FAB: An Intuitive Consensus Protocol using Raft and Paxos

Bhavin Thaker
Symantec Corporation[*]
bhavin_thaker@symantec.com
23rd March, 2015

Abstract

The Fast Atomic Broadcast (FAB) protocol provides atomic broadcast of messages across a pre-configured set of cluster members such that either all or none of the members receive the messages. FAB is a distributed consensus protocol that builds on concepts from the Raft and Paxos protocols, but is relatively intuitive and easier to understand. Similar to Raft and Paxos, FAB is faster than the usual 2-Phase Commit (2PC) protocol because FAB requires the broadcast of a message to reach only a majority of the cluster members. In contrast, the 2PC protocol requires all cluster members to receive the message. The cluster operates as long as there is a majority of the members in the cluster. All messages to be broadcast are routed through a single Leader with remaining members serving as Followers. The Leader is elected based on majority consensus among the members, and the messages are committed based on majority consensus. The broadcast messages are persisted in a transaction log and retained until all cluster members receive the messages. When appropriate, each cluster member deletes the messages (garbage collection). If a disconnected member lags in terms of the messages it has persisted, as compared to other members, that member is synchronized automatically when it connects back to the cluster. In addition, FAB supports dynamic addition and deletion of a member in a cluster while the broadcast of messages is in-progress, while continuing to provide consistency and high-availability at all times. FAB also provides weighted leader election to influence which member of the cluster becomes the Leader without compromising consistency.

1. Introduction

A plethora of distributed consensus protocols have been proposed, of which Paxos [10] is one of the most widely used. Raft [16] and Ark [7] make a sincere attempt to have a consensus protocol that is easier to understand. FAB is a newer protocol that combines essential ideas from the Raft and Paxos protocols to provide an intuitive distributed consensus algorithm.

The FAB protocol is implemented using the C language in a user-level library. The implementation is about 5,000 lines of production-quality source code (excluding the RPC communication layer), and it handles many corner-case scenarios reliably.

Explaining Distributed Systems in a paper is like describing a movie, where the element of time and multiple actors make it difficult to capture the story easily in diagrams. Inspired by comic-book-style illustrations, the time-lapsed illustrations in this paper make the core concepts intuitive and easy to understand. This paper borrows pedagogical concepts from ECKEL, B., MARSH, D., Atomic Scala [3]. The principles of "Show, don't tell" and "Practice before theory" aid in appreciating the design decisions.

2. Motivation for FAB

The main motivation to create FAB is based on the observation that Paxos and Raft have many elegant design concepts that could be combined and enhanced to build a protocol that is simple, intuitive and easy to understand without requiring correctness proofs or math to appreciate the beauty of a distributed consensus protocol. The "foundational" scenarios described in this paper are sufficient to understand the protocol in order to implement it easily.

FAB uses the notion of a strong Leader as well as terminology and data structures such as term, voted_for, etc., borrowed from Raft. FAB, however, avoids Raft's overlapping configurations and joint consensus, which can be complex and unreliable to implement. Instead, FAB requires only one configuration change to be pending at a time. It broadcasts and applies the configuration change via a message to the majority of the cluster members. Additionally, similar to Paxos but unlike Raft, FAB considers a message as committed implicitly if the message is present on a majority of the cluster members. This property is used by FAB's Leader elec-

tion to maintain message consistency and reliability. In contrast, Raft allows many messages to be written to the Leader log without requiring that they have been written to the majority of the cluster members. This can be a source of complexity for handling many corner-case scenarios. For example, having the Leader restart multiple times with many uncommitted entries makes recovery complex.

The primary use-case of FAB is replicating configuration updates persistently across a set of cluster members. In such use-cases, where the messages are given a global sequence number, the messages typically are heavily interdependent. Hence, it is believed that there is not much benefit gained if the design was to transmit multiple interdependent messages in parallel after getting a global sequence number and then implement complex recovery scenarios. Instead, by making the design decision to wait for message completion before transmitting the next message, the simple and minimalistic infrastructure design principle has eased implementation of features such as dynamic addition and deletion of cluster members, and weighted leader election for the highest weighted member to become the Leader. Additionally, similar to Paxos and Raft, waiting for a majority of the cluster members allows FAB to scale to a large number of nodes.

3. Design

3.1. Problem Overview

The problem of Distributed Consensus is to have cluster members agree on a proposal made to the cluster. The proposal is a message sent to any member of the cluster. Either all members agree to accept the message, in which the message is "**committed,**" or none of the members agree to retain the message, if less than a majority of the members happen to have the message. In the latter case, the members discard ("**rollback**") the message since the message was "**not committed.**" For each message received by the cluster, a globally unique message identifier (msgid) is assigned by the cluster, after which the message is either committed or rolled back before the next message is accepted by the cluster.

3.2 Configuration Overview

Information about all members in the cluster is configured in the fab/conf/fab.conf file, which is same on all cluster members. Each cluster member has a unique numeric identifier in the fab/conf/myid file, which is different for each cluster member.

…/fab/conf/fab.conf: is the FAB configuration file that lists information about all the cluster members. The format resembles the Zookeeper [4] configuration file.

```
server.1=10.182.198.21:12341:22221:10
server.2=10.182.198.22:12341:22221:20
server.3=10.182.198.23:12341:22221:30
#server.myid=IPaddr:Iport:Eport:Weight
```

Each line in the configuration file above refers to a particular cluster member. The IP address after the equals sign is the member's IP address. The two ports after the IP address denote the internal port number that is used for FAB-to-FAB member communication and the external port number that is used for client-to-FAB communication. The number at the end of the line denotes the weight of the member, which is used for Weighted Leader Election. This weight influences Leader Election such that the member with the highest weight becomes the Leader. In the above configuration, member 3 becomes the Leader since it has the highest weight of 30, as long as it can get a majority of votes from the cluster members. The fourth line is a comment line.

…/fab/conf/myid: is the member identification file. The id is an integer from 1 to 127, the largest member count curently supported by FAB. For example, the myid file on member 1 contains "1".

3.3 Interface Overview

This section provides a black-box overview of the FAB API. Each API has a flags argument that is reserved for use in special cases. All FAB API calls are blocking. The return value of each API call is 0 for success and non-zero for failure.

Figure 1 shows the flow of API calls made by FAB clients on each cluster member. The FAB client on cluster member S1 initiates a broadcast to the FAB Leader, which is elected through majority consensus among the cluster members on startup. The FAB Leader assigns a new message id to each message received for broadcast and triggers a broadcast of the message to all the cluster members. Each cluster member, on receiving the broadcast message, updates the FAB state, logs the message persistently, and sends an ACK back to the Leader that initiated the broadcast. The Leader uses the 2PC protocol to commit the message on at least a majority of the cluster members. Once the message is committed, it is ready to be received by the client on each cluster member. The FAB client on each cluster member calls fab_recv() to receive the message and get the associated message id for the received message. The client processes the received message and then receipt-acknowledges (called rack) the message id to FAB. The

fab_rack() acknowledgement is an operation local to the cluster member, and it does not send any network message. Mainly, it indicates to FAB what message the client has processed so that FAB does not need to redeliver the message to the client in cluster member failure scenarios. When the message is racked locally and is present on all the cluster members, it is deleted. Note that all cluster members can issue the broadcast call in parallel even though they are routed for broadcast through the single Leader.

