Jialiang Chen, Jose Luis Pueyo Viltres

Distributed storage systems and the CAP theorem

Sistemes Distribuïts

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Contents

Table of figures

1. Abstract

The present documents detail the implementation of two types of Distributed Systems, a network of independent computers that work together to appear as a single coherent system. Specifically, the centralized and decentralized versions of a distributed system will be implemented.

The function of this distributed system is to store {key, value} pairs (similar to dictionaries, the operations allowed are put (write value to the system) and get (read value from system). The systems will be implemented in Python using gRPC, a type of middleware that enables remote procedure calls.

The main objective of this project is to understand how a distributed system works, and different variants of it, their differences, advantages, drawbacks, etc. Once understood the algorithms used, structure and functionality, we will implement the centralized and decentralized versions of this system consisting of 3 nodes acting as a single system.

2. System Design and Discussion

To enable the communication between different instances of the application (node), the gRPC uses a series of "proto" files, used to define the interfaces and data (messages) types that gRPC uses to generate the necessary code for the intercommunication. In our case, we defined a series of methods and message types to enable the two protocols used in these two systems: 2 phase commit and Quorum:

```
service KeyValueStore {
  rpc put(PutRequest) returns (PutResponse);
  rpc get(GetRequest) returns (GetResponse);
  rpc slowDown(SlowDownRequest) returns (SlowDownResponse);
  rpc restore(RestoreRequest) returns (RestoreResponse);
  rpc prepare(PrepareRequest) returns (PrepareResponse);
  rpc commit(CommitRequest) returns (CommitResponse);
  rpc registerNode(NodeInfo) returns (Empty);
  rpc readVote(ReadVoteRequest) returns (ReadVoteResponse);
  rpc writeVote(WriteVoteRequest) returns (WriteVoteResponse);
  rpc registerSelfToOtherNodes(NodeInfo) returns (Empty);
  rpc doCommit(DoCommitRequest) returns (Empty);
}
```

Some of these methods are implemented in one of the two implementations, if any clients request a method that is not implemented in that implementation, an error will occur.

Both systems allow a delay to be added to the individual node to simulate slow performance of a specific node. This call is used by the evaluation tests (included with the project) to test the proper operation / scalability / failure tolerance of the system.

2.1 Centralized Implementation

The centralized version of the system consists of a **master** node and two **slave** (worker) nodes, the write operations are only allowed in the master node whereas the read operations can be requested from any node.

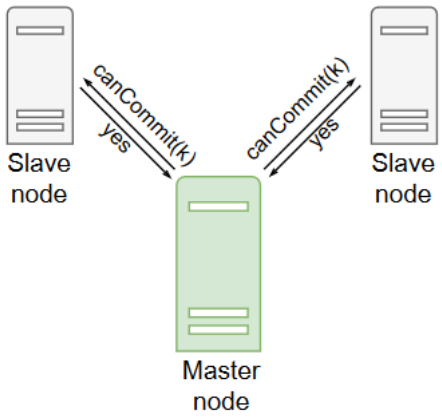


Figure 1: Centralized Structure

The put operation is carried out by the master when a user sends a PutRequesst to the master node, once received, master asks the other two nodes if it can commit (store) the key, values received, once both nodes agree to commit, it will proceed to save the data locally and send out two commit requests (one for each node) to update the other nodes.

Our nodes consist of a node_service, a proxy that the servicer (who implements the gRPC methods) delegates functions to, it contains method logic are used by the master_servicer as well as the slave_servicer (two implementations of the same system), each of those two servicers are the ones that are in charge of listening on a port to receive gRPC calls, and then delegates the logic to the proxy.

The initialization of the servicer initializes:

- configReader: A local service that oversees the obtainment of different node information (ip, port and id of other nodes)
- GRPCService: Used to get the stub of other nodes to send them messages if necessary (for example, slaves use this stub to register themselves to the master.
- NodeService: This is the proxy that contains all the application logic.

Inside NodeService we create a StorageService, which is the core of the application, it can store, reading values previously stored in the service. This class backs up data to a backup file each time it adds a new value or deletes a value. It is important to note that each node has their own storage service, so if any other node fails, we still have the data in the other nodes.

