# Urban Area Parking Spot Monitoring System

Thomas Kaufmann

TU Wien

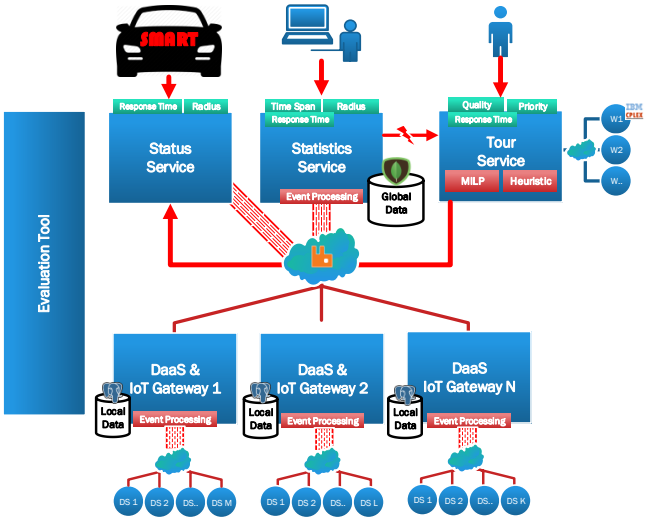
Advanced Services Engineering

Summer Term 2017

## Introduction

This project was developed for the course 'Advanced Services Engineering' at TU Wien. The goal was to devise a scenario that illustrates important aspects of modern IoT application. Furthermore, another goal of this project was to identify and implement different data-concerns and to incorporate analytical services to enable support for elasticity. To this end, we devised a prototype for a parking spot monitoring system for urban areas. The decentralized system can incorporate multiple so-called *cluster-masters* that monitor certain sub-areas (e.g. a district of a city) and provide preprocessed and filtered data over various interfaces (REST, MoM). In addition, the system contains several higher-level services that retrieve and process the data of several cluster-masters and make the aggregated data available over a single interface (instead of querying all cluster-masters). While a cluster-master only provides information about the parking spots in its monitored area, the Status Service gathers information from all available cluster-masters which allows to query the overall state while still avoiding to centrally store the entire information. In general, the status service allows a user to specify its location and a considered range/radius. Since the computation of distances for every record in the database can be rather compute-intensive, we decided on a distributed architecture that allows to be executed concurrently when the query spans over multiple clusters and hence a lot of distance computations have to be performed. Even though this architecture may still can be improved, we claim that it already allows for scalability and avoids single point of failures as well. In contrast to the cluster-masters that only store the current state of the parking spots, the Statistics Service records all state transitions of all parking spots in the entire system and persists them in a document based storage. In addition to the REST interface that allows to query statistical information, the service also performs analytical tasks, by processing the incoming events via a CEP engine and performing a MapReduce tasks on the entire dataset to detect peak loads in the system. The last high-level service that can be queried by end-users is the Tour Service. The service allows a user to query tours between occupied parking spots. For example, a typical group of users for this service could be the staff of the department of traffic-management that want to check the parking tickets of cars more efficient. Of course, this scenario could be extended to inform the user about changes in the system state (e.g. car on the route has left) and to re-compute a new route on-the-fly. However, we thought that this would be far beyond the scope of this class and would most likely be rather difficult to test. However, we claim that our current architecture would allow to extend the system by this feature rather easily. Finally, to be able to determine QoS & QoD metrics, the system employs an Evaluation Tool that records incoming requests, the corresponding results and the response time of the request. Even though we only provide a simple read-only interface for this data, in a more advanced scenario this data could be used by other services to determine prices or resource demands (e.g. for on-demand provisioning).

The following figure shows a system overview and illustrates the interactions between the different services.




## Data Concerns & Elasticity

As already mentioned previously, one major goal of this project was to devise and implement different data-concerns and to allow for elasticity.

