Replicated Photo Storage System

By Rohit Raikhy, Lei Feng, Jodeci Wheaden

Summary Description

Overview

The purpose of this project is to apply four algorithms from our distributed systems course to a distributed system. The algorithms chosen were two phase commit, replication with single fault tolerance, and replicated data management, and timeout, and encryption and decryption. Two phase protocol is the process of ensuring that a client's request is distributed across all servers. The client can choose between 3 requests: upload, download, or delete. These requests will be sent to the coordinator that is in charge of coordinating data distribution. It will loop through each server participants and confirm that they can commit. If all servers have acknowledged that they can commit, all servers will implement the client's request. If one or more servers can not commit, all the servers will abort the request. Replication with single fault tolerance is our implementation of having multiple databases to store our files. Thus, if a database fails the distributed system will continue to operate without loss of data. Timeout is the process of timing how long it takes the server to respond to a client's request and this is implemented in our project. If the server does not respond to the client's request in 100,000 milliseconds the program will exit. The next algorithm we added is replicated data management. Each server connects to a database and creates a collection in that database so the number of replicated databases depends on the number of servers successfully connected. The final algorithm we implemented is encryption and decryption of the files sent between server and client but we are still testing the robustness of our implementation. The result of joining all of these algorithms is a replicated storage system.

Features

- All user requests will be manually checked and if it is invalid the client will receive an error message and be redirected to the original prompt.

- A logger has been added to both the client, server and coordinator to log all of the requests, responses and errors processed. Every line of the record has a timestamp with the current system time.
- All user requests will be manually checked and if it is invalid the client will receive an error message and be redirected to the original prompt.
    - The path of the files will be checked to ensure they are valid
    - The name of the file in the path and the name of the file given by the user must match one another in the upload operation

Design Choices

- We chose to extend my project two implementation of the RMI design which allows communication between the client and server by managing requests by the client through remote service objects.

Architecture Overview Diagram

The following architecture shown in figure 1.1 highlights a multi-client and multi-server model of our file storage application. This system presumes a transaction involves multiple processes on multiple machines. Hence, a consensus algorithm was needed to ensure atomicity of data in the server nodes. The algorithm proposed in our design to ensure atomicity is the Two-Phase Commit Protocol. In this case, either every process involved in the transaction commits or all of them will abort. We can see in figure 1.1 that there is a coordinator node which acts as the mediator between the clients and the server nodes. The server nodes are shown to contain a database at each node. In our implementation the storage facility offered is a replicated Mongo database across all server nodes using Cloud Atlas. Upon server node creation a new database is created allowing the application to scale out as data files across clients can be terabytes or even petabytes of information. Unfortunately, as time permits the application does not handle the case of new servers being added to maintain state and atomicity across server node storage. Further

implementation would require each server node the ability to speak to one another and therefore when a new server node is added, it can replicate the data from another server node previously running maintaining state through two phase commit protocol.

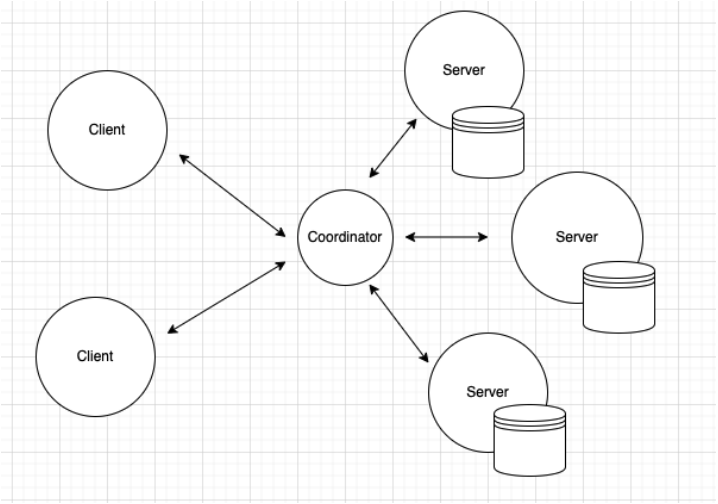

Figure 1.1, Two Phase Commit Consensus

Two phase commit is also not fault tolerant by nature, the algorithm fails to handle cases whereby server nodes begin to drop out of the system. This causes issues as the program scales or becomes ready for production. Each server node also contains a cache of file ObjectIds that were uploaded to the Mongo Atlas storage. If a server node restarts this cache will clear and therefore servers in this architecture are not fault tolerant by this nature. This can be improved by caching Object Ids in permanent storage to avoid this issue. Unfortunately, this also means a user would have to upload an image to be able to get the ObjectId which is saved in the server node's cache. Again this can be improved upon by storing all object ids in permanent storage. The coordinator node is built to be able to create multiple nodes, however, as previously stated the server node clusters do not have the ability to speak to one another and therefore cannot maintain consensus in replications by using this architecture. This can be implemented further to radically improve scalability in an application as the design would not have a single point of failure. A voting algorithm could be used in this case to maintain state in replica server nodes.

Replication with Total Ordering with Central Sequencer

Figure 1.2 shows a design that involves a central sequencer to handle replication atomicity. All requests are sent to the central sequencer, which assigns consecutive increasing identifiers to requests as it receives them. Our design is able to be improved upon to create the Total Ordering with Central Sequencer as the Coordinator node can be changed to an FE node and a central sequencer can be added. One of the server nodes could act as the central sequencer. This will therefore maintain atomicity in replications of server nodes and allow for scalability. There is a possibility one of the RM's could be a leader and send duplication requests to the other RM's but I will leave this to the reader to research further. There is a concern with fault tolerance in this design, as the load of the central sequencer can become very heavy as more requests are sent through the application and files can be very large in storage size. Hence, the application could fail in this regard at this central node. There is a way to remove the need of the central sequencer by using total ordering based on a distributed agreement, which is a method that will be seen in the following section on using PAXOS for consensus. Distributed agreement can be achieved using two phases. Firstly, each RM node can propose a candidate unique identifier cuid for a request which is then forwarded to the FE which issued the request. Secondly, one of the candidate identifiers can be selected by the FE which becomes the uid for the request. This uid is then communicated to the RMs. Again Mongo would be used as storage in each of the RMs.

