Deployment of Autopilot Car Sharing Service and requirements within IBM Cloud Cluster

Project Report

from the Course of Studies Applied Computer Science

at the Cooperative State University Baden-Württemberg Mannheim

by

Pascal Schroeder

11/03/2019

Time of Project	19.11.2018-08.02.2019
Student ID, Course	5501463, TINF16AI-BC
Division, Business Unit	IBM Research
Company	IBM Deutschland GmbH, Dublin
Supervisor in the Company	Martin Mevissen

Signature Supervisor ______________________________

Confidentiality Statement

The Project Report on hand

Deployment of Autopilot Car Sharing Service and requirements within IBM Cloud Cluster

contains internal resp. confidential data of IBM Deutschland GmbH. It is intended solely for inspection by the assigned examiner, the head of the Applied Computer Science department and, if necessary, the Audit Committee at the Cooperative State University Baden-Württemberg Mannheim. It is strictly forbidden

- to distribute the content of this paper (including data, figures, tables, charts etc.) as a whole or in extracts,
- to make copies or transcripts of this paper or of parts of it,
- to display this paper or make it available in digital, electronic or virtual form.

Exceptional cases may be considered through permission granted in written form by the author and IBM Deutschland GmbH.

Mannheim, 11/03/2019

Pascal Schroeder

Contents

Declaration of Sincerity

Hereby I solemnly declare that my project report, titled *Deployment of Autopilot Car Sharing Service and requirements within IBM Cloud Cluster* is independently authored and no sources other than those specified have been used.
I also declare that the submitted electronic version matches the printed version.

Mannheim, 11/03/2019
Place, Date

Signature Pascal Schroeder

Abstract

1 Introduction

1.1 Introduction to 'Autopilot' project

The AUTOPILOT project is a LSP (Large-Scale-Pilot) by the European Commission. Those LSPs include five pilots as well as two coordination and support actions. Thereby the AUTOPILOT project takes care of using IoT (Internet of Things for connecting autonomous driving to a larger environment like traffic monitoring systems, car parks, traffic light radars or just other vehicles.cmp.[1]

In this project 45 different partners from 15 different countries across Europe are working together in four different sectors - development of autonomous driving vehicles, development of IoT and networks, collection of data to evaluate the systems and their potential impacts and organisations that will use the results for developing innovative services.cmp.[2],[3]

This project not only connects different sensors, cameras and vehicles to the IoT Platform, but also includes autonomous driving modes like automated valet parking, car sharing or a highway pilot, as well as driving services like route optimisation, chauffeur services or electronic driving licenses and many more.cmp.[4]

In the IBM Research Lab the Optimization & Control team is working on implementing different IoT platforms and networks for creating a large environment with many vehicles, road sensors and infrastructure facilities. This solves the limitations usual on-board sensors cause, because it expends the point of view from the vehicles view only to a far larger environment. This approach enables possibilities like anticipating situations upfront, minimize risks like traffic jam or accidents and all in all improving safety and comfort while reducing costs through a better organized and connected traffic.cmp.[5]

One part of this is a car sharing service for autonomous driving vehicles including a scheduler, which matches the customers to its target and calculates the best fitting vehicle and optimizes the route. Thereby it also includes traffic information and recalculate the route dependent on traffic jams, accidents or other environmental changes.

For proving the functionality of this scheduler it is necessary to test it in different environments - first in a real-life environment with working, autonomous driving cars and second in a large scalable simulation for proving its functionality in a large environment with many cars and customers.

This also needs to be based on a reliable and scalable infrastructure with enough resources to execute all the necessary calculations. One modern approach for this are cluster systems, of which advantages will be described in chapter 1.2.

1.2 Advantage of cluster systems

Cluster systems are two or more computers connected to each other, so that they can share their resources and can be viewed as one system. Therefore they can be connected physically as well as virtually. Using such a cluster system is typically much more cost-efficient compared to using a single computer, which provides about the same power. But the advantage of such systems doesn't end with a better cost-efficiency.cmp.[6]

First clusters are not only used for one specific purpose, but there are different kinds of cluster configurations, which are build for different objectives.

