**LowPaxos Design Document**

**Preface**

* **Goals:** To design and implement LowPaxos – a distributed consensus algorithm for low resource settings. A low resource setting is faced with erratic network performance characterized by limited bandwidth (due to high costs), high and inconsistent latencies between communication endpoints, packet losses and limited or heterogenous resources to perform consensual actions in a consistent and uniform way most times. LowPaxos will use these properties to determine the most suitable long-term leader but also allow for leadership adaption to changes in the properties of the environment.
* **Desired fault model:** The system should be able to automatically adapt to environmental changes within the (2f + 1) quorum size to make consistent progress. The participant roles dictate the possible actions that can be performed, depending on the capacities and capabilities. Outside the quorum size, client reads may be allowed at the compromise of consistency. The writes can only be allowed if there is quorum for the operation. All nodes participate in the quorum determination (consensus), but the execute actions are performed on a subset of ‘able’ nodes. At the lowest performance mode, the system operates in a degraded state where the ‘slow’ nodes will execute the commit log actions. This state is reached when the ‘fast’ nodes are consensually unreachable.
* **Challenges:** The protocol elements operate in different geographical locations, and provide different guarantees of operation execution. As earlier described, the performance of the participants is non-uniform and varies over time. Messages can easily be dropped, reordered and retransmitted over a mostly unreliable network. The nodes may also be just slow to process and respond to requests. There are operations that may be resource-intensive and can take significant execution latencies to complete. The categorization of operations in such setups takes precedence, at the huge costs of operation classification. In addition, failures are more imminent and common. Designing a protocol for such a setup requires a clear balance between linearizability, consistency, availability and performance.
* **Assumptions**
  + The clients can make multiple read/write requests to any node in the distributed system
  + Consensus is reached when more than half of the nodes (quorum) agree on an operation and its subsequent commit/execute actions in the log and the SMR app respectively.
  + The roles of the participants are determined based on a weighted criterion of the low-resource properties. This creates an ordering of ‘fast’ 🡪 ‘mid-fast-slow’ 🡪 ‘slow’:
    - Fast (Leader) – Orders the operations by contacting and receiving acknowledgments from a quorum of nodes in the system. It is the in-charge of the distributed system over a given propose\_term and will commit/execute operations and communicate the decision to other nodes.
    - Mid-fast-slow (Candidate) – These commit and execute operations as instructed by the leader, and are potential candidates for the leader role.
    - Slow (Witness) – As instructed by the leader, the witnesses can commit but not execute operations. It is only after promotion to the candidate role that previous commits can be executed. The witnesses can participate in the rest of other operations of the distributed system.
  + At all points, the participating nodes need to have updated leader information
    - Communication from the leader to the rest of the replica nodes indicates its liveness. In addition, the candidates have to probe (or poll) the leader for explicit liveness checks as the system may not be busy at some points. At set times (not as fast as the candidate probes), the leader broadcasts its status to all the replica nodes (candidates and witnesses).
    - The leader is in charge of the system and is expected to have all the log entries committed and executed, and can also provide responses to the client.
      * Can the leader keep information of the nodes that have committed certain operations, so that client read requests can be load-balanced to improve on performance?
  + The candidates (or even witnesses in degraded mode) can allow for client read operations as long as its execution index with the leader is similar.
    - Can the candidate contact a quorum of other candidates or witnesses and allow the read operation?
  + ~~An operation may be classified based on its resource requirements – only the nodes with ‘capability’ shall execute this operation. The rest of the nodes shall commit it to the log, and optionally execute it (or them) in the degraded mode. This may require sharding which is outside the scope of the current work!~~

**Protocol**

* Kinds of node:
  + **Client:** A client **(C)** makes an initial request to perform an operation as part of the kvstore functionality. This request will be to any replica in the system.
  + **Replica:** A replica **(R)** is an entity that will provide a response to requests in the distributed system. The replicas participate in consensus on slot contention for commit operations of a distributed log. Consequently, these operations are executed on the SMR. The replica can take a role of a leader, candidate or witness during its lifetime.
  + **Monitor:** The monitor **(M)** in the testing setup provides metrical information used to determine leadership. It is a view on the current state of the network and resource performance on the replicas. The replicas use this information to determine the role configuration of the setup.
* **State at each kind of node:**
  + **Client (C)**
    - Request
      * ClientId (Integer) – A unique identifier for a client that is added to each request. A static integer value that uniquely identifies the client in the distributed system.
      * RequestId (Integer) – A unique identifier for each request and monotonically increases with time.
      * Operation (Bytes) – The requested operation and optional arguments that are part of its definition in the kvstore system functionality.
    - ~~Configuration~~
      * ~~ConfigId (a tuple of propose\_term and leader address) – A unique identifier of the current term of the system (propose\_term, leader). It includes the current term of the configuration and the leader address. Initially, the propose\_term is set to 0 and the leader address is set to None.~~
      * ~~Replicas (ReplicaId, role) – For a given configuration, the replicas are tagged with their current roles (leader, candidate, witness). A replica in the configuration is a set of ReplicaId, and its role.~~
    - Response
      * ResponseId(Integer) – A unique identifier for each response from the server and is monotonically increasing with time. This should match one of the request identifiers that has ever been issued by this client.
      * ResponseMessage(Bytes) – The results of the operation as executed by the replicas in response to the client operation request.
      * ~~Configuration (as before) – A set of information of the current configuration that serviced the request. The client will use this information for subsequent requests to be sent to the distributed system.~~
  + **Replica (R)**

A replica participates in the ordering of operations, commits, executions, leader election, receives client requests and provides responses. At the start of the protocol, the role of a replica is a member and is at the ***initialization*** state. The replicas elect a leader at the initialization state and create a configuration that will be used to service the client requests and operations on the kvstore application.

