Projektplan

DSE-2021S Gruppe 08 - Eder, Eidelpes und Zeisler

1. Projektzieleplan

In diesem Projekt wird ein Szenario simuliert, in dem mehrere autonom gesteuerte Fahrzeuge auf einer Fahrbahn mit Kreuzungen und Ampeln unterwegs sind. Dabei kommunizieren sie periodisch mit einem Cloud System, das die Geschwindigkeit der Fahrzeuge auf Grund von Ampel Sichtweiten und Daten automatisiert optimieren kann. Dadurch soll erreicht werden, dass die Fahrzeuge mit der Geschwindigkeit fahren, mit der eine “grüne Welle” erreicht werden kann (keine Ampel ist bei Ankunft auf rot gestellt). Die Simulation kann auch ein Szenario umfassen, wo ein Fußgänger beinahe mit einem Auto zusammen stößt und das Auto ein NCE (Near Crash Event) auslöst und an das Cloud System meldet.

Ziele:

- Fahrzeuggeschwindigkeiten so anpassen, dass für die teilnehmenden Fahrzeuge eine grüne Welle erreicht wird.
- Das Fahrzeug kann trotz auftretendem “Near Crash Event” (NCE) eine grüne Welle erreichen.
- Im Frontend ist die Simulationsgeschwindigkeit dynamisch regelbar / skalierbar.

Nicht Ziele:

- Ort des NCE ist nicht konfigurierbar, findet an festgelegtem Ort statt. Es kann nur festgelegt werden, ob für das gegebene Fahrzeug ein NCE auftritt oder nicht.
- Ein NCE kann nicht durch das ineinander Fahren von Autos ausgelöst werden, da davon ausgegangen wird, dass jedem Fahrzeug eine eigene Fahrbahn zur Verfügung steht (mehrspurige Fahrbahn).
- Im Frontend sind abgesehen von der Simulationsgeschwindigkeit sonst keine Parameter regelbar.

2. Projektorganigramm

Rolle / Aufgabe	Name	Info
Testkoordinator	David Eder	Testabdeckung, Unit Tests, Integrationstests
Technischer Architekt / System Engineer	Marco Zeisler	Technologiewahl, System Design
Integration Engineer	Tobias Eidelpes	Deployment auf GKE, Docker Containers, Billing für Cloud Services

3. Work Breakdown Structure (WBS)

IFeed

IF1	Fahrzeug sendet periodischen Status Updates (DAF) via AMQP an Message Broker -> Message Broker verteilt via AMQP an Event Store & Orchestration (Exchange)
IF2	Ampeln senden bei Statuswechsel den aktuellen Ampelfarbenstatus via AMQP an Message Broker -> Message Broker sendet Ampel Event an Event Store Service ● a. Farbe: „Grün“ oder „Rot“ (der Einfachheit halber verzichten wir auf das Senden von „Gelb“) ● b. Status „Error“ im Fall eines Fehlers der Anlage (wenn sie noch Energie besitzt, um den Fehlerstatus zu senden, ein Solarpanel kann hier helfen) ● c. Zeitstempel des Statuswechsels
IF3	Fahrzeug erhält potentiell neue Richtgeschwindigkeit von Message Broker via AMQP ausgehend von Orchestration Service
IF4	Fußgänger kreuzt Fahrbahn? <ja/nein> Wenn ja, dann löst das Fahrzeug nach 2400m (Ampel 1 + 400m vor Ampel 2) ein NCE aus. Übermittelt via DAF {NCE: True}.

