Projektdokumentation AveCaesar RaceController

Lars Carpagne - 1693335

Übersicht

Das Projekt **AveCaesar RaceController** ist eine Simulation eines antiken Wagenrennens in dem mehrere virtuellen Streitwagen auf einer simulierten Rennstrecke gegeneinander antreten. Mithilfe einer verteilten Systemarchitektur werden Ansätze für die Steuerung und Verteilung der einzelnen Rennsegmente gebaut. Das Projekt basiert auf **Docker-Containern** und **Apache Kafka** für Skalierung und Kommunikation.

Aufgabenstellung und Ziele

Aufgabe 1 – „Nur der Schnellste gewinnt“

- Verwendung von Apacha Kafka als Messaging-System
- Einbindung eines Strecken-Generators
- Implementierung von Segmenten als Clients die über Kafka kommunizieren
- **Streitwagen (Token)** werden zwischen den Segmenten weitergereicht.
- Simulation eines Rundkurs-Rennens ausgehend von **Start-Goal**
- Einfache **CLI-Steuerung** zum Starten des Rennens.
- Erstellen und Starten von Docker-Containern

Aufgabe 2 – Cluster

- Ersatz des zentralen Streaming-Servers durch ein **Kafka-Cluster** mit mindestens drei Instanzen.
- Verbesserung von **Skalierbarkeit** und **Zuverlässigkeit**.

Aufgabe 3 – „Ave Caesar“

- Erweiterung der Rennsimulation um **neue Segmenttypen**:
  - **Caesar-Segment** für Interaktion mit „Caesar“-Logik.
  - **Bottleneck**-Segmente für Engpässe.
- Einführung von **Spielerrestriktionen** (begrenzte Spieleranzahl pro Segment) und **zufälligen Verzögerungen** in Engpässen.
- Implementierung von **Aufspaltungen in Streckenabschnitte** mit mehreren möglichen Nachfolgersegmenten.

Spezifikationen und Anforderungen

Technologie-Stack

1. **Programmiersprache**: C# (Version 13.0, .NET 9.0)
2. **Containerisierung**: Docker und Docker Compose
3. **Streaming-Plattform**: Apache Kafka (mit Zookeeper)
4. **Topologie**:
   - Rennstrecken-Segmente werden individuell als Docker-Container betrieben.
   - Kommunikation erfolgt über das Publish/Subscribe-Muster.

Rennsimulation

- **Rennstrecke**: besteht aus Segmenten, die Spielerbewegungen anhand von Token verarbeiten.
- **CLI-Initiierung**: Rennen starten und Befehle an die Segmente weiterleiten.
- **Endbedingung**: Sobald eine bestimmte Rundenzahl erreicht ist, wird das Rennen beendet und die Laufzeiten aller Streitwagen ausgegeben.

Ansätze zur Lösung

1. **Streaming-Architektur**:
   - Zentrale Kommunikationsebene mit **Apache Kafka**
   - **Themen (Topics)** und Nachrichten zwischen RaceController und Segmenten.

2. **Dockerisierung**:
   - Alle Dienste (RaceController, Kafka, Zookeeper) in isolierten **Docker-Containern** betrieben.

3. **Strecken-Generierung**:
   Es gibt 2 Möglichkeiten:
   - Aus C# heraus das Python Program als Prozess starten und das erzeugte JSON aus der Datei lesen
     ODER
   - Der Track-Generator wird nach C# portiert ohne die Logik zu verändern (außer später für Aufgabe 3) und das erzeugte Strecken Objekt liegt direkt im Arbeitsspeicher und spart sich den Schritt des Serialisieren/Deserialisieren mittels JSON

4. **Segment-Management**:
   - Jedes Segment wird als eigenständiger Kafka-Client im RaceController betrieben.
   - **Kafka-Listener** in jedem Segment verarbeitet Token und leiten sie an passende Segmente weiter.

5. **Weiterleitungslogik**:

- Tokens werden durch Segmente weitergereicht.
- Zusätzliche Segmenttypen erweitern die Simulation durch Restriktionen und Interaktionen.

6. **Cluster-Konfiguration**:
   - Einrichten eines Kafka-Cluster für höhere Verfügbarkeit und Redundanz.

Datenstrukturen

Spieler (Token)

Dient als zentraler Punkt zur Verfolgung der Spielerbewegungen über alle Segment-Kafka-Clients.

```
public class PlayerToken
{
    //(1) Spieler-ID
    public int PlayerID { get; set; }

    //(2) Daten zum Senden und Empfangen
    public string SenderID { get; set; }
    public string ReceiverID { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.Now;
    public DateTime SentAt { get; set; }

    //(3) Renninformationen
    public int CurrentLap { get; set; }
    public int MaxLaps { get; set; }
    public ulong TotalTimeMs { get; set; }

    //(4) Zusatz-Elemente
    //TASK 3: Number of Caesar Greets
    public int CaesarGreets { get; set; }
}
```

Segments

Umfasst Eigenschaften wie **Segment-Typ**, Spielerrestriktionen und Nachfolgersegmente.

```
public class TrackSegment
{
    public string SegmentId { get; set; }
    public string Type { get; set; } // normal, start-goal, caesar, bottlenec
k
    public int MaxPlayers { get; set; }
    public List<PlayerToken> CurrentPlayers { get; set; }
    public List<string> NextSegments { get; set; }
}
```

Umsetzung und Implementierung

1. **Segmentrestriktionen**:
   - Jedes Segment erhält eine **MaxPlayers**-Eigenschaft.
   - Kontrolliert, wie viele Spieler gleichzeitig im Segment erlaubt sind.