* + - **Member**
      * **Initialization:** This is the initial state of a distributed replica system participant and must attain one of the subsequent states (leader, candidate or witness).
        + **ReplicaId** (Socket address (IP address + Port)) – A unique identifier of a replica composed of its IP address and a port number.
        + **Role (String, Default = ‘member’)** – This is current role of a replica and it defaults to member at the initialization state.
        + **State (const integer, Default = 0)** – This is the current state of the replica and defaults to 0. The following are additional states that a replica can take on regardless of its role:

0 – Initialization (A replica is starting up and able to participate in election of the leader.

1 – Normal (A replica in normal state can receive and respond to client requests. It can also participate in configuration change processing)

2 – Configuration (In a configuration change state, the replicas are participating in a new election due to the unavailability of the current leader. This is initiated by one of the candidate replicas when it is unable to poll the leader (shorter timer) or when it has not received heartbeat messages from the leader (longer timer). The leader propose and commit messages may be used for the heartbeats. Where the system is not busy, normal heartbeat messages should be sent by the leader. The heartbeat timer on the leader is reset at each message send to the replicas. The replicas also reset their timers (heartbeat for the candidate/witness and pollleader for the candidate)).

3 – Recovery (A replica enters the recovery mode when it rejoins the network and it needs to be updated with the latest log information from the current leader or if the replica has stale information about the configuration. It is possible that the replica was a leader before and has now rejoined the network and needs to catch up with the rest of the replicas.)

4 – Amend (A witness replica is being promoted to a new role as candidate and will be expected to **execute** all the operations in the log. In this state, one of the candidates has been marked offline by the leader and requires ***f + 1***)

5 – Offline (A replica is marked as offline if it is unable to reach a leader of the current active configuration or *f + 1* replicas. It is also possible that replica is just unreachable from the rest of the replicas. This will be helpful to the clients when they make future requests).

* + - * + Configuration (A map with the propose\_term as the key and the replicas): The configuration is initially set with the propose\_term 0 and the default replica information. A replica in the configuration is a set of ReplicaId, role and profile with their default values.
        + Quorums

Leadership – A set of ResponseVoteOk messages that a member replica will use for computation and possible ascension to leadership.

Propose – A set of ProposeOk messages used to confirm that the replicas have agreed to log the operation for the current term and proposal number.

Read – A set of ProposeReadOk messages from the candidates to allow fast reads of the kvstore application.

* + - * + Voted (A tuple with the propose\_term and replica\_id) – This is the vote information at the replica that contains the term in which the replica voted in and the replica voted to be the leader.
        + Log (A replicated log struct): The Log contains a vector of log entries. A log entry contains the following information:

Propose\_term (An integer with the leader term at the time of ProposeAccept). This is initially set to 0.

Propose\_number (A unique identifier attached to the request as proposed by the leader)

State (One of Propose, Accepted, Committed and Executed) as a request traverses the system towards execution.

Request (The client request)

* + - * + Transport (The transport mechanism that will be used by the replica for communication with the clients and other replicas in the system.): It will be set to send 1-1 messages but also broadcasts for mostly leader-initiated messages. For each transport call, a timeout is added and indicates what callback operation should be executed if no response is received within a set time.
    - **Leader** (The long-term leader that has received majority of the votes and all mutable operations have to be routed through it)
      * **Member (Role = ‘leader’)**
      * data {k, v} // all operations that have been successfully executed on the kvstore
      * log [] // the commit log on the leader. In normal operation mode, this is the most

UpToDate log of the distributed system.

* + - * commit\_index (integer) // the latest commit index
      * execution\_index (integer) // the latest execution index
      * configuration // the latest replica configuration information (includes the latest profiles)

updated as requests are processed or when there is a change in the environment

* + - **Candidate** (A potential leader of the distributed system as a result of a ***profile*** computation based on the properties of the operational environment)
      * **Member (Role = ‘candidate’)**
      * data {k, v}
      * log []
      * commit\_index (integer)
      * execution\_index (integer)
      * configuration
    - **Witness**
      * **Member (Role = ‘witness’)**
      * data {k, v}
      * log [] // A log of operations that have been committed
      * commit\_index (integer)
      * execution\_index (integer) // For the first time, the execution index will be 0. After promotion at some instance, this will contain the id of the last execution performed at the witness.
  + **Monitor (M)**

The monitor provides metrical performance information that will be used to compute the weighted profiles. A weighted profile represents the strength of a replica node against another, and can be used to authoritatively request for votes. The assumption is that the nodes shall not behave in a byzantine mode and will always provide the accurate profile value. In a low resource setting with heterogenous elements, there is a very high degree of obtaining different profiles between a pair of replicas. In a situation where the replicas have similar profiles, the voter should always respond positively to a vote request.