Message Broker

MB1	RabbitMQ Docker Image bereitstellen / einbinden / nutzen / konfigurieren

Orchestration Service

OS1	Holt sich Daten zur Initialisierung der RabbitMQ queues direkt via REST von Entity Ident (verfügbare Vehicle IDs)
OS2	Empfängt DAF Event von Fahrzeug(en) und prüft ob dieses in Reichweite von Ampel via Entity Ident Service via REST -> ignoriere wenn nicht sichtbar -> wenn sichtbar, retourniere Entfernung von Ampel zu Auto und Schaltintervall
OS3	wenn in Reichweite einer Ampel, -> hole letztes Event eben dieser Ampel (grün oder rot) via REST von Event Store -> berechne darauffolgend die Richtgeschwindigkeiten um Ampel in grün zu überfahren -> Sende Ergebnis via AMQP an Message Broker -> Message Broker verteilt via AMQP (Exchange) an Event Store & IFeed (Fahrzeug)
OS4	Stelle Daten für X-Way via simple Flask Webserver bereit

Event Store Service

ESS1	RedisDB zur Persistenz einbinden
ESS2	Subscribed auf alle RabbitMQ Exchanges und logged alle verteilten Nachrichten
ESS3	Stelle API für Ampel Events zur Verfügung
ESS4	Stelle Daten für X-Way via simple Flask Webserver bereit

Entity Ident Service

EIS1	MongoDB als Persistence Layer anbinden
EIS2	Speichert die Daten der verfügbaren Akteure (Ampeln & Fahrzeuge)
EIS3	Stelle API für Ampel und Fahrzeugdaten zur Verfügung
EIS4	Stelle API zur Verfügung um zu prüfen, ob Geo Daten von Auto in Sichtweite von Ampel, retourniere Entfernung und Schaltdauer wenn sichtbar
EIS5	Stellt Daten für X-Way via simple Flask Webserver bereit

X-Way API Gateway

XW1	Vereint / vereinfacht die verfügbaren APIs der Services und stellt via simple Flask Server eine Facade (API Gateway) für das Control Center bereit
XW2	Entity Ident Service Anbindung via REST
XW3	EventStore Service Anbindung via REST
XW4	Orchestration Service Anbindung via REST

Control Center

CC1	Angular Web App Setup
CC2	Beziehe Daten der Micro Services via X-Way REST API Gateway
CC3	Stelle Fahrzeuge in einer Web Oberfläche in einer Kartenansicht dar. Die Position wird dabei kontinuierlich aktualisiert.
CC4	Stelle eine Möglichkeit bereit, die Geschwindigkeit des Verkehrsflusses zu skalieren
CC5	Ein Klick oder Mouseover auf ein Fahrzeug oder eine Ampel zeigt die Stammdaten und auch aktuelle Daten zu dieser Entität an.

Deployment Lokal

DL1	Lokal mittels docker-compose deployen / testen

Deployment GKE

DC1	In cloud auf GKE deployen / testen

Weitere Dokumente

D1	Verfassen des Architektur & Design Dokumentes
D2	Verfassen des Wartungshandbuch
D3	Verfassen des Lessons Learned & Conclusio

4. Aufwandsschätzung

ID	Aufwand in h Eidelpes | Eder | Zeisler	Gesamtaufwand
IF1	5 | 0 | 0	5h
IF2	3 | 0 | 0	3h
IF3	1 | 0 | 0	1h
IF4	2 | 0 | 0	2h
MB1	1 | 0 | 0	1h
OS1	0 | 2 | 0	2h
OS2	0 | 2 | 0	2h
OS3	0 | 10 | 0	10h
OS4	0 | 4 | 0	4h
ESS1	0 | 0 | 1	1h
ESS2	0 | 0 | 5	5h
ESS3	0 | 0 | 2	2h
ESS4	0 | 0 | 4	4h
EIS1	0 | 0 | 1	1h
EIS2	0 | 0 | 5	5h
EIS3	0 | 0 | 3	3h
EIS4	8 | 0 | 0	8h
EIS5	5 | 0 | 0	5h
XW1	1 | 1 | 1	3h
XW2	1 | 0 | 0	1h
XW3	0 | 1 | 0	1h
XW4	0 | 0 | 1	1h
CC1	0 | 5 | 0	5h
CC2	0 | 5 | 0	5h
CC3	6,6 | 6,6 | 6,6	20h
CC4	2 | 2 | 4	8h

