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

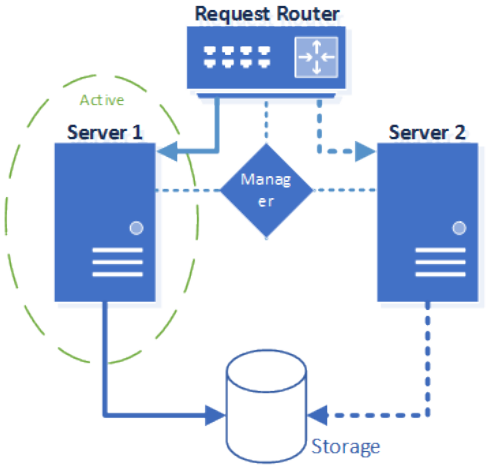

Figure 1.1.1 “High availability” cluster architecture

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

To run applications on a cluster it is necessary to isolate these applications inside of a virtual container. Inside of these containers everything is available, what the application needs to get executed, but noting more. It is comparable to a **VM!** (**VM!**, but much more light weighted. This enables a very light deployment without unnecessary services or applications running in the background, which leads to a very performant execution.

Docker is such a technology for virtualization of containers. It provides a industry-leading container engine technology with an easy way to build containers with a Dockerfile and manage and run those containers easily on your local device.

The main difference between a container and a VM is the weight. While an average VM is hosting an operating system on top of a hypervisor on top of physical hardware. And on top of the OS of the VM the application is running. This affects the performance and the speed of the application in a very negative way, because a lot of unnecessary background proccesses are running.

Docker solves this problem with a far more lightweight compute resource. The hypervisor of a docker container sits directly on the hosting operating system. This allows setting up several docker containers on only one phsyical machine and enables a high scalability. The differences between a VM and a docker container can be seen on the figure below.

On the left side the infrastructure of a VM can be seen, on the right side the infrastructure of a Docker container. Both needs a physical defice infrastructure and the host operating system. On a VM on this Host Operating System a Hypervisor is running and on this Hypervisor several Guest OS can be running. On those again the apps itself can be executed and the necessary libraries and binaries are running.

In case of a docker container those binaries and libraries are directly running on the operating system without the need of a hypervisor or a complete version of a Guest OS. This also enables the app to be running on top of that. The containers are isolated from each other in different namespaces and own network stacks. This means, that processes running within a container cannot see or interact with processes of other containers and they don't get privileged access to sockets or interfaces of other containers.

Additionally there is a Docker Daemon running in another process. The Docker Daemon has three main tasks - listening and processing API requests from the Docker client to run Docker commands, managing Docker objects (images, containers, volumes and networks) and parsing Dockerfiles for building docker images.

Those Dockerfiles contains the definition of a basic image, on which the new container should be based on, and then every operation to be executed to build the container. This includes copying necessary files, installing underlying services, compilers and libraries, setting up the einvironment, executing the commands to start the app and more. All those operations creating a new layer in the container. This leads a container to look like can be seen in the figure 2.1.x below.

This has the advantage, that similar containers, which has equal parts as other, already existing layers, can mount the existing layers from the other container and don't need to build every container for their own again. This makes the build of the docker containers much faster and the single images are smaller, which is especially a great gain for continious inntegration.

All in all Docker enables the creation of virtualized containers, in which applications can run isolated and independent from its environment. That's why it is a condition for deploying an application on a cluster, because there is no host operating system, on which the applications could be running otherwise.

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.

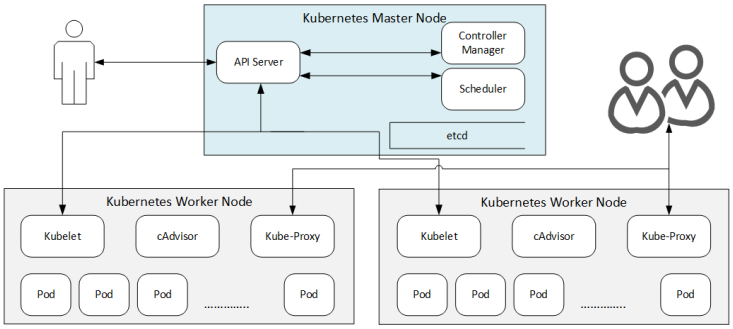

Figure 2.1.1 Kubernetes allocation of services

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**]]