On the one hand, there are "Load balancing" cluster configurations. Load balancing clusters are clusters with the objective to provide better computing performance, like minimizing response time and maximizing throughput. Therefore it splits the computation in different parts and runs them parallel on different nodes. Through that the capabilities of the systems can be combined.cmp.[7]

Furthermore there are "High availability" clusters, which are designed to prevent a total break down of the system. For that there are several redundant nodes connected to the system. That means, that when one component fails, a redundant node of this component is used to provide the service. Thereby availability is ensured even if one component fails. The more redundant nodes are used the better will be the availability of a system. To provide an example: If a system has an availability of 90% it has a downtime of 10%, which means that the system is not available on 36.53 days per year. If there is a second, redundant and independent node connected to the cluster, which acts as a stand-in for the first system in case of a failure, the downtime of those two systems has to be combined. That means that this cluster would only fail if there is an overlap between the downtimes of those two systems. Resulting of a downtime of 10% of each system there is a combined downtime of 1% or 3.65 days per year. If there is a third system connected to the cluster, the downtime will be reduced to 0.1% or 8.77 hours per year and so on. The job of "High availability" clusters is in that to reduce the downtime of the system and to provide a working system almost any time.cmp.[8]

Figure 1.1 shows an example architecture of such a "High availability" cluster. There are two servers, whereby one server is a replication of the other one. The request router handles the incoming requests and leads them to the active server. If there is a system

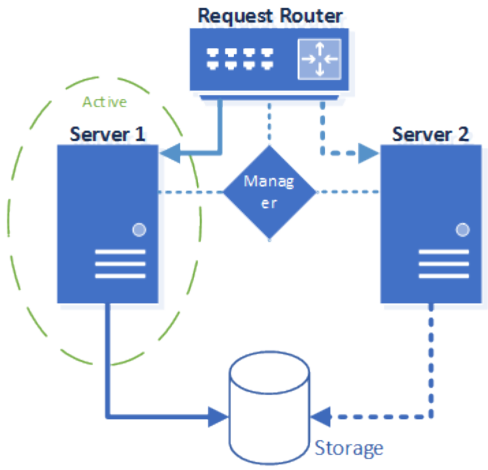

Figure 1.1.1 “High availability” cluster architecture

failure of the active server, the manager will recognize that, and sends a message to the request router to change the request address to the other server. From now on this server will deal with all the requests. Also there is a shared storage for both of the servers in one database. No matter which one is active they can both access this database.cmp.[9]

But those advantages don’t have to stand alone, but they can also be combined, so that you can provide high availability as well as high performance. Because there is an continuously increasing amount of calculations to be made cluster systems are becoming more and more important in modern IT for providing a solution for both problems.

1.3 Project objective

The objective of this essay is first to describe the car-sharing part of the AUTOPILOT project and all its underlying technologies. Those include Apache Kafka, OSRM (Open Source Routing Machine and MongoDB. Also the IoT Platform will be described roughly. For understanding the need of deploying these apps inside a cluster, the Kubernetes cluster solution will be portrayed.

In the next step the it will be explained how to deploy the car sharing service and its dependencies on a cluster and how a communication between them can be established. For that first it will be described how this project can be setup locally and afterwards how to containerize those applications with Docker, before the deployment itself will be explained. Afterwards it will be shown how this setup can be tested with a simulation running in a locally hosted UI (User Interface).

Last those results will be shown and discussed based on previously determined criteria.

2 Theory

2.1 Docker

2.2 Kubernetes cluster solution

One very common system for managing cluster systems as described in chapter 1.1 is Kubernetes - or short K8s. Originally developed by Google and now maintained by the Cloud Native Computing Foundation, Kubernetes is an “open-source platform for managing containerized workloads and services”.cmp.[**12**] For unterstanding what that means the concept of containers needs to be described first.

Containers are isolated, stand-alone packages of software, similar to processes. In those packages everything is included, which this piece of software needs, like runtime, libraries, settings and other system tools. These containers have a completely different environment within themselves than outside. This environment includes for example network routes, dns settings and control group limits. This enables the possibility to share common resources and still be isolated from any other process as well as the host system. Thereby containers are always working the same, no matter on what system they run or in which environment.cmp.[**13**], [**14**]

Kubernetes is for an automating deployment, scaling and management of these containers within a cluster of nodes. Thereby a cluster consists of at least one master node and any number of worker nodes. Figure 2.1.1 shows the different services owned by master and worker nodes.