In a controlled environment, the node will be injected with simulated performance information to test the different scenarios in which the metrics can change and necessitates election of a new leader. For our deployment environment, monitoring information is managed by an external module resident on the same node as the replica and can be contacted for metrical information. In order to expedite the profile computations, a cache will be implemented and runs in the background.

* + - MonitorMatrix (An n-dimensional array that stores the metrical information on the compute, network, availability and overall profile between pairs of clusters). It also defines methods for creation, update and retrieval of the results.
      * ReplicaId (IP address + Port) – A unique identifier of a replica composes of its IP address and a port number.
        + Compute (float)
        + Network (float)
        + Availability (float)
        + Profile (float) // A sum of the compute, network and availability computations for a pair of replicas.

The deployment environment (setup) will define a 1-dimensional monitor matrix with the above results for each of the replicas.

* **Messages:**
* **Leader Election**
  + **RequestVoteMessage (replica\_id, role, propose\_term, propose\_number, replica\_profile, election\_type)**

The replicas participate in an election to determine the long-term leader of a given propose\_term. The election is also used to determine the configuration of the distributed system. The profile in the message is of the destination replica with respect to the source replica. Any replica at the start of the system can request for votes to attain leadership. A replica increments the propose\_term and sends a broadcast message to the replicas seeking for votes to become a leader.

A replica that desires to be the leader changes its state to configuration and sends the RequestVoteMessage broadcast to the rest of the replicas.

* + - Source: Replica
    - Destination: Replica
    - Contents: replica\_id, replica\_role, propose\_term, replica\_profile, election\_type

The election\_type takes three variants:

* normal – This is the first election round that any replica can start.
* offline – This is an election that a candidate starts when it is unable to reach the leader, and the PollLeaderTimer or HeartBeatTimer have expired. A witness may also start this election type when its HeartBeatTimer has expired.
* profile – This is an election that a candidate starts when it notes a significant change in the profile of the leader.
* timeout – This is at the first election; a candidate may initiate this if it notices that a leader has not been elected within a specific period and the LeaderVoteTimer has expired.
* degraded – A replica member may start this election type if LeadershipVoteTimer has expired. This is a non-profile-based type of leader election. The eventual leader will still classify the rest of the replicas as per the profiles it has.

At the receipt of the RequestVoteMessage, a replica confirms that it is a valid recipient (different replica\_ids) and checks the vote status:

* (1) If the replica has never voted before (voted is none) or in this term (its propose\_term is < the propose\_term in the message), it will compare two profile values: its profile in the message, and the profile of the replica seeking leadership:
  + (a) If the profile in the message is less or equal to the profile information it knows about the source replica, it will vote for the replica with a ResponseVoteMessage as confirmation and change its status to configuration. It then updates its voted property with this information (replica\_id voted, propose\_term). The replica then starts the LeadershipVoteTimer within which it can start a degraded election type if not informed of a leader.
  + (b) Else: The RequestVoteMessage will be ignored by the replica, the replica then changes its role to candidate and starts the LeaderVoteTimer
* (2) If the replica has voted before (voted is not none):
  + (a) The replica checks that this is not a duplicate RequestVoteMessage by comparing the voted and propose\_term. If it is a duplicate, it will resend the ResponseVoteMessage to the replica earlier voted. If this is a new RequestVoteMessage, the replica checks on the propose\_term, the message replica\_id, the message role and the message election type:
    - (i) If the propose\_term is less than earlier seen (as part of the voted) or its own propose\_term (updated from a leader configuration message), the message will be ignored. The source is informed that the replica configuration has changed.
    - (ii) If the propose\_term is the same as earlier seen (as part of the voted), or its own propose\_term:
      * Confirm the role of the replica that sent the message (in the message):
        + *Candidate:* This means one of the candidates has had the LeaderVoteTimer expire, and no leader has been elected.

Confirm that the election\_type is a timeout and proceed with (1) (a/b), else ignore the request.

* + - * + *Witness:* Ignore as the witness role is only assigned after a leader has been elected for this propose\_term.
        + *Leader:* Ignore as a leader should be long-term for a given propose\_term and should never restart an election cycle.
    - (iii) If the propose\_term is greater than earlier seen (as part of the voted) or its own propose\_term – this represents a new election cycle which is a result of the leader not available or having a a worse profile:
      * Determine the election type and should be one of the following (match):
      * profile
        + If the role of the replica is a leader, proceed with with (1) (a/b). If the result of (1) (a/b) is No, the leader won’t change its role to candidate but will send HeartBeatMessage messages to other replicas.
        + [Else] If the role of the replica is a candidate/witness:

Confirm the role of the replica that sent the message.

*Leader:* Ignore as a leader should be long-term and should never restart an election cycle for any less, similar, or even higher propose\_term.

*Candidate/Witness:* Proceed with (1) (a/b) but no need to change the replica roles. There is no need as this will be communicated by the new replica leader if there is a successful election round.

*Member:* Ignore as this election type can only be started by a candidate or witness.

