Software Architecture For Enterprises
Übung 3

Aufgabe 1 „Nur der Schnellste gewinnt“

1. Ansatz

In der ersten Aufgabe bestand die Aufgabe darin, eine grundlegende Simulation eines Rundkurses zu erstellen. Dieser Rundkurs wurde durch eine Anzahl von kreisförmigen Bahnen definiert, wobei jede Bahn aus einer festgelegten Anzahl an Segmenten bestand. Jedes Segment wurde über eine einzigartige Segment-ID und eine Liste möglicher Folgesegmente (`nextSegments`) repräsentiert, die in einem JSON-Format definiert wurden. Der Rundkurs soll von der Datei “race_manager.py” aus generiert und gestartet werden. Die Kommunikation zwischen den Segmenten sollte mit Redis Streams umgesetzt werden.

2. Vorgehensweise

Die einzelnen Segmente werden vom race_manager erstellt. Dieser erstellt aus jedem Segment des Rundkurses einzelne Docker-Container mit Hilfe dessen, das Rennen simuliert wird. Der race_manager erstellt dabei sowohl die einzelnen Docker-Container für die Segmente (segment_program.py und Dockerfile.segment) als auch die Cluster von Aufgabe zwei von selbst und beendet diese auch nach dem Rennen selbstständig. Über client.hset werden die Nachrichten von den Segmenten als Streams an Redis versendet und in einem Redis-Hash gespeichert. Um ein realistisches Rennen zu simulieren, gibt es in jedem Segment eine kurze Verzögerungszeit, die zwischen 0,5 und 2 sekunden liegt. Zusätzlich wird in denr segment_program.py auch festgehalten, wie viele Runden bereits absolviert wurden. Damit keine zwei Token/Wagen in einem Segment landen können, werden diese gesperrt, damit es keinen Konflikt gibt.

Nach der Erstellung der Segmente wird dann von dem race_manager das Rennen gestartet, indem auf dem Start-and-Goal-Segment ein Token platziert wird.
Nach der Beendigung des Rennens, werden vom dem race_manager zusätzlich auch die Ergebnisse noch in der race_results.txt gespeichert und auch die finalen Zeiten ausgegeben. Während dem Rennen werden von dem race_manager nur alle 0.5 Sekunden die aktuellen Positionen ausgegeben. Ein kleiner Makel des Managers ist jedoch, dass dieser zum Beenden der Segmente durchaus etwas länger benötigt.

Redis wurde im Projekt als zentraler Baustein für die Kommunikation und Synchronisation zwischen den Segmenten eingesetzt. Dabei kam die Redis-Cluster-Funktionalität zum Einsatz, um eine skalierbare und fehlertolerante Infrastruktur bereitzustellen. Redis diente in erster Linie als Nachrichtenvermittler über Streams und als Speicherort für Renninformationen, wie die aktuellen Standorte der Streitwagen oder die Laufzeiten der Tokens.

Kommunikation zwischen Segmenten: Jedes Segment verfügte über einen eigenen Redis-Stream (`stream-{segment_id}`), auf dem es Nachrichten (Tokens) empfing. Die Tokens wurden in diesen Streams gespeichert und enthielten Informationen wie die Streitwagen-ID. Sobald ein Token ein Segment erreichte, wurde es über den entsprechenden Stream gelesen und verarbeitet. Nach der Bearbeitung wurde es an die Streams der Folgesegmente weitergeleitet.

Beispiel:

```
stream_name = f"stream-{segment_id}"
messages = client.xread({stream_name: "$"}, block=0)
for _, entries in messages:
    for entry_id, entry_data in entries:
        token = entry_data.get("token")
        client.xadd(f"stream-{next_segment}", {"token": token})
```

Speicherung der Standortinformationen: Die aktuellen Standorte der Tokens wurden in einem Redis-Hash (`token_locations`) gespeichert. So konnte zu jedem Zeitpunkt nachvollzogen werden, auf welchem Segment sich ein Streitwagen befindet. Diese Information war besonders nützlich, um die Rennfortschritte zu überwachen.