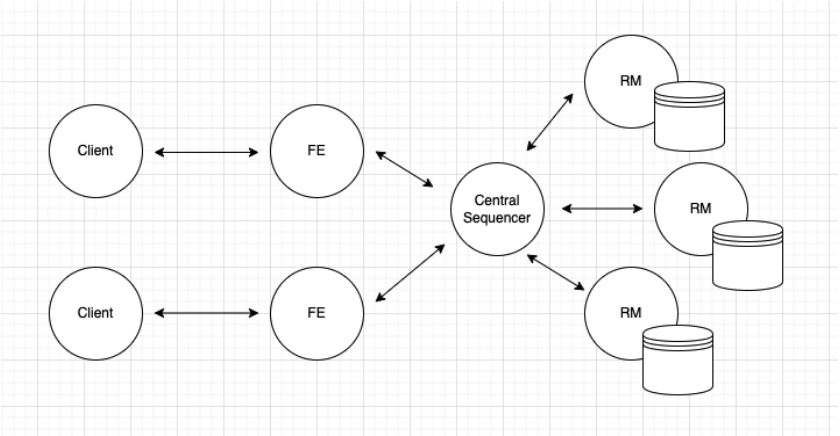

Figure 1.2, Total Ordering with Central Sequencer

PAXOS Consensus for Fault Tolerance

To be able to reach fault tolerance a better algorithm is needed in terms of consensus. There are a number of options such as RAFT but in our implementation PAXOS was considered as a means of achieving better fault tolerance. The algorithm would work as a message passing model to take consensus and would use a majority vote when committing files to the mongo database. The nature of the model is also scalable as the nodes can be created while maintaining state while some of the servers may also crash. There are three phases of PAXOS to consider in the file storage application (Proposal, Acceptance and Learning). The server nodes in the file storage system could act as all three methods. Firstly, the client would send a file request of upload or delete to the proposer node (download does not change state and therefore a consensus algorithm is not really needed.) The preparation stage would send a message to all acceptors in the cluster the proposer has the ability to speak to. Each of the acceptors in the system would then compare the message to the highest proposal it has saved and respond with a message back to the proposer. Again in our implementation for the future, the proposer could be selected by one of the server nodes. If a majority is reached then the request would be sent to the learner node to commit to the storage (Mongo Atlas in the application). This algorithm needs a majority of processes (2p+1) to survive and therefore would maintain fault tolerance much

better than the two phase commit algorithm implemented in our original implementation. Replication with data management can still be achieved to scale out server nodes for storage in this manner.

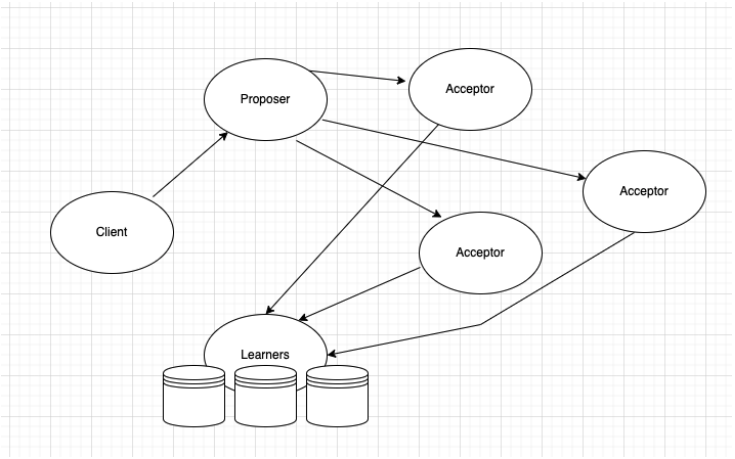

Figure 1.3, Fault Tolerant PAXOS Consensus

Key Algorithms

Two Phase Commit Consensus -

The two-phase commit protocol is responsible for distributing transactions across all servers in our network. The request is sent to the coordinator and the coordinator distributes the transaction across all of the servers. If all of the servers commit the transaction the transaction is successful but if one or more of the servers do not commit the transaction the transaction is abort for all of the servers.

Replication With Single Fault Tolerance -

Our project implements single fault tolerance by creating multiple databases so if a database fails our operation will continue since the data has been replicated.

Replicated Data Management -

Every time a server is connected a database is created with a collection in MongoDb. Hence, if there are five servers there are 5 databases with 5 collections such that there is a collection in each database. Therefore, we have replicated our data in the database.

Timeout -

Timeout was implemented by storing the time of the system before the coordinator is called. Then I call the system time again and subtract the start and end time. The result is the total time needed for the server to process the client's request.

Encryption -

Encryption function was implemented in the client part to encrypt the file after uploading the file. The encrypted file will be saved under our src directory. This encryption function increases the files' privacy and only the specific decryption function can unlock it. This algo now still needs further testing for the deployment on multi databases.

Conclusion

Our goal was to create a replicated photo storage system on a distributed network that satisfied the project requirements. We had to choose four algorithms that would allow us to achieve our goal and would also allow maximum success between our server and client. The client is sending files to the coordinator who is distributing the files across all the servers. The servers are connected to databases and that is where our data is stored. Once the request has been processed successfully the response is sent back to the client. The response could be as simple as a success or a file.