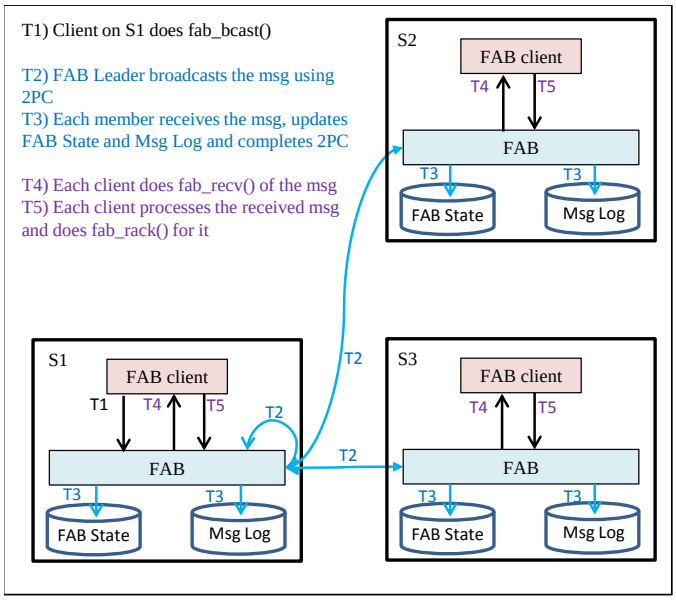

Figure 1: Flow of API calls made by FAB clients

3.3.1 fab_open()

```
int fab_open(struct rpc *rpc,
             struct fab **fabpp, int flags);
```

fab_open() opens the FAB module based on the information in the configuration files, viz. fab/conf/{fab.conf, myid} and returns a pointer to FAB via fabpp. The passed-in RPC pointer is used for RPC communication between cluster members.

3.3.2 fab_bcast()

```
int fab_bcast(struct fab *fabp,
              char *msgp, uint32_t msglen,
              uint32_t  *msgidp,
              uint32_t  prev_msgid, int flags);
```

fab_bcast() broadcasts messages atomically. The message pointer is passed via msgp and the message length via msglen. FAB returns the message id assigned to this message via msgidp. The argument, prev_msgid, is the msgid of the latest msgid known to the client. If prev_msgid is specified, the fab_bcast() call checks if the specified prev_msgid value matches the latest committed msgid in FAB. If it does not match, it implies that another msgid was committed recently by another cluster member. When this happens, fab_bcast() returns EINVAL. Then, the client reads the message that recently got committed, updates its prev_msgid accordingly and reinitiates the fab_bcast() call. If prev_msgid is 0, this check is skipped by FAB. The prev_msgid check acts like a test-and-set operation and is similar to the version check of Zookeeper's [4] znode in the zoo_set() API. It allows the broadcast to be atomic without requiring explicit locking.

3.3.3 fab_recv()

```
int fab_recv(struct fab *fabp,
             char *msgp, uint32_t *msglenp,
             uint32_t *msgidp, int flags);
```

The fab_recv() call receives the message from FAB via msgp. The client allocates memory for the message and passes in pointers to the message and its length via msgp and msglenp respectively. FAB returns the length of the message via the same pointer msglenp. The message id is returned by FAB via msgidp.

3.3.4 fab_rack()

```
int fab_rack(struct fab *fabp,
             uint32_t msgid2ack, int flags);
```

The client uses fab_rack() to "receipt ack" all the data received via fab_recv() and processed so far. Note that the ACK is not sent over the network; it merely updates the persistent pointer to the data successfully read and processed by the client so far. As an example, if the client has not yet completed ACK of msgid=150, on restart after a crash, the client gets msgid=150 from FAB. If the client has successfully completed ACK of msgid=150, on restart after a crash, the client gets msgid=151 onwards from FAB. In other words, on a restart that happens before the fab_rack() completion, the unacknowledged data is received by the client. Similarly on a restart that happens on fab_rack() completion, the acknowledged data is no longer received by the client from FAB.

3.3.5 fab_close()

```
int fab_close(struct fab **fabpp, int flags);
```

This closes the FAB module, freeing up resources.

3.3.6 A sample program to use FAB library

A self-explanatory sample program to use the FAB library for broadcast of three messages is given in Figure 2.

```
#include "fabapi.h"

#define NUM_TEST_MESSAGES       3

/*
 * Sample program to use FAB library.
 * No error checking shown to maintain code conciseness.
 */

int
main(int argc, char **argv)
{
        int rc = 0;
        struct rpc *rpc;
        struct fab *fabp;
        int txid;

        char cmdbuf[FAB_MSG_MAXLEN];
        uint32_t cmdlen;

        /* Initialize RPC handle */
        rpc = rpc_init(NULL, &rc);

        /* Open FAB and get a handle in fabp */
        rc = fab_open(rpc, &fabp, open_flags);

        uint32_t msgid, prev_msgid;

        /* perform atomic broadcast of 3 messages */
        prev_msgid = 0;
        msgid = 0;
        txid = 1;
        while (txid <= NUM_TEST_MESSAGES) {
                sprintf(cmdbuf, "HELLO WORLD from FAB: txid=%u\n", txid);
                cmdlen = strlen(cmdbuf);

                rc = fab_bcast(fabp, cmdbuf, cmdlen, &msgid, prev_msgid, 0);

                ++txid;

                prev_msgid = msgid;
        } /* while */

        /* Read the 3 messages and receipt-ack (rack) each message read */
        printf("NOW ... reading messages ...\n");
        txid = 1;
        while (txid <= NUM_TEST_MESSAGES) {
                msgid = 0;
                cmdlen = FAB_MSG_MAXLEN; /* defined in fabapi.h */
                memset(cmdbuf, 0, FAB_MSG_MAXLEN);
                rc = fab_recv(fabp, cmdbuf, &cmdlen, &msgid, 0);
                printf("fab_recv: msgid=%u, cmdlen=%u, cmdbuf=%s\n",
                                        msgid, cmdlen, cmdbuf);

                rc = fab_rack(fabp, msgid, 0);

                ++txid;
        }

        /* Close the FAB handle */
        rc = fab_close(&fabp, 0);

        return 0;
} /* main */
```

Figure 2: A sample program to use FAB library

3.3.7 Dynamic addition of a FAB member

The member that wants to join a cluster sets its myid configuration file to 0 and sets its member information in the first line with a member id of 0 in the fab.conf file. The second line lists the leader to which it needs to send an add request. An example configuration for dynamic addition is given below, where the current leader has myid = 2.

…/fab/conf/myid:
0

…/fab/conf/fab.conf:
server.**0**=10.182.198.24:12341:22221:5
server.2=10.182.198.25:12342:22222:3

Once the member addition process is complete, the myid file contains the new member id, and the updated fab.conf is replicated on all the cluster members. The configuration files for the above example, after the dynamic addition process has completed, are given below:

…/fab/conf/myid:
4

…/fab/conf/fab.conf:
server.1=10.182.198.21:12341:22221:2
server.2=10.182.198.22:12341:22221:3
server.3=10.182.198.23:12341:22221:4
server.**4**=10.182.198.24:12341:22221:5

3.3.8 Dynamic deletion of a FAB member

A member can be deleted only from the member server that is the Leader of the cluster. If the Leader deletes itself, it triggers reelection. The id of the member to be deleted from the cluster is written to the file …/fab/conf/deleteid, which FAB periodically reads and acts upon appropriately.