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

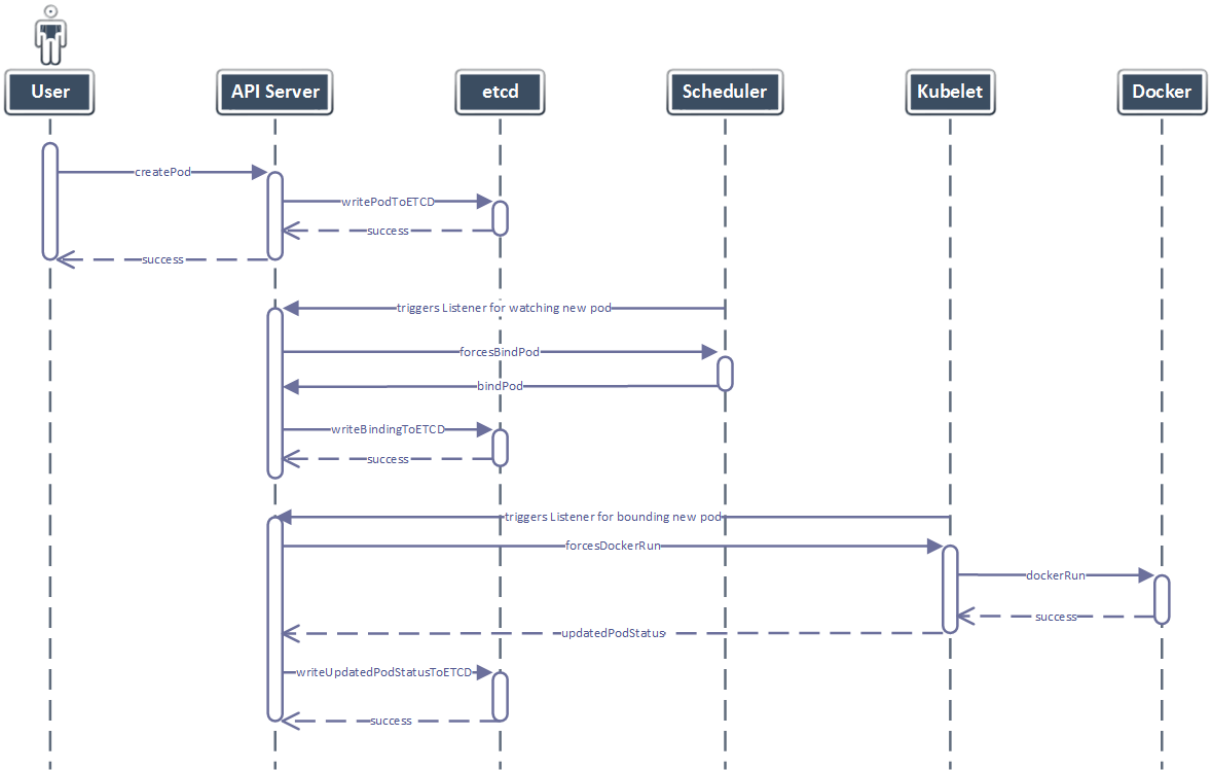

Figure 2.1.2 Kubernetes API Server watcher sequence diagram

Based on retrieved information from [**16**]

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.

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**]]

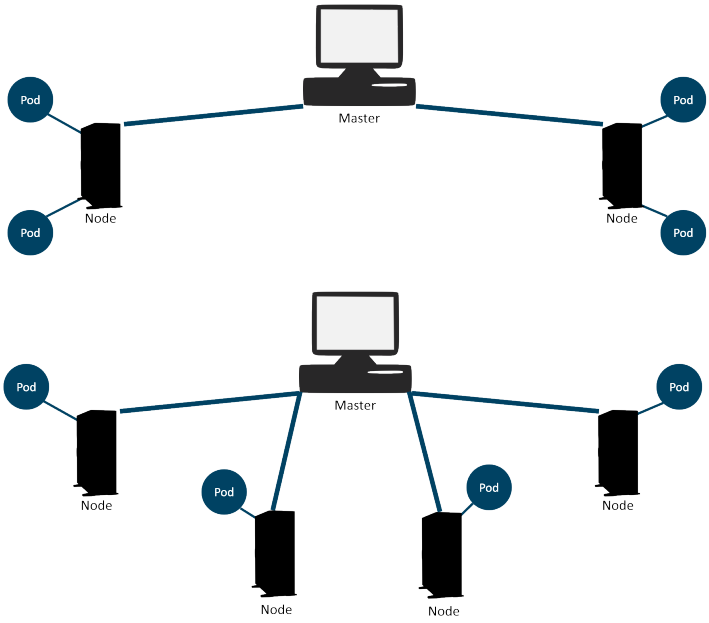

Figure 2.1.3 Kubernetes API Server watcher sequence diagram

Based on retrieved information from [**17**]

