlich ein Konzept, bei dem alle Spieler gleichzeitig einen gemeinsamen Track mit kritischen Engpässen nutzten, da die zuvor separieren tracks bzw. dedizierten bahnen pro Spieler jeweils eigene Kafka Topics erzeugten und somit keine echten Bottlenecks repräsentierten. Die Hauptaufgabe lag dabei in der korrekten Synchronisation der Zugriffe auf diese gemeinsamen Bottleneck-Segmente, in denen es zu Verzögerungen und potenziellen Kollisionen kommen konnte. Diese Engpässe stellten eine besonders realistische Simulation von konkurrierenden Prozessen in verteilten Systemen dar.

Technologien & Laufzeitumgebung

Bei allen drei Aufgaben verwendete ich Apache Kafka als Event-Streaming-Plattform. Die asynchrone Kommunikation zwischen den Segmenten (Kafka-Consumer) und den Spielern (RaceToken) wurde mithilfe von Topics realisiert. Die Java-Anwendung lief als Konsolenprogramm und bezog ihre Konfiguration für die Rennstrecken aus einer externen JSON-Datei (tracks.json), welche dynamisch mit dem Python-Skript erzeugt werden

konnte. Diese Methode erlaubte mir schnelle Anpassung und Flexibilität bei der Generierung neuer Streckenlayouts für unterschiedliche Szenarien.

Zum Einsatz kamen folgende Technologien:

- Apache Kafka (Version 7.5.0, Confluent) für Messaging & Event-Handling
- Jackson ObjectMapper zur Verarbeitung und Verwaltung der JSON-Track-Dateien
- Kafka CLI Tools für die Verwaltung und Inspektion von Topics sowie des Kafka-Clusters
- Docker Compose zur Erstellung und Verwaltung lokaler Multi-Broker-Setups
- GitHub Copilot wurde zur Generierung von Boilerplate-Code und zur Vervollständigung von Kommentaren verwendet.

Durchgeführte Testszenarien & Beobachtungen

Aufgabe 1

Im ersten Testszenario dieser Aufgabe wurde ein einfaches Rennsystem mit zwei vollständig unabhängigen Tracks umgesetzt. Jeder Track bestand aus fünf Segmenten, wobei ein Spieler jeweils einen Track durchlief. Die Simulation wurde für drei Runden ausgeführt. Die Tokens der Streitwagen wurden nacheinander durch die Segmente weitergeleitet, und da keine Synchronisationspunkte existierten, verlief das Rennen vollständig parallelisiert. Die Laufzeit lag bei rund **3 Sekunden**, und beide Spieler erreichten zuverlässig das Ziel.

Im zweiten Szenario wurde die Segmentanzahl pro Track auf sechs erhöht. Auch hier waren die Tracks getrennt, und es gab keine gegenseitige Beeinflussung. Dennoch stieg die Laufzeit auf etwa **38 Sekunden**. Die Ursache liegt weniger in der Komplexität der Strecke, sondern vielmehr darin, dass mit jedem zusätzlichen Segment ein eigener Thread gestartet wurde – ein Faktor, der sich vor allem bei begrenzten Systemressourcen bemerkbar macht. Die Segment-Threads liefen stabil, die Tokens wurden korrekt weitergeleitet, und das Rennen wurde fehlerfrei abgeschlossen und auch die Kafka-Topics waren sauber strukturiert.

Aufgabe 2

In Szenario 1 bestand jede Strecke erneut aus fünf Segmenten, jedoch wurden nun alle Topics mit **drei Partitionen** und einem **Replikationsfaktor von 3** erzeugt. Während die Struktur identisch zu Aufgabe 1 war, stieg die Laufzeit der Simulation deutlich auf über **36 Sekunden**. Diese Verzögerung lässt sich vor allem auf den erhöhten Koordinationsaufwand zwischen den Kafka-Brokern sowie den Overhead durch Replikation zurückführen. Die Kafka-Logs zeigten hierbei, dass die Daten tatsächlich über alle Broker verteilt wurden.