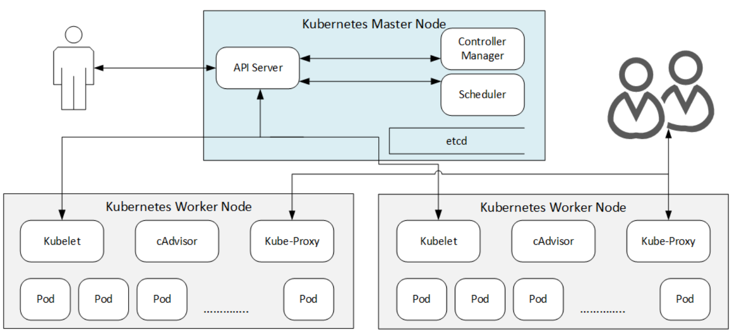

Figure 2.1.1 Kubernetes allocation of services

Based on retrieved information from [**13**]

First there are several pods on each worker node. Pods are the smallest unit in Kubernetes. They contain one or more containers, which are deployed together on the same host. Thereby they can work together to perform a set of tasks.cmp.[15]

On the master node there are an API (Application Programming Interface) Server, a Controller Manager, a Scheduler and a key-value store called etcd.cmp.[13], [16]

The API Server is for clients to run their requests against. That means the API Server is responsible for the communication between Master and Worker nodes and for updating corresponding objects in the etcd. Also the authentication and authorization is task of the API Server. The protocol for the communication is written in REST (Represantational State Transfer). For reacting on changes of clients there is also a watch mechanism implemented, which triggers an action after some specific changes, like the scheduler creating a new pod. This workflow is showed in figure 2.1.2.cmp.[13], [16]

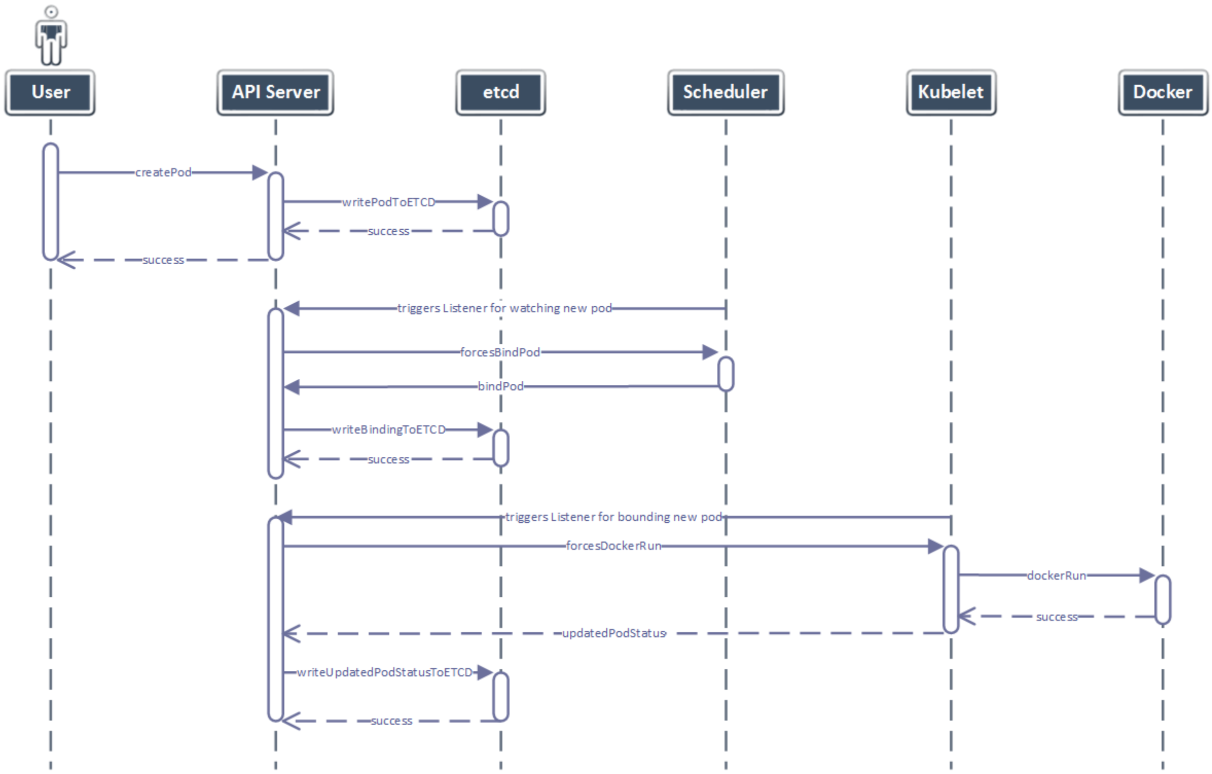

Figure 2.1.2 Kubernetes API Server watcher sequence diagram
Based on retrieved information from [16]

