Assignment 4 Report

Ole Rößler (7211)

December 2024

Contents

1 Design Exercise: Flight Booking System

1.1 Requirements

This section talks about the functional and non-functional requirements of the flight booking system wanted by the ACME Organization.

1.1.1 Functional Requirements

1. **Fetching Data:** Periodically receive flight data updates from partner airlines (every hour).
2. **View Flights:** The customer is able to see specific flight details. Therefore the customer enters a specific flight number/ flight ID and gets updates about the specified flight.
3. **Booking:** The customer is able to book a flight on a desired flight as well as make a seat reservation on the airplane (Most airlines offer different price-tiers for seats).

1.1.2 Non-Functional Requirements

1. **Price Consistency:** The system has to maintain consistency in pricing, therefore short term increases in flight-prices have to be avoided .
2. **Performance:** Low latency for user interactions (response within seconds).
3. **Scalability:** The system has to handle hundreds of thousands of concurrent users as well as hundreds of flights send in their data on the hour.
4. **Fault tolerance:** The system has to be reliable.

1.2 System Architecture

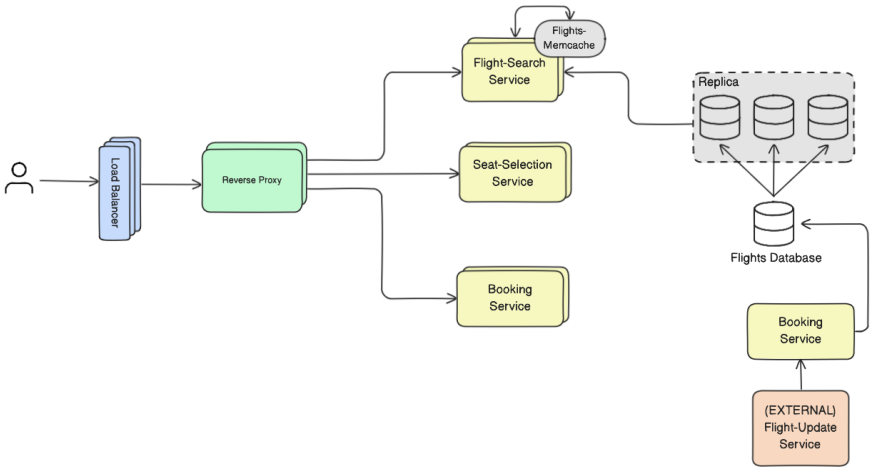

Figure 1: Redesigned flight booking system (FBS)

1.3 Discussion

The functional requirements are mainly implemented as micro-services. The Flight-Search service queries the database according to the users flight-data. The booking service is used to book a flight that meets the requirements and the Seat-Select service is called if a customer wants to make a seat reservation on a selected plane. The proxy handles security, also adding further loadbalancing before forwarding to the requested service.

In order to address the non-functional requirements and insure scalability and performance different measures were taken. The Load-Balancer distributes the incoming traffic equally to the proxy-servers so that the performance of the system isn't impacted by a single overloaded server (that would take way to long to respond to the user-requests). This also enables another point of scalability such that we can increase the number of proxy servers if the overall workload is to high. My analysis of the given flight-booking-system was, that it has a high write-to-read ratio (meaning the number of writes is way higher than the number of reads). Therefore we have database dublicates that all follow a single leader. Those replicas are read-only and help to satisfy the high read-demand. Furthermore the additional caching insures, that the database-reads don't bottleneck the application. In the future additional replica can be added to support the database layer. We could also add additional partitioning (by origin-destination). To scale the micro-services it is always possible to horizontally scale the micro-services by adding more instances. If the request arrives at the service it will be load-balanced internally and send to one of its instances.

2 Freestyle Exercise

2.1 My DS-Concept

I chose to implement a distributed version of our assignment server. The core improvement is, that the workload is distributed amongst various stateless servers that can access a database to update a representative "assignment-01"-score. To scale the number of servers horizontally, I implemented a loadbalancer that uses round-robin scheduling to distributes the work amongst the registered servers. The idea is that the clients can only communicate with the servers via the loadbalancer, that forwards the workload, therefore increasing the maximum number of requests that can be processed in parallel. To represent a distributed system on a local machine, I wrote a docker-compose file. This file includes multiple servers in different networks (net-1 and net-2; representing two data-centers) as well as references to the loadbalancer, the database storing the student data and an API. The API is used to read certain database information and ultimately display the students and the passed/failed exam-entries in a neat static front-end[2].

Assignment Results

Assignment	User ID	Passed
1	4328	❌
1	5231	❌
1	6220	❌
1	6430	❌
1	7014	❌
1	7211	✅

Users

User ID	Name
4328	student.name1
5231	student.name2
6220	student.name3
6430	student.name4
7014	student.name5
7211	ole.roessler
8888	test.user

Figure 2: Screenshot of the front-end

I also implemented a heart-beat mechanism that sends a udp-package from the servers to the loadbalancer every few seconds to signal that they are still alive and able to receive new data. I supported the processing of MessageTypes specified in the given ds-repo. The idea is, that the servers process given message-data and are able to write "assignment passed" or "assignment failed" into the

database. As it's only a prototype, I implemented a very easy test to simulate the behaviour of the real test-server: If the received ChatMessage equals 'TEST 1 USER ID: ' + UID and the userId equals the userId specified in the ChatMessage the test gets passed. The server will return a message contaning "TEST 1 USER ID CORRECTNESS FAILED/PASSED", depending on the results of the message analysis. By the way, all other ChatMessages are just echoed. Therefore the database only needs a table to save the assignment-scores as well as a table for all the registered users, as well as a few constraints(e.g. only one entry per user per assignment, only valid userIds in the assignments table). The resulting postgresql database-schema looks accordingly[3].