Das zweite Szenario diente als Lasttest mit **vier Spielern**, jeweils mit acht Segmenten und drei Runden. Insgesamt wurden hierbei **über 30 Topics mit jeweils 3 Partitionen**

und 3 Replikas erstellt. Trotz dieser hohen Komplexität verlief das Rennen stabil und ohne Synchronisationsfehler. Die Laufzeit betrug pro Spieler etwa **31 Sekunden**, was angesichts der verteilten Struktur und der zusätzlichen Netzwerkkommunikation zwischen den Brokern sehr zufriedenstellend war.

Aufgabe 3

In diesem Szenario befuhren zwei Spieler einen gemeinsam genutzten Track mit sechs Segmenten. Das Segment `segment-2` verzweigte sich dabei in zwei mögliche Pfade – einer davon (`segment-3`) enthielt einen Bottleneck. Die Simulation wurde für drei Runden durchgeführt.

Der Ablauf zeigte eine klare Auswirkung der Bottlenecks auf die Spielverläufe: Streitwagen mussten an Engpässen zwischen **500 und 1500 Millisekunden** warten. Die Verzögerungen wurden durch `Thread.sleep()`-Aufrufe in den entsprechenden Segment-Threads realisiert. Dennoch konnten beide Spieler das Rennen zuverlässig beenden. Die Laufzeiten lagen bei ca. **34 Sekunden**, was angesichts der Wartezeiten und der gemeinsam genutzten Pfade zu erwarten war.

Erkenntnisse & Bewertung

Das entwickelte System zeigt eine hohe Modularität, Parallelisierung und eine effiziente Event-gesteuerte Architektur. Durch die Kafka-Architektur war es möglich, eine beliebige Anzahl an Spielern parallel fahren zu lassen, solange ausreichend Ressourcen wie Threads und Partitionen vorhanden waren. Die simulierten Bottleneck-Segmente ermöglichten zudem eine realistische Nachbildung von asynchronem Verhalten.

Die größte technische Herausforderung stellte die korrekte Cluster-Konfiguration dar, insbesondere bei Listener-Einstellungen (INSIDE vs. OUTSIDE), DNS-Auflösung innerhalb von Docker und der zuverlässigen Erstellung von Topics mit mehreren Partitionen via Java (ohne Autocreate).

Performancetechnisch bewies Kafka auch unter erhöhter Last Stabilität, wobei deutlich wurde, dass umfangreiche Partitionierung und Replikation die Systemreaktivität reduzieren. Tests mit mehr als vier Spielern wurden aus zeitlichen Gründen nicht durchgeführt, wären jedoch sinnvoll für weitere Evaluierungen.

Fazit

Ich habe das Streitwagenrennen erfolgreich mit drei unterschiedlichen Architekturkonzepten: vollständig parallelisiert (A1), verteilt über ein Kafka-Cluster (A2) und als gemeinsamer Track mit Synchronisationspunkten (A3) realisiert. Dabei konnte ich die Vorteile von eventbasierten Systemen mit Kafka hinsichtlich Skalierbarkeit, Echtzeitfähigkeit und Entkopplung demonstrieren, gleichzeitig wurde mir aber auch deutlich, welche Herausforderungen im Bereich der Konfiguration und Synchronisation bestehen.

Quellen

- Apache Kafka Documentation. https://kafka.apache.org/documentation/
- Docker Compose Docs. https://docs.docker.com/compose/
- Confluent Platform for Kafka. https://docs.confluent.io/platform/current/overview.html
- GitHub Copilot wurde zur Generierung von Boilerplate-Code und zur Vervollständigung von Kommentaren verwendet.

