Implementation of Classic Paxos in Orc

Project Report for CS 380D: Distributed Computing
May 9 2013

Hemanth Kumar Mantri Makarand Damle
The University of Texas at Austin
{mantri, mdamle}@cs.utexas.edu

Abstract

Paxos[1] is a widely used algorithm for implementing consensus in a fault-tolerant distributed system. It was introduced by Leslie Lamport in 1990 and published in 1998. In this project report, we present an implementation of classic Paxos in Orc[2]. We also present some optimizations to the classic algorithm included in our implementation, such as an efficient leader election and efficient handling of multi-proposer cases which are not required for correctness.

Keywords Distributed Systems, Consensus, Paxos, Leader Election, Orc

1. Introduction

Consensus is the problem of getting a set of nodes in a distributed system to agree on something such as a value or course of action or a decision. Consensus algorithms help a group of machines work as a coherent unit in presence of failures with every individual machine aware of and in agreement with the actions of the whole system. Consensus algorithms are needed to:

- decide on committing a transaction to the database
- synchronize clocks by agreeing on current time
- agree to move to the next stage of a distributed algorithm
- elect a leader among a set of nodes

As a result, consensus protocols have assumed a key role in building reliable large scale software systems and is important to understand how consensus is achieved in distributed systems across different network configurations. **Paxos** has dominated the discussion of consensus algorithms in the last two decades with many implementations either based on or influenced by it. It has also found acceptance in the industry with Google using it in its *Chubby*[7] service for *Big Table*[8] and Microsoft using it in *Autopilot cluster management service* from Bing.

Paxos is very notorious to be too complex to understand, in spite of numerous attempts to make it more approachable. Thus to understand it better, we have chosen to implement this in our course project. In this paper, we describe an implementation of classic Paxos algorithm and few of its optimizations in Orc.

This paper is organized as follows. Section 2 provides the background information necessary to help understand advent of Paxos. In section 3 we describe implementation details and optimizations done to classic Paxos in Orc. We conclude in section 4 and finally, identify some future work in section 5.

2. Background

In a distributed system with N nodes, any consensus protocol is correct if and only if it can satisfy the following rules.

- **Agreement:** all N nodes decide on the same value
- **Validity:** the value that is decided upon must have been proposed by some node in the system
- **Termination:** all nodes eventually decide

Two-phase commit (2PC)[3] and three-phase commit (3PC)[4] protocols are generally used to build consensus among nodes of a distributed system. However they have some deficiencies which arose the need for a Paxos like protocol. Paxos can be best appreciated only when we can understand what limitations of 2PC and 3PC are being solved by it. Hence, we first describe 2PC and 3PC protocols and their deficiencies and then describe the classic Paxos algorithm.

2.1 Two-Phase Commit

The 2PC protocol works in 2 phases as its name suggests.

1. **Propose:** A designated leader(proposer) proposes a value to every participant and gathers their responses.
2. **Commit-or-Abort:** If everyone agrees, the proposer instructs every participant to commit. Otherwise an abort message is sent to all the participants.

2PC satisfies all three properties of consensus protocol in absence of failures. However, when the proposer(leader) fails at the same time when a participant failed, all the remaining nodes *block* and this dramatically reduces its applicability.

2.2 Three-Phase Commit

3PC solves the issue of blocking in 2PC by adding an extra phase and hence the name, three-phase commit.

1. **Propose:** The leader (proposer) proposes a value to every participant and gathers their responses.
2. **Prepare to commit:** Proposer sends this message to all participants if it has received a positive acknowledgement from all the nodes in phase-1. On receiving this message, participants get into a state where they are able to commit the transaction but do not do any work that they cannot later undo. The purpose is to communicate the result of the vote to every participant so that the state of the protocol can be recovered no matter which participant dies.

3. **Commit-or-Abort:** If proposer receives confirmation of the delivery of the 'prepare to commit' message from all participants, it commits the transaction, else it aborts.