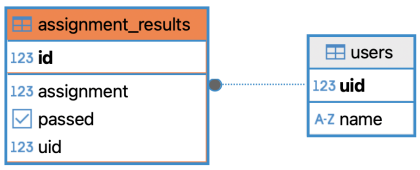

Figure 3: Enter Caption

To create a better overview of the created docker-compose "system", I included a screenshot of the running application in docker-desktop. We can see 3 different servers, a loadbalancer, the database as well as the api containers. The different servernodes are all running the same server-image but use different ports of receive data that was forwarded by the loadbalancer. The database instance is just a postgres-14:alpina container specified to allow connections via the db-net. For further details consult the dockerfiles as well as the docker-compose file and the source-code.

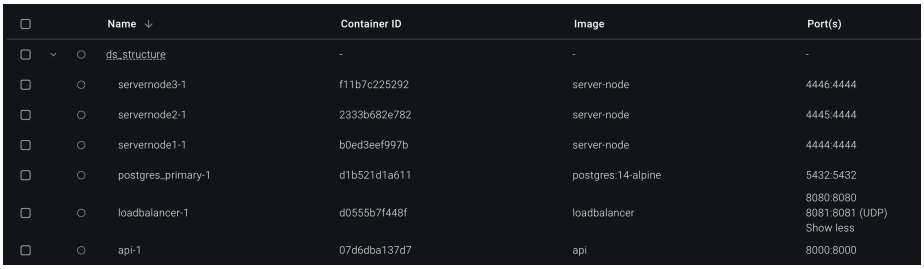

Figure 4: Screenshot of components via docker-desktop

2.2 Evaluation

2.2.1 Pros

1. **Functional Requirements:** The implementation matched all the functional requirements a ds-assignment server has. Even though I only implemented one simple test-case it can be seen as a POC that all the components interact in the intended way. The docker-logs also provide further evidence, that the loadbalancer relays the incoming requests in its intended way. The API interface also shows, that the test can be passed and the results are updated in the database. Therefore the functional requirements are definitely matched.

2. **Fault-Tolerant Heartbeat Mechanism:** Fault-Tolerant Heartbeat Mechanism: The UDP heartbeat ensures that if a server is unreachable, it is quickly removed from the rotation, minimizing potential downtime.

3. **Simplicity in Load Distribution** A simple design of components can be a good thing, especially when a more complex solution is not needed. The current design of the loadbalancer will be able to handle the current maximum number of requests with ease, therefore keeping the implementation straightforward is a plus.

2.2.2 Cons and Limitations

1. **Suboptimal Load Balancing:** Although round-robin is simple, it does not account for the actual server utilization. More advanced techniques like dynamic load balancing could distribute requests more efficiently when workloads are uneven across different requests. Especially when more complex test-cases are implemented, the workload might differ a lot (simple string comparison vs multi-step message protocol).

2. **Security Weaknesses:** Even though this is only a prototype, it has to be mentioned that I didn't consider security at all. The API currently lacks authentication (e.g., bearer tokens) and all ports are exposed through localhost. In a real environment the student should only have access to the loadbalancer and every following request, even to the server should include some sort of authentication.

3. **Single Points of Failure:** Both the load balancer and the database represent single points of failure. If the load balancer fails, the entire system becomes inaccessible(currently the servers could be accessed directly, but that would defeat the purpose of the entire system).

4. **Modularity:** Currently the database gets accessed from two different points, the API as well as directly from the server(in sql). Because we aim for a high degree of modularity, the split access-points are suboptimal. A future goal should be to remove the direct access of the database via the

server, but go through a API-Layer that specifies all possible database interactions very clearly.

2.2.3 Improvements and Future Work

As mentioned in the previous paragraph, moving to a clearly seperated API-Layer that handles all database interactions is definitely a clear future improvement. Also moving towards a distributed leader-follower database that implements horizontal optimization via replication(followers) and a RAFT implementation to handle leader failure, would add a significant part to the systems scalability and resilience.

2.2.4 Conclusion

The implemented system successfully demonstrates key distributed systems concepts through a custom load balancer, a simple API, and a single-leader database. While I am aware of the fact that 1000 clients can most likely be handled by a single server, this implementation aims to act as a prototype. Therefore the round-robin load balancing is more than sufficient for the expected user load, as well as an potential increase of comp.-sci. students. The loadbalancer could still benefit from utilization-based strategies. Additionally, enhancements in security, database distribution, and modularity are recommended to align the system more closely with nonfunctional requirements and to eliminate single points of failure.

Overall, the project fulfills its educational purpose of experimenting with distributed system design, while also highlighting areas for further improvement.