3.4 Design Overview

The following sections explain foundational scenarios of the overall design.

3.4.1 Leader election

To ensure that there is only one cluster member that assigns the globally unique message id, and to coordinate the message commitment across the cluster member, one of the cluster members acts as a "Leader**,**" while the remaining cluster members act as "Followers." By default, when a cluster member starts, it enters the state of being a Follower. After a wait of Heartbeat timeout period, the cluster member switches from the state of Follower to "Candidate" and requests votes from all the cluster members.

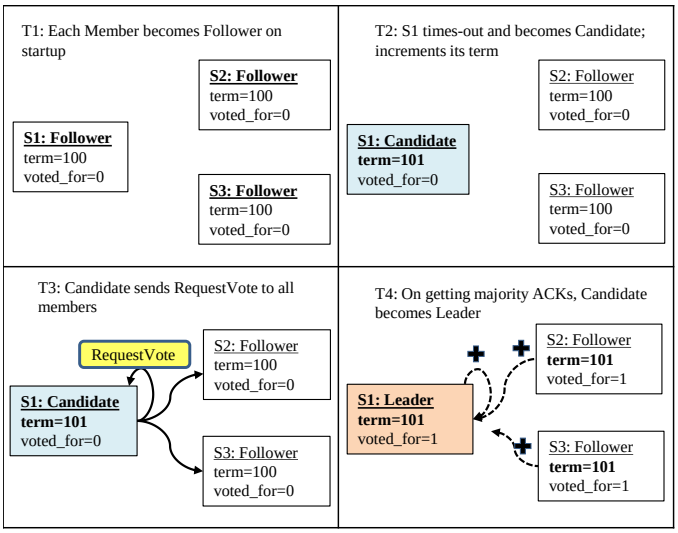

Figure 3: TLI for Leader election

On receiving votes from the majority of the cluster members, the cluster member switches its state from "Candidate" to "Leader." The time-lapsed illustration (TLI) for a 3-member cluster (server S1, server S2 and server S3) with the starting term 100 is shown in Figure 3. T1, T2, T3, and T4 represent the sequenced time-steps. The updated state variables are highlighted in bold. A member grants a vote to a Candidate for a particular **"**term,**"** similar to the way a citizen votes for the election of the President of a country for a particular term. The **"**term**"** helps identify stale messages after a configuration change like electing a new leader or adding or deleting a cluster member. Once a server has granted a vote to a candidate for a term, it cannot change its vote. However, it can communicate its already granted vote to the same candidate again in response to a request vote message. A member sends a positive acknowledgment as a grant vote and a negative acknowledgment as a reject vote. A member always grants a vote to a request vote with a term higher than it has already voted for, and always rejects a request vote with a lower term than it has already voted for.

3.4.2 Convergence of Leader election

If a Leader is not elected within the election timeout period, then the Candidate waits a random period less than the Election timeout and starts reelection by incrementing the term and sending the broadcast of RequestVote with the incremented term value. The reason for a random wait during reelection is to make it improbable to have multiple Candidates sending broadcasts of RequestVote messages at the same time and therefore causing reelection repeatedly. In some cluster configurations (say, an even number of cluster members, for example, four), there could be stalemate scenarios where two candidates receive votes from half (e.g. two) of the total number of cluster members (e.g. four) and not achieve a majority number of votes (e.g. three or more votes). The random wait makes this unlikely to occur. If it does, the Leader Election is retried, thereby helping in convergence of the Leader Election process. Additionally, each member's current term is updated to the term in the request vote message if the request vote's term is newer than its current term. The Candidate increments its term at the start of a new election so that previous lower-numbered terms are considered stale. The members are aware of the global state of each other and this helps in fast convergence of the Leader election.

3.4.3 State transition diagram

Figure 4 summarizes the state transitions.

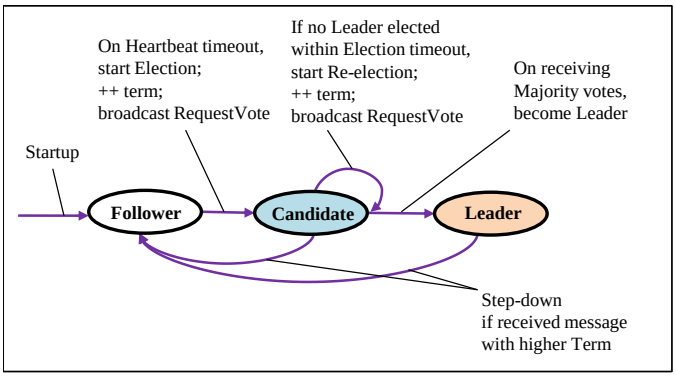

Figure 4: State transition for member type

3.4.4 Leadership assertion

The Leader sends heartbeats at periodic time intervals (called timer timeout, for example, 3 seconds) to all the cluster members and expects acknowledgments (ACKs) from a majority of the cluster members to assert its leadership**.** If the majority of the ACKs do not arrive in a pre-determined time interval (called heartbeat timeout, for example, 15 seconds, that is, 5 missed heartbeats), then the Leader "steps-down" from its Leader state and becomes a Follower. Each Follower expects a periodic heartbeat from the Leader, and if it does not arrive in the pre-determined time-interval (heartbeat timeout, 15 seconds), then the Follower assumes that the Leader is no longer alive and "steps-up" to the Candidate state, thereby triggering Leader Election. After the Candidate sends RequestVote messages to all cluster members, if a majority of votes are not received with an Election timeout period (for example, 10 seconds), the Candidate increments the term and starts a new reelection. The timer out (3 seconds) is less than the election timeout (10 seconds), which is less

than the heartbeat timeout (15 seconds). The values are chosen based on the principle that leader election step should be deferred as much as possible to handle spurious network failures but once started, the leader election step should complete as soon as possible. Leadership assertion is illustrated in Figure 5.

