CMPE 275 Project 1 Report

Climate Facts

CMPE 275

Submitted by
Aastha Kumar
Maulik Bhatt
Kshitij Sood
Ardalan Razavi

Submitted To

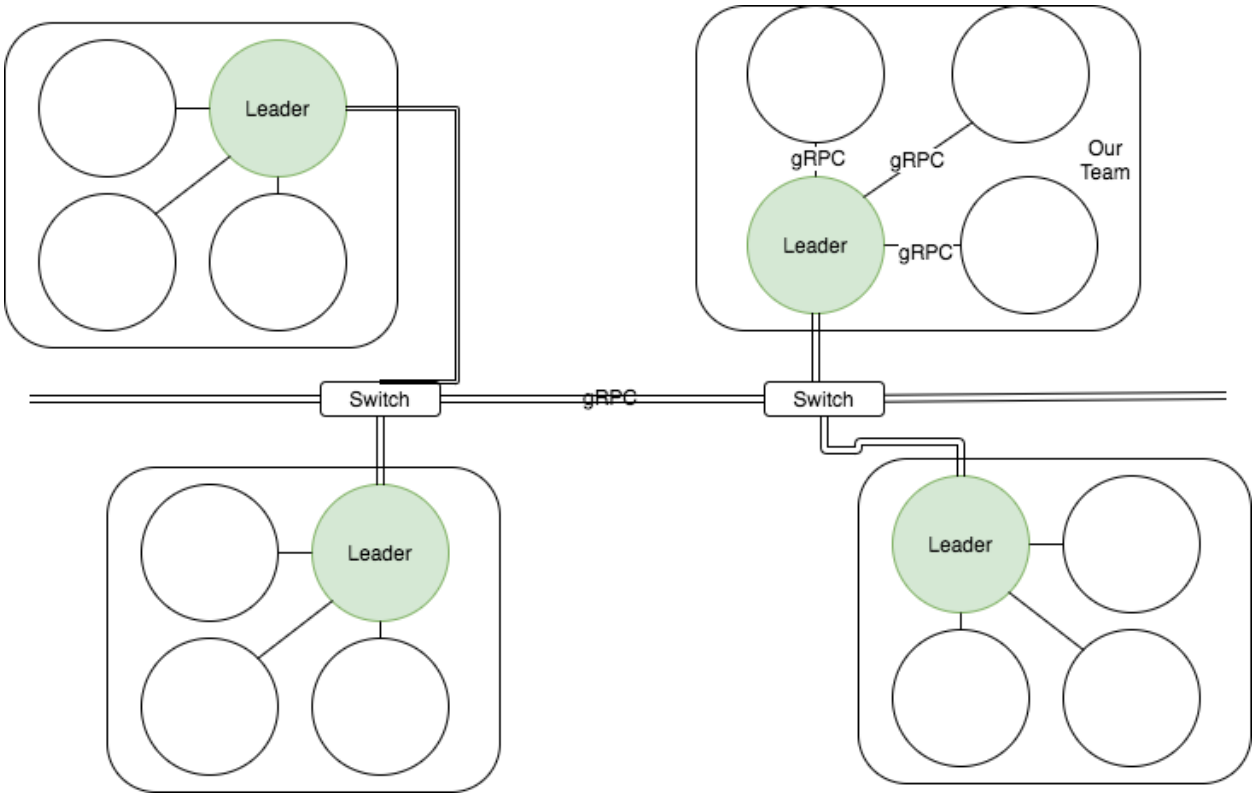
Professor John Gash
San Jose State University