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

```
1 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

For a scalable system with a potential of a huge amount of requests every second a realtime streaming platform is necessary. This platform needs to be able to consume messages without blocking the producer even for a second and to produce messages without the need to know the final addressee. Such a streaming platform is Apache Kafka, originally developed by LinkedIn and managed by Apache Software foundation with Open Source license.

Apache Kafka thereby characterizes through the possibilities of a very high throughput, so that it can support millions of messages per second, and real time functionalities, which means, that every produces message is immediately visible to consumer threads. Additionally Kafka supports multiple client systems from different platforms like Java, Python, PHP and more and also it supports distribution over a cluster of consumer machines while maintaining its ordering semantics.

For the realization of these characteristics the architecture of Kafka looks like can be seen in figure 2.3.x. Any amount of producers can produce messages to the Kafka brokers. For that it uses a key-value pair as well as a topic, to which it will produce the message. The broker then stores the message with a timestamp to the topic.

One broker contains several topics, which can be considered as a logical collection of messages. The topics can be divided into one or more partitions. Those partitions are distributed across multiple brokers, which enabled Kafkas abilitiy for dynamically scaling. For that all brokers are connected to one, big cluster of brokers. This cluster is then managed by a Zookeeper Client, which maintains the confguration information across its nodes.

Consumer read from Kafka topics. This means they are listening on a topic and as soon as a new message is produces to it, the consumer recognizes that and reads the newly produced message.

This makes Apache Kafka to a simple-to-use real time streaming platform, which is very scalable, performant and stable through its cluster structure, which enables replications, distribution and partitioning.

2.4 Open Source Routing Machine

One requirement of a car sharing app is of course the routing of a vehicle to its given destination. In the drl car sharing app this is done with OSRM (Open Source Routing Machine), which is designed for use with OpenStreetMaps.

Before calculating the shortest path from A to B OSRM needs an Open Street Map in osm.pbf format. This map needs then to be prepared, which includes extracting the map, partitioning the graph recursively into cells and customize it by calculating routing weights for all cells. When extracting the map it can also include a vehicle profile, which initialized the map with knowledge about which paths can be taken by the specific vehicle and which cant. For example some routes are only accessible by bikes or pedestrians, others can only be accesses with a car.

For the calculation of the route OSRM describes the map as a graph. In a simple model intersections are described as entities and the roads are used as connections between those entities. Those connections are called edges. Every edge gets a weight added, which denotes the cost to traverse the connection, for example time or distance. The goal is then to calculate the path with the minimum overall weight, when all connections leading from the origin edge to the destination edge are summed together.

After the visualization of the road network as a weighted graph, the shortest path needs to be found. For that OSRM uses the Dijkstra's Algorithm. This algorithm looks for every node connected to the source node and stores the weight of the connecting edge. In case there are several connecting edges it takes the edge with the smallest weight. Then it continues recursively and looks for every node connected to the new nodes. Every node, for which the connections have been calculated, are added to a set "visited". The recursion stops when it reaches a node, which is already in the "visited" set. When visiting a new node it stores this to a path log and summing the weight of all the edges it took on its way. This way it calculates the distance between every node by summing the weight of every connection from the origin node to the destination node and storing all nodes visited on the path to it.

This is done for every node of the map when preparing the map for its use. So there is every possible path stored from any node to any other node. When looking for the shortest path OSRM only needs to look for every path leading from the origin node to

the destination node and picks the smallest one. The car sharing app uses this routing machine for calculating the shortest path for the vehicles to its destinations.

2.5 Mongo Database

For a performant and scalable processing of huge amounts of data on a Cloud platform a special database technology is necessary. A very popular database technology for that purpose is **MongoDB!** (**MongoDB!**). MongoDB was developed by the compand 10gen since 2007 and the name Mongo stands for “Humongous”.

MongoDB characterizes through its simple data replication and scalability. For that MongoDB consists out of a cluster of server, which can be easily extended, so that there is no need for upgrading single servers for scaling the database. Additionally MongoDB is a lot faster than relational databases. As an example in figure 2.5.1 it can be seen a comparison between update requests of MongoDB and traditional **MySQL!** (**MySQL!** requests at different scales.

The x axis is the scale of records to be updated and the y axis shows the time it needs in milliseconds. With a small amount of requests the difference is quite low, but as larger the scale get, the greater the difference between those technologies get. With a scale of 100000 records MongoDB is already more than twice as fast as MySQL (30 milliseconds for MongoDB and 76 milliseconds for MySQL). When increasing the scale to 1000000 records MongoDB is almost four times faster than MySQL.

Other than MySQL databases Mongo stores its data as documents instead of relational tables. These documents are encoded as **BSON!** (**BSON!**). The data of the documents are stored as key value pairs. The difference between simple key-value stores and a document store is, that the value column in document stores contains semi structured data. This means, that the data does not conform a formal structure of a data model but it has its own structure for separating semantic elements. But this structure can vary from document to document and from row to row and is no subject of a schema. Additionally a single column can contain any amount of attributes. This can be summarized under the characteristic of a schema-free database.

For backup and recovery of data Mongo offers different replication models. The most common one is the master-slave model. There one node of the cluster act as master and every additional one as salve node. The data stored on the master node is automatically replicated on the slave nodes.

Additionally Mongo offers a feature called AutoSharding. This features enables to automatically divide a database server to several, physical machines, which allows horizontal scaling. For that the documents are divided in different shards by a specified ShardKey. The documents are then divided to different groups selected by an evenly range of this ShardKey. This doesn't affect the application itself at all, because the MongoDB still represents itself as a single database to the outside.

All in all MongoDB combined the advantages of a fast database and a high flexibility because of its missing schemas. This makes it to a very efficient way for storing huge amounts of data.

2.6 DRL Car Sharing Service

3 Method

3.1 Catalogue of Criteria

For measuring the success of this project afterwards some criteria need to be defined of what should be achieved during this project.

The longterm goal for the car sharing project is to move away from a local setup to a deployment on a cluster with all its components and dependencies.

The first step for that is to outsource the Kafka client from a local setup to a Cloud service. For that the IBM Cloud Message Hub service can be used. This also enables the possibility to send request not only from the local device, on which the car sharing app itself is running, but also from other applications like a mobile device, which is a necessary requirement for a real use of the car sharing app. Because currently there is no clean way of testing the Kafka client also a web application for testing purposes of the Kafka client should be created. This app should be able to produce messages as well as consuming them and printing them on its UI, so that it can be checked if the production of messages is successfull and, that they can be consumed afterwards.

After that the DummyScheduler should be deployed on the Cluster. This requires to create a container of the car sharing app. Because the DummyScheduler does not use the cplex engine or any other third party service those dont need to be included to the Docker container.

Because the cluster is an internal Research Cluster there are some differences to "normal" Kubernetes cluster. This causes some additional security stepts to be made for deploying apps on it. These additional stepts should be figured out and documented, so that it can be done with other apps without the need of additional research in the future.

If the DummyScheduler is running on the cluster, all its functionalities are working and the Kafka client is outsourced to the IBM Cloud Message Hub, the project can be considered as successful.

If the deployment of the car sharing app, for example with the real scheduler and its requirements, can get continued beyond that, the project can be considered as overachieved.

3.2 Local 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

The locally running Kafka service has two disadvantages: The first one is, that it needs to be setup along with Zookeeper every time when running the app. For deploying the car sharing app within a cluster on the cloud the Kafka client as well as an underlying Zookeeper would also have to get deployed on it to guarantee the functionality of the car sharing app.

The second problem is, that a local Kafka client can only be accessed from local side. This means, that no messages can be produced to the Kafka client from outside the host, which prevents the car sharing app from receiving customer requests from the outside, for example an app with which the customers can order a ride.

These problems can be solved by outsourcing the Kafka client to a Cloud service like the IBM Cloud Message Hub. This service contains a fully managed, highly scalable and high performing Kafka platform. In this chapter it will be described how the configurations of the car sharing kafka client has to be changed, so that it can connect to the IBM Cloud Message Hub.

Currently there are several configurations made for the local Kafka client. These configurations can be found in the following figure