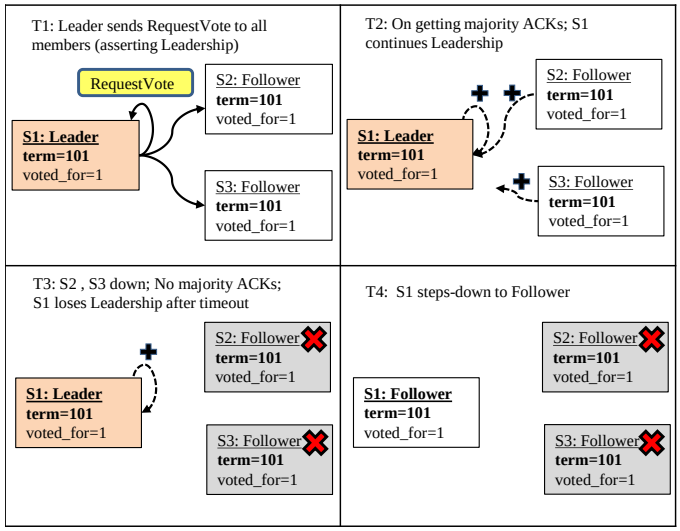

Figure 5: TLI for Leadership assertion

3.4.5 Stale Leader

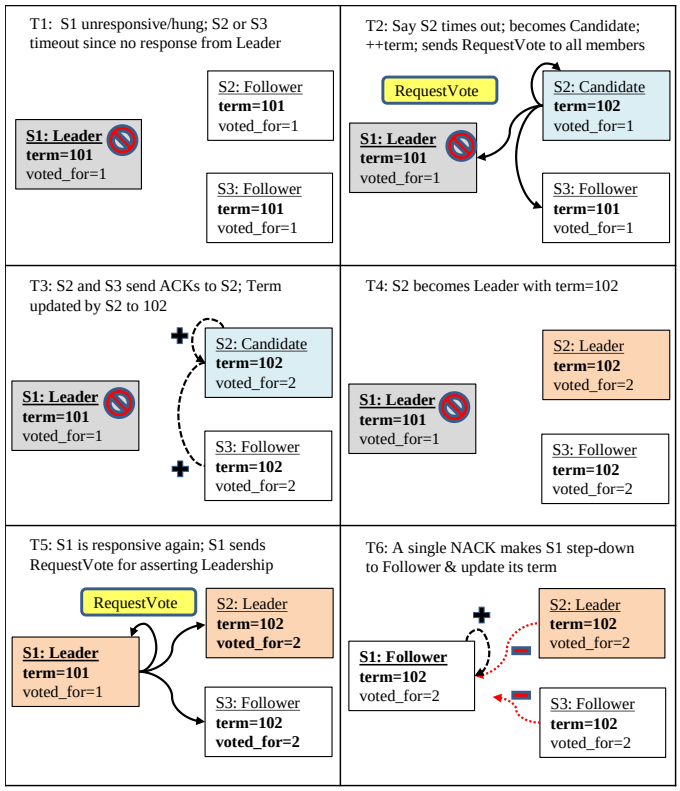

Figure 6: TLI for handling a Stale leader

It is possible that network connectivity could be lost temporarily between members of the cluster such that a Leader is disconnected from all the cluster members, the remaining cluster members elect a new Leader and then the original Leader is connected back to the cluster. Another way to get into this scenario is when the current Leader is unresponsive (as illustrated by a crossed-circle in time-steps T1 to T4 of Figure 6), so that another cluster member becomes the Leader.

In order to handle this scenario of two Leaders in the cluster, the notion of "term" is used as a logical clock. Each Leader election process starts with a new term number, incremented during the Candidate state. If we have two Leaders in the cluster, the Leader with the higher term survives, whereas the Leader with the lower term is a stale Leader and steps-down to the Follower state, as illustrated in Figure 6. Additionally, a cluster member grants its vote to a candidate for a particular term after which it is not allowed to regrant its vote for that term. This ensures convergence of the Leader Election process and is similar to the voting process, where a citizen is allowed to grant only one vote to a candidate of the Presidential election for a particular term.

3.4.6 Split Votes during Leader Election

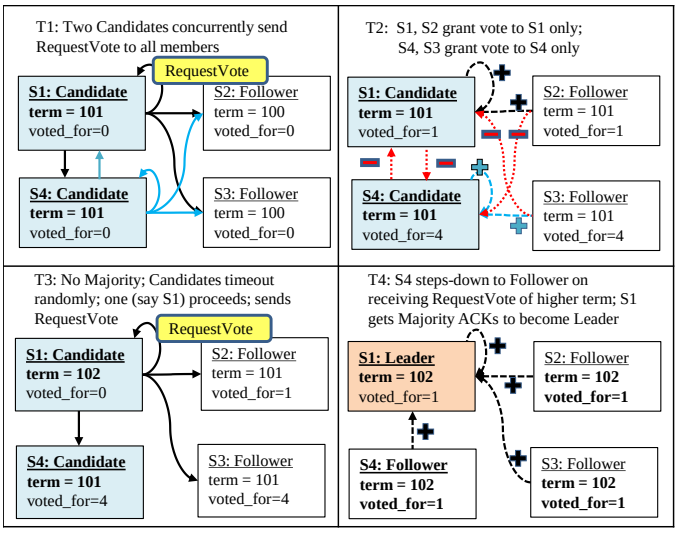

Figure 7: TLI for handling Split Votes during Leader Election

It is rare but possible that multiple candidates request votes from the cluster members and get an equal number of votes, causing a tie and leading to an indeterminate state. As an example, if we have four cluster members, it is possible that two candidates could get two votes each, leading to a tie. If a candidate does not get a majority of votes, then it waits for a random time period, times out and restarts the election. In the example described in Figure 7, if two candidates get same num-

ber of votes, then each one of them waits for a random time period and restarts the election. Next, both candidates will most probably wait for different time periods and one of the candidates will win the election before the other candidate, thereby causing convergence and completion of the Leader election process.

3.4.7 Log Replication

A message submitted to the cluster is broadcast to all members of the cluster so that eventually all members have the same message or none of them have this message. This is the atomic property of the atomic broadcast. In order to be able to recover from many failure scenarios, for example, a member that is offline when the broadcast happens, or a member that goes offline after receiving the message, the broadcast is done using the 2 Phase Commit (2PC) protocol. In the current implementation, each cluster member is the client doing the broadcast and is aware of the Leader in the cluster. The Leader receives the message and performs the 2PC of the message to the cluster members.

Assume that the message that needs to be broadcasted is "aa", as illustrated in Figure 8. After receiving the message, the Leader assigns a message id to the message (say msgid=1) and sends an AppendEntries (AE) request with the msgid and the message (="aa") to all the cluster members. This is Phase 1 (P1) of the 2PC protocol. Each cluster member, including the Leader itself, receives the msgid (=1) and the message (="aa") and stores it persistently in a log file. The message is stored in a log file in order to be able to recover from node crash or reboot. The failures are assumed to be non-malicious (non-Byzantine); that is, on a crash or during data transfer the data is not tampered with and remains unchanged. Two values related to the message id, curr_msgid and cmted_msgid, are required to complete the 2PC protocol. They are abbreviated as curr_m and cmted_m respectively in the illustrations. As an example, when the msgid=1 is in Phase 1, curr_m is updated from 0 to 1, whereas cmted_m remains 0. As part of Phase 2, cmted_m is updated to 1 completing the 2PC protocol. Each member records the log message (="aa") and curr_m (=1) persistently and then sends a positive ACK back to the Leader. This completes Phase 1 of the 2PC protocol, as illustrated in steps T1 to T4 of Figure 8.

Note that if majority of the cluster members have persistently written the log message, the transaction is considered committed, even though the ACKs have not been sent by the cluster members. In other words, once the message is persistently written to a majority of the cluster servers, if there is a crash of the complete cluster, this message will be recovered and be present in the cluster. If the majority of the cluster members did not store the message, then there is no guarantee of the message being present in the cluster, and the results may be indeterminate.

