DISTRIBUTED SYSTEMS REPORT

Marios Papamichalopoulos CS3190006
Christos-Charalampos Papadopoulos CS3190009

INTRODUCTION

Our goal for this project is to design a robust system for e-voting with horizontal scaling for high throughput that could manage the votes of a small country like Greece. Based on [1], the people that voted in Greece were approximately 5,769,644. This results in 68 votes/sec or 34 votes/sec if the voting phase lasts one or two days respectively. Thus, what we wanted to achieve is our system to be able to manage 34 votes/sec and not lose one of them in the process.

We started by trying to implement our approach using a web framework such as *Spring Boot* or *Django*. We tried to find a way to create the clusters from scratch and not use something preexisting, since this is the point of the assignment.

All database services have their own implementations of replicas and clusters, like *MongoDB*, *PostgreSQL*, *MySQL*. Because of that, most web frameworks do not support custom implementations of database transactions. What that means is that we could not set up Spring Boot or Django to have multiple databases that communicate with one another or return custom error codes the way we wanted to. For example, we could not return "-2" in case of a replica not responding in time, or declare a custom upper limit for a replica to be considered as non responding.

We wanted to low-level tweak the database requests whereas these frameworks provided high-level repositories like JPA. This enables developers to avoid writing boilerplate code so as not to make mistakes or lose precious time. By using such interfaces they do not meddle with petty tasks but are unable to implement some heavy request customizations.

Thus, we used another python web framework called *Flask*. Flask is a pretty straightforward web framework that could do what we requested. It provides a lightweight API for endpoints which we could use in harmony with *psycopg2*, a python library for *PostgreSQL* queries, to write to the databases, return custom errors, make asynchronous requests etc.

Our system and the test cases used can be found here:

Github URL

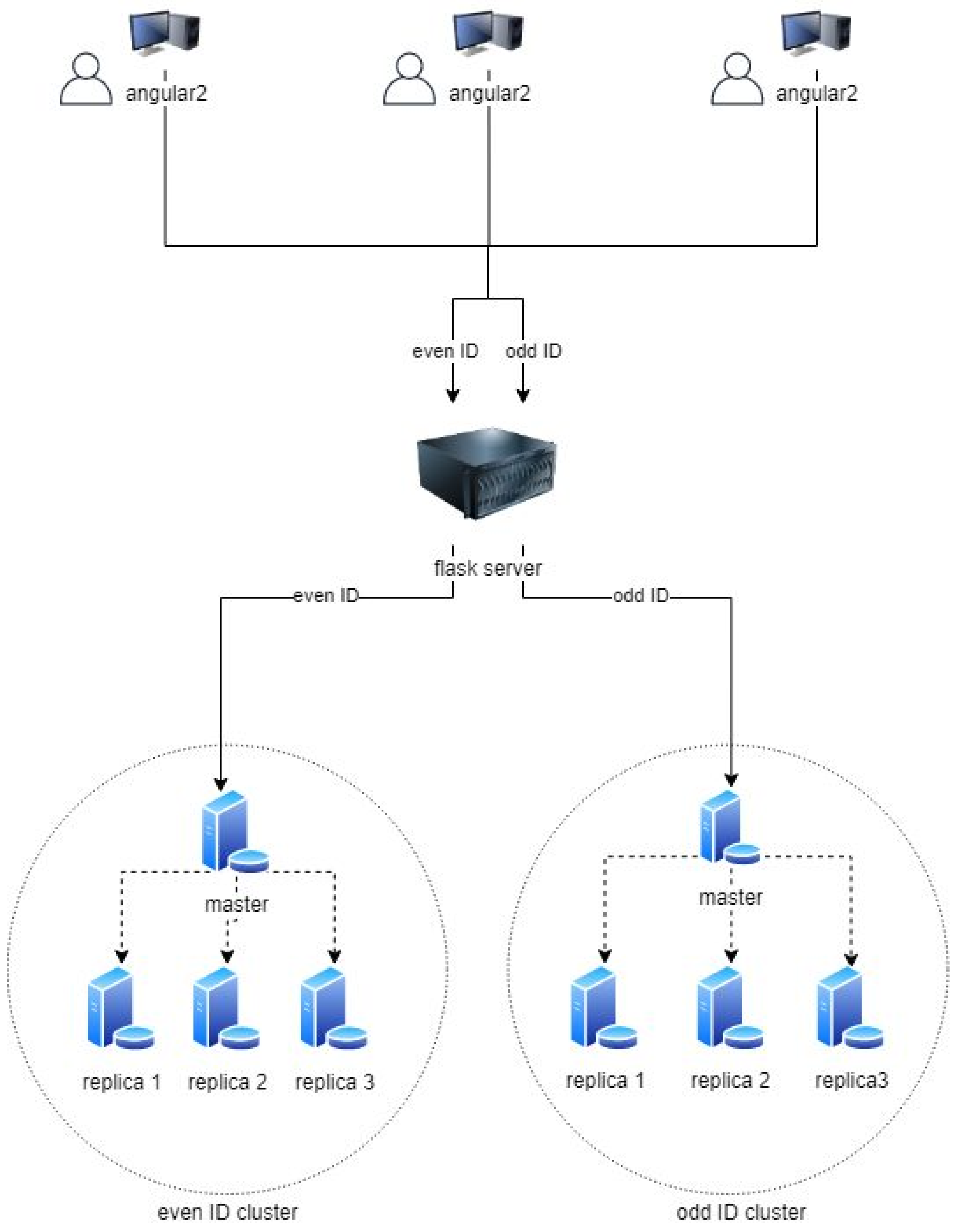

Figure 1. System Snapshot

DESIGN

Our main goal for our system is high throughput. Hence, we chose to implement an eventually consistent policy using async requests for replicas. For this, we looked up CURP [2] which is ideal for our system. The major goal in CURP is to take advantage of the commutativity between certain operations in order to reduce the 2RTTs needed for a request to 1 RTT.

In our case, a user can only vote **once** rendering all the database transactions taking place consist of write operations. No vote is going to change. Each user casts a single vote which is immutable.

By having these in mind, we designed a system that is highly available and highly partition tolerant with eventual consistency. Based on CAP Theorem we can achieve only two of the three mentioned, so we opted for an eventual consistency scheme to manage highest throughput possible.

The user casts the vote, the vote is written on the master DB, the master makes async write requests to the replicas and issues a confirmation to the user. The master is going to eventually receive the successful write requests from the replicas.

IMPLEMENTATION

Our project is developed as a web application. A snapshot of our system can be viewed at Figure 1.

Front-end

Our front-end consists of *Angular*, a typescript front-end framework. The user is able to cast a vote after submitting the form. There is no authentication whatsoever since it is just for testing purposes.

The view checks what ID is submitted from the user, meaning odd or even, and routes it to the appropriate API back-end method. The front-end actually acts as a router for the horizontal scaling API. A sample image of the dummy view can be seen below:

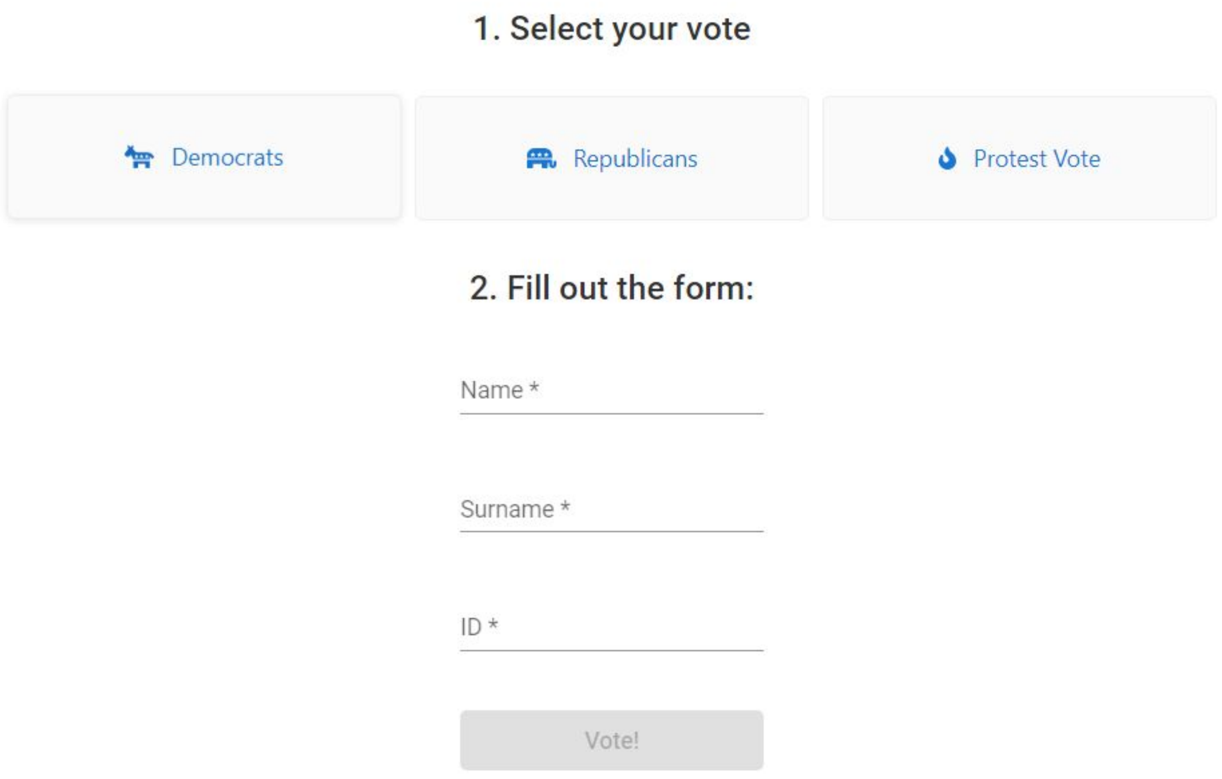

Figure 2. Angular View

Back-end

Our back-end is implemented using pure *Python* along with the *Flask* web framework. For our DBs we use PostgreSQL. In order to communicate with the DB we use *psycopg2* Python library.

The master node consists of a PostgreSQL DB which saves the data in a synchronous way meaning that incoming requests block the application until they are actually saved. Each request is then forwarded in an event-based fashion in the other databases (replicas). That means that each replica receives the vote but if something goes wrong the application is not halted. The master node has the vote saved and, since PostgreSQL is ACID, we can safely inform the clients that their vote has been submitted in 1 RTT rather than 2 RTTs. To store a vote, the asynchronous replicas establish a new connection to the database each time, submit the vote to be stored and then close the connection when the vote has been stored. In the witness (synchronous master node), the same connection is used each time, thus we save ourselves the overhead of recreating connections each time. The Flask server offers a REST API for the frontend to use and we have added custom errors and test/debug modes in the backend using pure python instead of using some framework.

BENCHMARKS

For the benchmarks we constructed a Postman collection, which can be also found in our github project. Using Postman's Collection runner one can sequentially make an arbitrary number of requests to our Flask server. Taking a look at CURP one can see that for their throughput they implement sequential requests as well.

Because there are two requests in the collection, when running the runner for 10 iterations there are going to be 20 requests, 10 for odd ID votes and 10 for even ID votes. In order to achieve random data and IDs we use Postman's pre-request scripts to construct them. The collection can be imported through Postman for a better look.

