Single-Decree Paxos

Miguel Alves, Nº 49828

1 Introduction

The **Single-Decree Paxos** protocol is used to achieve consensus on a single value. To handle log replication amongst peers it is more appropriate to use the **Multi-Decree Paxos** protocol.

Each entity involved in the protocol is called a **peer**, and each peer can assume the role of **Proposer**, **Acceptor** and **Learner** - each peer can assume multiple roles simultaneously, or assume a single role depending on the implementation.

This document considers an implementation where each peer assumes all roles simultaneously.

2 Message Structure

The following is the base structure of the messages sent between peers in the **Single-Decree Paxos** protocol. Note that implementing the protocol for different programming languages and use cases may require adding more information to these messages.

- `PREPARE`
  1. `id` → the proposal number that the **Proposer** intends to use in a future proposal
- `PROMISE`
  1. `id` → the smallest proposal number that a proposal needs to have for the **Acceptor** to accept that said proposal
  2. `accepted_id` (optional) → if the **Acceptor** has already accepted a proposal, `accepted_id` is the proposal number of the accepted proposal; otherwise, this field is `null`
  3. `accepted_value` (optional) → if the **Acceptor** has already accepted a proposal, `accepted_value` is the value that has been accepted; otherwise, this field is `null`
- `PROPOSE`
  1. `id` → proposal number
  2. `value` → value that is being proposed
- `ACCEPT`
  1. `id` → proposal number of the accepted proposal
  2. `value` → value associated with the accepted proposal
- `REJECT`
  1. `id` → proposal number that has been rejected

3 High Level Description

3.1 Proposer

When a peer receives a value from a client, it begins behaving like a **Proposer**.

A Proposer starts by sending a `PREPARE` message to all the other peers, which contains the proposal number that it intends to use in a future proposal.

The Proposer then waits for the responses to it's `PREPARE` message. Upon receiving a `PROMISE` response from a majority of peers, the Proposer proceeds to send a `PROPOSE` message to **every** peer (not just the ones from whom it received a `PROMISE`) - note that if any of the received `PROMISES` contained a non-null `accepted_value` field, the Proposer **must** propose that same value.

Finally, if the Proposer receives an `ACCEPT` response from a majority of peers, then it has achieved consensus on it's proposal and can inform the **Learners** - implementation dependant - and the client.

3.2 Acceptor

A peer begins behaving like an **Acceptor** upon receiving a `PREPARE` message from another peer. The Acceptor will then respond in one of three ways:

- If the proposal number in the `PREPARE` message is the largest the Acceptor has seen so far and it hasn't accepted any other proposals yet, the Acceptor will respond with a `PROMISE` message with the same proposal number, and with the `accepted_id, accepted_value = null`, meaning it will not accept any proposals containing a smaller proposal number
- If the proposal number is not the largest the Acceptor has seen so far and it hasn't accepted any other proposals yet, it will simply ignore the `PREPARE` message
- If the Acceptor has already accepted another proposal, it will respond with a `PROMISE` message containing the highest proposal number it has seen so far, and the pair `id,value` of the proposal it has accepted

Upon receiving a `PROPOSE` message from another peer, the Acceptor will respond in one of two ways:

1. If it hasn't accepted any proposals yet, and the proposal number is larger or equal to the largest proposal number it has seen so far, the Acceptor will accept the proposal and will respond with an `ACCEPT` message
2. If it has already accepted a proposal, or it has seen a larger proposal number, the Acceptor will respond with a `REJECT` message or simply ignore the proposal (implementation dependant)

3.3 Learner

The **Learner** role represents the entity that will "learn" the value for which consensus was reached. In real systems the Learners are, for example, data bases.

Therefore, in the articles that describe the protocol, there is very little information about the Learner's behaviour.

4 Internal State

The following is the state stored by each peer. To improve readability, the state is split amongst the roles.

4.1 Proposer State

1. `pid` → the peer's identifier
2. `state` → the proposer's current state
   (a) `IDLE` → waiting for a client request
   (b) `PREPARED` → waiting for responses to a `PREPARE` message
   (c) `PROPOSED` → waiting for responses to a `PROPOSE` message
   (d) `ACCEPTED` → has achieved consensus on it's proposal
3. `id` → the proposal number that the proposer will use
4. `propose_val` → value that will be proposed
5. `membership` → the list of known correct peers
6. `rcvd_promises` → a list containing the received `PROMISE` messages
7. `rcvd_accepts` → a list containing the received `ACCEPT` messages

4.2 Acceptor State

1. `pid` → the peer's identifier
2. `state` → the acceptor's current state
   (a) `IDLE` → waiting for `PREPARE` messages
   (b) `PROMISED` → has sent `PROMISES`, waiting for proposals
   (c) `ACCEPTED` → has accepted a proposal
3. `max_id` → the highest proposal number the acceptor has seen so far
4. `accepted_val` → the value associated with the accepted proposal, or `null` if it hasn't accepted a proposal
5. `accepted_id` → the proposal number associated with the accepted proposal, or `null` if it hasn't accepted a proposal

5 Internal State Changes

The following are the steps where there are internal state changes. To improve readability, these are split amongst the roles.

