# Ergebnisbericht SA4E Übung3

**(Marvin Mokelke, s4mamoke@uni-trier.de, MatrNr.: 1779643)**

Hinweis: Eine Anleitung zur Ausführung meiner Lösung befindet sich im README.me des Repository. Auf der letzten Seite dieser PDF finden Sie ein UML-Diagram zur Lösung der Aufgabe 3. Die UML-Diagramme zu den Aufgaben 1 und 2 sind Teilmengen dieses Diagrams (ohne die Caesar-, Bottleneck- und SplitSection Klassen).

Jeder Aufgabenordner enthält eine eigene docker-compose.yml, mit welcher der/die Kafka-Server gestartet werden.

**Erstellung der Rennstrecke (Track)**

Im Ordner *CuCuCo* befinden sich folgende Dateien:

- **TrackGenerator.py**: Fragt über die Konsole die Anzahl der Segmente pro Segmenttyp ab und generiert daraufhin die Rennstrecke.
- **MyTrackConfig.json**: Speichert die generierte Rennstrecke als JSON-Objekt. Bei Neuerstellung wird die bestehende Datei überschrieben.
- **ExampleTrackConfig.json**: Enthält eine vorkonfigurierte Beispielstrecke.
- **TrackVisualizer.py**: Plottet die Rennstrecke als Graph.

**Beschreibung der Segmenttypen**

Im Ordner *Sections* sind die verschiedenen Segmenttypen definiert. Jede Datei enthält eine Klasse zur Abbildung eines spezifischen Abschnitts der Rennstrecke. Die Segmente kommunizieren über Topics des Kafka-Servers miteinander. Jedes Segment besitzt eine eindeutige ID und kennt die ID seines Nachfolgers. Ein KafkaConsumer-Thread hört permanent auf neue Einträge im Topic. Sobald ein neuer Eintrag erscheint (die Daten eines Spielers), wird dieser an das nachfolgende Segment weitergeleitet (siehe Abb. 1).

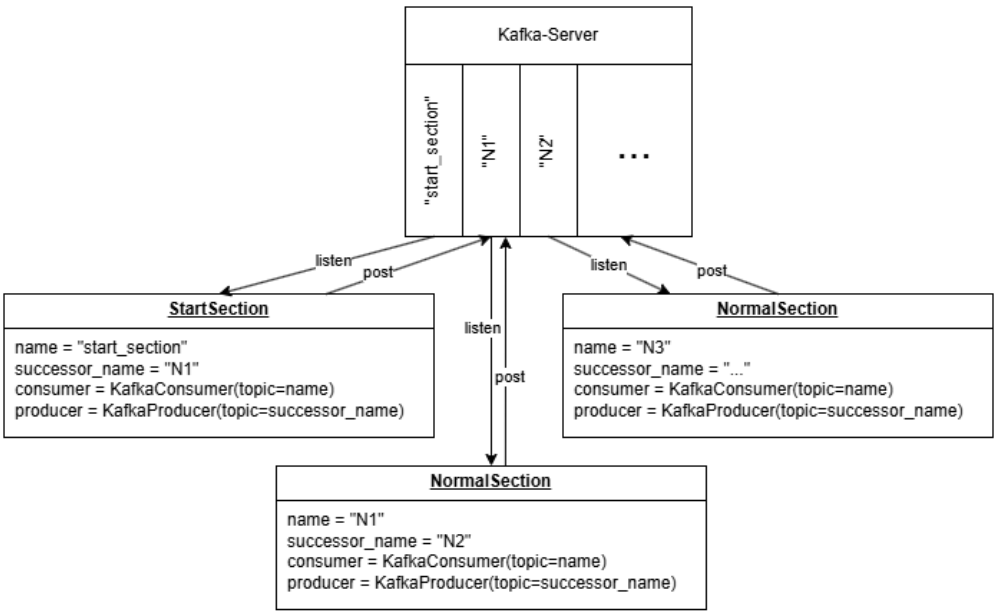


*Abb. 1: Funktionsweise Sektionskommunikation*

Zusätzlich zur grundlegenden Funktionalität besitzt jeder Segmenttyp spezifische Logikerweiterungen:

- **StartSection.py**: Registriert alle Spieler zu Beginn des Rennens und erstellt für jeden Spieler einen eigenen Thread. Über die Methode start_race() kann ein start_event ausgelöst werden, das alle Threads simultan startet.
- **NormalSection.py**: Leitet Spieler ohne Verzögerung an das nachfolgende Segment weiter.
- **CaesarSection.py**: Spieler müssen zufällig zwischen 1 und 5 Sekunden für Caesars niederknien, bevor sie weiterziehen dürfen. Es können mehrere *Caesar*-Segmente in einer Strecke existieren.
- **BottleneckSection.py**: Erlaubt nur einem Spieler gleichzeitig das Passieren. Die Durchgangszeit ist zufällig zwischen 1 und 3 Sekunden.
- **SplitSection.py**: Erlaubt eine Aufteilung des Weges in mehrere SubPaths. Dies ermöglicht komplexe Verzweigungen und Abkürzungen. Der Spieler entscheiden anhand einer gleichverteilten Zufallsvariable welchen SubPath er begeht. Der *TrackGenerator* aus Aufgabe 3 erstellt immer zwei SubPaths mit je drei *NormalSections*, bevor die Strecken wieder zusammengeführt werden. Die Datei ExampleTrackConfig.json enthält ein Beispiel für eine komplexere Konfiguration.
- **FinishSection.py**: Überprüft, ob ein Spieler die erforderliche Anzahl an Runden absolviert hat. Falls ja, wird die Gesamtzeit erfasst; andernfalls wird der Spieler zurück zur Startsektion verwiesen, um eine weitere Runde zu drehen.

Die Lösungen zu Aufgaben 1 und 2 enthalten wie gefordert ausschließlich *StartSection*, *NormalSection* und *FinishSection*.

**Umsetzung der Aufgaben**

Das Rennen kann mit SectionInitializer.py gestartet werden. Dazu wird eine der Konfigurationsdateien eingelesen und die notwendigen Segmente initialisiert. Zudem werden die Spieleranzahl und die Rundenanzahl per Konsoleneingabe festgelegt und an die Start- und Zielsektionen weitergegeben.

Der Rennstart erfolgt durch den Methodenaufruf start_race() der *StartSection*. Der gesamte Rennverlauf kann über die Konsolenausgabe verfolgt werden.

**Zu Aufgabe 1:**

Das Einrichten der Kafka- und Zookeeper-Server war dank vorgefertigter Docker-Images und Compose-Files sehr einfach.
Anfangs hatte ich jedoch alle Player in einer for-Schleife gestartet, wodurch sie immer in der Reihenfolge des Loops ins Ziel kamen. Um einen fairen Start zu gewährleisten, habe ich die Architektur auf Player-Threads mit einem gemeinsamen Start-Event umgestellt.

**Zu Aufgabe 2:**

Zu Beginn gab es einige Probleme mit der Abstimmung innerhalb des Kafka-Clusters. Alle Topics (Datenbereiche) werden auf allen drei Servern repliziert, wobei jedes Topic einen Leader hat, der die Replikation steuert. Im Normalfall übernimmt der Zookeeper-Server die Wahl eines neuen Leaders, falls ein Kafka-Server ausfällt.

Das Problem dabei: Fällt der zentrale Zookeeper-Server aus, versagt das System bei einem weiteren Ausfall eines Kafka-Servers, da kein neuer Leader für die betroffenen Topics bestimmt wird. Eine sichere Lösung wäre, einen weiteren Ersatz-Zookeeper-Server zu betreiben.

Stattdessen habe ich mich entschieden, den in Kafka integrierten Konsensmechanismus „KRaft“ zu nutzen. Dadurch erkennt das Kafka-Cluster selbstständig den Ausfall eines Nodes und wählt automatisch einen neuen Leader für die betroffenen Topics.

**Zu Aufgabe 3:**

Durch die zusätzlichen Section-Typen, insbesondere die SplitSections, können die TrackConfig-JSON-Dateien schnell unübersichtlich werden. Um dem entgegenzuwirken, habe ich den TrackVisualizer entwickelt.