```
KAFKA_BROKER = "localhost:9092"
KAFKA_BOOKING_TOPIC = "book"
KAFKA_CONFIRMATION_TOPIC = "confirm"
KAFKA_TOPICS = [KAFKA_BOOKING_TOPIC, KAFKA_CONFIRMATION_TOPIC]
KAFKA_GROUP = "group1"
KAFKA_TOPIC_CONFIG = {"auto.offset.reset": "smallest"}
KAFKA_CONSUMER_SESSION_TIMEOUT_MS = 30000
```

First the broker is set to the localhost with port 9092 before the booking and confirmation topics names are set. After that there are just some general settings to be made for the Kafka client like session timeout. They will stay the same with the Message Hub configurations. For this there are some additional settings to add for enabling access to the Kafka Client running on the IBM Cloud service. The new configurations are listed below.

```
KAFKA_BROKER = os.getenv("BOOTSTRAP_SERVERS")
KAFKA_BOOKING_TOPIC = "book"
KAFKA_CONFIRMATION_TOPIC = "confirm"
KAFKA_TOPICS = [KAFKA_BOOKING_TOPIC, KAFKA_CONFIRMATION_TOPIC]
KAFKA_GROUP = "group1"
KAFKA_TOPIC_CONFIG = {"auto.offset.reset": "smallest"}
KAFKA_CONSUMER_SESSION_TIMEOUT_MS = 30000
KAFKA_SECURITY_PROTOCOL = 'SASL_SSL'
KAFKA_SSL_CA_LOCATION = '/etc/ssl/certs'
KAFKA_SASL_MECHANISMUS = 'PLAIN'
KAFKA_SASL_USERNAME = 'token'
KAFKA_SASL_PASSWORD = os.getenv("API_KEY")
KAFKA_API_VERSION_REQUEST = True
KAFKA_BROKER_VERSION_FALLBACK = '0.10.2.1'
KAFKA_LOG_CONNECTION_CLOSE = False
KAFKA_CLIENT_ID= 'client1'
KAFKA_GROUP_ID = "group1"
```

First there are the same configurations to be made as for the local client. Only the broker needs to be changed. In this case it takes the broker out of an environment file, in which the url of the broker is defined. Addtionally there are settings for a secure access to the broker, like used security protocol or ssl location. For the identification an API token is used. That's why the username is set to "token". The password has to equal the API key in this case, which is also read from the environment file. Both - the API key as well as the broker - can be found on the IBM Cloud Message Hub dashboard. Last there are some more configurations for the API access, version and logging to be made.

These configurations are added as additional configuration mode called “MessageHubConfig” in the configuration file. To use these configurations the used mode has to be changed to the newly added “MessageHubConfig”.

When configuring the Kafka client later in the app sequence it will access this configurations to set it up. Because there are some new configurations and not only existing ones with different values, these have also to be added to the setup of the Kafka client. There a JSON is created like below:

```
        'bootstrap.servers': config.KAFKA_BROKER,
        'session.timeout.ms': config.KAFKA_CONSUMER_SESSION_TIMEOUT_MS,
        'default.topic.config': config.KAFKA_TOPIC_CONFIG,
        'batch.timeout_s': config.
            BOOKING_REQUEST_PROCESSOR_BATCH_TIMEOUT_S,
        'batch.topics': [config.KAFKA_BOOKING_TOPIC],
        'batch.job.trigger.default.interval_s': config.
            BOOKING_REQUEST_PROCESSOR_TIME_WINDOW_S,
        'batch.executor.default.pool': 1,
        'batch.job.max_instances': 1,
        'security.protocol': config.KAFKA_SECURITY_PROTOCOL,
        'ssl.ca.location': config.KAFKA_SSL_CA_LOCATION,
        'sasl.mechanisms': config.KAFKA_SASL_MECHANISMUS,
        'sasl.username': config.KAFKA_SASL_USERNAME,
        'sasl.password': config.KAFKA_SASL_PASSWORD,
        'api.version.request': config.KAFKA_API_VERSION_REQUEST,
        'broker.version.fallback': config.KAFKA_BROKER_VERSION_FALLBACK,
        'log.connection.close': config.KAFKA_LOG_CONNECTION_CLOSE,
        'client.id': config.KAFKA_CLIENT_ID,
        'group.id': config.KAFKA_GROUP_ID,
```

This takes all the configurations made in the CONFIGURATION file and add these to a JSON, which is used for the initialization of the Kafka producer as well as its consumer. After adding these configurations the Kafka client is successfully setup for accessing the IBM Cloud Message Hub Kafka service.

3.4 Dockerizing car sharing app and underlying technologies

For deploying the car sharing app and all its underlying technologies on a Kubernetes cluster it needs to be dockerized beforehand. This means, that the application needs to be packed in a container, in which only the app itself and a very light weighted **OS!** (**OS!**) with only necessary installations are running. This is made possible by Docker.

In every Docker container only one application can be running at the same time. That's why for every application needed for the car sharing app, an own Docker container is needed. That means all in all three Docker containers are needed - one for the MongoDB, one for OSRM and one for the car sharing app itself. To build a Docker container it is necessary to create a Dockerfile. This Dockerfile is usually based on one basic image, like a very light weighted Ubuntu OS, and then every necessary step to be taken before the app can be executed. This includes copying necessary data, setting environment variables, installing necessary compilers and libraries etc.

For Mongo and OSRM there are already prebuilt Docker containers exisiting. They need to be extended by the necessary data for the car sharing app and the operations needed to be executed on these data. In case of OSRM such a Dockerfile looks like this:

```
FROM osrm/osrm-backend

COPY ./osrm/data/new-york.osm.pbf /opt
RUN osrm-extract -p /opt/car.lua /opt/new-york.osm.pbf
RUN osrm-partition /opt/new-york.osrm
RUN osrm-customize /opt/new-york.osrm

CMD ["osrm-routed", "--algorithm", "mld", "/opt/new-york.osrm"]
```

First the base image is defined with the "FROM" command. Then the map is copied and the necessary preprocessing commands described in chapter 3.2 are executed. Last the osrm service is started with the prepared map in the "CMD" statement.

Extending the Mongo Docker container works similarly:

```
FROM mongo:3.4
MAINTAINER Pascal Schroeder <pascal.schroeder@de.ibm.com>

COPY ./mongo/seed /tmp/seed

ENV MONGO_DATA_DIR=/data/db
ENV MONGO_LOG_DIR=/dev/null

RUN chmod 700 /tmp/seed/mongo-seed.sh

CMD mongod --fork --logpath /var/log/mongodb.log; \
    ./tmp/seed/mongo-seed.sh; \
    mongo db_rs --eval 'db.network.createIndex( { "lat_lon" : "2d" } );'
        ; \
    mongod --shutdown; \
    docker-entrypoint.sh mongod
```

First the base image is declared. Then the seed folder is copied into the docker container, so that it can use the seed for restoring the database afterwards. Then a few necessary environment variables are set before the "mongo-seed.sh" file is set to executable. When starting the mongo database it first runs the "mongo-seed.sh" script for restoring the database and creates the necessary index before restarting the mongo database.

Last the car sharing app itself needs to be dockerized. For that there is no prebuilt docker container, so this image is based on a light weighted ubuntu version and installs every requirement itself. This Dockerfile looks like this:

```
1  FROM ubuntu:latest
2  MAINTAINER Pascal Schroeder <pascal.schroeder@de.ibm.com>
3
4  RUN apt-get update
5  RUN apt-get install -y locales
6  RUN apt-get dist-upgrade -y
7  RUN apt-get update
8
9  ENV PYTHONPATH=/drl-car-sharing
10 ENV TZ=Europe/Dublin
11
12 RUN mkdir -p /drl-car-sharing
13 WORKDIR /drl-car-sharing
14 COPY . .
15
16 RUN apt-get install python3 python3-pip -y
17
18 RUN cd ./cplex/x86-64_linux && python3 setup.py build && python3 setup.
       py install
19
20 RUN pip3 install -r requirements.txt
21
22 WORKDIR /drl-car-sharing/drl_car_sharing/srv
23 CMD ["python3", "launch-query-proc.py"]
```

First the **apt!** (**apt!**) get updated and the locales package gets installed as well as the distribution gets upgraded. Then some environment variables for the pythonpath and the timezone are set before the work directory is created and set. Then all the project files are copied into the docker container except for the not-required files defined in the ".dockerignore" file.

After that python3 gets installed as well as the python package manager **pip!** (**pip!**). Then it installs the requirements of this project defined in the "requirements.txt".

Last the directory is changed and the file to be executed for starting the app is defined. In this case it is the file for the DummyScheduler. For starting another file, for example the simulation, it is necessary to create a new Dockerfile based on this image, which changes the file to be executed in the end. This looks like following code snippet:

```
FROM registry.eu-de.bluemix.net/drlautopilot/drl-car-sharing

ENV PYTHONUNBUFFERED = 0

WORKDIR ../demo
CMD ["python3", "-u", "start_demo_cli.py"]
```

To build these Dockerfiles the docker build command has to be run. This command can be found below

```
docker build -t <tag-of-docker-image> -f <name-of-dockerfile> <path-to-
    dockerfile>
```

The <tag-of-docker-image> is needed for the identification. Also with the tag it can be defined to which registry the image will be pushed. This is explained in more detail in chapter 3.5. The <name-of-dockerfile> is only needed when the Dockerfile is not called Dockerfile. Otherwise this parameter can be removed of this command. The <path-to-dockerfile> parameter is necessary and defines the relative path to the Dockefile.

After the Docker images are all built they can be pushed to a container registry. How this works and how they can be deployed on a cluster afterwards is described in chapter 3.5.

3.5 Deploying and exposing docker container on cluster

Before deploying a Docker container on a Kubernetes cluster, the containers need to be pushed to a container registry, from where the cluster can access and pull the image. One possibility is the open platform DockerHub. It is the largest community for container images and for example the base images for mongo and OSRM are already stored on it. The disadvantage is, that they the containers are stored on public servers and even, if they can be set to private containers, unpermitted access to the data can not be guaranteed.

That's why for confidential data private container registries are usually preferred. In case of the car sharing project this is the IBM Cloud container registry. To push a container to this registry it needs to be correctly tagged. For that the tag needs to start with *registry.<region>.bluemix.net/<namespace>*. The *<region>* placeholder has to be replaced with the location of the used container registry, for example "eu-de". In those

container registry services of IBM cloud there can be created different namespaces. For defining them in the tag the *<namespace>* placeholder has to be replaced with the chosen namespace. The docker image with id *<image-id>* can be tagged with

```
docker tag <image-id> registry.eu-de.bluemix.net/<your-namespace>/<name-
    of-image>
```

Before pushing the container the user has to be logged in to IBM Cloud **CLI!** (**CLI!**) and an account, a region and a Cloud space has to be chosen. After that it can be pushed with the docker push command and the newly generated tag:

```
docker push <image-id> registry.eu-de.bluemix.net/<your-namespace>/<name
    -of-image>
```

Now the docker image is ready to get deployed on the Kubernetes cluster. This has to be repeated for every docker container to be deployed. This means in case of the car sharing project this has to be done for the extended mongo and osrm docker container as well as the car sharing containers for the simulation as well as the scheduler component itself. For deploying those containers to the cluster a deployment.yaml file is needed. Before that a secret for pulling the image from the cloud registry needs to be created beforehand. When a secret is created it needs to be defined in the deployment.yaml along with other properties. All in all this file has to look like below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: <name-of-deployment>
spec:
  selector:
    matchLabels:
      app: <name-of-deployment>
  replicas: 1
  template:
    metadata:
      labels:
        app: <name-of-deployment>
    spec:
      containers:
      - name: <name-of-container>
        image: <tag-of-container>
        resources:
          requests:
            cpu: <requestes-cpu-power>
            memory: <requested-memory>
          limits:
            cpu: <max-cpu-power>
```

```
            memory: <max-memory>
      imagePullSecrets:
      - name: <name-of-secret>
```

In this example deployment.yaml several properties needs to be changed. First, the <name-of-deployment> placeholders have to be set to the name, which the deployment itself will get. The <tag-of-container> placeholder has to equal the tag with which the container was tagged before. The <name-of-secret> field has to be the name of the created secret for the container registry. Last the resources has to get defined. The requested resources are the resources, which the deployment always request from the cluster, so that those are always accessible and reserved for this specific deployment. If they need more, they can request more but only up to the limit defined in the limits section.