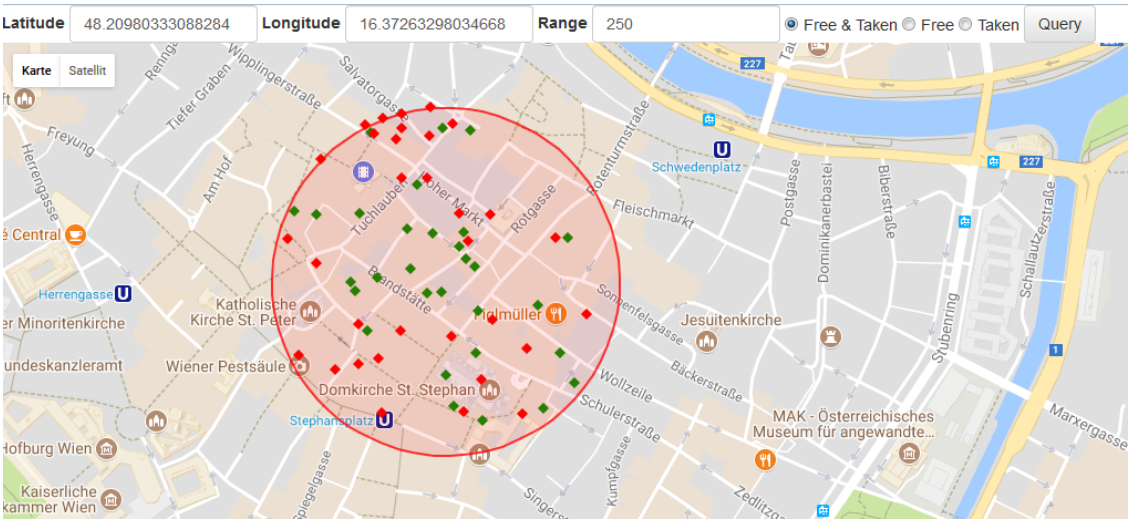
In general, the *major* data concern used in this project is quality of data. By allowing the user to specify different ranges and time-spans for the queries at all services, the user can influence the quality/granularity of the data (e.g. by queried range in meters) and specify a required confidence of the results (e.g. longer timeline yields more confident statistical results). Since it is inherent in the scenario that the system state changes rapidly and continuously, we claim that the response time is an important QoS metric of all our services.Therefore, we make use of AOP techniques to record the response time of all requests and persist the data at the evaluation tool. While in this scenario the data can only be viewed for informative purposes, more advanced scenario could use the information in various ways. The response time could be injected into the result objects which would allow a client to decide whether the received results are already obsolete (e.g. a request with an unusual high response time of >5s could already be obsolete for a car driving with 130km/h on the highway). Furthermore, other services could also consider this information for pricing models and SLAs enforcement.

Finally, we introduced elasticity at the tour service by implementing a master-worker pattern that is controlled by an analytical service. In general,once a tour request is received, the service forwards the job to a set of worker nodes that process the jobs as soon as possible. On one hand, our system enables elasticity, by making use of a three-level priority queue that allows jobs with higher priority to overtake others. On the other hand, our system allows the user to specify a desired quality level of the solution, that the system should try to achieve. To this end, we have implemented three different solvers for the Traveling Salesman Problem. While for low quality requests, a simple nearest-neighbor heuristic is used, the system tries to improve such a simple solution with a Simulated Annealing-like meta heuristic for medium quality requests. For high-quality requests, we make use of aMiller-Tucker-Zemlin MILP formulation that is solved by the IBM CPLEX MILP Solver(it is important to note that this Solver is a commercial product and no sources and libraries are contained in this project. However, the solver is available on https://onthehub.com/ibm/ for academic usage). However, since the TSPand MILP in general are known to be NP-hard and therefore cannot be solved efficiently in the general case (unless NP=P), one cannot expect a solution infeasible time for large instances. Therefore, our system makes use of some heuristics to decide when an input instance is too large to be solved by the MILP solver. An important feature that enhances elasticity in the system is the ability to autonomously degrade the requested quality level when peak-loads are detected. To this end, the statistics service uses a MapReduce task on the global system state to detect peak loads (unexpectedly high number of state transitions in a specified time window) and forwards this information to the tour service. The tour service then uses this information together with information about its own load to reduce the quality level autonomously (i.e.decrease of the quality level speeds up the jobs and allows for a higher throughput at peak loads). But since the result could differ a lot from the expected one, information about this autonomous change is included into the result object and hence also recorded by the evaluation tool. Again, a more advanced scenario could use this information for pricing models that consider the actual quality level rather than the requested one.