There you can see the user creating a pod through a belonging request to the API Server. The API Server writes this change to the etcd. Afterwards the API Server recognizes a new pod in the etcd and invokes the Scheduler to create this new pod. What is the exact task of the scheduler is described later. After successfully creating and binding the new pod, the API Server writes this change to the etcd. Because of this change within the etcd the API Server invokes the kubelet, which is also described later in this chapter, of the corresponding node. This kubelet starts docker to create the containers of the pod.

Kubelet responses with the new status of the pod, which is then written to the etcd by the API Server. After that the creation of the new pod is successfully finished.cmp.[13], [16]

The Controller Manager is a daemon, which embeds all of the Kubernetes controller. Examples for them are the Replication Controller or the Endpoint Controller. Those controllers are watching the state of the cluster through the API Server. Whenever a specific action happens, it performs the necessary actions to hold the current state or to move the cluster towards the desired state. For providing an example: If the Replication Controller recognizes, that one replication of a pod has been destroyed for some reason, it will take care of triggering the creation of a new replication.cmp.[13], [16]

The scheduler manages the binding of pods to nodes. Therefore it watches for new deployments as well as for old ones to create new pods if a new deployment is created or recreating a pod whenever a pod gets destroyed. The scheduler organizes the allocation of the pods within the cluster on the basis of available ressources of the pods. That means, that it always create pods, where the most resources are available, or reorganize the allocation if there is a change in the resource allocation of the cluster.cmp.[13], [16]

Figure 2.1.3 shows the way the Scheduler works, when new nodes are connected. As long as there are only two nodes and four pods need to be deployed, it allocates those four pods to the two nodes . As soon as there are more nodes connected, the scheduler recognizes free resources and reallocate the pods to these new nodes.

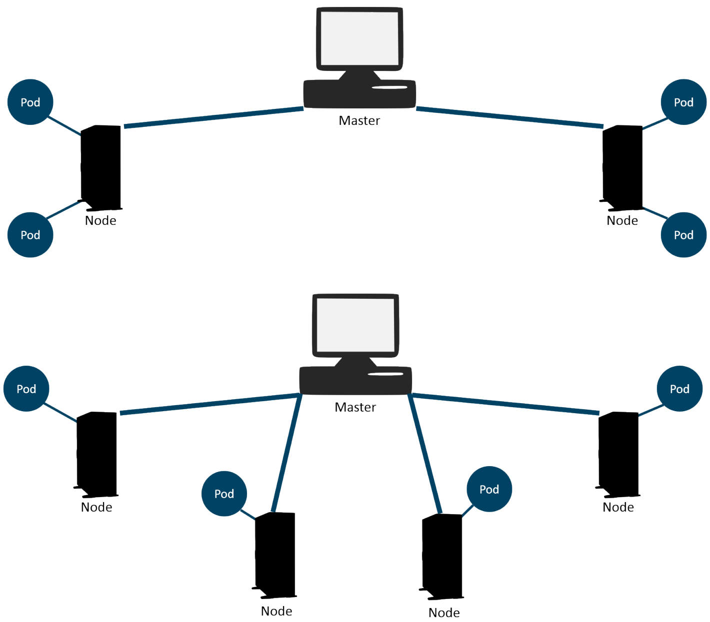

Figure 2.1.3 Kubernetes API Server watcher sequence diagram

Based on retrieved information from [17]

The etcd is a key-value store, which stores the configuration data and the condition of the Kubernetes cluster. The etcd also contains a watch feature, which listens to changes to keys and triggers the API server to perform all necessary actions to move the current state of the cluster towards the desired state.cmp.[13], [16]

The worker node consists of a Kubelet, a cAdvisor, a Kube-Proxy and - as mentioned before - several Pods.

