CONSENSUS ALGORITHMS WITH EC2

In distributed computing, a classic problem is achieving consensus among multiple parties. In a design with multiple parties, there may be times when a majority (or potentially all) of the processes involved need to agree. The protocols relating to achieving this agreement are typically called consensus algorithms.

One of the harder problems with achieving consensus is dealing with and recovering from failure of processing during the consensus activity. Most algorithms are focused on achieving as consistent and as reliable a consensus while anticipating failure. The goals of most consensus algorithms usually include:

- **Validity**—The final answer that achieves consensus is a valid answer.
- **Agreement**—All processes agree as to what the agreed upon answer was by the end of the process.
- **Termination**—The consensus process eventually ends with each process contributing.
- **Integrity**—Processes do not vote more than once.

Many consensus algorithms contain a series of **events** (and related messages) during a decision-making **round**. Typical events include **Proposal** and **Decision**.

One of the most important consensus algorithms was the Paxos algorithm, first defined by Leslie Lamport in the 1990s. It defines several roles played by processes during the consensus algorithm:

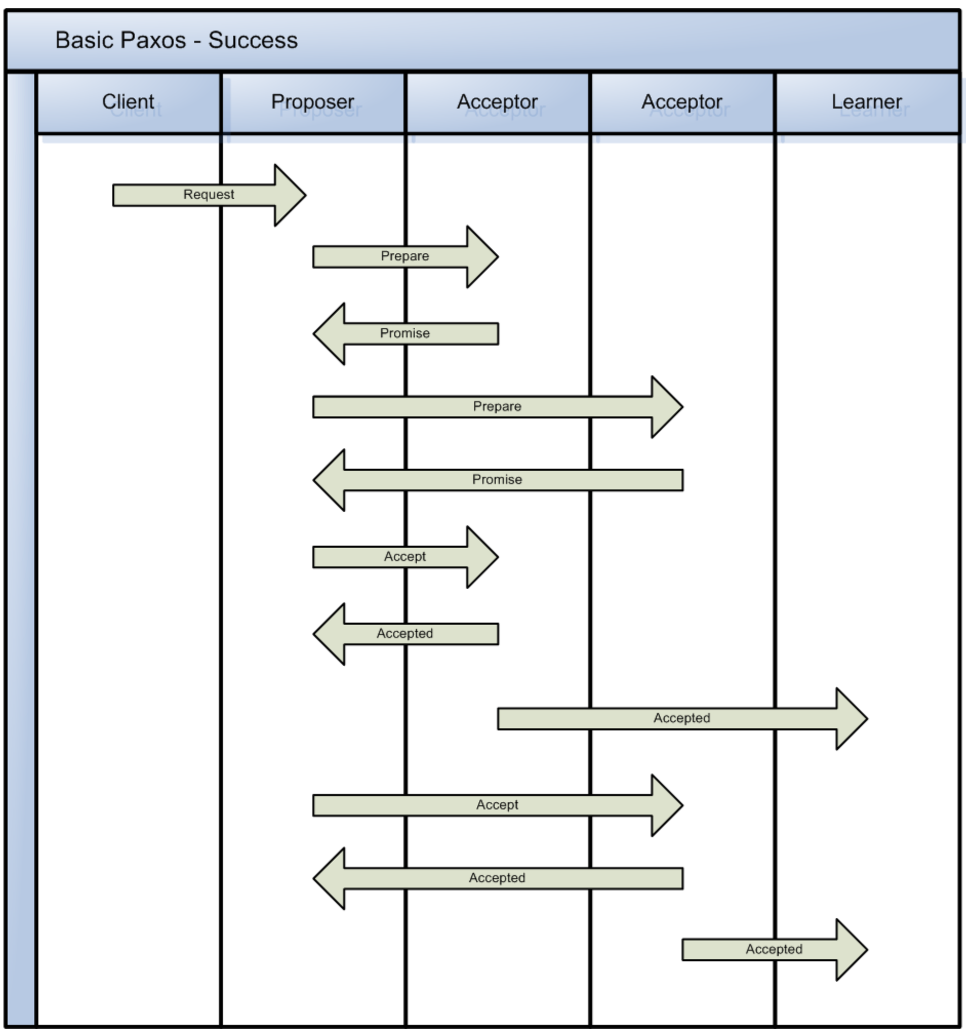
- **Client**—This is the process that issues a request to the Paxos system. Typically, this request is a system or business process message. For instance, a distributed file system notification (such as file write.)
- **Proposer**—This is the role that accepts the client request and publicizes and advocates the state change to the other processes. In some consensus systems, there is a single Proposer (who acts as a de facto leader); in other systems, any or all processes can be the Proposer.
- **Acceptor**—This role is aggregated into a collection called a Quorum. Essentially, all or most of the Paxos processes act as Acceptors, which essentially exist to record the state change once agreement is made.
- **Learner**—A Learner is a client of a Paxos process that learns of the state change and may (or may not) act on it. You may want to think of the Learner as the algorithm's output reception process.
- **Leader**—The leader is the Proposer for any particular round. As stated earlier, Leaders may rotate. Even within a single decision-making round, there may be multiple processes which believe they are the leader (for instance, due to fault tolerance protocols). The Paxos algorithm includes a method to distinguish which one's Proposals will win that round.

The simplest Paxos algorithm (though not the most popular or usable) is called Basic Paxos. It includes the following steps (and corresponding messages):

1. **Prepare**—The Leader and Proposer sends a Prepare message to the Quorum of Acceptors. This Prepare message includes a proposal number, n.
2. **Promise**—The Acceptors respond with a Promise to not accept Proposals with numbers less than the proposal number in the Prepare message. They must first verify that they have not already made a similar Promise with a proposal number greater than *n*.

   If they cannot Promise, they send a denial message. If they can Promise, they may optionally send a proposed value for the Proposal they believe is the correct one.
3. **Accept**—The Proposer sends a value for the Quorum to agree upon. If no values were included in the Promise messages, the Proposer is free to choose any value it believes is correct. If values were returned with the Promise messages, the Proposer must select one of those.
4. **Accepted**—This is the response message from the Acceptors within the Quorum, indicating they have accepted the value. This message is sent by the Acceptors to the Proposer and by each Acceptor to each Learner (think of the Leaner as a secondary storage mechanism in this case.)

Figure 1 illustrates a successful Basic Paxos round.



Basic Paxos fails when the Quorum does not successfully respond in either the Promise or Accepted stage.

PROBLEMS

Follow the instructions in the document "Getting Started with Amazon's Web Services for Lessons Plans" for getting an Amazon Web Services account and setting up your system.

1. Implement a Basic Paxos that uses RMI to send messages from processes on distinct machines. Use Amazon's EC2 (Elastic Compute Cloud) to create a set of machines.
2. Bonus: Basic Paxos is considered to use more network messages than needed. There are multiple and competing optimizations, such as Cheap Paxos or Fast Paxos. Research these alternatives and then describe their strengths and weaknesses. Which would be easier to modify your implementation toward?