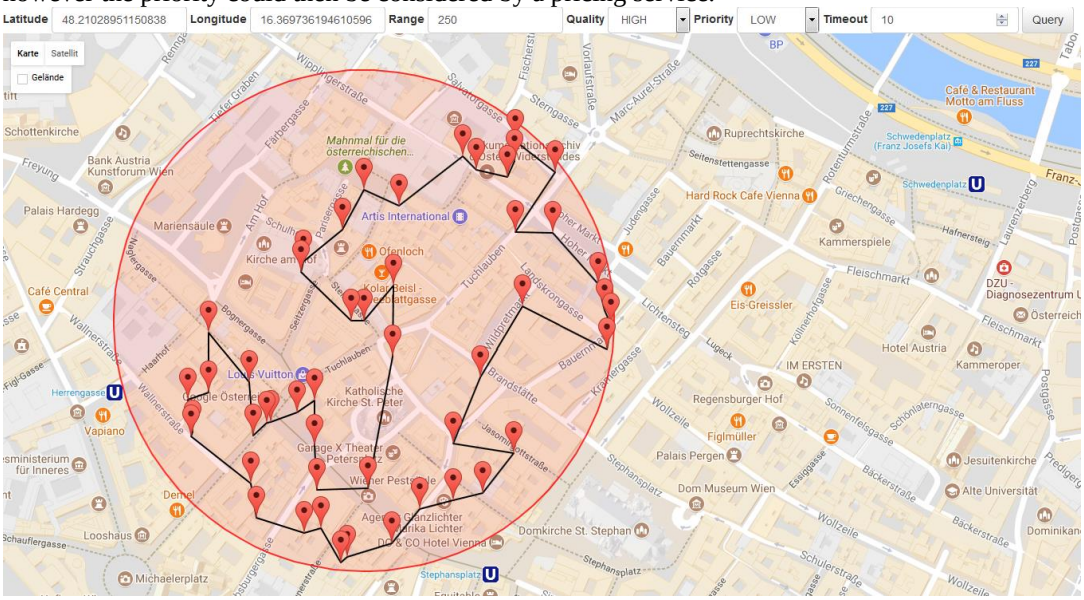
## Simulation & Input Data & UI

Since we were not able to find any real data for simulation purposes, we provide a small .NET tool that simulates the data sources by directly feeding datainto the respective message queues. Since the tool is based on .NET core, it can be also run on Linux environments. For the input data, we randomly generated 1000 parking spot locations in the centre of Vienna. For demonstration purposes, we implemented a rudimentary GUI based on bootstrap, jQuery and the Google Maps API to display the results of the simulation. The following figures show screenshots of the implemented UIs:

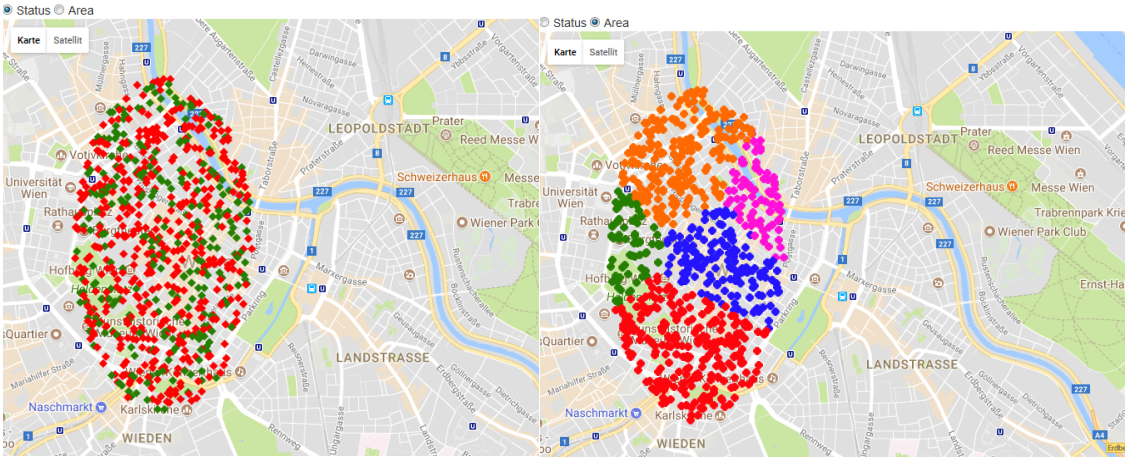
1. Status Query: The status query UI simply allows a user to select an arbitrary GPS coordinate and to query the status of the parking spots in a certain area. As the screenshot illustrates, this service is similar to a loupe over a map, that allows the user to view the current status of the parking spots. The size of the loupe allows a user to obviously inspect a larger area, increasing in some sense the quality of the data.