Unlike 2PC, 3PC does not block on proposer failure. However, it can have potential unsafe runs in case of *network partition*. When all participants that received *prepare to commit* are on one side of the partition, and those that did not are on the other, the system will have an inconsistent state when the network merges. Also 3PC works well in a fail-stop fault model but can have inconsistent behavior in a fail-recover fault model. Paxos, as we'll see below, overcomes all these problems.

2.3 Classic Paxos

The classic Paxos consensus algorithm is resilient to node failures and network partitions. However, it only guarantees correctness(Safety), but not progress (liveness).

The nodes in the distributed system are assigned the following *roles*

- ***Client:*** It issues a request for consensus and waits for response.
- ***Proposer:*** It is responsible for initiating the protocol. It advocates/proposes the client's value and convinces the *acceptors* to arrive at a consensus on this value.
- ***Acceptor:*** Each acceptor responds to proposals from the proposer either by rejecting them, or agreeing to them. The chosen value if any, may be disseminated to nodes which are interested in it (also called *learners or listeners*).
- ***Listener:*** It serves as a replication factor for the learned/chosen values.

Paxos is similar to 2PC with two important exceptions.

- **Ordering:** The proposals are ordered using mutually exclusive sets of proposal numbers to allow the acceptors determine which proposals should be accepted. Acceptors prefer higher proposal number (also called *ballot number or sequence number*) when they have to make a choice.
- **Quorum:** Paxos considers a proposal as accepted when a majority of acceptors indicate their acceptance. This is different from 2PC/3PC where proposals are agreed only if every acceptor sends an acceptance. Hence, Paxos can tolerate F failures in a distributed system of 2F+1 nodes.

The Paxos algorithm operates in two phases.

1. **Phase1:**
   (a) **Proposer (Prepare):** selects a unique proposal number N and sends a *prepare message* with number N to all acceptors.
   (b) **Acceptor (Promise):** If number N is greater than that of any prepare request it saw before, acceptor responds with a *promise message* not to accept any more proposals numbered less than N. Otherwise it rejects the proposal and also indicates the highest proposal number it has already accepted.
2. **Phase2:**
   (a) **Proposer (Accept):** If N was promised by a majority of acceptors, sends an accept message to all acceptors along with a value *v*.
   (b) **Acceptor (Accepted):** On receiving an *accept message (N, v)*, it accepts the value unless it has already responded to a *prepare message* with a sequence number greater than N. If accepted, this value is also sent to all listeners for replication.

Figure 1 shows a successful run of Paxos algorithm in absence of any failures in a distributed system consisting of 1 client, 1 proposer, 3 acceptors and 2 listeners. The client sends a consensus request to proposer and waits for the response. Proposer initiates phase-1 of the protocol by sending a `Prepare(N)` message to all acceptors. The 3 acceptors send a `Promise(N)` (since this is the only proposal they have seen and thus the proposal number is the highest that they have seen until now) to the proposer. When the quorum of promises is met, the proposer initiates phase-2 by sending `Accept(N,v)` message to all acceptors which in turn respond by sending `Accepted(N)` messages to proposer and the accepted value to listeners. Once a majority of acceptors reply, the proposer sends the response back to client and terminates the protocol.

3. Paxos implementation in Orc

As detailed in [1], our implementation of Paxos satisfies the following three key properties:

- **P1:** Every proposal number is unique. If there are T nodes in the system i^{th} node uses numbers in the sequence {i, i+T, i+2T, ...}.
- **P2:** Any 2 set of acceptors have at least 1 acceptor in common.
- **P3:** Value sent out in phase 2 is the value of the highest-numbered proposal of all the responses in phase 1.