Beispiel:

```
client.hset("token_locations", token, segment_id)
```

Aufgabe 2 Cluster

Durch die Verwendung eines Clusters konnte gewährleistet werden, dass auch bei hohem Datenverkehr und einer großen Anzahl von Tokens keine Engpässe oder Ausfälle auftreten. Die Cluster-Konfiguration wurde in separaten Dateien (`redis-node-1.conf, redis-node-2.conf und redis-node-3.conf`) definiert, wodurch die Einstellungen für jeden Cluster-Knoten individuell angepasst werden könnten.

Die Konfiguration der Knoten ist wie folgt:
port 7001 (7002, 7003: der Port ist unterschiedlich für jeden Knoten)

cluster-enabled yes

cluster-config-file nodes.conf

cluster-node-timeout 5000

appendonly yes

Durch die Option “cluster-enabled yes” wird die Cluster-Funktion aktiviert. Die Option “appendonly yes” sorgt dafür, dass Daten dauerhaft gespeichert und auch nach einem Neustart des Clusters wiederhergestellt werden könnten, wobei dies jedoch nur zum Debuggen genutzt wurde.

Die Unterschiedlichen Ports führten anfangs zu einem Problem bei der Erstellung der Cluster, da die Knoten ansonsten nicht richtig im Cluster erkannt wurden.

Aufgabe 3 Ave Caesar

Für Aufgabe 3 wurde die neue Datei circular-course_AVE_CEASAR.py erstellt.
In dieser werden sowohl das Ceasar-Segment als auch das Bottleneck-Segment an unterschiedlichen Stellen der Bahn erstellt.
Ein Beispielhafter Track könnte folgendermaßen aussehen:


```
{
  "trackId": "1",
  "segments": [
    {
      "segmentId": "start-and-goal-1",
      "type": "start-goal",
      "nextSegments": [
        "segment-1-bottleneck"
      ]
    },
    {
      "segmentId": "segment-1-bottleneck",
      "type": "bottleneck",
      "nextSegments": [
        "segment-global-bottleneck"
      ]
    },
    {
      "segmentId": "segment-1-bottleneck-ret",
      "type": "bottleneck-ret",
      "nextSegments": [
        "segment-1-2"
      ]
    },
    {
      "segmentId": "segment-1-2",
      "type": "normal",
      "nextSegments": [
        "segment-1-3"
      ]
    },
    {
      "segmentId": "segment-1-3",
      "type": "normal",
      "nextSegments": [
        "segment-1-4"
      ]
    },
    {
      "segmentId": "segment-1-4",
      "type": "normal",
      "nextSegments": [
        "segment-1-caesar-link",
        "segment-1-caesar-ret"
      ]
    },
    {
      "segmentId": "segment-1-caesar-link",
      "type": "caesar-link",
      "nextSegments": [
        "segment-global-caesar"
      ]
    },
    {
      "segmentId": "segment-1-caesar-ret",
      "type": "caesar-ret",
      "nextSegments": [
        "segment-1-6"
      ]
    },
    {
      "segmentId": "segment-1-6",
      "type": "normal",
      "nextSegments": [
        "start-and-goal-1"
      ]
    }
  ]
},
```

Wie in der Abbildung zu sehen ist, werden die Bottleneck- bzw. Ceasar-Segmente über link und ret (return) an den Rest der Strecke angebunden. Die beiden Segmente selbst werden dann anschließend als “globalSegments” erstellt:

Aufgrund von Problemen mit dem Weiterfahren, nach einem Stop, sowie einer fehlenden Kontrolle des “Ave Ceasar” habe ich es jedoch nicht geschafft, die Fahrer das Rennen mit dieses Schikanen abschließen zu lassen. Hierfür wären erneut einige Änderungen im race_manager von nöten gewesen.