2. **Spielerbewegung**:
   - Spieler werden aus dem aktuellen Segment entfernt und hinzugefügt:

```
public void RemovePlayer(PlayerToken player)
{
    CurrentPlayers.Remove(player);
}
public bool AddPlayer(PlayerToken player)
{
    if (CanAddPlayer())
    {
        CurrentPlayers.Add(player);
        return true;
    }
    return false;
}
```

3. **Kafka-Integration**:
   - Konsumenten (Consumers) registrieren sich für bestimmte Themen:

```
var config = new ConsumerConfig
{
    BootstrapServers = "localhost:9092",
    GroupId = "race_controller_group",
    AutoOffsetReset = AutoOffsetReset.Earliest
};
```

   - Token werden verarbeitet und zum nächsten Segment weitergeleitet.

4. **Cluster-Setup**:
   - Nutzung eines Docker-Compose-Files für Kafka mit drei Brokern:

Ausführung

1. **Starten der Docker-Umgebung**:
   - Befehle zum Starten, Stoppen und Neustarten der Umgebung:

     ```
     docker-compose down -v
     docker-compose up -d –build
     ```

2. **Starten eines Rennens**:
   - Der CLI-Befehl zum Initialisieren eines einzigen Rennens (immer eins nach dem anderen; nur 1 Rennen gleichzeitig):

     ```
     echo "admin:start_race laps=2 segments=3 players=4" |
     docker exec -i kafka1 kafka-console-producer.sh --broker-list kaf
     ka1:9092 --topic race_api
     ```

   - `admin:` - ID für den Absender.
   - `start_race` - Befehl zum Starten des Rennens.
   - `laps=2` - Anzahl der Runden (hier 2)
   - `segments=3` - Anzahl der Segmente (hier 3)
   - `players=4` - Anzahl der Spieler (hier 4)
   - `race_api` - Kafka-Topic für die Rennsteuerung.
   - `docker exec -i` - Führt den Befehl im Kafka-Container aus.
   - `kafka-console-producer.sh` - Sendet Nachrichten an Kafka.
   - `--broker-list kafka1:9092` - Broker-Adresse und Port.

3. **Fehlerbehebung**:
   - Prüfen der laufenden Container:

     ```
     docker ps
     ```

   - Logs auslesen:

     ```
     docker logs <container_name>
     ```

Ergebnisse

- **Simulationsergebnis**: Nach Ende des Rennens werden die Gesamtlaufzeiten aller Spieler ausgegeben.


```
[
 {
  "playerID": 1,
  "senderID": "start-and-goal-1",
  "currentLap": 4,
  "maxLaps": 3,
  "sentAt": "2025-03-16T19:59:32.7435114+00:00",
  "totalTimeMs": 1909
 },
 {
  "playerID": 4,
  "senderID": "start-and-goal-1",
  "currentLap": 4,
  "maxLaps": 3,
  "sentAt": "2025-03-16T19:59:32.7435788+00:00",
  "totalTimeMs": 1910
 }
]
```


- **Cluster-Effizienz**: Die Aktualisierung auf ein Kafka-Cluster ermöglicht höhere Skalierbarkeit und Zuverlässigkeit. Der Code macht den Server performanter oder schlechter. Bei Hunderten von Rennen, Segmenten, Wagen und Runden werden mehrere GB an Arbeitsspeicher verbraucht bis sich alles wieder abgebaut hat.

- **Neue Segmenttypen**: Einführung von Caesar- und Bottleneck-Segmenten erhöht die Komplexität und Realitätsnähe.

Weitere Verbesserungen / Potenziale

- **Nachrichten-Größe und Inhalte**: Aktuell werden die Segment-Objekte als JSON Strings serialisiert, gesendet, empfangen, deserialisiert, aktualisiert und wieder zum Senden serialisiert.

  Dieses Design wurde gewählt, da die Anwendung auch als komplett Verteiltes System mit mehreren Service-Instanzen laufen könnte anstatt direkt in 1 Docker-Container.

  Solange alle Segment-Instanzen in der gleichen C# Anwendung / Service (RaceController) sind wäre es sinnvoller wenn Nachrichtengrößen dauerhaft verringert werden, um den Speicherverbrauch zu optimieren sowie Aufwand für das Serialisieren und Deserialisieren in der CPU zu eliminieren

- **Kommunikationen zwischen Clients**: Dadurch dass Kafka ein Messaging-System basierend auf Topics mit Subscribe/Unsubscribe Methodik ist, läuft die Kommunikation ähnlich eines Peer-2-Peer Kommunikationssystems. Alle Clients in einem Topic werden bei einer Nachricht benachrichtigt. Durch schlauere Struktur der Topics/Kanäle und auch Gruppierung der Clients kann der Datenverkehr reduziert werden

  Im Aktuellen Design wird eine Nachricht gesendet, jedes Segment prüft ob es der zuständige Empfänger ist und nur wenn ja , dann wird die Nachricht bearbeitet, ansonsten verworfen.

  Wenn man eine Rennstrecke als ein Kreis von N Segmenten sieht, dann könnte es auch N viele Kanten geben zwischen Segmente (0-1), (1-2), (2-3), (3-4) etc. So könnte ein Topic immer eine Kante zwischen 2 darstellen. Wenn aber ein Segment mehrere Nachbarn (X-viele) hat, dann würde sich die Anzahl der Topics um das X-fache erhöhen. Es wäre trotzdem eine Möglichkeit weiterhin, aber fragwürdig ob die Masse an Topics sinnvoll gewählt ist.

Aufgrund von Prüfungen und diversen Portfolio Prüfungen wurden diese Optimierungen hier nur beschrieben und nicht implementiert. Ich möchte damit zeigen, dass das Verständnis der Aufgaben da ist und noch viel Optimierungs- und Verbesserungspotenzial besteht.