3.1 Classes

Our Paxos implementation in Orc is structured into 4 major classes and can be seen in [9]:

1. **Process Class:** This class encapsulates all data and methods related to *proposer, acceptor or listener* and the current role of a process is determined by its `role` variable. It is built flexible enough to let a process take multiple roles at a time and to change roles dynamically in the middle of the protocol. For example, proposer behaves as both proposer(PROP) and acceptor(ACCP), in order to listen to messages from other potential proposers. Also, an acceptor can change its role to proposer when it detects a proposer failure. Similarly, a proposer can change itself to an acceptor when multiple processes start acting as proposers.
2. **Faulty Channel Class:** It simulates channel (and thereby, node) failures and message losses in the distributed system. The parameter *p* determines the probability with which any message from this channel is lost. A value of p=0 indicates a perfect fault free channel and p=100 for a node's out-going channels simulates the node failure.
3. **Message Class:** Encapsulates the data being sent in messages of the type { `type, sequence num, value` } where *type* indicates the type of the message which depends on the sender and its intended recipient.
4. **Distributed System Class:** It encapsulates the entire distributed system being simulated. It parametrizes the distributed system by accepting
   - `numNodes`: total number of nodes in the distributed system
   - `failProb`: the probability of failures which translates to the probability at which all channels in the system fail

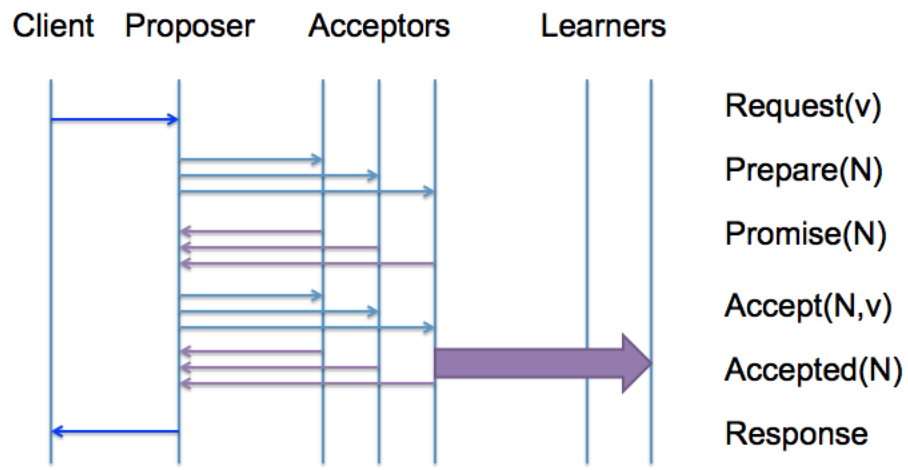

Figure 1. Successful Paxos Round

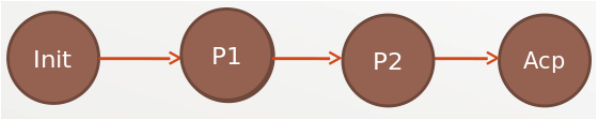

Figure 2. Proposer State machine: No failure

Figure 3. Proposer State machine: Phase 1 failure

- `timeout`: the time to wait before declaring failures
- `numListeners`: number of listeners/learners to tweak the desired replication factor

It also contains methods to initialize the system (create processes, channels, assign roles and channel mappings) where every node is connected to every other node via channels.

3.2 Proposer States

In our implementation of the Paxos algorithm, proposer goes through various states namely, `Init` (Initial), `P1` (Phase-1), `P2` (Phase-2), `Rjt` (Rejected) and `Acp` (Accepted). State transition diagrams for 3 important cases is shown below:

1. **Successful round completion:** Figure 2 depicts the scenario in the absence of any failure. Here the proposer transitions from `Init` to `Acp` through `P1` and `P2` that correspond to the two phases of Paxos algorithm.
2. **Phase 1 failure:** Figure 3 depicts the scenario when protocol fails in phase 1, because a quorum of acceptors fails to send promise message to the proposer. Here the proposer transitions from `Init` to `Rjt` through `P1`. The protocol does not enter phase-2 of the Paxos algorithm and hence proposer never enters state *P2*.
3. **Phase 2 failure:** Figure 4 depicts the scenario when protocol fails in phase 2, because a quorum of acceptors fails to send accepted message to the proposer. Here the proposer transitions from `Init` to `Rjt` through `P1` and `P2`.