2. The tour query UI simply forwards a request to the tour service and display the route on reception of the results. In addition, notifications are use to display the actual cost of the route. While the tour service can also be parametrized by a certain range, here the quality of data measure is the actual tour quality. As already mentioned before, the user is able to specify different quality levels and depending on the current load the system will try to meet these demands. In addition, the user can also specify a priority of the request, which introduces some kind of elasticity: higher priority request will be processed sooner, however the priority could then be considered by a pricing service.



3. Finally, just for debugging purposes we provide a monitoring service that polls the current state every second. Moreover, the user is able to select from two different views. While the Status View simply displays the current status, the Area View shows by which cluster a parking spots is managed.




**Technologies & Requirements**

In general, our services are based on Spring (Boot, Data, AMQP) and can be deployed in Docker. Since the cluster-masters only maintain the current state and should be able to perform queries on records quite fast, we decided to make use of a relational database (PostgreSQL, but could easily be replaced due to the use of Spring Data). On the other hand, the amount of data stored by the Statistics Service and the Evaluation Tool is obviously much higher (entire history vs. current state). Therefore, we decided to make use of NoSQL technologies (MongoDB).

While the higher-level services provide a REST-based interface, the communication within the system (i.e. between the components) is based on RabbitMQ. As already mentioned before, for the data analytics we make use of the MapReduce feature of MongoDB and perform stream-processing (CEP) with Apache Flink (input filtering, generation of higher-level events, detection of specific state transition sequences, …).

Finally, the worker nodes of the Tour Service make use of the CPLEX MILP Solver of IBM. Once more we want to stress, that this is a commercial product and no sources or libraries are included in this repository. However, for academic usage it is available at https://onthehub.com/ibm/. To build this project, one needs to add the respective jar file into the local Maven repository and place the respective library file at the java library path. Alternatively, one could also replace the given MILP solver by open source alternatives like Google OR-tools or entirely remove it from the project by modifying the respective factory.

**Deployment**

In general, this project is meant to be run within a Docker environment. To this end, we provide a docker-compose.yml file to setup the entire application, several docker-files to create images for each service and Spring application.properties files for the deployment in docker. This allows to easily deploy the entire application in a Docker environment just with a few manual steps:

1. Build with maven: (e.g. mvn install in the root directory).
    a. To make use of the IBM CLEX MILP Solver, one needs to install the jar file at the local maven repository. The following groupId, artifactId and version should be used for installation:

```
<dependency>
                <groupId>com.ibm.cplex</groupId>
                <artifactId>cplex</artifactId>
                <version>1.0</version>
</dependency>
```

2. Copy and rename the generated jar-files from the target directories into the respective sub-directories in the provided Docker directory of the repository. The expected name of the jar-file can be found in the respective dockerfiles.
3. To make use of the IBM CLEX MILP Solver, one needs to put the files *cplex.jar* and *libcplex<version>.so* (e.g. *libcplex1263.so*) into the directory Docker/TourWorker/cplex.
4. To use the experimental GUI (only a couply of static HTML files) one has to acquire a API Key of the Google Maps API use the key in the respective files.
5. Once these steps have been performed, the application can then be simply deployed with *docker-compose up –build*

In addition, the sub-projects contain Spring Application.properties files in the resource directories with all necessary default values. These configs can be used when the application is run from an IDE. However, the all services expose their REST interface on different ports, use default configs for credentials (e.g. postgres-123456) and assume that the basic environment has already been setup manually (i.e. postgres, mongodb, rabbitmq)