The kubelet needs to be used if a new pod should be deployed. Then it gets the action to create all needed containers. For that it uses Docker to create them. Afterwards it combines some containers into one pod. Conainers in one pod are always started and stopped together. This pod will then be deployed on the node, on which the kubelet is located.cmp.[13], [16]

The cAdvisor measures the usage of CPU-resources as well as demanded memory on the node, on which it is located, and notifies the master about it. Based on those measurements the scheduler allocates the pods within the cluster to ensure the best possible allocation of resources.cmp.[13], [16]

The kube-proxy is a daemon, that runs as a simple network proxy to provide the possibility of communicating to that node within the cluster. Additionally it runs a load balancer for the services on that node.cmp.[13], [16]

Through this architecture Kubernetes enables different possibilities to deploy pods within the cluster. The simplest one is to deploy a specific pod directly through the

```
kubectl create -f ‘image\_path’
```

Listing 2.1: Create Kubernetes pod

command. This deploys the pod described in the file, but it doesn't ensure the failure safety. That means if the pod gets destroyed for some reason, no matter if it has been destroyed by accident or intended, it won't be recreated and deployed again automatically.

Another possibility is to create a deployment. Therefore the image has to be embedded within a replicaset and this replicaset within a deployment. If this deployment is created, it will automatically create every pod of the replicaset. If one pod gets terminated, no matter if manually or because of an error, it will be directly recreated and deployed.cmp.[13], [18]

This procedure also enables the possibility to execute dynamical rolling updates. If you execute the command

```
kubectl set image deployment/name-deployment name=name:1.1.1
```

Listing 2.2: Create Kubernetes deployment

Kubernetes will start a Rolling Update. This causes the creation of a new replicaset, which uses pods of the new version. While the new replicaset will be scaled up, the old one will be scaled down step by step. This enables the maintenance of a deployment even while an update is enrolled, so that the users can still use the services while those are updating. That means, that there is never a need of a system to shut down because of an update, which needs to be enrolled, and the system guarantees its high availability.cmp.[13], [18]

A third possibility to deploy pods are services. Services are used to enable the usage of pods from outside the cluster. Therefore it gets the pod from the targetPort of the belonging node and creates a random port on this node. This port serves as endpoint for the LoadBalancer. Through this, everybody can now communicate with this pod.cmp.[13], [18]

This is how Kubernetes ensures High Availability as well as Load Balancing at the same time. Through the possibility of replicate every pod several times it is ensured that whenever one pod fails for some reason, another is already prepared to help out and take over the job. Even the master can and should be replicated to ensure a functional system, even if one master has been destroyed. For production environment it's recommended to have 3, 5 or 7 master nodes running at the same time. For facilitating the High Availability almost every unit of a Kubernetes cluster is stateless and could be run redundant in several instances. Only the etcd key value store is stateful. For solving that problem in case of failure of the node, which owns the etcd, there is a leader election between all master node replicas to determine the new leading etcd.cmp.[13], [18]

All in all Kubernetes combines all the benefits of cluster applications in one software. High availability as well as load balancing is guaranteed and it also ensures a high scalability, rolling updates and auto-scaling. Compared to other cluster systems, like Docker swarm, it has the biggest community and it can support clusters of up to 5000 nodes, which enables using Kubernetes even for very big clusters. For example the whole Google Cloud System runs on a Kubernetes cluster. Performing tests of Kubernetes shows, that in 99% the API responses in less than one second and also their pods start in less than five seconds in 99% when starting containers with pre pulled images. This all makes Kubernetes maybe one of the most flexible and fastest cluster systems. In exchange for that flexibility it is more difficult to set it up, because you have to do it all on your own. But only that way it can be ensured, that it runs fast and flexible in the same time, which is the reason for Kubernetes being so popular.cmp.[19], [20]

2.3 Apache Kafka

2.4 Open Source Routing Machine

2.5 Mongo Database

2.6 IoT Platform

2.7 DRL Car Sharing Service

3 Method

3.1 Catalogue of Criteria

3.2 Locally setup of car sharing app

For a better understanding of how the car sharing app is working and how the infrastructure for the communication with its underlying services is build up, first the actual state with local setup of all the single components will be described and explained.

For enabling all functions of the car sharing app, three services need to be setup beforehand. Those are Apache Kafka, OSRM and Mongo Database. First it will be explained how those services can be setup and what they are needed for.