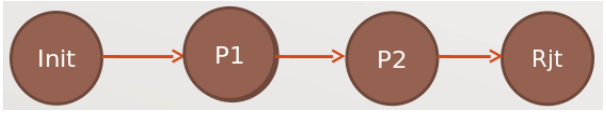

Figure 4. Proposer State machine: Phase 2 failure

3.3 Acceptor States

When an acceptor receives an *accept message* from the proposer, with a sequence number higher than what it has seen earlier, it transitions from `undecided` state to `accepted` state. Otherwise, it remains `undecided`.

3.4 Failure and Recovery handling

In the *default* run (provided in the source code), 1 node is selected as proposer and 1 node as listener and the rest as acceptors. If the failure probability is set to 0, a successful Paxos run in absence of failures occurs as described in Figure 1. However, our implementation is robust to handle node/channel failures that may occur in real world.

To detect node failures, every node sends a special *heartbeat message* (`HBT`) to every other node. Each node maintains its view of the state of all other nodes in the system using an array named `alive`. If no message is received from a node, it is marked not alive. This then helps it to detect if leader has failed.

- **Acceptor failure:** As discussed earlier, the algorithm can tolerate maximum F acceptor failures if total nodes in the system are 2F + 1 (of course, excluding the client and listeners/learners). This is because the remaining F+1 *alive* nodes can still form a quorum and drive the protocol to successful conclusion. This is shown in figure 5.
- **Proposer failure:** If a proposer fails, a new leader/proposer has to be elected by the remaining nodes. We decided to let

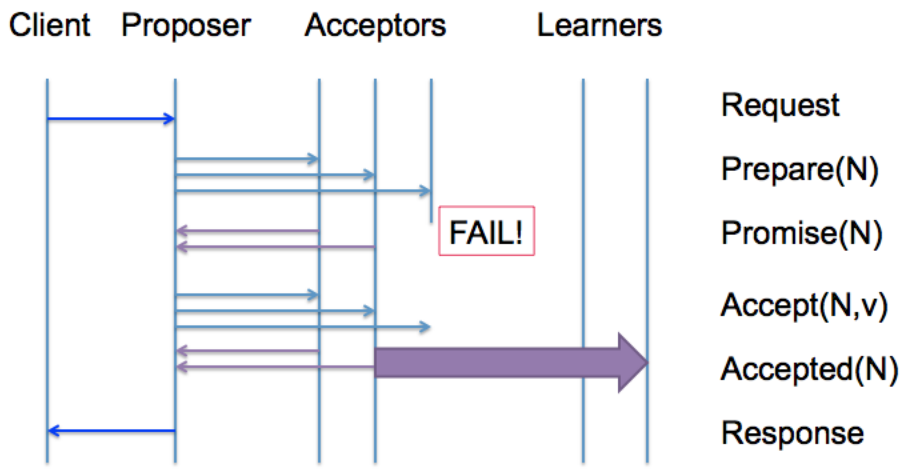

Figure 5. Acceptor failure

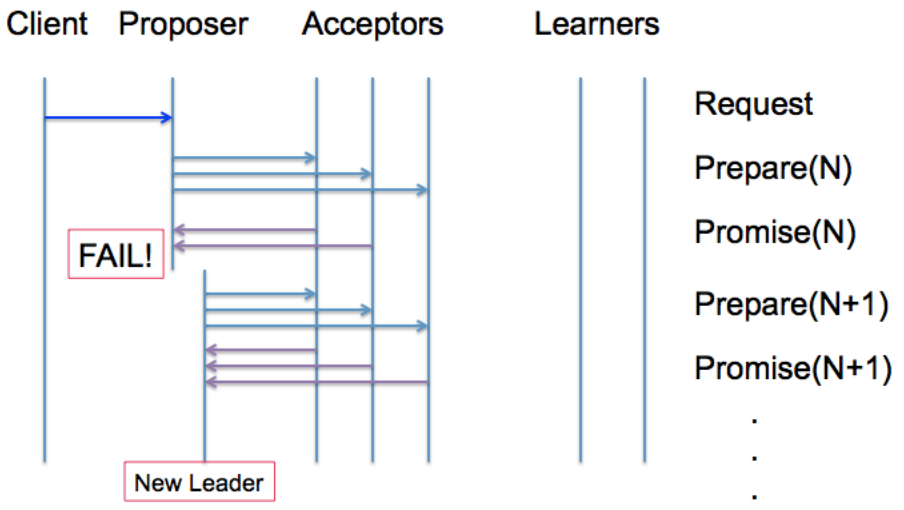

Figure 6. Proposer failure

the *alive node with the highest id* to assume itself as the next leader in the event of a proposer failure.

