VU Advanced Internet Computing (184.269)

Project Topic 1 – Stream Processing for Spatially-Distributed IoT Systems

Institut für Informationssysteme – Distributed Systems Group

Description

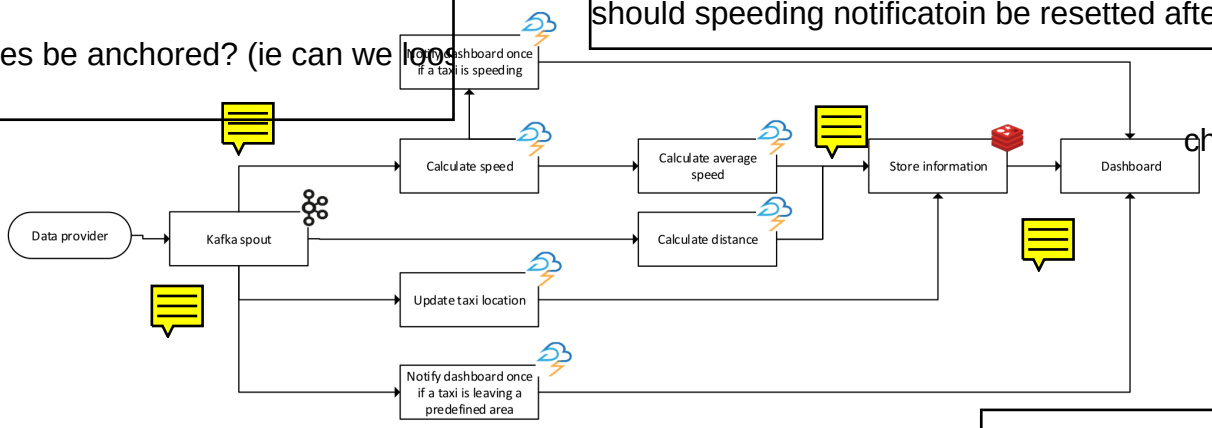
IoT systems which operate within a dynamic spatial environment are becoming ubiquitous; think of a taxi fleet within a smart city optimizing spatial distribution with respect to passenger demand, or a manufacturing floor co-habited by humans and robots. Those represent an important class of cyber-physical systems, which are faced with the manifold challenges that a dynamic spatial environment brings. They demand operational management to handle large amounts of data, often in real time.

This project entails realizing a monitoring dashboard, which is updated in real-time based on spatial streaming data obtained from IoT devices. To provide information in real-time, data needs to be processed by a sequence of different stream processing operators, such as filters or aggregators.[1] These individual stream processing operators form a stream processing topology, as you can see in Figure 1.

The goal of this stream processing topology is to transform a series of geographical locations by different taxis into value-added information that can be visualized on a dashboard to monitor the taxi fleet. In this task, you are going to combine several data processing frameworks, such as Apache Storm,[2] Apache Kafka,[3] and Redis.[4]

Should tuples be anchored? (ie can we loos

should speeding notificatoin be resetted after it



check: https:/

Use something like sockets f

Figure 1: Stream Processing Topology.

The preliminary task for this project is to populate an Apache Kafka instance with the location information of the taxis.[5] This data set contains the location information of different taxis and your task is to replay the location information according to their timestamps, and to submit the replayed data stream to Apache Kafka. Apache Kafka acts then as a spout for Apache Storm, which processes the location information to obtain the value-added information and forwards the value-added information to a data storage as well as dashboard. The processing operators of this stream processing topology are implemented by several stream processing operators, which are described in detail below:

The **Calculate speed** operator calculates the speed between two successive locations for each taxi, whereas the distance between two locations can be derived by the Haversine formula. This operator represents a stateful operator because it is required to always remember the last location of the taxi to calculate the current speed.

[1] Bugra Gedik et al. "SPADE: the system s declarative stream processing engine". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1123–1134.

[2] https://storm.apache.org

[3] http://kafka.apache.org

[4] http://redis.io

[5] *T-Drive trajectory data sample*. https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/.

The **Calculate average speed** operator aggregates all speed information from the previous stream processing operator and calculates the average speed of all single speed information for a specific taxi. You may use this formula for calculating the average speed:

$$S_{avg} = \frac{1}{S_n} \sum_{i=1}^{S_n} S_i \qquad \boxed{\text{Savg\_n = Savg\_n-1 + (Sn - S}} \tag{0.1}$$

S_n represents the number of all individual speed information and S_i represents the individual speed information. This operator is stateful, since it has to collect all individual speed information to calculate the average speed for a taxi.