Apache Kafka needs to run on top of a Apache Zookeeper instance. Zookeeper is a a centralized service for maintaining naming and configuration data and providing synchronization within distributed systems. For Kafka it keeps track of status of the cluster nodes as well as its topics, partitions etc.

The Kafka service as well as the Zookeeper are running inside of two separated docker containers. The kafka container listens on port 9092 for predefined topics. In the case of the car sharing app those topics are “book” and “confirm”. A Kafka client inside the car sharing app provides a producer as well as a consumer. The consumer listens on new messages for the “book” topic. This way the app is able to receive booking request for the car sharing service, so that customer can request a ride. This ride needs to be confirmed then, which the producer is for. It can produce messages to the “confirm” topic, so that the consumer can be informed about the approval of his ride request and additional information like estimated arriving time or car identification.

For testing the Kafka client within the in this essay described work a test web application was developed. This enables entering a topic, a key and a value and producing a message. This message is then produced by a provided producer. Also the test apps provides two consumers - one listening on the “book” topic, another one listening on the “confirm” topic. This way it can be controlled, if all produced messages of this topic are successfully produced to the Kafka service and if they can be consumed by the consumer. All consumed messages are shown directly on the Web UI, which allows dynamic changes via the Python SocketIO library.

The scheduler uses this for listening to requests, calculating the best fitting car and its route and confirming the ride after all calculations have been made. In the current state of the project there are two different Schedulers - first the “real” scheduler, which is yet only implemented for a simulation, so that it gets its request from a predefined file instead of a Kafka consumer. The other scheduler is a “dummy” scheduler, which is used for a demonstration with a real car, for which no complicated scheduling is necessary, because it is only one car. This “dummy” scheduler uses kafka for consuming requests and confirms them directly afterwards with an equal message for the “confirm” topic.

The second underlying service for the car sharing app is OSRM. As described in chapter INSERT OSRM is for calculating the shortest paths in road networks. For this a map is needed in PBF (Protocolbuffer Binary Format) format. This map then needs to be extracted with a car profile, which determines which routes or streets can be used for this kind of vehicle and which cannot (private road, barriers etc.). This also converts the map into an osrm file. Next this needs to be partitioned into cells and last these cells have to get customized by calculating its routing weights. This OSRM container is also setup within a docker container listening to port 3000 of the localhost for requests for calculating routes on the given map. For running the necessary steps for preparing the map inside the docker container there is a preprocessing shell script prepared, looking like this:

```
rm -rf "$(pwd)"/data/*.osrm
rm -rf "$(pwd)"/data/*.osrm.*

docker run -t --rm -v "$(pwd)"/data:/data osrm/osrm-backend osrm-extract
    -p /opt/car.lua /data/new-york.osm.pbf
docker run -t --rm -v "$(pwd)"/data:/data osrm/osrm-backend osrm-
    partition /data/new-york.osrm
docker run -t --rm -v "$(pwd)"/data:/data osrm/osrm-backend osrm-
    customize /data/new-york.osrm
```

The map data are not stored inside the docker container itself but on the local system. This data is then mounted to the docker container, so that it can access it without the need of storing it inside, which keeps the docker container more light weighted.

The third service is a Mongo Database, on which all vehicles, customers, routes and the history are stored. Through this a demo UI can access all necessary informations to represent the vehicles, customers and its calculated routes. This database can be created by a mongo seed, in which all necessary informations are stored in a compact format, so that the database can easily be restored.

The MongoDB is also running inside a Docker container. The needed seed data are mounted to the Docker container, so that it can access these data stored on the local machine from within the container. Also the created database is created in a local directory which is then mounted to the Docker container. The database is restored by running a mongo-seed shell script looking like this, which first drops the existing table and then recreate a new one by the stored seed:

```
mongo db_rs --eval "db.dropDatabase()"
mongorestore --db db_rs --dir /tmp/seed/data
```

This script is executed within the docker container by running this command:

```
docker exec -it cs-mongo sh -c "chmod 700 /tmp/seed/mongo-seed.sh && /
    tmp/seed/mongo-seed.sh"
```

With all those services running now the car sharing app itself can be started. For the DummyScheduler only the Kafka service is needed. For the simulation all of those services are necessary but the Kafka service. How the communication works can be withdrawn by the following figure, which uses the simulation as example application.