Requirements

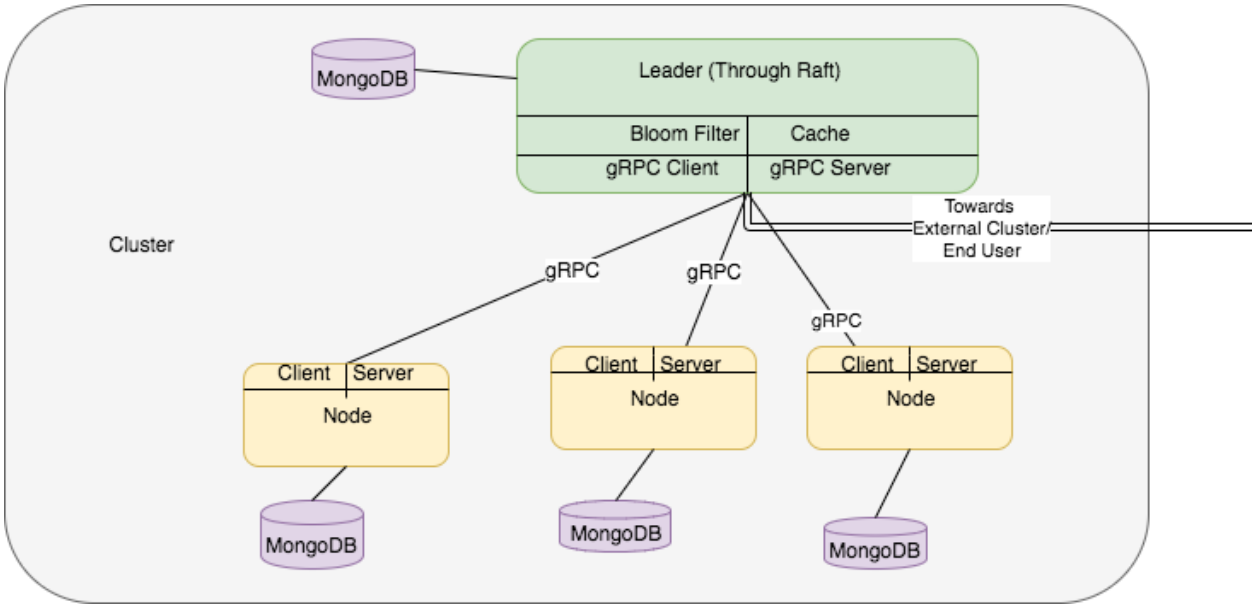
1. Create management strategies to store large datasets in distributed fashion.
2. Data will exceed the capacity of large system, so data should be rebalanced, routed or system should be allowed to grow dynamically.
3. Handle system/ nodes failure.
4. Handle events of request forwarding to other nodes in case the requested data is unavailable in node.
5. Manage frequently requested data using cache.

Architecture

In this project we have multi-cluster architecture like below diagram. Each cluster has different technology stack but the only common component is communication channel which is gRPC with Protobuf. All clusters agreed upon one proto file which contains structure of all methods and objects that are used for inter cluster communication.



Below is the architecture of our cluster. We followed hub-spoke architecture and point to point communication style. Also we used Raft for leader election so once a leader is selected it becomes point of contact for external cluster or end user. Each node has MonogoDB instance on it to store all data. When we load data into our cluster we create bloom filter at leader node and also keep updating it as new data comes. We used gRPC for intra cluster communication as well. If our leader receives request for data from external client/end user, leader will serve the data and also cache last few request's output into cache.



Components

The project included many components. We will go over a brief description the components.

Communication

The whole project was based on communication between nodes in a cluster, between clusters, and end-user with the system. The technology choice for communication was the most important decision that we had to make. We had many choices for inter cluster and intra cluster communication like RESTful services with JSON, gRPC, netty etc. We chose gRPC with Protobuf for inter as well as intra cluster communication.

gRPC with Protobuf

We decided to use gPRC as our mechanism for inter cluster communication. We compared RESTful+JSON and gRPC+Protobuf. gRPC+Protobuf showed better performance than REST+JSON for sample data.

gRPC+Protobuf	REST+JSON
Marginally Fast (after tested on sample + from other references)	Slow
Less readable format	More readable format
Fast serialization/deserialization as Protobuf is binary format	Slow serialization/deserialization

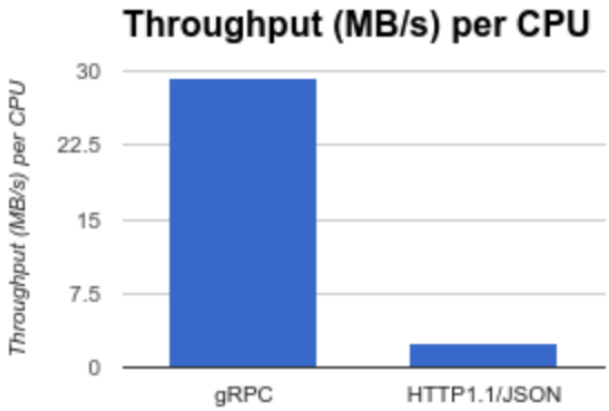


Figure: Comparison between gRPC and REST[1]

For intra cluster communication we also chose gRPC over other communication technologies like Messaging Queues etc. Because :

1. As we are running gRPC servers on each of our node so using these same servers for intra cluster communication will be better than running netty server on each node. Because running extra messaging server in addition with gRPC will consume more computing power than running only gRPC.
2. For using messaging queues we will need to pass incoming data from gRPC server to messaging server at leader node and also vice-versa for outgoing data. It may slow us down.
3. We decided for point to point communication architecture so in this case gRPC suits more than messaging queue.