The **Calculate distance** operator calculates the overall distance for each taxi, whereas you can also use the Haversine formula.

The **Store information** operator stores the average speed for each individual taxi and the distance of the taxi ride so far to a Redis data structure store.

The **Dashboard** retrieves the data every 5 seconds from the data store, calculates the amount of all currently driving taxis, aggregates the overall distance covered by all taxis and displays the calculated numbers and current location of the taxis on the dashboard.

The **Update Taxi Location** operator forwards the current location of the taxi to the data storage.

The **Notify dashboard once if taxi is leaving the predefined area** and the **Notify dashboard once if taxi is speeding** operator analyze the speed respectively location of each taxi and whenever a taxi is faster than a predefined speed limit, i.e., 50 km/h or leaves a predefined area, this operator forwards the information to the dashboard. The dashboard should issue a warning whenever a taxi is leaving the area with a radius of 10 km around the center of the Forbidden City in Beijing. The dashboard should discard any information about taxis who leave the area with a radius of 15 km around the center of the Forbidden City in Beijing.

Outcomes

The expected outcomes of this project are two-fold: (1) the actual project solution, (2) two presentations of the results.

Project Solution

The project has to be hosted on a Git repository provided by the Distributed Systems Group – you will get instructions how to apply for such a repository at the lab kickoff meeting. Every member of your team is assigned a separate Git account. *We will check who has contributed to the source code*, so please make sure you use your own account when submitting code to the repository. Further, it is required to provide an easy-to-follow README that details how to deploy, start and test the solution.

For the submission (see below), you also have to package your services in Docker containers and create a docker-compose script, which starts your implemented solution. You can find detailed instructions on how to setup your build and run configuration on TUWEL.

Presentations

There are two presentations. The first presentation is during the mid-term meetings, and covers at least the tasks of Stage 1 (see below), the second presentation is during the final meetings and contains all your results. The actual dates are announced on TISS.

Every member of your team is required to present in either the first *or* the second presentation. Each presentation needs to consist of a slides part and a demo of your implementation. Think carefully about how you are going to demonstrate your implementation, as this will be part of the grading. You have 20 minutes per presentation (strict). We recommend to use only a small part of the presentation for the slides part (e.g., 5 minutes) and to clearly focus on the presentation of your results.

Grading

A maximum of 60 points are awarded in total for the project. Of this, up to 35 points are awarded for the implementation (taking into account both quality and creativity of the solution as well as code quality), 20 points are awarded for the presentations (taking into account content, quality of slides, presentation skills, and discussion), and 5 points are awarded for individual accomplishments during the meetings as well as teamwork.

A strict policy is applied regarding plagiarism. Plagiarism in the source code will lead to 0 points for the particular student who has implemented this part of the code. If more than one group member plagiarizes, this may lead to further penalties, i.e., 0 points for the implementation.

Deadline

The hard deadline for the project is **January 20, 2020**. Please submit the presentations as a single ZIP file via TUWEL. The submission system will close at 18:00 sharp. Late submissions will not be accepted.

Test Cloud Infrastructure

Although the topic implementation can be developed locally without any cloud resources, we recommend every group to also run their implementations in a cloud environment. To this end, computational resources will be provided on Amazon Web Services for every student. However, a cloud deployment is not required and will not be taken into account for grading purposes. You will receive by email further instructions on how to access the cloud resources provided.

Stage 1

Your task in Stage 1 is to setup and configure Apache Kafka as well as Apache Storm. Furthermore, you have to implement the data provider to fill Apache Kafka with data. Therefore you may need to preprocess the data provided by the T-drive data set[6] dynamically, i.e. using a script. The data in the text files are structured as follows: The taxiId represents a unique identifier for one taxi, across all locations, the timestamp the time when the location was recorded and latitude respectively longitude describe the geographical location. It is advisable that the data provider emits an end token as soon as the data for one taxi, i.e., one data file, has been submitted to the system, to signal that this taxi has stopped and does not emit any further information.