REQUESTS	TIME			
	master	master + 1 replica	master + 2 replicas	master +3 replicas
10 + 10	0.34 sec	1.10 sec	1.60 sec	2.97 sec
100 + 100	3.38 sec	8.72 sec	15.82 sec	20.32 sec
500 + 500	18.31 sec	43.2 sec	65.03 sec	98.63 sec
1000 + 1000	33.69 sec	88.45 sec	140.89 sec	192.01 sec

Table 1. Time benchmarks for system

However, there is a huge overhead due to psycopg2 library requiring to open a connection and close it if it is asynchronous for every request. Furthermore, we only have two shards.

Keeping those in mind, the benchmark results are encouraging. They can get even better by increasing the number of shards we could accommodate for more votes at the same time. More improvements can be found at the section below.

IMPROVEMENTS

In this section we reference some problems we ran across during our implementation or some future improvements that could be easily implemented in our E-Voting APP:

1. As mentioned for the DB operations we use a python library called *psycopg2*. This library as most of the open source ones has some pretty painful problems. Because psycopg2 is not thread safe some asynchronous operations on the replicas had complications.
    - We could not leave the connection open to the database. As a result, whenever an insert was called we had to open the connection, insert the vote and close it. This adds a huge handicap to the application.
    - Running simultaneous requests would not work, because of psycopg2.poll method. This function is used during an asynchronous connection attempt to check if a file descriptor is ready for reading, writing or an error occurred. The problem seems to be that when this function is called by different threads, the *POLL_OK* returned to one thread applies to all of them leading to erratic behaviour. Most of the time, the first thread writes successfully while the other threads encounter an error and the writes are never committed.

    In order to solve these problems we would find a new library to communicate with the Postgres DBs, since implementing a mutex would require tweaking the open-source methods.

2. Another feature which could be implemented is making the horizontal scaling "wider". In our implementation for the sake of the demo we only had two clusters, one for even IDs and one for odd IDs. As a result, each cluster accommodates half the population.

    In order to reach peak performance and throughput we would need 10 clusters one for each final user ID digit. For example, IDs ending in 0 would go to cluster 0, IDs ending in 1 would go to cluster 1, etc. This would actually take all the load and divide it by a factor of 10 making each cluster work less.

3. RAFT for consensus and partition tolerance. As mentioned in the CURP paper [2] the master node can be further enhanced using a protocol such as RAFT so that the "witness" set of data (the data structure/database that saves the data

synchronously) can be partition tolerant and prevent data being lost upon failure. We didn't implement this because a RAFT implementation in the context of a class project would be better done if implemented as a standalone project rather than part of an application like ours.

4. A feature that was omitted due to restricted resources and complexity was booting up a Virtual Machine for each master and replica. Instead of having a different DB for each master and replica in the same PC, we could use VMs that accommodate the respective DBs. The system would be more realistic and the requests would be from VM to VM, like in a real distributed system.

   The goal would be for each VM to boot up a Flask server instance and receive requests from the master node to the appropriate endpoints. After the vote insert in the local DB it would return back a confirmation message. If the master did not receive one from the replica under a fixed time limit it would render it as offline.

CONCLUSION

We implemented a prototype voting application based on CURP and our own design. After two unsuccessful attempts using Spring Boot and Django frameworks, we decided to use Flask and customize the DB requests, error codes, asynchronous requests, using psycopg2.

Our application supports storing replicas in 1 RTT and is partition tolerant, available and (eventually) consistent. Our benchmarks seem promising bearing in mind some of the overhead imposed by psycopg2. They actually are viable for countries like Greece if more than two shards are used.

Further improvements such as using some consensus protocol on the master database or creating more shards for a better throughput can upgrade the system to a whole new level.

REFERENCES

1. ypes.gr, “Εθνικές Εκλογές Ιούλιος 2019”, 2019[Online]. Available: https://ekloges.ypes.gr/current/v/home/parties/ [Accessed: 5 September 2020].

2. Seo Jin Park, John Ousterhout, “Exploiting Commutativity For Practical Fast Replication”, Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’19), USENIX, 2019, pp. 47-64.