Such a yaml file has to be created for every deployment - this means for Mongo, OSRM and every car sharing component, so the DummyScheduler and the simulation client as well as the simulation server. Before creating these deployments on the cluster kubectl has to be configured with the KUBECONFIG as described in chapter 2.2. After that they can be deployed with the command

```
kubectl create -f <name-of-deployment-file>.yaml
```

The containers are now deployed on the cluster, but they can not communicate with each other. For that services are necessary, so that they can communicate via the Kubernetes network. Such services can be created with another yaml file, looking like this:

```
apiVersion: v1
kind: Service
metadata:
  name: <name-of-service>
  labels:
    app: <name-of-service>
spec:
  type: NodePort
  ports:
  - port: <port-of-app>
    targetPort: <exposed-port>
    name: http
    protocol: TCP
  selector:
    app: <name-of-deployment>
```

Like before in the deployment.yaml file the <name-of-service> has to be defined first. After that it is important to define the <name-of-deployment> for allowing the service to connect to the right deployment for exposing the port. For that it is also necessary

to define the <port-of-app>, which should be the port, on which the running container of the deployment is listing on. The <exposed-port> has to equal the port from which all requests to the service will be forwarded to the specified port of the deployment. A service has also be created for every deployment, with which another deployment has to communicate. This means it is needed for the Mongo and OSRM deployment as well as for the DummyScheduler and the simulation server. The creation of the services is done equally to the creation of the deployments:

```
kubectl create -f <name-of-service-file>.yaml
```

Because these deployments are all on the same Kubernetes network it is sufficient to send requests to the host equalling the name of the requested service. This means, that the deployments need to set the host to the the name of the service they want so speak to before deploying the container.

Services do not enable access from outside the cluster, but only from the inside. To access it for external requests it is necessary to create an ingress. This ingress also needs a yaml file for its creation. An example ingress can be found below:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    ingress.kubernetes.io/allow-http: "false"
    ingress.kubernetes.io/ssl-redirect: "true"
    kubernetes.io/ingress.class: f5
    virtual-server.f5.com/balance: round-robin
    virtual-server.f5.com/ip: <your-ip>
    virtual-server.f5.com/partition: RIS-INT-FRA02
  name: <ingress-name>
  namespace: <namespace>
spec:
  rules:
  - host: <host>
    http:
      paths:
      - backend:
          serviceName: <name-of-service>
          servicePort: <port>
        path: /
  tls:
  - secretName: /Common/BlueMix
```

In this file it needs to be defined the ip of the cluster namespace (<your-ip>) as well as the host of it (<host>). After that the <name-of-service> needs to be defined, which has

to equal the service, which should be exposed to the outside. Lastly also the port the service is listening on has to be defined. After creating this the same way as the service and the deployment has been created the service is accessible from the outside.