* + - * + [Else] If the role of the replica is a member:

Ignore as the member may attain a candidate role but with will need quite a larger state to be transferred.

* + - * offline
        + If the role of the replica is a leader, ignore and send HeartBeatMessage messages to other replicas.
        + [Else] if the role of the replica is a candidate/witness:

Confirm the role of the replica that sent the message.

*Leader:* Ignore as a leader should be long-term and should never restart an election cycle for any less, similar, or even higher propose\_term.

*Candidate:* This means one of the candidates has had the PollLeaderTimer or HeartBeatTimer expire:

Proceed with (1) (a/b)

*Witness:* This means that one of the witnesses has had the HeartBeatTimer expire and none of the candidates has attained leadership:

Proceed with (1) (a/b)

*Member:* Ignore as this state should not be reached – A replica with this role cannot send a RequestVoteMessage with a higher propose\_term or even start this election type.

* + - * + [Else] If the role of the replica is a member:

Ignore as the member may attain a candidate/leader role but with will need quite a larger state to be transferred.

* + - * other (normal, timeout, degraded)
        + Ignore, only accept new elections cycles of a higher propose\_term with **only** profile and offline types)

**Timers**

* + - RequestVoteTimer: At the start of each RequestVoteMessage send, the replica initiates this timer. It expects a response within this time, else the message or the response could have been dropped. The replica can resend the message.
    - LeaderVoteTimer: The timer is initiated by a replica that has changed its status to candidate and expects a leader to be voted within this period. At the end of an election and there is a leader, configuration information is distributed to all the replicas. If a candidate receives this configuration, the timer will be stopped otherwise, the candidate will start another election round with the same term.
    - LeadershipVoteTimer: This timer is started by a member replica after it has voted for another replica and expects that a leader decision will be reached at soon.
    - HeartBeatTimer: The leader sends heartbeat messages to the rest of the replicas explicitly or as part of any broadcast message. At each message broadcast (Propose or Commit), the leader resets this timer after which it will explicitly send the HeartBeatMessage as a broadcast. As a candidate or witness, this timer represents a period within which a heartbeat is expected from the leader.
    - PollLeaderTimer (< HeartBeatTimer): The timer is initiated by the candidate at the start of the PollLeaderMessage. It is reset by a leader response (PollLeaderOkMessage) to this message and the HeartBeatMessage broadcast.
  + **ResponseVoteMessage (replica\_id, propose\_term, propose\_number)**

The ResponseVoteMessage will be sent by a replica in response to a RequestVoteMessage to confirm it has provided a vote to the replica that had requested for it. A replica will change its state to configuration and send out this message.

* + - Source: Replica
    - Destination: Replica
    - Contents: replica\_id, propose\_term, propose\_number

When a replica receives the ResponseVoteMessage:

* It checks that propose\_term matches own propose\_term to confirm the correct election term
* It actively checks for quorum (the ResponseVoteMessages are uniquely identified by a pair of (propose\_term, replica\_id) to ensure it has received enough votes for the current propose\_term that guarantees leadership.
* If it receives the required votes, the replica will change its role to leader and inform the rest of the replicas by sharing its configuration information in a ConfigureReplicaMessage. The Leader also starts the LeaderLeaseTimer as the lease period of the election term. This configuration information provides an ordering of the replicas using the leader profile information:
  + - * Leader – current replica (1)
      * Candidate – f replicas
      * Witness – f replicas

If a replica doesn’t receive any ResponseVoteMessage after the LeaderElectionTimer (high probability it has a worse profile compared to the rest of the replicas), it will automatically change its role to a witness. If it receives at least a vote, it changes its role to a candidate. As a candidate, it will start the LeaderVoteTimer and perform other entry actions in this role.

**Timers**

* + - ResponseVoteTimer: For each vote request that a member issues, this timer is started within which a response from a replica is expected. If this timer expires, the request is resent as the request or response may have been dropped.
    - LeaderElectionTimer: The timer is started at each election round as the period within which an election should be completed. At the end of this timer, a member will have counted the available votes to determine its new role as either leader, candidate or witness.
    - LeaderLeaseTimer: The timer is started at the point that a replica attains the leader role. The timer is used to maintain the leadership status of a leader replica and will be reset at the dispatch of the PollLeaderOkMessage to the candidates. If the timer expires, the ‘current’ leader will change its role to a candidate to allow for a new election.
  + **RequestReplicaConfigMessage (replica\_id)**

This message is sent by a replica when there is a leadership conflict – A stale leader as per the propose\_term and/or propose\_number will send a request to the latest known leader. A replica changes its state to configuration and sends the message to the leader.

* + - Source: Replica
    - Destination: Leader
    - Contents: replica\_id
  + This request can only be processed by the leader, other roles are ignored.The leader sends the ConfigureReplicaMessage to the requesting replica.
  + **ConfigureReplicaMessage (replica\_id, propose\_term, role)**

The leader sends the replicas the propose\_term and their roles for the duration of the propose\_term after the outcome of the election results

* + - Source: Replica
    - Destination: Replica
    - Contents: replica\_id, propose\_term, role

When a replica receives the ConfigureReplicaMessage, it confirms that the propose\_term is new otherwise it will ignore it. If it is fresh (new compared to own propose\_term), the replica will now update its role and configuration information. It also updates its propose\_term as per the ConfigureReplicaMessage.