CC5	4 | 2 | 2	8h
DL1	3,3 | 3,3 | 3,3	10h
DC1	6,6 | 6,6 | 6,6	20h
D1	3,3 | 3,3 | 3,3	10h
D2	2 | 2 | 2	6h
D3	1 | 1 | 1	3h
Gesamtaufwand	55 | 55 | 55	165h

5. Ressourcenplanung und Meilensteinplanung

Die Aufwandsschätzung definiert wer welches Paket implementiert (Siehe Stundensätze pro Person). Wann die Pakete zu implementieren sind, wird nicht genauer definiert, da wir Agil arbeiten. Jede Person hat ihre Zuständigkeiten und ist dafür verantwortlich, dass die untenstehenden Meilensteine erreicht werden:

Projektplan fertig 18.04.2021

Draft Architektur & Design Dokument 02.05.2021

50% Features umgesetzt - 18.05.2021

Acceptance Testing - 100% Features umgesetzt - 11.06.2021

Alle Dokumente fertig gestellt - 15.06.2021

Projekt Deadline: 18.06.2021

6. Technische Planung

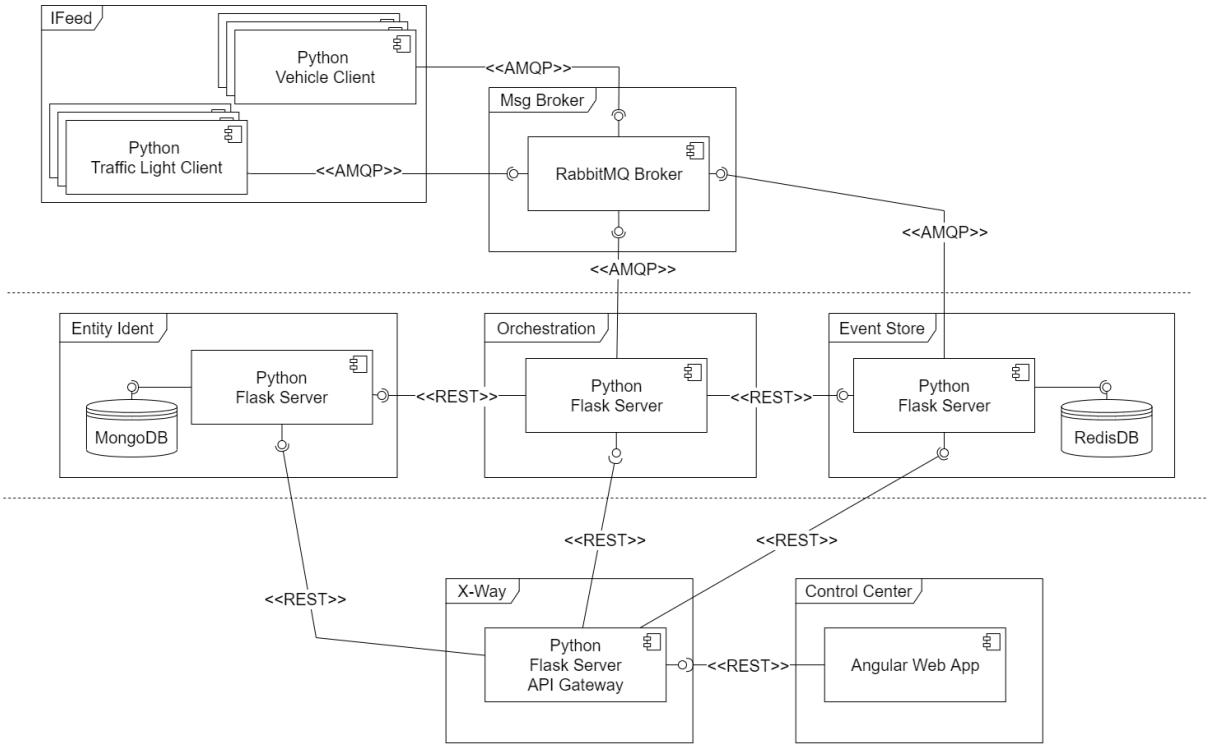

Abbildung 1 - System Architektur Entwurf

Als Container Technologie wird Docker[1] verwendet.