- The new leader, then changes its role to `PROP`, if not done already, and sends a *prepare message* with a higher sequence number than the one currently under consideration. It also update its `leader` variable to indicate itself as the new leader.
- All acceptors that respond to the above *prepare message* with a *promise message*, change their `leader` variable to indicate the new leader. If the earlier leader recovers and receives a *prepare message*, it checks to see if its proposal sequence number is less than that of the one it received and if so changes itself to an acceptor, without starting another round of leader election. This helps cut down the number of messages broadcast during failure-recovery and results in faster termination. The proposer failure scenario is shown in figure 6.

3.5 Optimizations

We implemented two optimizations on top of the classical Paxos algorithm to allow it to progress faster. It must be noted that neither of these are required for correctness.

1. **Multicast Vs Broadcast:** Instead of broadcasting the *prepare message* to all nodes, number of messages can be reduced by sending it to only alive nodes. Also, it is easy to extend it to include only a majority of acceptor nodes. This is key to the implementation of Cheap Paxos[10].
2. **Early Rejections:**

   Every acceptor can reject

   - `prepare(N)` if `prepare(M)` was answered and `M` $>$ `N`
   - `accept(N,v)` if `accept(M,u)` was answered and `M` $>$ `N`
   - `prepare(N)` if `accept(M,u)` was answered and `M` $>$ `N`

4. Conclusion

We successfully implemented Classical Paxos algorithm with some optimizations in Orc that:

- reaches consensus in absence of any failures.
- reaches consensus in case of acceptor failures. It is resilient to F acceptor failures when total nodes in the system are 2F + 1 and F+1 of then are `alive` nodes.
- performs on demand *leader election* in case of proposer failure. The alive node with maximum ID is chosen as the leader.
- handles failure recovery of proposer gracefully by having the proposer with lower proposal sequence number change itself to acceptor. This guarantees liveness.
- handles multi-proposer gracefully by having the proposer with lower proposal sequence number change itself to acceptor thereby guaranteeing liveness.
- includes two optimizations to the classical algorithm to allow it to make progress faster.

5. Future Work

The implementation provided in this project is a classic, non-byzantine version of Paxos with some added optimization flavors. In future, we think that it should be possible to extend this implementation to build other variants including but not limited to Cheap Paxos, Fast Paxos and Multi-Paxos.

Also, the paxos implementation can be easily ported to a true distributed channel class once that is available natively in Orc. Current implementation uses a Channel() site that simulates network channels between processes.

*A*cknowledgements:

We are greatly indebted to Prof.Seif Haridi for his Youtube video lectures on Distribtued Computing that included basics of Paxos. They helped us better understand the protocol. We also like to acknowledge the help provided by Henry Robinson's (researcher at Cloudera and Zookeeper committer) blog [6] in providing the background and motovation for Paxos as well as some implementation details.

References

[1] Lamport, Leslie (2001). Paxos Made Simple ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.

[2] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. 2009. The Orc Programming Language.

[3] http://research.microsoft.com/pubs/64636/tr-2003-96.pdf

[4] Skeen, Dale; Stonebraker, M. (May 1983). "A Formal Model of Crash Recovery in a Distributed System". IEEE Transactions on Software Engineering 9 (3): 219228

[5] http://en.wikipedia.org/wiki/Paxos_(computer_science)

[6] http://the-paper-trail.org/blog/consensus-protocols-paxos/

[7] http://research.google.com/archive/chubby.html

[8] Bigtable: A Distributed Storage System for Structured Data

[9] https://github.com/mantri/Paxos/blob/master/paxos.orc

[10] Cheap Paxos, Leslie Lamport and Mike Massa, Microsoft Corporation