Multi-PAXOS Consensus Protocol

Kunal Goyal 160050026
Aman Bansal 160050028
N. Jagadeep Sai 160050104

November 2019

1 Problem Statement

We want to implement a system which can take requests from clients and process them. The requests from the clients include read and write requests and can potentially change the state of the server including addition/overwriting of the server data. The system should satisfy the following properties:

- **Safety**: This property states that no bad event should occur. For example, if a read and write occur simultaneously then it should not be the case that the read request get a fraction of data as it was before overwriting and remaining as it is after overwriting.

- **Liveness**: This property states that a good event should occur eventually. For example, if a user sends a read requests then it should be answered eventually.

- **Fault Tolerance**: This property states that the system should work even if one (or more) components fail. It should have a very high effective uptime. For example, if the system has a single server taking all requests and responding to them then the system is not fault tolerant as it has a single point of failure.

2 Proposed Method

The one plausible solution that we explore here is using a distributed system with a State Machine Replication approach. A lot of all this have been developed by two major researchers Edsger W. Dijkstra and Leslie Lamport 1(a) and 1(b). A distributed system is a system whose components are located on different networked computers, which interact with one another in order to achieve a common goal. A state machine replication or state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. The approach also provides a framework for understanding and designing replication management protocols.
2.1 Problems

- **No Shared Memory**: There is no memory that is shared by the nodes for the purpose of communication. Hence proper message passing should be done for any communication. We need a proper protocol for message communications.

- **Consistency**: Since each node has a replica of the system state and only one node is getting any request and all nodes will be getting requests simultaneously, it can be easily seen that maintaining consistency is a very major issue.

- **Handling Failure**: If one (or more) node fails then the other nodes should not get affected in the sense that they should not stop working. To the outside world it should appear as if nothing happened.

2.2 A Major Setback

**Theorem 1.** The FLP theorem [1]. In an asynchronous network where messages may be delayed but not lost, there is no consensus algorithm that can guarantee all three of Safety, Liveness and Fault-Tolerance.

**Theorem 2.** The CAP theorem [2]. In an asynchronous network where messages may be lost, it is impossible to implement a sequentially consistent atomic read / write register that responds eventually to every request under every pattern of message loss.

2.3 Solution

We will use a consensus protocol for solving all of the above mentioned problems.
The consensus protocol we will be using is PAXOS (or more precisely multi-PAXOS).

There are four phases in the protocol:

- **Prepare Phase**: A Proposer creates a message, which we call a 'Prepare', with a unique proposal number n. Then, it sends the Prepare message containing n to a Quorum of Acceptors.

- **Promise Phase**: If an Acceptor receives a Prepare message, the Acceptor must look at the identifier number n of the just received Prepare message. There are two cases.
  - If the highest proposal number received by the accep to r till now is less than n then it must return a message, which we call a 'Promise'. If the Acceptor accepted any proposal at some point in the past, it must include the previous proposal number, say m, and the corresponding accepted value, say w, in its response to the Proposer.
  - Otherwise the Acceptor can ignore the received proposal or send a 'Nack' message.

- **Accept Request Phase**: If a Proposer receives enough Promises from a Quorum of Acceptors, it needs to set a value v to its proposal.
  - If any Acceptors had previously accepted any proposal, then they’ll have sent their values to the Proposer, who now must set the value of its proposal, v, to the value associated with the highest proposal number reported by the Acceptors.
  - If none of the Acceptors had accepted a proposal up to this point, then the Proposer may choose the value it originally wanted to propose.

The Proposer sends an 'Accept Request' message, (n, v), to a Quorum of Acceptors with the proposal number and its chosen value.

- **Accept Phase**: If an Acceptor receives an Accept Request message from a Proposer, it must accept it if and only if it has not already promised for some higher proposal number.
– If not, it should register the value of the just received Accept message as the accepted value of the Protocol, and send an ‘Accepted’ message to the original Proposer as well as every other proposer.
– Else, it can ignore the Accept message or send a 'Reject' message.

Note that an Acceptor can accept multiple proposals. This can happen when another Proposer, unaware of the new value being decided, starts a new round with a higher identification number n. In that case, the Acceptor can promise and later accept the new proposed value even though it has accepted another one earlier.

However, the Paxos protocol will guarantee that the Acceptors will ultimately agree on a single value.

One important thing to note is that this protocol doesn’t guarantee any progress. It ensures safety and fault-tolerance and hence, as a result of FLP theorem, cannot guarantee liveness.

3 Simplifying Assumptions

We will not focus on the exact effects of the requests. We will just represent a request received as a basic entity and will represent it with a unique identifier. For simulating its execution, we will just add in the system state log that this request is executed. In this scenario consistency means that the log of each node should have the identifier of the requests in the same order. We will also assume that no message can get lost. It can only get delayed. Hence the CAP theorem is not applicable here.

4 Solution Details

Let the number of nodes be $N$. The main phases of the protocol are as follows:

- **Initialization Phase**: We run the standard PAXOS consensus protocol to decide upon a leader. Once a leader has been agreed upon and all the nodes have received the information of the leader, we continue to the next phase

- **Operating Phase**: Any node upon receiving a request sends it to the leader. The leader receives requests from all nodes and processes them and then it sends the requests to all the other nodes along with the order in which to apply the requests to their memory.

- **Recovering Phase I**: If any of the non-leader node fails then the system goes into this phase. Operationally, there is no difference than the operating phase. The only difference is that the messages sent to failed node get dropped in the network. The node should be brought up as soon as possible.

- **Recovering Phase II**: System goes into this phase when the leader fails. There are few issues we have to handle in this phase:
– To inform the other nodes that the leader has failed we send a regular heartbeat from the leader to everyone else. A node assumes that the leader has failed if the heartbeat doesn’t come within a specified timeout.

– To re-elect another leader every node starts the PAXOS protocol again and a new leader is elected. When the old leader comes back it looks for heartbeat for finding the current leader.

– If a heartbeat gets delayed and a node incorrectly assumes that the leader has failed and stats the protocol then the other nodes upon receiving a leader re-elect message send a message saying leader is alive.

5 Problems Faced

• Understanding how and why the protocol works was quite challenging and it took us time to understand every aspect of the protocol

• We were facing some issues because of inherent infinite loop of the protocol and had to use various techniques like exponential back-off to avoid that. This really increased our work a lot. Understanding why the system is going in infinite loop and how to avoid it really added to our understanding.

• Leader failures are detected by timeout of heartbeats due re-election is non-trivial due to presence of nodes which still believes the previous leader is present (due to delays in heartbeat causing differences in heartbeat timeout).

• For practical purposes having a single timeout on reaching majority of promises was causing a significant delay for larger n. For this introduced another timeout to wait after reaching majority.

References