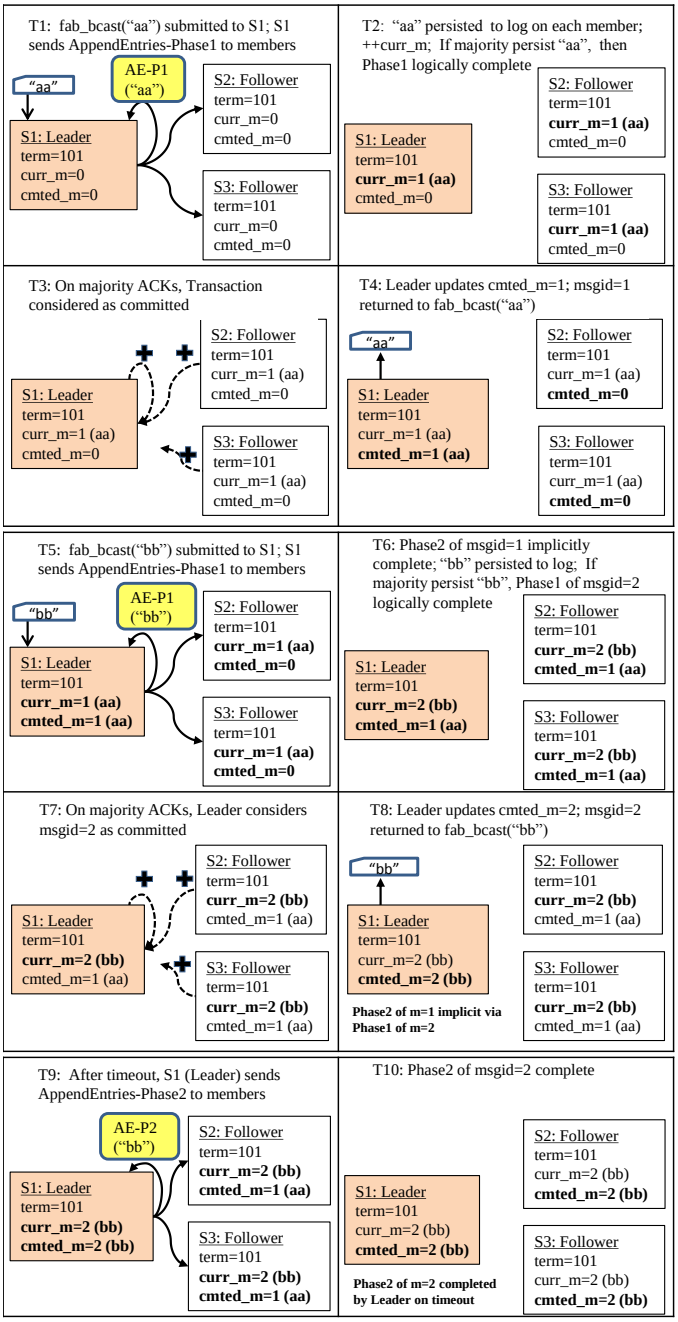

Figure 8: TLI for Log Replication via 2PC

In the 2nd phase of the 2PC protocol, if the Leader receives a majority of positive ACKs from the cluster members, it learns that the message was committed from the ACKs and communicates this information to all the cluster members as part of the 2nd phase. As an optimization for performance reasons, this 2nd phase is not immediately carried out by the Leader. The Leader

waits for a new message to arrive, and if it does, then receipt of the Phase1 of a subsequent message implicitly indicates completion of Phase2 for the previous message, as illustrated in steps T5 to T8 of Figure 8.

Finally, if a newer message does not arrive, as illustrated in time-steps T9 to T10, the Leader times out and sends an explicit AppendEntries (AE-P2) message for Phase2 of the 2PC protocol, thereby completing 2PC.

3.4.8 Replay of messages

Consider the scenario illustrated in Figure 9, where a message is uncommitted since the message from the Leader reached itself but did not reach the other cluster members. This means that the message is not present on a majority of the cluster members. Therefore, curr_m=1 and message "aa" got recorded persistently on the Leader whereas other cluster members have curr_m=0 and and no message "aa". Now, if the Leader crashes and one of the cluster members, S2, becomes the Leader with term=102. It is not aware of the uncommitted message on S1. So when a new message ("bb") arrives, S2, being the Leader, assigns msgid=1 to it and does a 2PC of this message to get it committed. When S1 comes online, since its "aa" message was uncommitted, it is overwritten via replay of the messages from the Leader S2. Thus, it is guaranteed that committed messages are retained by the cluster, whereas the presence of the uncommitted messages after the cluster recovery is indeterminate.

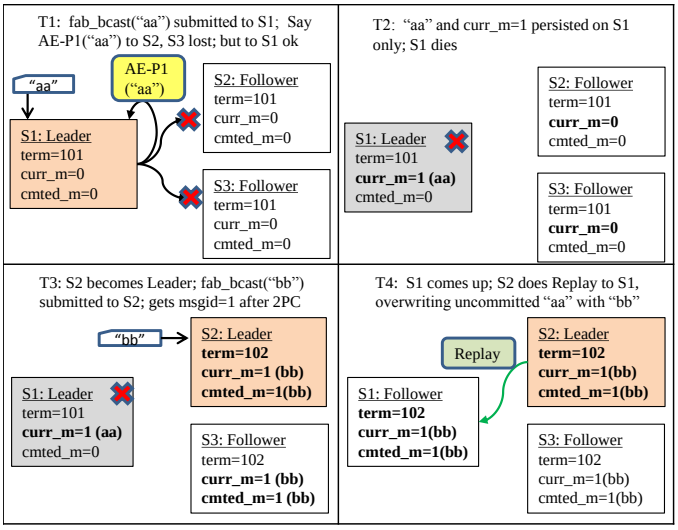

Figure 9: TLI for uncommitted message overwritten via Replay

When a disconnected member joins the cluster and the Leader finds that the disconnected member lags in terms of the messages, the Leader sends the pending messages to the lagging member. This step is called replay of the messages from the Leader to the lagging member. The lagging member cannot participate in the 2PC of new messages unless it is up to speed with the Leader. This ensures that there are no holes or gaps in messages on a cluster member. This design choice also makes it easy to implement dynamic addition or deletion of a cluster member.

3.4.9 Recovery of messages

This section explains a scenario where a message is present on a majority of cluster members in Phase 1, but the Leader crashes before 2PC completes. The presence of the message on the majority of the cluster members influences the selection of the Leader deterministically. The Leader performs recovery of this committed message so that it is retained and is eventually present on all the cluster members. As illustrated in Figure 10, message "aa" is submitted to Leader S1 who assigns msgid=1 and sends AppendEntries request to the cluster members. Assume that the connection between S1 and S3 breaks, and the AppendEntries message for "aa" is lost. Next, assume that S1 crashes and S3 becomes a candidate. Note that S3 is stale, whereas S2 is more recent with respect to the messages received. When S3 sends a RequestVote message during Leader election to S2, S2 sends Negative ACK denying its vote, due to which S3 steps-down from candidate to follower and S2 steps-up from follower to candidate to leader, as illustrated in steps T1 to T6 in Figure 10.