On top of this storage service, it also has a transaction_service, in charge of keeping track all the write requests it is receiving (during the prepare phase), since each request contains a unique is (UUID) that is related to a write (key, value). When master sends the prepare request, the slave nodes save the UUID and store values relation, if master sends a commit request for said UUID, it will proceed to store said value.

Lastly, a node_registrator_service is also used to keep track of all the nodes that the system registered, when other nodes register themselves to the node, it stores their data in this class.

The functionality of the centralized system is simple, if any of the slave nodes receive a put request, it will simply return success=False. In the master's case, if it received commit or prepare requests, it will also return error.

When master receives a put request, it needs to have the approval of both slave nodes to save that information, in our implementation, the master waits for the node's answer synchronously, meaning that if a node fails to respond, the master will very likely cause a big overhead for write requests.

2.2 Decentralized

The decentralized version of this system works similarly to the centralized, except instead of a master and two slaves, we now have 3 equivalent nodes. And the algorithm used to write / read values is the Quorum protocol, where each node has a vote weight, and each node is capable of reading / writing to the storage.

To carry out the write or read operation, the node first needs to ask the other nodes for their vote (yes = their vote weight, no = 0), once it collects enough votes (including its own) it can proceed with the operation, in this context, our vote threshold for read is 2 while it is 3 for the write operations.