```
kubectl create -f <name-of-ingress-file>.yaml
```

This can be necessary for example for the simulation server for being able to observe the simulation from the outside. This way the frontend can connect to the defined host to get the results of the simulation. How this looks like can be found in chapter 4.

4 Result

4.1 Outsourcing of Apache Kafka Client

The first task for the movement of the car sharing app to a cluster instead of a local setup was to oursource the kafka client to the IBM Cloud Message Hub. Basically this needed a few configurations of the kafka client inside the car sharing app as decsribed in chapter 3.x. Because yet there is no app with which the functionality could have been tested it was necessary to create a test application, with which the producing as well as the consuming of messages can be tested.

This app has three different text fields in total - One for specifying the topic of the message to produce. Another one for the key of the message and a third one for the value. With the submit button this message will be produced to the defined topic.

Additionally there are two predefined topics - confirm and booking . and for each of them a consumers is running in another thread. The consumers are continiously listening for newly produced messages for those topics. If one of them consumes a new message while it is running this message is shown below the matching header.

How the app looks like can be seen in the picture below.

With this testing app also the DummyScheduler can be tested. When producing a message for the booking topic in a JSON format the DummyScheduler mirrors this message to the confirm topic. This means it produces an equal message for the confirm topic. Because the test app also includes a consumer for this topic and prints the consumed messages on the web app the mirrored message can be seen there, if the DummyScheduler is working. If there is no new message for confirm topic a few seconds after the book message got produced, the DummyScheduler is not working correctly.

All in all this way the app enables testing both - the outsourcing of the Kafka client itself and the correct functionality of the DummyScheduler.

4.2 Communication between car sharing app and services

The movement of the car sharing app from a local setup to a cluster not only changes the location, where the app is running. It also changes the way of communication between

the single apps and services. This means this movement did not only needed a setup on a different location, but also some changes of the code for maintaining the communication.

As described in chapter 4.1 the Kafka client was outsourced to IBM Cloud message hub. The way of communication does not change in general, only some additional configurations like security protocols and access data had to be made and the broker had to be changed from localhost to an online client. Also the topics had to be created in the online client of Kafka as well, so that messages can be consumed as well as produced. After those changes have been made the workflow with Kafka didn't change at all, so that there is no further need of adjustments. The functionality can be tested with the testing web application descibed in chapter 4.1.

The other services - mongo and osrm - weren't necessary for the DummyScheduler, but they were for the real scheduler. Same applies to the cplex engine.

For moving this to the cluster as well, first it was necessary to isolate the needed part of the cplex engine - the python library installer - and copy it inside the docker container to be created. Inside of this docker container this cplex python library is automatically installed, which enables all needed functionalities directly inside the created docker container.

The mongo and osrm services had already basic images available, on which could be built up. They got extended by the necessary data and also by preprocessing those data for restoring the database in case of mongo or preparing the map in case of osrm. Those extended docker containers were also deployed on the cluster.

The communication inside a cluster is working differently from a communication on a local device. Even if the deployments are running inside the same network, as described in chapter 2.2, it is necessary to create services for allowing the deployments to talk to each other. Then the host has to be changed from local host to the name of the service, to which it has to be talking to. How the communication is working after creating these services can be seen in the picture below. It described the communication flow for the deployment of the simulation, which accesses the real scheduler for calculating the car-customer mapping, as well as the DummyScheduler.

There can be seen, that the DummyScheduler, the simulation client and the simulation server are running on different containers on the cluster. In two other additional containers the mongo and the osrm containers are running. The DummyScheduler only needs the KafkaClient for its calculations. That's why it only accesses the Kafka client on IBM Cloud Message Hub, which is outside the cluster.

The simulation client doesn't get real requests via the Kafka client, but predefined ones of a JSON file located in the project itself. That's why it needs no connection to the Kafka

client. Instead it accesses the OSRM service for calculating the route for the vehicles to the customers. For optimizing the scheduling it also uses the cplex engine installed inside the docker container. The results are then written to the MongoDB, which is the reason why it has to access the mongo container too.

The simulation server only reads from the Mongo database for returning the current state of the simulation. The service of the simulation server is exposed with an ingress as described in chapter 3.x. That's how an external client can access the simulation server, for example a demo UI. This demo UI can then visualize the state of the simulation as can be seen in the picture below.

Those results will be compared to the formulated criteria in chapter 3.1 in the next chapter 5.

5 Discussion

The defined criteria in chapter 3.1 determined the goals of this project. In the following chapter those will be compared to its results and evaluated for its outcomes.

The first goal was to outsource the Kafka client from a local client to IBM Cloud Message Hub. For testing its functionality a web application had to be created, which can produce and consume messages. The results were described of this goal was described in chapter 4.1 and can be considered as successfull. The web app can produce messages of any topic and there are also consumer for the two predefined topics book and confirm, which enables an accurate testing of the Kafka client. This client was also moved from local docker containers running on a Zookeeper client to a completly independent IBM Cloud service calles "Message Hub". This enables an access to the Kafka client without the need of any local setup more than running the car sharing app.

The second goal was to deploy a DummyScheduler on a Kubernetes cluster hosted internally on IBM Cloud. The internal hosting caused some additional security restrictions. That's why it was part of the work to figure out the differences from deploying apps on this internal cluster compared to a "normal" Kubernetes cluster. For that some additional configurations for initializing the Kubectl client (cmp. chapter 3.2) were necessary. Also the docker container had to be uploaded to a internal container registry, which caused the need of creating secrets on the cluster for being able to pull those containers on the cluster. Last also the exposing of the services for external use is different and needs some more configurations in the ingress.yaml file (cmp. chapter 3.5???).

All those differences could be figured out and were completly documented for the rest of the team. Also the DummyScheduler and its dependencies could be dockerized, which was a condition for deploying it on the cluster. This container could then be uploaded to the container registry and deployed on the cluster, which enabling the project to be considered successfull, because all the goals were fullfilled. The DummyScheduler is running on the cluster and all its functionalities are working, which can be tested with the Kafka test app described in chapter 4.1.

Additional to deploying the DummyScheduler during this project also the simulation of the car sharing app, which includes the usage of the real scheduler, could be successfully deployed on the cluster. This needed the docker contaienr of the car sharing app to include a small part of the cplex engine. Additionally for that an extended docker container for each - the mongo database as well as the OSRM service - had to be created and deployed

as described in chapter 3.x. All this work was successfull, which makes it possible to run a simulation on the cluster and visualize it with a locally running demo UI, which can be seen at the end of chapter 4.2.

All in all this additional deployments of more than just the DummyScheduler allowing the project to be considered as overachieved, because the goals were exceeded.

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