* + **HeartBeatMessage (replica\_id, propose\_term, profile)**

The leader sends the HeartBeatMessage broadcast to all the replicas to confirm its liveness. This is at the expiry of its HeartBeatTimer – this will be reset on each broadcast at the leader.

* + - Source: Leader
    - Destination: Replica
    - Contents: replica\_id, propose\_term, profile

On receipt of the HeartBeatMessage:

* *Leader:* Ignore the message
* *Candidate:* 
  + Checks its profile and compares with the message profile for any significant differences (better by > **∂**) and if so, starts a new election cycle with a new term and type profile as the next possible leader.
  + If no significant differences, the candidate will reset the PollLeaderTimer and HeartBeatTimer
* *Witness:*
  + Checks its profile and compares with the message profile for any significant differences (better by > **σ**) and if so, change state to a candidate.
  + If the leader profile is still better than the witnesses’, the witness resets the HeartBeatTimer.
* *Member:*
  + Request for configuration information (RequestConfigMessage) from the leader and update accordingly.
  + **PollLeaderMessage (replica\_id, propose\_term)**

The candidate polls the leader for liveness within a short period compared to HeartBeatMessage intervals.

* + - Source: Replica (Candidate)
    - Destination: Leader
    - Contents: replica\_id

When the leader receives this message, it responds with a PollLeaderOkMessage to the candidate replica and resets the LeaderLeaseTimer. If the candidate does not receive a response within a PollLeaderTimeout (the candidate can retry at this point as the request/response may have been lost/dropped), it will start a new election cycle with type offline.

* + **PollLeaderOkMessage (replica\_id, propose\_term, profile)**

The leader sends the PollLeaderOkMessage to the candidate replica as a liveness check confirmation in response to a PollLeaderMessage. The leader checks if it is responsible for the propose\_term in the message, and will drop the request if otherwise.

* + - Source: Leader
    - Destination: Replica (Candidate)
    - Contents: replica\_id, propose\_term, profile

On receipt of the PollLeaderOkMessage:

* *Leader:* Ignore the message
* *Candidate:* 
  + Checks its profile and compares with the message profile for any significant differences (> **∂**) and if so, starts a new election cycle with type profile new term as the next possible leader.
  + If no significant differences, the candidate will reset the PollLeaderTimer and HeartBeatTimer
* *Witness:* Ignore the message
* **Request Processing**
  + **RequestMessage (client\_id, request\_id, operation [arg0, arg1, … argn])**

A client sends/issues a request that includes its identifier (socket (address + port)), a uniquely generated and random request identifier and the operation payload with optional arguments. The arguments depend on the kind of operation that the client is invoking on a replica.

* + - Source: Client
    - Destination: Replica
    - Contents: client\_id, request\_id, operation [arg0, arg1, … argn]

The message is sent at the onset of an operation request from a client to perform an operation on the kvstore application. The client sends this message to any of the replicas. When a replica receives this message:

1. It determines the type of the operation in the RequestMessage
   1. Mutable (set, append, delete) - The following operations are performed based on the role at the replica:
      1. Leader – Confirm it is not a duplicate request and broadcast the ProposeMessage to other replicas and creates a log entry in its Log with the status *Request*. If the request is a duplicate, then provide the response to the client as before.
      2. Candidate – Confirm it is not a duplicate and send to the leader. If the request is a duplicate, then provide the response.
      3. Witness – Forward the request to the leader.
      4. Member – Ignore as this replica should attain one of the three operation states.
   2. Non-mutable (get) – The following operations are performed based on the role at the replica:
      1. Leader – Provide the response directly.
      2. Candidate – A quorum read proposal (ProposeReadMessage) is initiated (The assumption is that at least a quorum of the candidates *(f + 1)/2 + 1* have committed and executed the operation). If there is no quorum within ProposeReadTimer, the request is forwarded to the leader. *This can introduce delays on request processing. Maybe the request should directly be sent to the leader.*
      3. Witness – Forward the request to the leader.
      4. Member – Ignore as this replica should attain one of the three operation states.

**Timers**

* RequestTimer: The timer is initiated at the onset of an operation call by the client within which it expects a response from the replica. If there is no response, the client will resend the operation.
  + **ProposeMessage (replica\_id, propose\_term, propose\_number, RequestMessage)**

A leader sends this proposal message as a broadcast with its unique identifier, the propose\_term, propose\_number and the RequestMessage to other replica members (or candidates for fast reads).