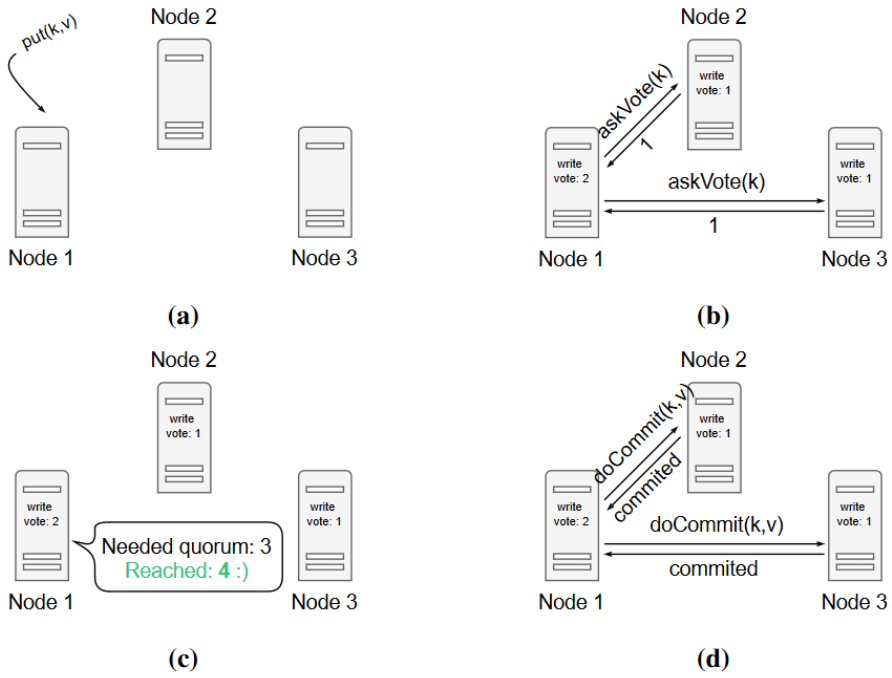


Figure 2: Decentralized Structure

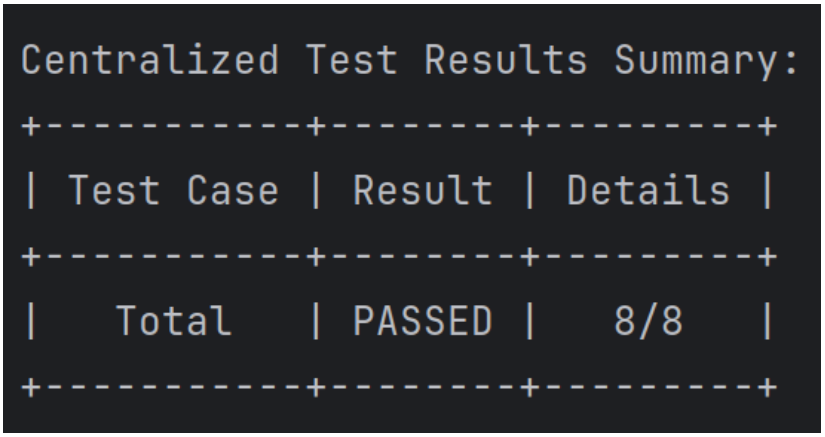
To enable this protocol, in the decentralized system we use threads when asking the other nodes for their vote, and we use a global variable that has a "mutex" to control controlled access and avoid race conditions. The requesting node creates n threads to ask n nodes for their votes, and starts waiting for the global variable's update, when this variable is changed, the requesting node checks if the minimum quorum has been reached, if so, it stops waiting and carries out the write / read operation. If it is a write operation, it will notify the other nodes (registered in the requesting node) of the change with a CommitRequest.

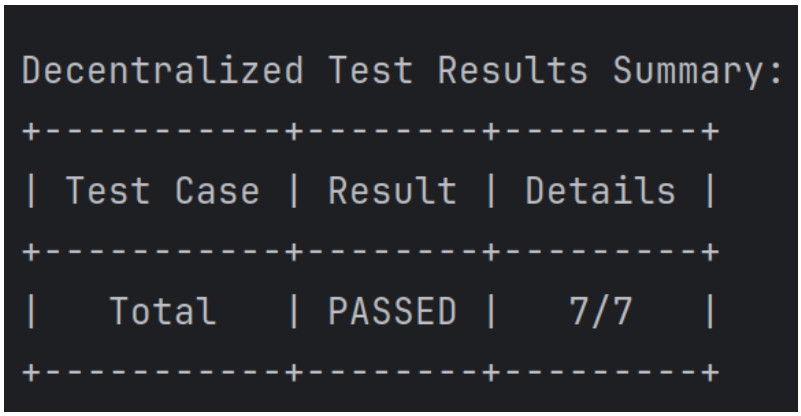
All the internal structure remains like the centralized version, taken the difference in gRPC methods implemented and the put / get methods' logic.

2.2 Main application

The main application (main_app.py) allows a simple yet functional way to interact with any node of any of the two systems, it asks the user if they want to use the centralized or the decentralized version of the system, followed by the operation they want to carry out and the value / key required to perform the request.

3. System comparison





Our implementations both pass the tests that were provided by the teachers, given the results we obtained:

Centralized:

- Performed 400 operations in 1.03 seconds.
- Performed 400 operations in 21.03 seconds (slowing down master).
- Performed 400 operations in 55.83 seconds (slowing down slave).

Decentralized:

- Performed 400 operations in 0.96 seconds.
- Performed 400 operations in 31.62 seconds (slowing down node).

Given the results, we conclude that the centralized system suffers much more when a slave is slow, because the master needs to wait for the slave node to prepare (with delay), then commit (with delay), resulting in double the wait time compared to a slowed master, where put / read is only slowed once.

The decentralized system has a similar throughput if no nodes are failing / slowed, but when one node is slowed, the other two are still capable of functioning normally, providing a faster performance compared to centralized.

4. Questions

4.1 Justify the consistency level of your centralised implementation.

Our implementation enables a prepare phase of the 2-phase commit protocol, enabling the consistency required in critical applications.