Raft Algorithm

A consensus algorithm is: a cluster of servers should record a series of records ("log entries") in response to requests from clients of the cluster. (It may also take action based on those entries.) It does so in a way that guarantees that the responses seen by clients of the cluster will be consistent EVEN in the face of servers crashing in unpredictable ways (but not losing data that was synced to disk), and networks introducing unpredictable delays or communication blockages.

Raft works as follows. First, it elects a leader, then the leader records the master version of the log, telling other cluster servers what's in that master record and "committing" a log entry then responding to the client of the cluster to acknowledge that entry only when more than half the cluster has recorded a given entry. That works unless the leader crashes or loses communication with too many others; in such a case Raft elects a new leader. The election process is designed to guarantee that any newly elected leader will have (at least) all of the already-committed entries.

Raft nodes are always in one of three states: follower, candidate or leader. All nodes initially start out as a follower. In this state, nodes can accept log entries from a leader and cast votes. If no entries are received for some time, nodes self-promote to the candidate state. In the candidate state nodes request votes from their peers. If a candidate receives a quorum of votes, then it is promoted to a leader. The leader must accept new log entries and replicate to all the other followers. In addition, if stale reads are not acceptable, all queries must also be performed on the leader.

Optimisation in Favour of Speed, Fast Response to minimise fault tolerance

- **Fair leader election:** Leader is being elected efficiently and fairly. The candidate's term is matched with the follower's term every time a vote request is made. Also the candidate with the most updated logs is favoured.
- **Faster Recovery:** If the network is currently without a leader, the election is started as soon as first HeartBeat (after the leader went down) is transmitted in the network, this allows an efficient network recovery.
- **Minimal Network Clogging during App network communication:** Although we are using a mesh topology followers respond directly to Leaders and not through any other path (not via any node) so in effect its point to point communication for serving client requests.
- **Faster fault recovery:** If a node is down immediately the node is marked inactive in clients map.
- **Minimal Payload:** The accepted message format is of type LeaderRequest as defined in proto file so as to reduce payload weight over the network.

Implementation

Leader election using Raft Algorithm

1. Each node joins in a FOLLOWER state. A timer starts for each node. The nodes check cluster for the leader by asking other nodes in the defined config.
2. If Leader exists, the node functions as a follower.
3. If Leader does not exist, the node changes its state to CANDIDATE, increments the term and sends a requestVote message.
4. Once the requestVote message is received by the other nodes in the network, it checks candidate's term is greater than or equal to its term or leader is not reachable.
5. If both the conditions are satisfied, then it casts its vote to the candidate.
6. If the Candidate receives majority, it declares itself as Leader, changes the state to LEADER and sends the message to all the other nodes about its Leadership.
7. If the Candidate does not get the majority, then no Leader is elected for this term, the term ends and new term starts.
8. For the next term, new Election process starts, where all the nodes wait until the timeout and the election process is carried about as stated above.

High level flow of the solution

1. When a node joins the network it reads the config file and tries to connect with other nodes in the cluster.
2. Then node request the cluster to know the leader node, if leader has already been assigned for the cluster it stores the leader node details and continuously check leader state through ping request.
3. If leader has not been assigned to cluster, it participates in the leader election by changing it state to candidate and requests votes from other nodes in the cluster through requestVote rpc.
4. If majority of the votes calculated as math.ceil((active_nodes+1)/2) is received, it broadcasts itself as leader and publishes the its details to external end point for intercluster communication.
5. If majority of votes is not received and leader node remains indecisive, node times out for random election timeout and step 3 is repeated again till leader is found.

Cluster Communication

A cluster is a connection of nodes in the network. The cluster components are connected to each other through LAN. Performance and availability is increased by the use of cluster.

- **Inter-Cluster Communication:** Inter cluster communication is handled by leader node whose details are published to well known endpoint (external api hosted), so that other nodes can forward read/write request to our cluster.
- **Intra-Cluster Communication:** For intra-cluster communication, leader node is responsible for routing the request (read/write) to followers. Leader node maintains states of other node in the cluster. Read requests are only send to active nodes and write requests is send to nodes which are capable of writes i.e. node is not full.

