DisVote - Introduction

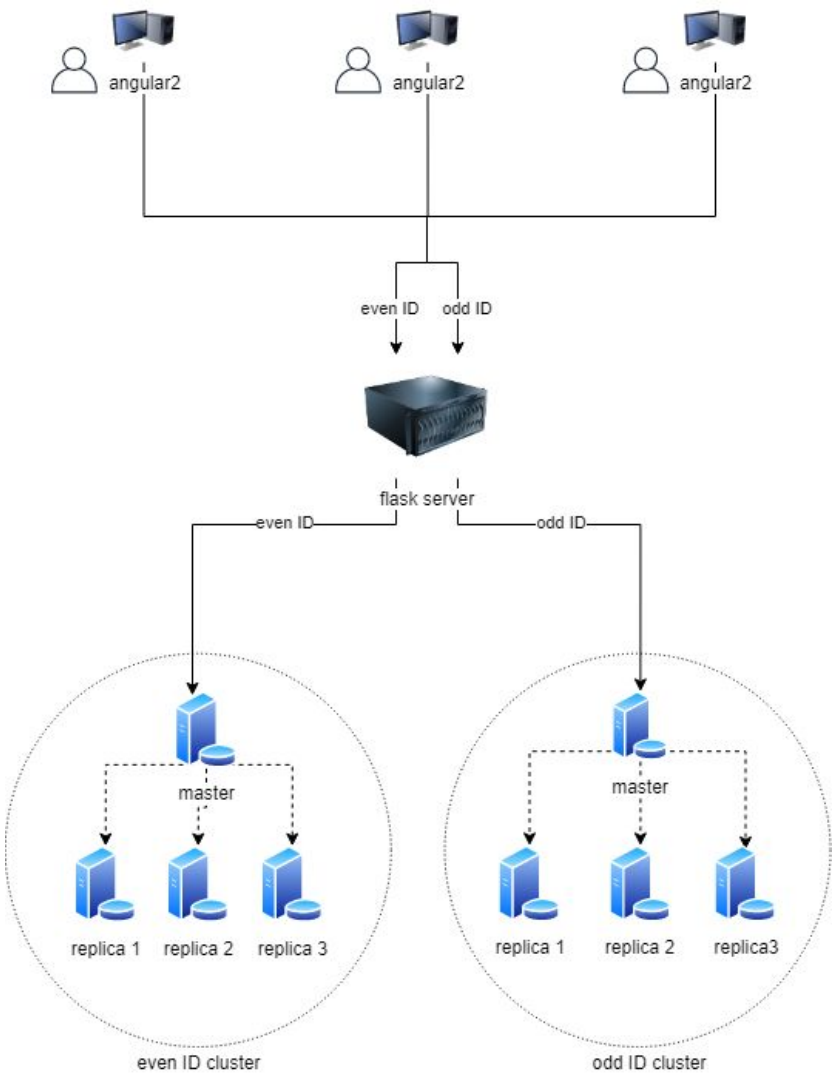
Our goal for this project is to design a robust system for e-voting with horizontal scaling for high throughput that could manage the votes of a small country like Greece.

Our main goal for our system is high throughput. This could be possible by implementing an eventually consistency policy using async requests for replicas.

We designed a system that is highly available and highly partition tolerant with eventual consistency. 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.

DisVote - Overview

Clients connect to the application, send their votes which are split according to the ID for better load handling and then stored in their respective node first to the master node (which ends the request succesfully for the user) and then to the replica nodes in an asynchronous way



DisVote - Details

All of our project is implemented using pure python and Flask+psycopg2. The master node consists of a postgresql database 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).

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.

DisVote - Benchmarks

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

DisVote - Benchmarks

there is a big 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.

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.

DisVote - Improvements

1. 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.
2. RAFT for consensus and partition tolerance. As mentioned in the CURP paper 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.

DisVote - Conclusion

We implemented a prototype voting application based on CURP and our own design. Our application supports storing in replicas with 1 RTT and is partition tolerant, available and (eventually) consistent.

Further improvements would be using some consensus protocol on the master database and create more shards to improve the throughput of our system