Anhang:

Anhang – Übersicht der getesteten Szenarien mit Konsolen- und Docker-Ausgaben

Aufgabe 1: Einfache parallele Tracks

Szenario 1: 2 Tracks, je 5 Segmente, 3 Runden

Docker-Container Übersicht:

```
CONTAINER ID   IMAGE                              STATUS
PORTS                          NAMES
b2f7e667adf4   confluentinc/cp-kafka:latest       Up 41
seconds   0.0.0.0:9092->9092/tcp         aufgabe1-kafka-1
b56263d84812   confluentinc/cp-zookeeper:latest   Up 41
seconds   2181/tcp, 2888/tcp, 3888/tcp   aufgabe1-zookeeper-1
```

Erzeugte Topics:

- start-and-goal-1, segment-1-[1 bis 4]
- start-and-goal-2, segment-2-[1 bis 4]

Ergebnisse:

- Streitwagen 1: 3091 ms
- Streitwagen 2: 3063 ms

Szenario 2: 2 Tracks, je 7 Segmente, 3 Runden

Erzeugte Topics:

- start-and-goal-1, segment-1-[1 bis 6]
- start-and-goal-2, segment-2-[1 bis 6]

Ergebnisse:

- Streitwagen 1: 38052 ms
- Streitwagen 2: 38024 ms

Aufgabe 2: Kafka-Cluster mit 3 Brokern

Szenario 1: 2 Tracks, je 5 Segmente, 3 Runden

Docker-Container Übersicht:

```
CONTAINER ID   IMAGE                                  STATUS
PORTS                                              NAMES
6bbde301db41   confluentinc/cp-kafka:7.5.0        Up 2 minutes
9092/tcp, 0.0.0.0:29094->29094/tcp             kafka3-a2
c4e9c66efad0   confluentinc/cp-kafka:7.5.0        Up 2 minutes
9092/tcp, 0.0.0.0:29092->29092/tcp             kafka1-a2
368a4ae1689d   confluentinc/cp-kafka:7.5.0        Up 2 minutes
9092/tcp, 0.0.0.0:29093->29093/tcp             kafka2-a2
b66f41171a84   confluentinc/cp-zookeeper:7.5.0    Up 2 minutes
2888/tcp, 3888/tcp, 0.0.0.0:22181->2181/tcp    zookeeper-a2
```

Erzeugte Topics: (jeweils 3 Partitionen, Faktor 3 Replikas)

- start-and-goal-1, segment-1-[1 bis 4]
- start-and-goal-2, segment-2-[1 bis 4]

Ergebnisse:

- Streitwagen 1: 36215 ms
- Streitwagen 2: 36191 ms

Hinweis: Laufzeit erhöht durch Cluster-Overhead.

Szenario 2: Lasttest mit 4 Spielern, je 8 Segmente, 3 Runden

Erzeugte Topics: (jeweils 3 Partitionen, Faktor 3 Replikas)

- start-and-goal-[1 bis 4], segment-[1 bis 4]-[1 bis 7]

Ergebnisse:

- Streitwagen 1: 3150 ms
- Streitwagen 2: 3146 ms
- Streitwagen 3: 3147 ms
- Streitwagen 4: 3144 ms

Hinweis: Stabile Performance unter erhöhter Last.

Aufgabe 3: Gemeinsamer Track mit Engpässen

Szenario 1: Gemeinsamer Track „shared“, 2 Spieler, 6 Segmente (davon 1 Engpass), 3 Runden

Docker-Container Übersicht:

```
CONTAINER ID   IMAGE                                  STATUS
PORTS                                              NAMES
c486419a015b   confluentinc/cp-kafka:7.5.0        Up 4 seconds
9092/tcp, 0.0.0.0:39092->39092/tcp             kafka-a3
```

```
ab23b41efb42   confluentinc/cp-zookeeper:7.5.0   Up 4 seconds
2888/tcp, 3888/tcp, 0.0.0.0:32181->2181/tcp   zookeeper-a3
```

Erzeugte Topics:

- start-and-goal, segment-[1 bis 5] (segment-3 als Engpass/Bottleneck)

Ergebnisse:

- Streitwagen 1: 34404 ms
- Streitwagen 2: 34372 ms

Beobachtung: Erfolgreiche Simulation und Verzögerung durch Engpässe.