If the message is not present on the majority of the cluster members, then it will get recovered only if one of the members containing the message participates in Leader election. However, this cannot be deterministic since it is possible that these cluster members containing the message may not participate in determining the majority for a new leader election, in which case this message will be lost. In other words, the presence of the message on the majority of the cluster members influences the selection of the leader deterministically in a guaranteed manner, while the presence of the message on a minority can influence the selection of the leader only if a cluster member containing the message participates in the new leader election, which may or may not happen.

After becoming the Leader, S2 notices that it has a message with msgid=1 ("aa") that it needs to "recover" and so it initiates 2PC on this message, resending messages for both Phase1 and Phase2 for this message so that it completes 2PC on a majority of the cluster nodes. When S1 comes online later, it gets a copy of the committed message from the Leader via replay of the pending messages that were committed while it was offline.

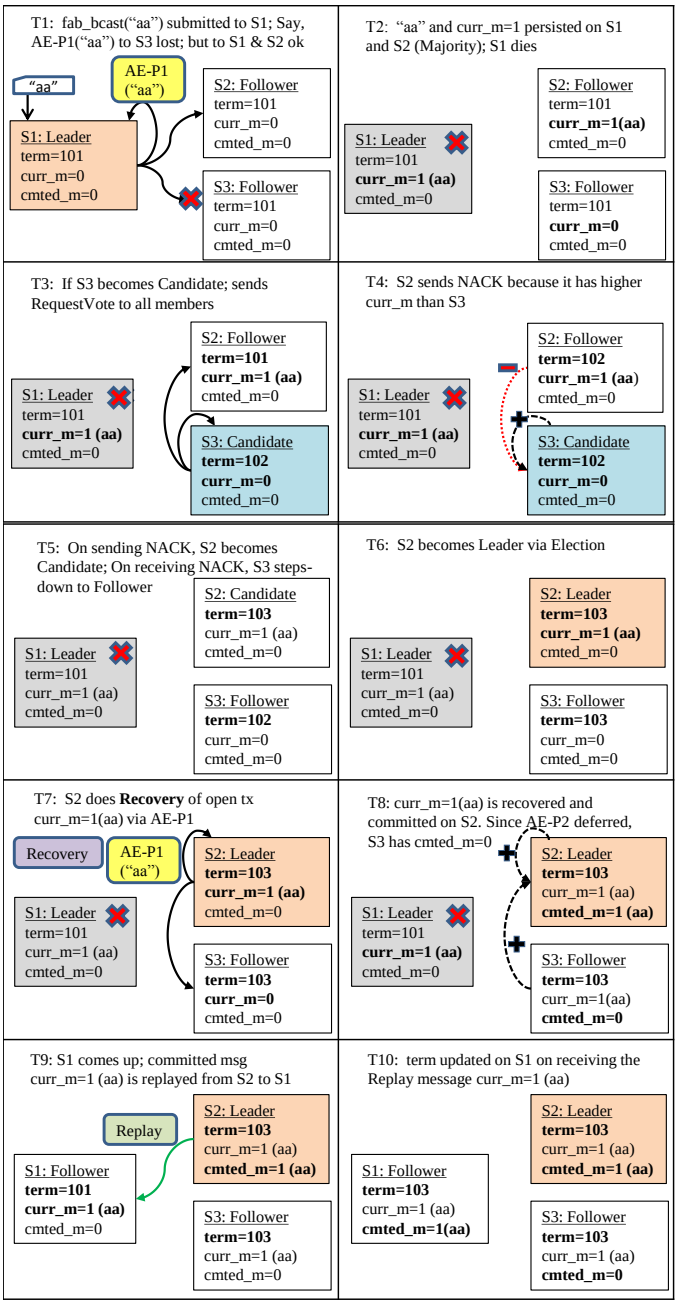

Figure 10: TLI for recovery of a committed message

Note the difference between the terms "replay" and "recovery." Replay of messages means synchronizing ready committed messages from the Leader to a lagging Follower, whereas recovery of a message completes the 2nd Phase of a committed message from the Leader to a majority of the cluster members. Replay happens for one or more messages that have completed 2PC, whereas recovery happens only for a single uncommitted message for which the 2PC is not yet complete. The replay happens to a single lagging cluster member at a time, though many independent, simultaneous replays are possible. In contrast, the recovery of a message happens to a majority of the cluster members, and there can only be one instance of recovery of a message happening at any point of time. Thus, an uncommitted message influences a Leader election so that the Leader can recover the message to complete the 2PC for the message and retain it in the cluster.

3.4.10 Garbage Collection

The scenario illustrated in Figure 11 is sufficient to understand the garbage collection design. Consider the scenario where cluster members S1 and S2 have seven messages committed so that their curr_m=7 and cmted_m=7, whereas cluster member S3 has been offline during the 2PC of these 7 messages, due to which S3 has curr_m=0 and cmted_m=0. This is time-step T1 in Figure 11.

Now, if S3 comes online and the Leader S1 does a replay of messages to the lagging S3, as in time-step T2. However, if S3 crashes after getting msgid=5 from the Leader S1. In this state, msgid=5 is committed on all the three members of the cluster and hence cmtedall_m=5 on all the cluster members. The cmtedall_m field indicates the msgid that has been committed on all the members of the cluster. The Leader determines the cmtedall_m value based on the cmted_m value in the response messages (both Request Vote and AppendEntries) from the cluster members. The Leader piggybacks the cmtedall_m value in the periodic RequestVote message (or AppendEntries message) to all the cluster members, making all the cluster members aware of this global state. The cmtedall_m field allows each member to know that messages earlier than this cmtedall_m msgid can be safely deleted since they are present on all the cluster members. However, a message on a cluster member cannot be deleted if it has not been read by the cluster member, and hence the racked_m field is used in conjunction with cmted_m field for garbage collection.

In time-step T4 of Figure 11, S1 has read all the seven committed messages due to which racked_m=7. However, it cannot delete all the seven committed messages because another cluster member (S3 in this example) will need to get messages from S1, if S1 is the Leader when S3 comes online. Hence S1 can delete messages only up to 5. In other words, messages can be deleted or garbage collected up to the minimum value of cmted_m and racked_m. So for S1, messages can be deleted up to:

MIN(cmtedall_m=5, racked_m=7) = 5,
which is indicated by deleted_m after the completion of the deletion. Similarly, for cluster member S2,

deleted_m = MIN(cmtedall_m=5, racked_m=3) = 3.