Tasks:

1. Setup and configure Apache Kafka.
2. Implement the data provider, which represents a producer for Apache Kafka. This data provider retrieves the location information from text files or an SQL database and submits it to Apache Kafka. The data needs to be replayed in the same order as it was recorded, i.e., all location information with the same timestamp need to be submitted at the same time. It is required to make the submission speed configurable to apply different submission speeds during testing and actual performance tests of the system. Furthermore it is necessary to make the amount of displayed taxis configurable for demonstration purposes.
3. Setup and configure Apache Storm.
   - Implement a spout for Apache Kafka.
   - Implement the required operators for the topology. For those operators that maintain a state, you have to implement some kind of state management. To achieve this, you can either use the Trident State[7] or implement your own state management based on a simple key-value storage solution, e.g. based on Redis.
   - The processed data needs to be stored in a Redis in-memory instance.

[6] *T-Drive trajectory data sample*.
[7] https://storm.apache.org/documentation/Trident-state

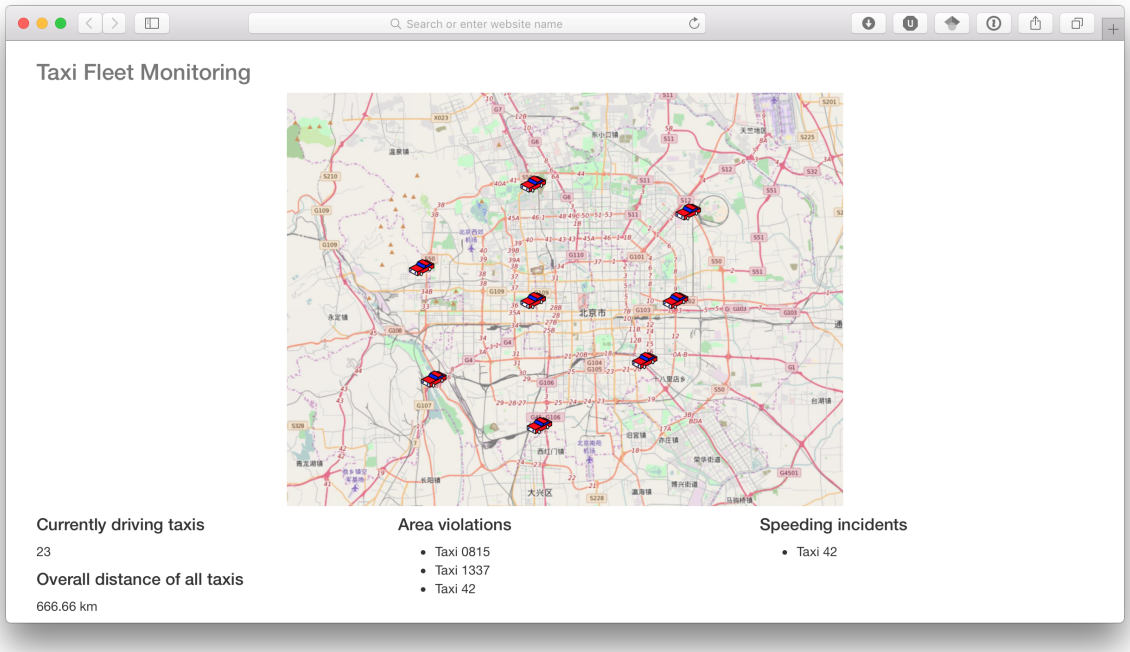


Figure 2: Fleet Monitoring Dashboard

Stage 2

In Stage 2, you have to implement the dashboard (see Figure 2), which retrieves the data from the Redis data storage or the stream processing topology and visualizes it for the user. You also need to evaluate the performance of your implementation, by determining the maximal submission speed your implementation can handle.

Tasks:

1. Implement a Web-based GUI, featuring all the elements in Figure 2.
2. Prepare a reduced dataset to visualize the movements on the dashboard.
3. Retrieve the processed information from the Redis data storage or stream processing topology and display the data in real-time, i.e., show taxis in the map and add a new incident for every violation retrieved from the stream processing topology.
4. Optimize your implementation to achieve a high throughput while maintaining valid results. For optimization ideas, you may have a look at the publication by Hirzel et al.[8]
5. Document and benchmark your applied stream processing optimizations and configurations for Apache Storm and their effects on the throughput, which can be monitored by the Apache Storm GUI.

[8] Martin Hirzel, Robert Soul, Scott Schneider, Bura Gedik, and Robert Grimm. A catalog of stream processing optimizations. ACM Computing Surveys (CSUR), 46(4):46, 2014.