Availability and Scalability

Availability: Our system is a cluster with leader elected by Raft algorithm. If leader goes down our, new leader will be elected using Raft algorithm. So even though nodes are going down, our cluster still works perfectly until last node goes down. Even because of replication factor we can recover data of lost nodes.
Scalability: As our cluster contains of nodes that each have mongoDB installed on it. So we are following distributed database concept hence any number of nodes can be added. Once new node is added to cluster it will start receiving new data in round robin pattern. Hence we can add as many node as we want according to our requirement.

Load Balancing

Whenever the leader gets the write requests from the client or external cluster, it distributes the data to internal nodes in round robin fashion. Each node can define its capacity, When the leader sends the write

request to a node, and if node responds with 'Capacity full', then that node will be removed from the node list for write request, read requests will still be sent to that node.
For read request, we have implemented bloom filter. Whenever the leader gets the request, it checks in bloom filter is the data is present in our system or not saving the database call and internode communication. If the data is not present in our cluster, then the requests is forwarded to external system and if the bloom filter responds that data is present then, the requested to forwarded to all the nodes. Response from all the nodes is sent back to client/ or external node.

Caching

We implemented caching using memcache, which is the high performance distributed memory object caching system. In order to decrease the database load and avoid the round trip to database, we thought of caching the frequent requests. But we also wanted to cache only those requests with small responses, as caching large responses like something in GB's could have overloaded the system, memcache was the ideal choice for that as it could only cache data upto 3 MB.

Hashing

Bloom filter - A bloom filter which is the space efficient probabilistic data structure is used to test the existence of the element in the set. A bloom filter might sometimes give false positives but it always correctly determine true negatives. Another interesting feature of bloom filter is that unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements. We non cryptographic hash - murmurhash for generating hashes..

We created the bloom filter for the set of dates existing in our cluster and used its definite true negatives property to determine if we have data existing in our cluster or if we need to forward the requests to external cluster.

Persistent Storage

Our initial choice for the persistent storage was Cassandra. We used the gossip protocol and all great features that came with Cassandra like gossip protocol for forming a cluster within our nodes. Since, the goal of the project was to implement everything like gossip protocol ourselves, we switched to MongoDB.

Mongo Database

The project used MongoDB in order to help our system to be dynamic as possible. There was no need for relational databases, even though it could be used. This way our system would be more flexible and adaptive to changes in our data models (if any change in future). Plus, it was a good choice to be used with Python which was our main language used.

Software Choices

Python

Python was an ideal choice for our project. It provided all the necessary drivers and libraries to interact with all our technologies and components. MongoDB driver and gPRC full support was in place for Python. Also, all team members felt comfortable with language choice which made the progress more efficient. The gPRC interfaces took care of data type validation for us, which again it made the python ideal choice.

Java

Java was used for our external client which was provided to our end users (Professor). The Java was able to communicate with Python code via gPRC interfaces.

Future Improvements

For Future, we can focus on two important factors:

1. Optimizing Raft algorithm: we have successfully implemented Raft algorithm with our own strategy. But the time for election is little longer and can be costly in critical situations. We can optimize our code and reduce time for election.
2. Replication Factor: When we fetch data from multiple nodes, its hard to figure out overlapping data because of replication in cluster. We are planning to optimize this part.
3. Support for multiple data type: Currently we little tightly coupled with data format. So in future we planning to support images and videos also.

Conclusion

In conclusion, the team learned a lot about the different technologies and algorithms used for consensus.

Git code base

https://github.com/soodkshitij/ClimateFacts

Contribution :-

We volunteered for load testing in class and our system performed well. We were able to integrate and test our system with all other teams in class.

	Maulik	Kshitij	Aastha	Ardalan
MongoDB	80%	-	-	20%
Java Client	-	85%	15%	-
BloomFilter + Hash	-	20%	80%	-

Cache	30%	-	70%	-
Python Client	25%	25%	25%	25%
Python Server	20%	40%	20%	20%
Raft	-	80%	-	20%
System Design	25%	25%	25%	25%
Load Balancing	50%	-	50%	-
Parser Code	20%	-	-	80%
Testing and Bug Fixing	25%	25%	25%	25%
Integration with other teams	25%	25%	25%	25%
Code Versioning and Repository Maintenance	30%	20%	20%	30%

Feature V/S Developer Table

References

[1]. https://cloud.google.com/blog/big-data/2016/03/announcing-grpc-alpha-for-google-cloud-pubsub