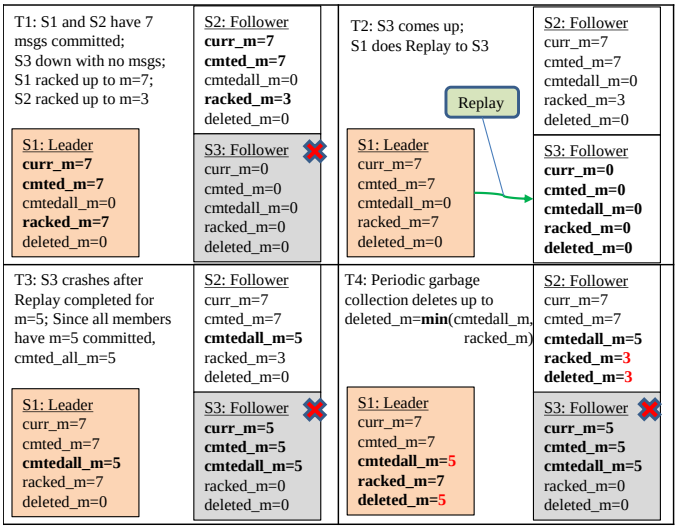

Figure 11: TLI for Garbage Collection

3.4.11 Dynamic addition and deletion of a member

FAB can dynamically add or delete a member in the cluster configuration without requiring the cluster to be brought offline. The dynamic addition or deletion of a cluster member can happen while the messages are being broadcast. This is possible because the dynamic addition or deletion itself is a special message that is broadcast similar to other messages. It is a special message because instead of the client message, the message content is the new configuration that the Leader generates and broadcasts to all the cluster members atomically.

Consider the scenario illustrated in Figure 12 where we have a cluster containing two members, S1 and S2, with a new cluster member S3 that needs to dynamically join the cluster. S3 configures its myid=0 in the fab/conf/myid file and its own configuration (that is, IP address, ports, etc.) plus the Leader configuration in the fab/conf/fab.conf file as mentioned below:

server.**0**=10.182.198.23:12341:22221:4
server.1=10.182.198.21:12341:22221:2

The line containing server.0 has the configuration for S3, which wants to be dynamically added to the cluster, and the line containing server.1 has the configuration for Leader S1. The information about the Leader is known to the administrator and is required to be set manually. This could be dynamically determined in a future implementation.

On startup, S3 sends an AddMember message to the Leader S1 with the configuration details of itself. The Leader S1 assigns a new member id (=3) to S3 and generates a new configuration file that also contains S3. The Leader S1 then atomically broadcasts this new configuration file as an AppendEntries message flagged as a configuration update. This special flag is used by the receivers of the message to apply the message inline after the message is determined to be committed, that is, after receiving majority ACKs on the Leader and after receiving Phase2 AppendEntries on the Followers. Applying the message means updating the configuration of the cluster member and recognizing the new server S3 to be part of the cluster. In other words, after the new configuration is applied, the Leader S1 replays of any pending messages to S3, if necessary. After the new cluster member S3 is synchronized with the Leader S1, S1 includes S3 in the broadcast of the 2PC of all the messages.

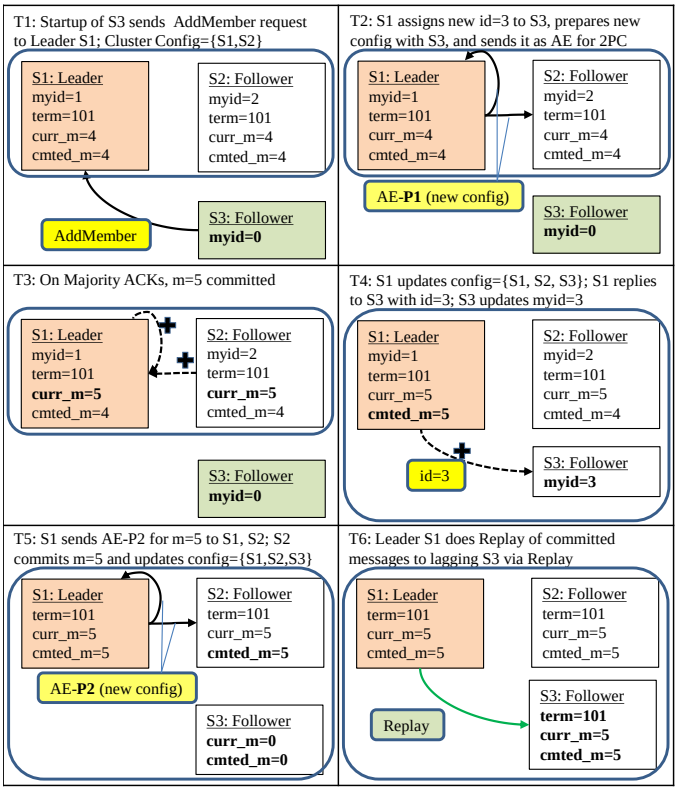

Figure 12: TLI for dynamic addition of a member

Dynamically deleting a cluster member uses the same procedure as dynamically adding a cluster member. The server id that needs to be dynamically deleted is written to the file fab/conf/deleteid on the Leader. The Leader periodically checks for the presence of this file and if present, the file is read, deleted, and then acted upon by the Leader; that is, the id of the cluster member mentioned in the file is deleted from the cluster. If the id is the same as the Leader, then the Leader steps-down and another cluster member goes through Leader election. A server that is deleted from the cluster brings itself down after it knows that it is no longer part of the cluster.

Note that the Leader always ensures that it is one of the members in the majority requirement for the configuration update to be considered committed so that the deleted server is not present in subsequent broadcasts. This design choice avoids the need for complexity of supporting multiple configurations as is the case in Raft.

3.4.12 Weighted Leader Election

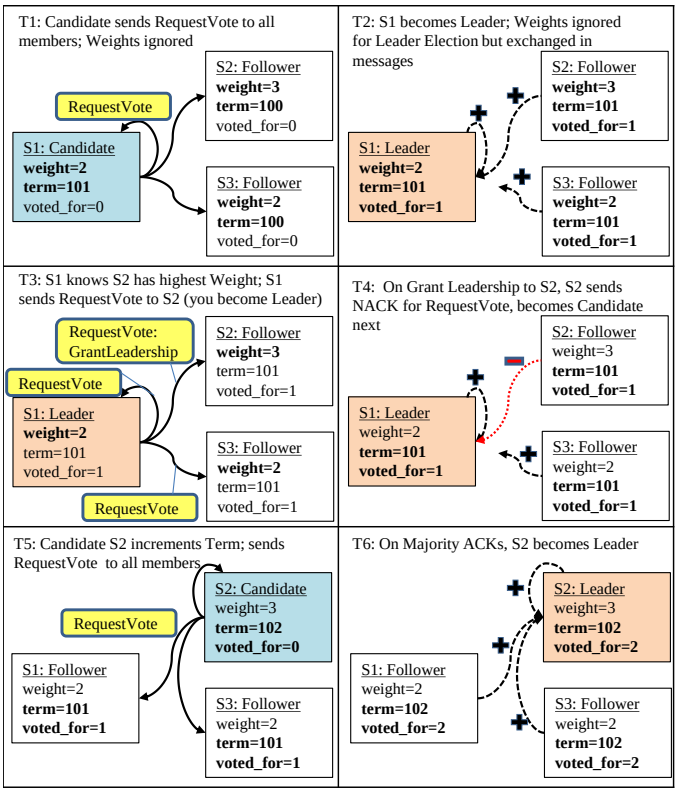

Figure 13: TLI for Weighted Leader Election

As illustrated in Figure 13, the Leader selection based on weight happens only after the Leader election based on consistency requirements (message recovery) is complete. This ensures that consistency is not compromised and a higher-weighted server does not win the election when another lower weighted server with a higher message number (higher curr_m or cmted_m) should win the election and become the Leader. Once the Leader election based on consistency requirements is complete, if the Leader is not the highest-weighted member of the cluster, then the Leader grants leadership to this highest-weighted server via a GrantLeadership flag in the RequestVote message to that server. When the highest-weighted server gets a RequestVote message with the GrantLeadership flag, it replies negative ACK to the RequestVote message so that the current Leader steps-down from its Leadership and transitions to Candidate state, thereby triggering Leader Election and becoming the Leader.