Hier ein Überblick über die Logik der hinzugefügten Section-Typen:

- Caesar Section: Wartet zufällig zwischen 1 und 5 Sekunden, bevor sie eine Nachricht an die nächste Sektion weitergibt.
- Bottleneck Section: Prüft mit einer Instanzvariable, ob sich ein Spieler bereits auf dem Feld befindet. Falls ja, müssen alle nachfolgenden Spieler in einer while-True-Schleife warten, die alle 200ms überprüft, ob das Bottleneck frei ist.
- Split Section: Der SectionInitializer erzeugt die SplitSection-Instanz und übergibt ihr die definierten SubPaths. Die Sektionen der SubPaths werden dann von der SplitSection-Instanz initialisiert.

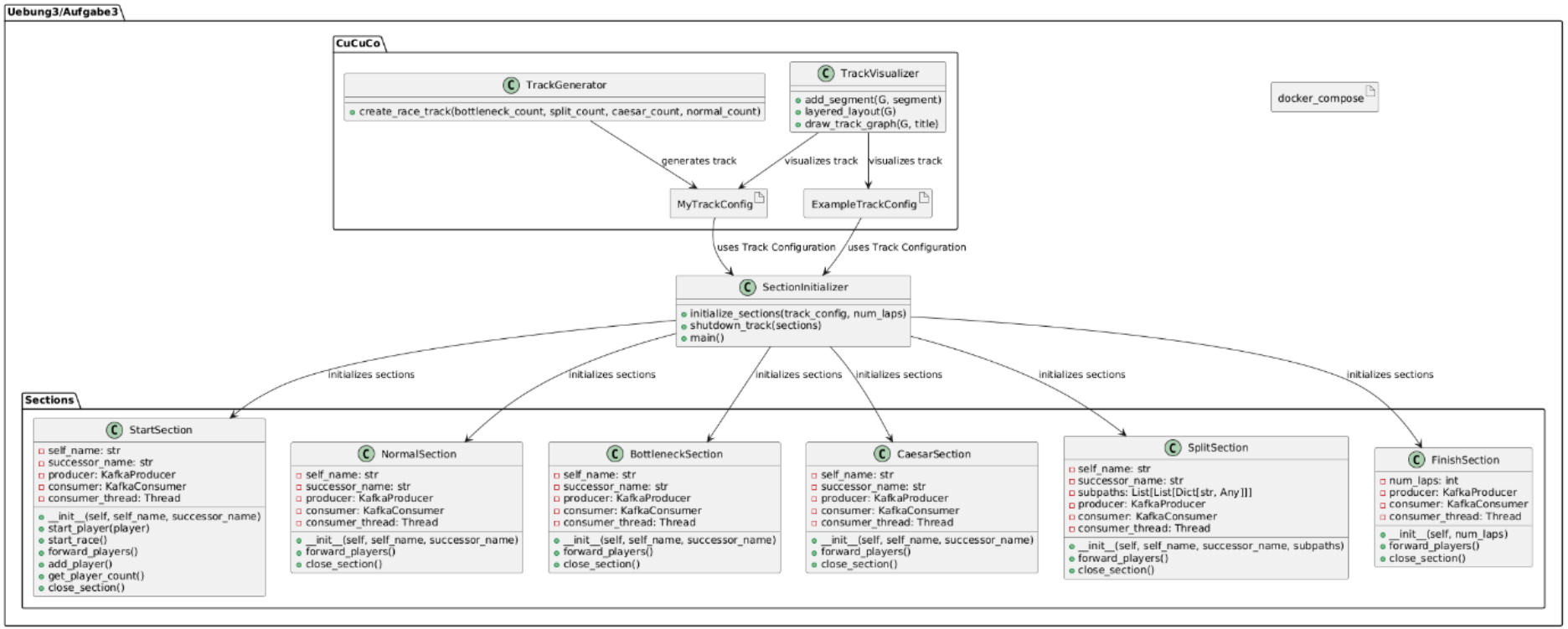


*Abb. 2: UML-Klassendiagramm zu Aufgabe 3*