* + - Source: Leader Replica (or candidate)
    - Destination: Replicas
    - Contents: replica\_id, propose\_term, propose\_number, RequestMessage

The message is sent when a leader or candidate receives a client operation to change or retrieve the state of the kvstore application. It issues the ProposeMessage to other replicas to lead this operation call. When a replica receives the ProposeMessage, it checks its current role:

* 1. If the role is a leader, and the replica\_id is similar, the message will be ignored as the leader by default votes for itself in the operation.
  2. If the role is a leader, and the replica\_id is different, the propose\_term will be compared (this is possibly an abnormal state where there are at least two leaders in the system).
     1. If the propose\_term of the request is new, this means the replica is an old leader and should be updated with the latest configuration and information. The old leader will send a RequestReplicaConfigMessage.
     2. If the propose\_term of the request is old, the current leader will send a ConfigureReplicaMessage to the old leader.
  3. If the role is a candidate/witness:
     1. Replica R checks if the propose\_term is the current term to confirm the freshness of the election:
        1. If the propose\_term is the current term known (same as own propose\_term), it proceeds to (ii) else it will be ignored as an earlier operation.
     2. R checks if the propose\_number is new (compare with its own propose\_number)
        1. If the proposal number is the latest ever seen, it proceeds to (iii) else it will be ignored as an earlier seen operation (it could be a duplicate operation request).
     3. R sends a ProposeOkMessage to the leader as a vote confirmation. It also updates its log entry with this request and status *Propose.* R additionally updates its propose\_number as per the ProposeMessage.
  4. If the role is a member, the request is ignored as the replica needs to attain one of the operational roles.

**Timers**

* ProposeMessageTimer: The timer is initiated at the onset of a ProposeMessage broadcast sent to the replicas to vote on an operation. It is expected that all the replicas shall respond within this period for a successful election. If the timer expires, the following are the possible reasons:
  + The leader role should change as the propose\_term is old. It is possible that there is now a new leader in the setup that should be coordinating the operations and is at a newer proposal term. A replica will most likely have sent a ChangeConfigMessage with the latest information to the proposer or the new leader has contacted the proposer with a similar message.
  + There might be communication issues to reach the replicas or in their responses back. If there is no ChangeConfigMessage received (and the ProposeMessageTimer is still active), the leader will re-broadcast the message to the replicas.
  + **ProposeOkMessage (replica\_id, propose\_term, propose\_number, commit\_index)**

A replica (candidate or witness) sends this message to the leader as acceptance of a proposal for an operation to be performed.

* + - Source: Replica
    - Destination: Leader
    - Contents: replica\_id, propose\_term, propose\_number, commit\_index

On receipt of the ProposeOkMessage:

1. A replica confirms that it is a leader (Any other role candidate/witness/member cannot process a ProposeOkMessage), and for the propose\_term and propose\_number in the message received else it will ignore the message.
2. For each ProposeOkMessage:
   1. Add the ProposeOkMessage to the Quorum Set and actively check for quorum:
      1. If quorum is reached, the leader:
         1. Updates the entry in the log with status *Committed*
         2. Updates the commit\_index with the propose\_number
         3. Executes the operation on the kvstore, and updates the log entry to *Executed*
         4. Provides a response to the client in form of a ResponseMessage
   2. Check the commit\_index in the ProposeOkMessage:
      1. If the leader.commit\_index > replica.commit\_index by 1:
         1. The leader sends the CommitMessasge to the source replica
      2. Else if the leader.commit\_index > replica.commit\_index by > 1
         1. The leader transfers the log in form of UpdateStateMessage from the replica.commit\_index upto the leader.commit\_index

The commit\_index is piggybacked on the ProposeOkMessage to enable the leader to have the current state of other replicas. This eliminates the need for CommitOk messages at each round of the Commit phase.

* + **CommitMessage (propose\_term, propose\_number)**

The leader sends the CommitMessage to all the replicas to commit and execute the operation

* + - Source: Leader
    - Destination: Replica
    - Contents: replica\_id, propose\_term, propose\_number

On receipt of a CommitMessage, a replica performs the operations below based on its role:

1. Leader
   1. If the replica\_id is similar, the message will be ignored.
   2. If the replica\_id is different, the propose\_terms are compared:
      1. If the propose\_term in the CommitMessage is new, this means the replica is an old leader and should be updated with the latest configuration and information. The old leader will send a RequestReplicaConfigMessage from the current leader, that will respond with a ConfigureReplicaMessage.
      2. If the propose\_term of the request is old, the current leader will send a ConfigureReplicaMessage to the old leader.
2. Candidate/Witness
   1. If the candidate/witness propose\_term is greater than the message propose\_term:
      1. Ignore the message.
   2. If the candidate/witness propose\_term is less than the message propose\_term:
      1. Candidate/witness requests for new configuration from the leader (message replica\_id) by sending the RequestConfigMessage
   3. If the candidate/witness propose\_term is the same as the message propose\_term, the replica checks the propose\_number
      1. If the candidate/witness propose\_number is greater than the message propose\_number
         1. Ignore the message.
      2. If the candidate/witness propose\_number is less than the message propose\_number
         1. Candidate/Witness requests the new state with a RequestStateMessage sent to the leader (message replica\_id)
      3. If the candidate/witness propose\_number is the same as in the message:
         1. Candidate/witness checks for a log entry with this propose\_number and its propose\_term
            1. If there is an entry:

The candidate/witness updates the log entry with the status *Committed.*

The candidate/witness updates the commit\_index with the propose\_number

Only the candidate executes the operation on the kvstore.

