Food Delivery Application

Davide Remondina, Marco Rodino, Andrea Zanin

March 1, 2023

1 Project overview

In this project we have built the infrastructure for a food delivery service. The platform is controlled using an admin dashboard through which items can be made available or not for purchase. The customers can make orders and check the status of the ones they already made. Valid orders are shown to delivery men which can then mark them as delivered.

We designed the system using the microservices architecture with a strong focus on fault-tolerance and horizontal scalability. Each microservice can continue to operate even if all the others fail and the failure of any service does not lead to data loss. All these desirable properties are achieved assuming that the Kafka topics are always available and do not lose data; this assumptions are reasonable in a Kafka deployment with sufficient replicas of each topic.

2 Microservice design

The system is composed of the following microservices: Users, Items, Orders, Validation, Shipping and Web; these communicate through Kafka topics. The users can interact with the system through HTTP requests, in particular the Web service hosts a website through which the users can interact with the system.

All the microservices keep local caches of all the information that they need to perform their function. This was a key design decision to ensure that each service can operate regardless of the failure of the others. These caches are built from scratch every time their parent service starts.

The Users microservice exposes an HTTP API to register a new customer (identified by their email) and stores the user information in the `users` Kafka topic.

A user can update its information (name and address) by performing a new registration with the same email; the old data is delete using log compaction.

The Items microservice works similarly to the Users microservice and it allows the admin to change the availability of items; this data is stored in the `items` topic. Furthermore the service keeps a cache of the available items, which is populated from the Kafka topic and thus receives also the availability updates made through another replica of the Items microservice. The cache is used to provide the users with an HTTP API from which they can load the available items.

The Orders microservice allows users to place an order, which is stored in the `orders` topic, the Kafka event contains a status field set to `REQUESTED`. Similarly to the Items microservice it keeps a cache of the orders and provides an HTTP API through which each customer can see all their orders.

The Validation service polls the `orders` topic to receive new orders, then it validates them by checking the availability of the requested items, it then stores the order with the updated status (either `INVALID` or `VALIDATED`) in the `orders` topic.
The validation is implemented by checking the requested items against a local cache of the available items; this implementation choice removes the need for remote calls in the validation process and thus improves the performance of the system, furthermore it allows the system to operate independently from the Items microservice.
If the order is valid the updated event is enriched with the user address, which is loaded from a local cache of the user data.

The Shipping service exposes an HTTP API that allows delivery men to load the `VALIDATED` orders which should be delivered and to mark them as delivered `DELIVERED`. The service keeps a local cache of the `VALIDATED` orders and stores the `DELIVERED` orders as events on the `orders` topic.

A drawback of the cache-based design of the microservices is that the caches are updated asynchronously, this can lead to temporary inconsistencies between different parts of the system. For example if an item is made available on the platform and a customer orders it, but the order is validated before the Validation service cache is updated to include this new item, the order will be marked as `INVALID` even though it was shown as available on the dashboard.

3 Fault tolerance analysis

The Web service simply serves static files, thus it is unaffected by crashes of any other service. If the Web service crashes, another instance can be spawned

without risk of data loss; indeed to ensure high availability many instances would typically be run at the same time.

The Users, Items, Orders and Shipping service has analogous behaviour as far as fault tolerance is concerned. When they crash and are restarted the cache may be lost, but it is automatically rebuilt from the original Kafka topic.
If the system crashes while handling a request to modify data (e.g. add a user, change the availability of an item, ...) the user will not receive a response, furthermore the requested action could be performed successfully or not depending on when the crash happens. We deem this behaviour to be acceptable, because the user can retry the operation manually if they don't receive a response.

The Validation service ensures that all orders appear to be validated exactly once even if the service crashes while validating an order.
This behaviour is achieved using the transaction feature of Kafka: the updated order and the new partition offset are stored atomically in Kafka; this ensures that either both updates are stored in Kafka, meaning that the order has been correctly processed, or both are not stored, meaning that the order will be processed again when a new instance of the service is started.

4 Scalability analysis

Several instances of the Web service can be deployed simultaneously without needing any particular implementation.

The Users, Items, Orders and Shipping service use a unique group id for each instance of each service to ensure that each instance reads all the events from the topics they cache locally. Thanks to this each service can be replicated to multiple instances, a load balancer can then be added to distribute the load of user requests over all the available instances of a given service.

Each instance of the Validation service uses a unique group id to read the `items` and `users` topic, in this way each instance has a cache of all the items and users. However all instances use the same goup id to read the `orders` topic, in this way each topic is read and validated only by one instance.
This design allows the Validation service to be scaled to multiple instances to handle a higher throughput of orders; the maximum throughput manageable by the service scales linearly with the number of replicas until the number of replicas reaches the number of partitions of the `orders` topic, further increasing the number of replicas has no performance benefit.

5 Demonstration deployment

To test the correct behaviour of the system we have deployed it as a swarm of Docker containers using the Docker compose tool: each instance of each service was deployed as a separate container and additional containers were used to deploy the Kafka broker and the Zookeeper service.

With this setup we tested that the system behaved correctly, both in a "one instance per service" configuration and in a "multiple instances per service" configuration. All the tests were successful.

The various containers are isolated and communicate through the Docker networking layer, thus the behaviour of the system executed within Docker accurately simulates the behaviour the system would have in a multi-server deployment.