Prinzipiell werden die verschiedenen Komponenten und Akteure in Python 3 programmiert.

Einzig das Control Center wird mit Angular[2] entwickelt.

Der Message Broker wird mit RabbitMQ[3] realisiert.

Die Microservices (Event Store, Entity Ident und Orchestration) und IFeed Akteure (Vehicle und Ampel) nutzen

- Python pika[4] um asynchron via AMQP mit dem RabbitMQ broker zu kommunizieren.
- Python circuitbreaker[5] für Fault Tolerance

Die Microservices und X-Way stellen ein jeweiliges REST interface via Flask[6] zur Verfügung, einem lightweight Python Server, der sich ideal für Micro Services eignet[7].

---

[1] https://www.docker.com/
[2] https://angular.io/
[3] https://www.rabbitmq.com/
[4] https://pypi.org/project/pika/
[5] https://pypi.org/project/circuitbreaker/
[6] https://pypi.org/project/Flask/
[7] https://cloudacademy.com/course/mastering-microservices-with-python-flask-docker-1118/course-introduction/

Das API Gateway (X-Way) vereint die REST Schnittstellen der Micro Services und vereinfacht diese in einer abstrahierten REST Schnittstelle für das Control Center mittels eines weiteren Flask Servers.

Für die Implementierung der Modultests wird Pytest[8] verwendet.

7. GCP Budget Schätzung

Zonal Cluster kosten je nach Region (US Iowa) nur $0.10/h und es gibt ein $74.40 gratis Budget. Der GKE Cluster an sich kostet nichts, wenn er in Iowa steht. Eine f1-micro Instanz ist immer gratis. Leider sind f1-micro sehr schwach (vCPU: shared; RAM: 0.6GB), weswegen das Frontend mit einem einfachen Webserver auf dieser laufen wird. Alle anderen Container werden auf einem Pool aus acht n1-standard-1 Instanzen (1 vCPU; 3.75GB RAM) laufen. Zusätzlich wird eine Ingress Forwarding Rule benötigt, damit Zugriff auf das Frontend von außerhalb möglich ist. Egress Traffic wird geschätzt 1GiB umfassen, mit Platz nach oben, da dieser sehr billig ist.

Es wird angenommen, dass die Instanzen nicht länger als zwei Stunden pro Tag laufen. Das ergibt 14 Stunden pro Woche, die der Cluster zu Entwicklungszwecken laufen kann. Akkumuliert über mehrere Wochen sollte dieser Zeitraum ausreichen um die nötige Konfiguration vorzunehmen. Die restliche Entwicklung wird lokal am eigenen Rechner durchgeführt.

Insgesamt rechnen wir auf zweieinhalb bis drei Monate (Halber April + Mai + Juni) und kommen damit auf Gesamtkosten von $105 bis $125. Damit sind wir im Rahmen des zur Verfügung gestellten Guthabens von $150 und haben Luft nach oben sollten zusätzliche Cloud Services benötigt werden.

$74.40 Free Credits Zonal Cluster

$0.10/h Cluster Fee (free)
$18.26 Load Balancer 1 Forwarding Rule
$0.0072/h f1-micro 1x (vCPU: shared; RAM: 0.6GB) ← free for 28h/day
$0.045/h n1-standard-1 8x (vCPU: 1; RAM: 3.75GB)

$41.92/month cluster running 2h/day 7 days/week

[8] https://pypi.org/project/pytest/