4. Implementation

FAB has an intuitive, efficient and maintainable implementation. FAB borrows data structures and protocol from Raft [16], uses the 2PC optimization from Paxos [10, 11, 12] and simplifies the overall design and implementation. Refer to the Raft data structures and protocol for help in understanding the FAB design and implementation.

As shown in Figure 14, FAB maintains a control block at each server to record its current volatile and persistent state. The persistent state is stored in the …/fab/state directory.

Similar to Raft, FAB has two message types for FAB-to-FAB communication on the internal port:

- RequestVote message:
    a. Used by a Candidate to request votes from members during Leader election.
    b. Used by the Leader to send heartbeat and assert Leadership.
- AppendEntries message:
    a. Used by the Leader to atomically replicate or broadcast messages to cluster members.
    b. Used by the Leader to atomically perform dynamic addition or deletion of a cluster member.
    c. Used by the Leader to replay messages to a lagging Follower.
    d. Used by the Leader to recover a message that is on a majority of the cluster members.

Each member is aware of the committed msgid state of the online members of the cluster since this information is exchanged during the RequestVote and AppendEntries messages. A timer thread runs periodically in each member to perform many housekeeping tasks. For example, if the member is the Leader, the timer thread sends heartbeat RequestVote messages to ensure that a message is not stuck for a long time in a particular phase of 2PC (after which the Leader steps-down to Follower). Additionally, the timer thread ensures that the AppendEntries message for the 2nd Phase of the 2PC is sent after a timeout. FAB uses negative ACKs judiciously to transfer critical information to cluster members and simplify the protocol and its implementation.

FAB has extensive tracing capabilities with detailed diagnostic messages that helps to see the protocol in action. The fab_state.sh utility displays the FAB state interactively so as to debug and view the FAB protocol behavior while it is running. The tables in Figure 14 show the main FAB data structures.

FAB Persistent State	
type	Follower or Candidate or Leader
term	Current term
cmted_term	Last committed term in log
curr_msgid	Current message id in log
cmted_msgid	Committed message id in log
racked_msgid	Receipt-ACKed msgid in log
deleted_msgid	Highest deleted msgid in log
cmtedall_msgid	Msgid committed on all members
voted_for	I have voted for this member

RequestVote: Send Message	
leader_id	Sender's current Leader id
id	Sender's id
term	Sender's current term
flags	Flags to denote Grant Leadership
curr_msgid	Sender's current msgid
cmted_msgid	Sender's last committed msgid
cmted_term	Sender's term for cmted_msgid
cmtedall_msgid	Committed msgid at all members

RequestVote: Response Message	
leader_id	Responder's current Leader id
id	Responder's id
term	Responder's term
vote_granted	ACK/NACK/0
curr_msgid	Responder's current msgid
cmted_msgid	Responder's committed msgid
cmted_term	Responder's term for cmted_msgid

AppendEntries: Send Message	
id	Sender's id
msgtype	Msg Type: Data/Config Update
logtype	Log Type: Phase1/Phase2/Reply
leader_term	Sender's (Leader's) current term
cmted_msgid	Sender's last committed msgid
cmted_term	Sender's last committed term
cmtedall_msgid	Last cmted msgid at all members
msgid	Log entry index
term	Log entry term
msg[]	Log entry message
msglen	Log entry message length

AppendEntries: Response Message	
leader_id	Responder's Leader id
id	Responder's id
ack	ACK/NACK
curr_msgid	Responder's current msgid
term	Responder's term
cmted_msgid	Responder's last committed msgid
cmted_term	Responder's last committed term

Bcast: Send Message	
id	Sender's id
msgtype	Msg Type: Data/Config Update
prev_msgid	Client's cmted_msgid so far
msg[]	Log entry message
msglen	Log entry message length
cfg_ipaddr	Member add/del: IP address
cfg_eport	Member add/del: External Port no.
cfg_iport	Member add/del: Internal Port no.

Bcast: Response Message	
Id	Responder's id
msgtype	Msg Type: Data/Config Update
ack	ACK/NACK
cmted_msgid	Committed msgid
leader_id	If non-0, redirect to new leader
leader_ipaddr	Leader's IP address
leader_iport	Leader's Internal Port no.
leader_eport	Leader's External Port no.
cfg_myid	New id set by Leader to receiver

Figure 14: FAB Data structures

5. Summary

FAB combines essential ideas from Paxos and Raft protocols to provide an intuitive distributed consensus algorithm. FAB also simplifies design choices like waiting for a message to complete its Phase1 before acting on the subsequent message. The tradeoff is that the sequenced messages are not sent in parallel. This tradeoff is insignificant because the sequenced messages are interdependent. Better network latency has also helped in making such design choices for FAB as compared to the distributed systems that were built about a decade ago. The hope for a simpler system is that it is intuitive and easier to understand and therefore more reliable due to a rock-solid implementation that can be tested well. FABulous, isn't it?

Acknowledgments

I would like to thank Sharad Srivastava and Partha Seetala for the opportunity to work on the distributed consensus problem. I would also like to thank my team members who reviewed this design paper and the FAB protocol. Finally, this work would not have been possible without the fabulous work done by the authors of Paxos and Raft, which formed the foundation for FAB.

References

[1] BURROWS, M. The Chubby lock service for loosely coupled distributed systems. In *Proc. OSDI'06, Symposium on Operating Systems De-*

sign and Implementation (2006), USENIX, pp. 335–350.

[2] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[3] ECKEL, B., MARSH, D., Atomic Scala, http://www.atomicscala.com

[4] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc ATC'10, USENIX Annual Technical Conference* (2010), USENIX, pp. 145–158.

[5] JOHNSON, B. Raft: The Understandable Distributed Consensus Protocol http://www.infoq.com/presentations/raft

[6] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M., Zab: High-performance broadcast for primary-backup systems. In *Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks* (2011), IEEE Computer Society, pp. 245–256.

[7] KASHEFF, Z. WALSH. L., Ark: A Real-World Consensus Implementation, http://arxiv.org/abs/1407.4765

[8] KIRSCH, J., AMIR, Y. Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.

[9] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM 21*, 7 (July 1978), 558–565.

[10] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[11] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25

[12] LAMPORT, L. Fast Paxos. *Distributed Computing 19*, 2 (2006), 79–103

[13] LogCabin source code. http://github.com/logcabin/logcabin

[14] MAZI `ERES, D. Paxos made practical, Jan. 2007 http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf

[15] ONGARO, D. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014 http://ramcloud.stanford.edu/~ongaro/thesis.pdf.

[16] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm http://ramcloud.stanford.edu/raft.pdf.

[17] Latest version of this paper (Symantec -internal): http://socialtext.ges.symantec.com/samg/fab