5.1 Proposer State Changes

1. Sending a prepare message (`snd_prepare`)

   (a) `pre_prepare`
   ```
   assert(state == IDLE)
   id = id + 1.0
   ```
   (b) `post_prepare`
   ```
   state = PREPARED
   ```

2. Sending a propose message (`snd_propose`)

   (a) `pre_propose`
   ```
   assert(state == PREPARED)
   assert(length(rcvd_promises) ≥ (length(membership) / 2))
   ```
   (b) `check_promises_contain_value`
   ```
   if ∃ promise in rcvd_promises : promise.accepted_val ≠ null
   then propose_val = promise.accepted_val
   ```
   (c) `post_propose`
   ```
   state = PROPOSED
   ```

3. Receiving a promise message (`rcv_promise`)

   (a) `pre_rcv_promise`
   ```
   assert(state == PREPARED)
   assert(promise not in rcvd_promises)
   ```
   (b) `post_rcv_promise`
   ```
   rcvd_promises.add(promise)
   ```

4. Receiving an accept message (`rcv_accept`)

   (a) `pre_rcv_accept`
   ```
   assert(state == PROPOSED)
   assert(accept not in rcvd_accepts)
   ```
   (b) `post_rcv_accept`
   ```
   rcvd_accepts.add(accept)
   ```

5. Check if we achieved consensus (`check_consensus`)

   (a) `pre_check_consensus`
   ```
   assert(length(rcvd_accepts) ≥ (length(membership) / 2))
   ```
   (b) `post_check_consensus`
   ```
   state = ACCEPTED
   ```

5.2 Acceptor State Changes

1. Sending a promise message (`snd_promise`)
   (a) `post_snd_promise`
   state = PROMISED
2. Sending an accept message (`snd_accept`)
   (a) `pre_snd_accept`
   assert(state = PROMISED)
   (b) `post_snd_accept`
   state = ACCEPTED
3. Receiving a prepare message (`rcv_prepare`)
   (a) `case_idle`
   max_id = prepare.id
   (b) `case_promised` or `case_accepted`
   max_id = max(max_id, prepare.id)
4. Receiving a propose message (`rcv_propose`)
   (a) `pre_rcv_propose`
   assert(state == PROMISED)
5. Accept a proposal (`accept_proposal`)
   (a) `pre_accept_proposal`
   assert(proposal.id $\geq$ max_id)
   (b) `post_accept_proposal`
   accepted_id = proposal.id
   accepted_val = proposal.value
   max_id = proposal.id

6 Implementation in Rust

For the **Rust** implementation, the following implementation choices were made:

- Each peer is simulated by a single thread - therefore it processes messages received in a sequential order (does not process messages concurrently)
- The message exchanges are made usign the std::sync::mpsc module
- Includes an adjustable probability for a message to be "lost" - for the tests below, that probability was set to 10%
- To better test concurrent scenarios, a random delay of 1 to 100 milliseconds is included before sending any message
- Since the proposal numbers must always be different between peers, each peer has it's own id and it's proposal number is initialized with id/1000 - note that this value should be adjusted if we increase the number of peers

The Rust implementation uses the same state variable names as referred in **.4** and the function names for each step are identical to the ones presented in **.5**.

7 Tests and Results

Running the Rust implementation results in a very verbose output that aids in understanding whether or not the protocol is executing as expected.

To test the performance of the implementation, measurements of the time taken to reach consensus were registered, obtaining the following table:

	1 concurrent proposal	10 concurrent proposals	100 concurrent proposals	1000 concurrent proposals
10 peers	601ms	678ms	-	-
100 peers	6479ms	6591ms	7089ms	-
1000 peers	65672ms	67271ms	69215ms	139325ms

Measurements of the time taken to obtain majority of promises were also taken, allowing us to see how much time is spent is each phase of the protocol - prepare phase and propose phase - obtaining the following table:

	Prepare Phase	Propose Phase
10 peers	305ms	373ms
100 peers	3454ms	3137ms
1000 peers	34358ms	32913ms

From these results, we can see that the time taken to achieve consensus increases linearly as we increase the number of peers used in the protocol.

We can also see that the time taken to achieve consensus barely increases when we have between 1 and 100 concurrent proposals, but it does increase significantly from 100 to 1000 concurrent proposals - this might be due to having a lot more software threads than hardware threads, and better performance might be attainable using a language like Go where we can maintain good performance when creating a large number of goroutines.

Finally, the time spent in each phase of the protocol is close to a 50% split.

8 Sources

Paxos Made Simple, Leslie Lamport, https://lamport.azurewebsites.net/pubs/paxos-simple.pdf