The car sharing simulation needs two components running - first the demo client and second the demo server. The client runs through predefined requests stored in a json file. Those requests are processed by the scheduler, which calculates the route with OSRM. For optimizing the route the cplex engine is used. After completing the calculations the results are written on the mongo database. The demo server consists out of an API, which can be accesses for requesting the current state of all vehicles, customers and calculated routes. For that it connects to the Mongo database and returns the requested data. This is how the simulation can be visualized on a demo UI, which sends requests to the server and represents the current state on a graphical map.

- Kommunikation schaubild

3.3 Outsourcing Kafka client to IBM Cloud message hub

3.4 Dockerizing car sharing app and requirements

3.5 Deploying and exposing docker container on cluster

4 Result

4.1 Outsourcing of Apache Kafka Client

4.2 Communication between car sharing app and services

Those results will be discussed in chpater 5.1

5 Discussion

Acronyms

LSP Large-Scale-Pilot

IoT Internet of Things

OSRM Open Source Routing Machine

UI User Interface

K8s Kubernetes

API Application Programming Interface

REST Represantational State Transfer

PBF Protocolbuffer Binary Format

Bibliography

[1] IoT European Large-Scale Pilots Programme Team (2018): IoT European Large-Scale Pilots Programme Retrieved from autopilot-project.eu/wp-content/uploads/sites/16/2018/03/220315_SD_IoT_Brochure_A4_Brief_LowRes.pdf

[2] AUTOPILOT website (2018): Facts and figures Retrieved from https://autopilot-project.eu/facts-and-figures/

[3] AUTOPILOT website (2018): Partners Retrieved from https://autopilot-project.eu/partners/

[4] M. Djurica, G. Romano, G. Karagiannis, Y. Lassoued, G. Solmaz (tbp 06/2019) oneM2M-Based, Open, and Interoperability IoT Platform for Connected Automated Driving Internal sources

[5] AUTOPILOT website (2018): Driving Modes Retrieved from https://autopilot-project.eu/pilot-sites/driving-modes/

[6] David A. Bader, Robert Pennington (2001): The International Journal of High Performance Computing Applications, Retrieved from www.cc.gatech.edu/~bader/papers/ijhpca.pdf

[7] Aho, Alfred V.; Blum, Edward K. (2011): Computer Science: The Hardware, Software and Heart of It,

[8] Patent by Omar M. A. Gadir, Kartik Subbanna, Ananda R. Vayyala, Hariprasad Shanmugam, Amod P. Bodas, Tarun Kumar Tripathy, Ravi S. Indurkar, Kurma H. Rao for NetAppInc (07-23-2001): High-availability cluster virtual server system, US6944785B2, Retrieved from patents.google.com/patent/US6944785B2/en

[9] OV (09-21-2017) - Bitbucket: High availability for Bitbucket Server, Retrieved from confluence.atlassian.com/bitbucketserver/high-availability-for-bitbucket-server-776640137.html

[] OV (2018) - Kubernetes: What is Kubernetes, Retrieved from kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[] Dr. Thomas Fricke (01-16-2018): Kubernetes: Architektur und Einsatz – Eine Einführung mit Beispielen, Retrieved from www.informatik-aktuell.de/entwicklung/methoden/kubernetes-architektur-und-einsatz-einfuehrung-mit-beispielen.html

[] OV (2018) - Docker: What is a container?, Retrieved from www.docker.com/what-container

[] OV (2018) - CoreOS: Overview of a Pod, Retrieved from coreos.com/kubernetes/docs/latest/pods.html

[] Jorge Acetozi (12-11-2017): Kubernetes Master Components: Etcd, API Server, Controller Manager, and Scheduler, Retrieved from medium.com/jorgeacetozi/kubernetes-master-components-etcd-api-server-controller-manager-and-scheduler-3a0179fc8186

[] Picture retrieved from www.informatik-aktuell.de/fileadmin/_processed_/b/6/csm_K8s_abb1_fricke_b5ce6bd912.png

[] OV (last viewed 08-05-2018) - deis: Kubernetes Overview, Retrieved from deis.com/blog/2016/kubernetes-overview-pt-1/

[] Christ Wright (22-06-2017): Kubernetes vs Docker Swam, Retrieved from platform9.com/blog/kubernetes-docker-swarm-compared

[20] OV (last viewed 08-13-2018) - redhat: What is Kubernetes, Retrieved from www.redhat.com/de/topics/containers/what-is-kubernetes

Appendices