Distributed Systems Project Final Report

Winter Term 2022, Period II

Edgars Gaisins, Veli-Matti Laamala, Simon Lechner

The source code is publicly available on GitHub:
https://github.com/Vekkumasa/Distributed-systems-fall22

Project Overview

The main goal of this project was to create a webshop based on a distributed system. Basic functionality like showcase of products, user accounts and a shopping cart are implemented. All these functionalities can be accessed through the front end which completely hides the distributed backend from the user. Due to the time constraints of the project focus has been on the distributed backend, some features that are usually found on a webshop are not yet implemented but the implementation can easily be extended to all common features.

Design principles

Coming into the design part no member had developed a distributed system before, a lot of completely new aspects of system design had to be taken into consideration. Our first plan as stated in the design plan has been very focused on fault tolerance, splitting the system into two independent parts. One part related to product data and one to consumer data, this allows partial operation of the webshop even when the other component is unavailable. Each component has its own MySQL database and NodeJS server which handles requests from the frontend. REST API is used as middleware and allows for straightforward communication between the servers and the frontend.
After these parts were implemented our next focus was on node discovery, after extensive online research we came to the conclusion that implementing all wanted functionalities by hand might be too time consuming for such a short project spawn. Kubernetes offers all the features we were looking for and is tested extensively. We decided to deploy our system using Kubernetes although this meant learning a completely new technology for all of us.
Node discovery is handled by Kubernetes which creates a DNS for both services and pods. It further offers fault tolerance by allowing to replicate stateless deployments by simply specifying the wanted number of replicas. We decided to stick with the already implemented two component system although we knew this was more complex than the minimum requirement of the project description which could have been met with just a single database and server deployed and replicated on kubernetes.
The original idea of a shopping cart which connects the user and product data and is saved redundantly on both databases was not implemented due to several difficulties that would have

to be dealt with. Rather the shopping cart has been moved to the frontend, after feedback during an exercise session this was confirmed to be a reasonable decision.
Consistency is still implemented in the system by the replication of the database. Similarly to the servers, the product database is replicated and holds a consistent state. The difference from the NodeJS server replication is that not all database replicas are equal. There is always only one master database which is the only replica that accepts both writes and reads. The variable number of slave databases only accept reads. All writes from the master database are passed on to the slave databases - consistency and synchronisation. MySQL itself does not provide this functionality so it is done by an XtraBackup container running in the pod together with the MySQL database.
This also has a positive impact on fault tolerance, if one slave database fails there are still replicas with the same data which can be used. Since there is only one master database, if it fails it will be impossible to write to the products database until Kubernetes restarts it. It restarts quickly, however, and everything is back to normal with the previous state. State remains unchanged when the master database dies because the state is stored in Persistent Volumes.
It would be even better to also have multiple database instances that accept writes but this would lead to much more complicated consistency mechanisms. It is enough for a webstore to support more reads than writes because customers will be browsing product pages (database reads) more often than buying products (database writes).

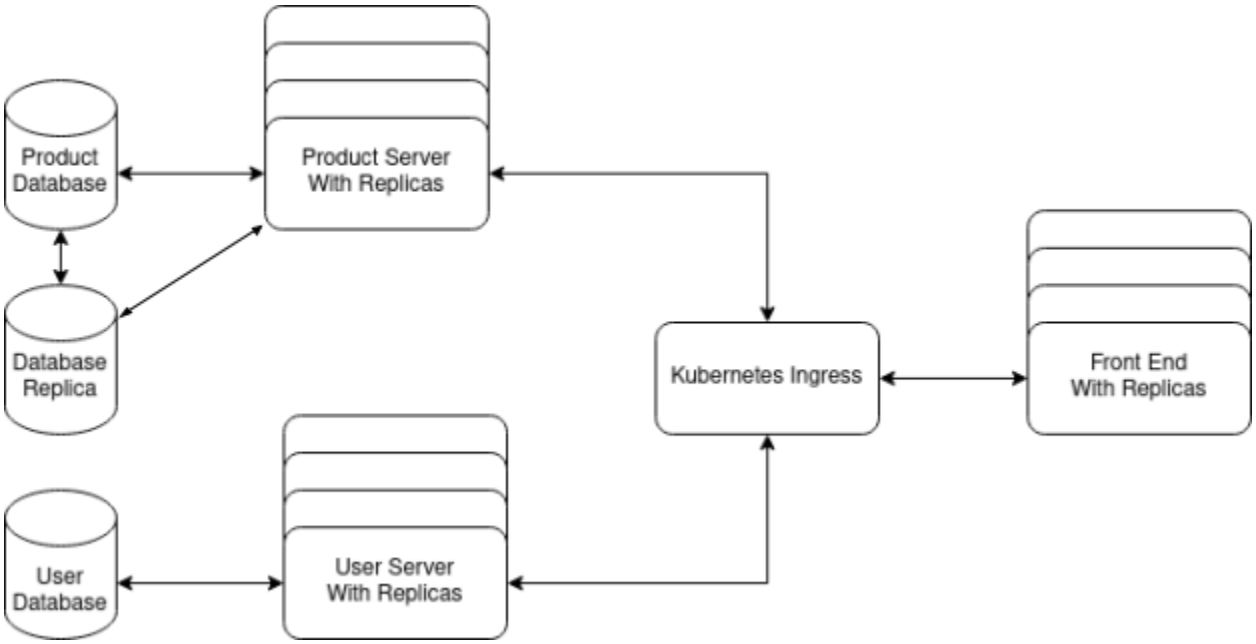

Figure 1: Complete Distributed System Architecture

As shown in Figure 1, a three tier architecture has been implemented, the frontends allow users to access the servers and databases. Both vertical and horizontal distribution is present, the two component architecture allows for placement of the product and server backend on separate