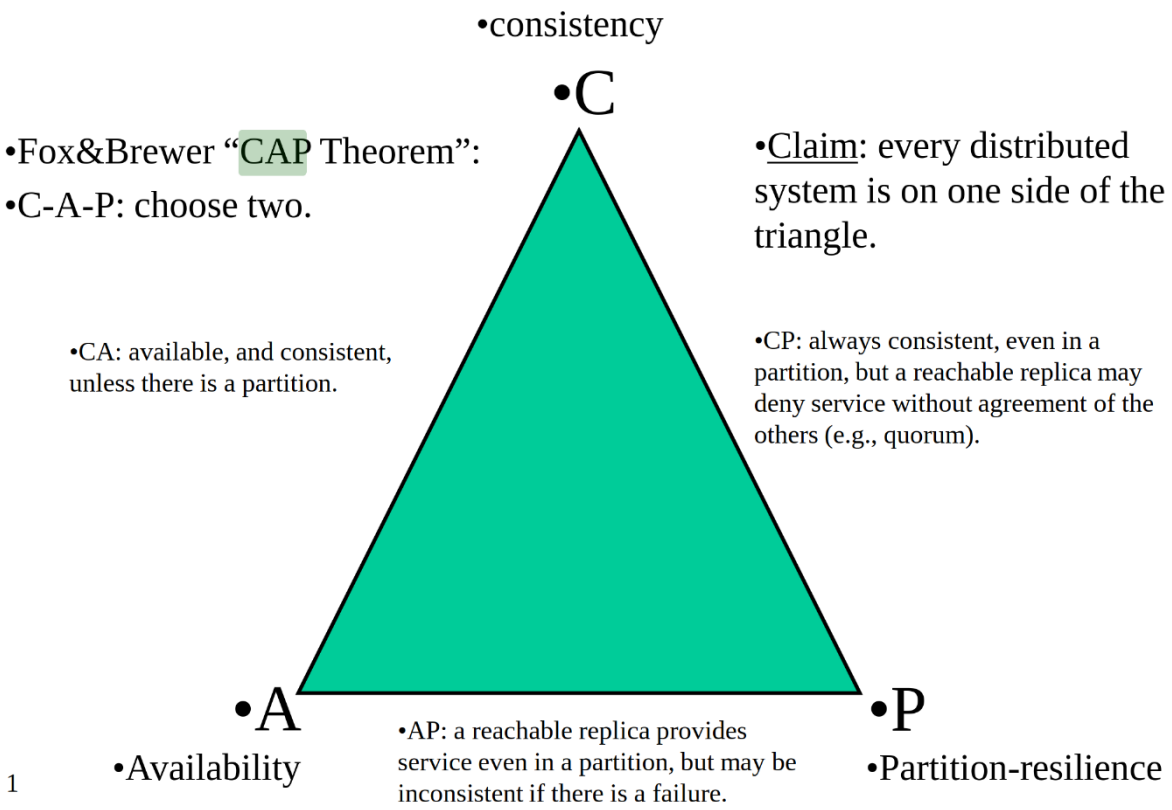
Specifically, when master sends out a prepare request, the slaves that receive the request will store the UUID and the key, value pair to a local service, and it **deletes** the stored value (in its local storage) that it had previously, this prevents read requests to return an old value, because any read request for a key that does not exist will result in error).

Once the master collects all favourable votes, it will issue commit requests (containing the UUID associated with the new key, value pair) to all slave nodes, successfully storing the updated value to their storage and enabling the read requests for said keys.

If any slave votes "no" to the prepare request, the master will issue a doCommit request with the old values of the key value pair to all nodes, restoring the values that slave nodes may have deleted previously, unlike commit, doCommit does not require prepare phase. This is because of the disagreement of one of more slave nodes during the prepare phase.

Our implementations also back up the data each time a node saves / deletes data to the local memory and it reads the local backup storage at startup. This enables a failure recovery, preventing data loss.

4.2 Characterize each of your implementations according to the CAP Theorem.



The centralized system we have developed fits more in the CA category, as it ensures availability and consistency across all nodes if there isn't a partition in the cluster. If a partition

occurs, no put operations can be carried out because the master won't be able to reach the node that was registered to it when it first initiated, therefore all prepare phases will at least have 1 False vote, leading to a system wide problem when partitions occur.

It ensures consistency because of the points commented in the 4.1 section.

The decentralized system we have developed fit in the CP category, as it ensures consistency across all nodes (even if it is in a partition) using the quorum voting system, where data is read / stored if and only if a minimum number of nodes agree previously. But a reachable node may deny service if insufficient nodes agree to read / write requests (doesn't provide availability).

GitHub page for the project:

https://github.com/xDontStarve/SD_Task2