Only the candidate updates the log entry with the status *Executed.*

Only the candidate updates the execution\_index with the propose\_number

* + - * 1. If there is no entry:
      1. Candidate/witness requests for new state with a RequestStateMessage sent to the leader (message replica\_id)

1. Member
   1. Ignore the message as the member must attain one of the three roles.
   * **RequestConfigMessage (replica\_id, propose\_term, propose\_number)**

The RequestConfigMessage is sent from any of the replicas that is behind in the propose\_term and/or propose\_number. These attributes are checked at the receipt of ProposeMessage and CommitMessage and in the election transactions:

* + - Source: Replica (Any role) – R1
    - Destination: Leader – R2
    - Contents: replica\_id, propose\_term, propose\_number

Upon receipt of this message, the replica:

1. Checks that its role is leader otherwise it will discard the message
2. Checks that at least one of the propose\_terms or propose\_number is behind the leader otherwise it will discard the message
3. The leader will then send the ChangeConfigMessage to the source replica.
   * **ChangeConfigMessage (leader\_id, propose\_term, propose\_number, Config)**

This message is sent from the current leader to a replica that has a stale propose\_term or propose\_number. It is a response to the RequestConfigMessage:

* + - Source: Replica (Any role) – R1
    - Destination: Replica (Any role) – R2
    - Contents: leader\_id, propose\_term, propose\_number, Config

When a replica receives the ChangeConfigMessage:

1. If the role is a leader (R2):
   * 1. If the replica\_id is own (R1 == R2) and the propose\_term (R1.propose\_term == R2.propose\_term) and propose\_number (R1.propose\_number == R2.propose\_number) values are the same, the message will be ignored
     2. If the replica\_id is different (R1 != R2), the propose\_term and propose\_number is checked:
        1. If the propose\_term is less than at the recipient (R1.propose\_term < R2.propose\_term), a ConfigureReplicaMessage response is sent back to the source with updated information (R1 as a leader is outdated) and further processing is ignored.
        2. If the propose\_term is greater than at the recipient (R1.propose\_term > R2.propose\_term), the role is changed to a member and a request for new configuration in a RequestReplicaConfigMessage is sent to the leader and the rest of the processing ignored
        3. If the propose\_term is similar to the recipient’s, then the propose\_number is checked (there is need to determine the leader based on the current progress in advancing the operations):
           1. If the propose\_number is less than at the recipient (R1.propose\_number < R2.propose\_number), then R2 sends a ConfigureReplicaMessage to R1
           2. If the propose\_number is greater than at the recipient (R1.propose\_number > R2.propose\_number), the role is changed to a member, and a request for new configuration in a RequestReplicaConfigMessage is sent to the leader and rest of the processing ignored
           3. If the propose\_number (R1) similar to the recipient (R2), this implies 2 leaders at the same state. This state should never be reached.
   1. If the role is a candidate/witness (R2):
      1. The replica checks the propose\_term and propose\_number to confirm if there is need to update the configuration
         1. If the propose\_term and the propose\_number in the message are similar or less than R2’s, the message is ignored as R2 is updated. It could also be a duplicate message.
         2. If the propose\_term in the message (R1) is greater than in R2, R2 updates its configuration information with the new leader and propose term. This also means the propose\_number at R2 will be likely lower than at R1 (checked), and there is need to request for the new state in form of a RequestStateMessage. If the propose\_numbers are similar (probably an idle system and hence unlikely), then there is a new leader but no progress on the log has been made and hence no need for state transfer.
   * **RequestStateMessage (replica\_id, propose\_term, propose\_number, commit\_index)**

This message is sent by any of the replicas when they realize they don’t have the latest state as per an earlier message received from the leader (ProposeMessage, CommitMessage, ChangeConfigMessage).

* + - Source: Replica (Any role)
    - Destination: Leader
    - Contents: replica\_id, propose\_term, propose\_number, commit\_index

When a leader of this propose\_term and propose\_number receives this message, it will be ignored as it should be originating from another replica (Possible for a witness to ask for state from a candidate?).

The leader:

* Checks that the propose\_term (==), propose\_number (<), commit\_index (<) attributes in the message are less than at the leader, else it will ignore
* Sends the UpdateStateMessage with log entries from the commit\_index (+1) of the message to the leader’s commit index.
  + **UpdateStateMessage (propose\_term, propose\_number, commit\_index, log)**

The UpdateStateMessage contains a log of operations sent from the leader to a replica.

* + - Source: Leader
    - Destination: Replica
    - Contents: propose\_term, propose\_number, commit\_index, log

The replica confirms that its role is either candidate or witness and checks that the commit\_index is a (+1) of its local commit\_index:

* Change the state to recovery
* Commit – Log (candidate/witness) – Execute – Log (Candidate) each of the operations in the received log, as the propose\_number and
  + **ResponseMessage (request\_id, payload)**

The message is sent from the candidate (read) or leader (write) as a result of successful operation on the kvstore application:

* + - Source: Replica (Candidate/Leader)
    - Destination: Client
    - Contents: request\_id, reply

**Correctness / Liveness**

All messages sent between nodes are over an asynchronous and unreliable network. The messages may arrive out of order, the replicas may time out while processing a message or even crash and become unavailable for extended periods of time, replicas may be recovering after a failure, the messages may be lost at any point in time and they may be duplicated. These issues can adversely affect the operation of the protocol. In the next parts, we look at the different elements (messages, timers and nodes) and how the protocol can satisfy the enumerated safety and liveness properties:

1. In each propose\_term, a unique leader will be elected within a reasonable period of time.
2. The profile of the elected leader should be greater than or equal to profiles of other *f + 1* replicas.
3. A leader will eventually be elected for each propose\_term.
4. If a leader is not elected within a reasonable period of time, new leader election attempts should be initiated by the candidates for the same term.
5. An elected leader should be able to coordinate all actions of the replica system.
6. A new election term shall be initiated if:
   1. If a new replica with a **significantly** higher profile is available
   2. If the profile of the current leader drops beyond a defined threshold
   3. If the current leader is unreachable or unable to periodically communicate with the replicas
7. A leader configuration should remain active for a fairly long-term unless there is a drastic change to the environment properties.
8. The replicas will always accept an elected leader even when they have better profiles compared to it.

RequestVoteMessage

* Properties
  + A unique RequestVoteMessage can be processed only once at a replica.
  + A RequestVoteMessage shall be delivered to all the replicas.
  + For a given propose\_term, there is a RequestVoteMessage that will eventually lead to election(receipt) of a single leader (information).
* A duplicate RequestVoteMessage should not change the state of the recipient replica. The replica will provide the same response as before.
* A destination replica will ignore the message if at an advanced state as per the propose\_term and propose\_number. Any advanced replica has to inform the source to change its state, configuration and possibly the data.
* If the destination replica has a better profile, it has to change its role to a candidate and can restart a similar election cycle (same propose\_term) if a leader decision is not made within a specific timeout period.
* A RequestVoteMessage with the replica\_role as candidate with the same propose\_term is allowed as leader decision may have not been reached after the LeaderVoteTimer. This is only allowed at the initialization state.
* Propose\_term 1 is allowed only at the initialization phase. A successful election cycle will promote the replicas to an operational state and roles, and hence later elections shall be with a higher propose\_term.
* The propose\_number should always be higher or equal to the propose\_term. This will eliminate possibility of stale elections.
* The RequestVoteMessage is issued by the replicas at random times to improve the decision times and limit conflicts
* Given a set of RequestVoteMessages (Rvi, Rvi+1, Rvi+2, … Rvn), where Rv = (id, role, propose\_term, propose\_number, profile) for any replica (R) seeking election:
  + At a source replica R, Rvi == Rvn (duplicate), if RequestVoteTimer is None, and role == member (no leader has been elected yet);
  + At a destination R, Rvi == Rvn (duplicate), if the source resent the message or the network duplicated it. If there is no leader elected yet, the previous response is resent or else it will be ignored and the source informed of the current replica configuration.

ResponseVoteMessage

* Properties:
  + A ResponseVoteMessage is a response (vote) to a RequestVoteMessage for a propose\_term that a pair of replicas have agreement over.
  + A quorum of ResponseVoteMessage’s guarantees election of a single leader for a given propose\_term.
* The source replica will provide a ResponseVoteMessage only for a new term or a known term if it had voted before by resending this response.
* The destination replica will only process unique ResponseVoteMessages as votes which are actively checked for quorum.
* A count and eventual leader status is attained only if a replica receives *f + 1* votes from other replicas in the system
* All the votes have to be counted and decision made within the LeaderVoteTimer (candidate) and LeadershipVoteTimer (candidate/witness/member)
* After a leader is elected, and a local replica has been notified, only higher level (> propose\_term) ResponseVoteMessages are issued or processed.

ConfigureReplicaMessage

* Properties:
  + There can be at most one ConfigureReplicaMessage for a propose\_term.
  + The ConfigureReplicaMessage will always be delivered to the replicas.
* The message is a confirmation that a replica was voted and has authority to declare the roles of the other replicas.
* This will reset (invalidate) any other election cycle for this propose\_term as the roles of the replicas are reset.
* After processing, any ConfigureReplicaMessage for a propose\_term is ignored.
* Replicas may receive this message at any point in their transitions in which they fall behind a leader in the propose\_term.

HeartBeatMessage

* Properties:
  + At all times, there is a ‘good’ leader in the replica system.
* If the replicas don’t receive the HeartBeatMessage, an election will be started as this may signal a network reachability issue between the current leader and other replicas.

PollLeaderMessage

* Properties:
  + In a replica system, a candidate is primed to take on the leader role if the current leader is unreachable or faulty.
  + A PollLeaderMessage may be delivered more than once
* Multiple PollLeaderMessages don’t change the state of the leader/system but can always be responded to.
* For each PollLeaderMessage, there must be only PollLeaderOkMessage response to it. //Do we identify them?
* With *f+1* candidates, at least one of them is expected to reach the leader else there will be need for a new election cycle.

PollLeaderOkMessage

* Properties:
  + The PollLeaderOkMessage must be delivered exactly once to the candidate in response to the PollLeaderMessage
* The PollLeaderMessage/PollLeaderOkMessage may be dropped, and a candidate may assume that the leader is unreachable. The candidate initiates an election, but this will require a quorum in which the candidates (and the leader) constitute over 50%.
* A leader may not vote in a new election if it believes it is still alive, and the election type is not profile change.
* The candidate will only vote in a new election round if the PollLeaderTimer has timed out or the type of election is profile change.