machines (vertical distribution). Kubernetes then allows for replicating each server, the frontend and product database (horizontal distribution).
The product and user NodeJS servers offer the following API endpoints:

Users

`GET /api/users`
Returns passwords and usernames of all users (Clearly only for development)

`POST /api/users`
Expects username and password in the request body and sends them back if it matches a user in the database

Products

`GET /api/products`
Returns all details about products in the database

`POST /api/buy`
Expects a list of product IDs and quantities to be bought. If there are enough in stock, returns the sum of the purchase.

`GET /dbinit`
Only for development, initializes the master products database with the products table and some example products

Both the user and product servers also have a **`GET /ping`** endpoint which returns a “pong” which was useful in debugging to determine server availability.

Functionalities of the System

Naming and Node Discovery

The first implemented feature in our distributed system has been the names of the services and how they connect to each other. After a failure the system should be able to automatically connect to the service once it has restarted. This is done by Kubernetes, during deployment services with names are defined which have a static IP address, other services always know their location from the DNS server which is automatically built during deployment of the Kubernetes services and pods.
For example, for the user NodeJS server we only need to specify that the database is hosted on by the *mysql-user* service without needing to know its IP address or other details. Kubernetes makes the connection for us.

```
env:
- name: MYSQL_HOST
  value: mysql-user
```

< node-user

mysql-user >

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-user
```

Fault Tolerance and Recovery

As described in the first section we followed the idea stated in the preliminary design plan. The general idea of our system design is that independent components can still function while the other component is down. Users can still look through products while the user server is down, the system still offers some functionality. Further it can easily be specified how many pods of each the user and product server should be deployed in the Kubernetes configuration files. This allows for full functionality even if one or more servers fail. The number of desired pods can also be increased or reduced at any time to adjust to increased and reduced demand. Stopped pods are automatically restarted by Kubernetes which provides recovery. Database state is also not lost if the databases die because their data is stored on Persistent Volumes.

Consistency and Synchronization

As described in the Design Principles section, consistency and synchronization are implemented by replicating the products database. All replicas have the same consistent state and updates to the master database are synchronized with the slave databases.

System scaling

The simplest form of our system consists of two databases, two servers, one ingress and the frontend. Due to the deployment of the backend on Kubernetes the product database and the servers can be replicated by any desired amount. This has a great impact on fault tolerance, availability and performance. There is a point of diminishing return in terms of performance when the consistent databases become the bottleneck. Due to the two component design of our system which reduces the number of operations on each server and database drastically, this should only be the case in very extreme cases. Kubernetes offers a load balancer for service replication which handles a high number of requests without any programming overhead from the developer.

Improve Performance

There are still several aspects which could be improved in terms of performance, but the coarse grain planning goal was to make all parts as easily extendable as possible. By splitting the system into two independent components the resource requirement of each part is reduced by a significant amount. Further Kubernetes allows for replication of each backend component with very minimal effort. Currently the replicated database is the biggest bottleneck if there are too many writes but it improves the fault tolerance significantly. For a big webshop some kind of caching technology should be used to bring data closer to the user. This would also improve performance drastically in our system but would have been too complex for such a short time. An easy but potentially expensive improvement would be to deploy the kubernetes cluster on a cloud provider such as Google Cloud, Azure, or AWS. They can provide more processing power

and scale to many more pods. We only deployed our system on Minikube containers running on our laptops which was good enough for development but certainly not for production.

Lessons learned

None of our group members had any experience with distributed systems before this project. The design phase has shown a rather steep learning curve due to the fact that such a great variety of things had to be taken into account. Very early on in the project we decided to implement the two component system design which has turned out more complex than expected but gave us great insight into aspects of real world distributed applications.
The biggest decision during development has been the deployment with Kubernetes, it offers many desired features of distributed systems. Learning a new technology from scratch in such a short time was challenging but has given us knowledge that is very applicable also after this course. One of the main lessons learned that we came across in a lot of computer science applications is that features should not be implemented from scratch if there is a library or technology which already offers the desired functionality. For small projects it is impossible to reach the same quality and amount of testing. Leveraging the key functionalities to Kubernetes allows for great scalability to a high number of nodes and guarantees a smaller number of bugs compared to a self implemented solution.

Group member participation

Edgars Gaisins: System planning and design decisions, set up of databases / node js servers, replication of databases, report / presentation work
Veli-Matti Laamala: System planning and design decisions, front end development and deployment, work on backend, report / presentation work
Simon Lechner: System planning and design decisions, deployment of backend, connection of backend services, report / presentation work