Raft Protocol Implementation

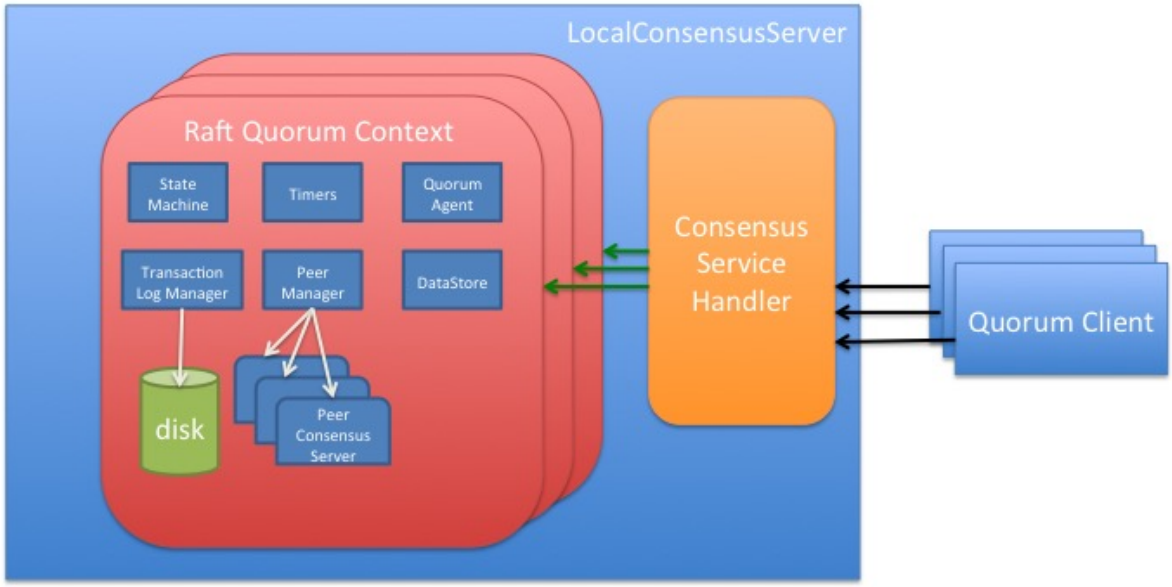
TABLE OF CONTENTS

Introduction

This is the design document for the Raft Protocol implementation in the https://github.com/apache/hbase/tree/HBASE-12259 branch. It describes the overall layout of the different modules and how important phases like (AppendRequest/Election process) are carried out. As a word of advice, please read the original RAFT paper to understand the protocol and the correctness.

Modules

High level diagram about the modules in the RAFT Protocol implementation.



RaftQuorumContext

This object stores the state of the quorum on every peer. There is only one RaftQuorumContext object per quorum per peer. It is the central place where the entire state of the quorum can be retrieved from. The information it stores can be divided into :

Static Information

1. Identification related information (Name, QuorumInfo, Address, Rank)
2. References to other modules like RaftStateMachine, progress/heartbeat timers, TransactionLogManager, PeerManager, DataStore
3. Configuration

Dynamic Information

1. Current Role (LEADER/FOLLOWER/CANDIDATE)
2. Current Leader
3. Current EditID, Committed EditID, Previous EditId
4. Session Objects (AppendConsensusSession/VoteConsensusSession)
5. Uncommitted Transactions
6. Metrics

PeerManager

Provides an abstraction layer to send requests to the peers in the quorum. It exposes methods to send AppendRequest/VoteRequest. It is aware of the current mode of the quorum, i.e whether it is undergoing a quorum configuration change(JointConsensusPeerManager) or is in a single quorum mode(SingleConsensusPeerManager). Every RaftQuorumContext will have only one PeerManager at any given point in time.

PeerConsensusServer

This class is responsible for sending/receiving request from a single peer. There will be one PeerConsensusServer for each peer in the quorum. This class is responsible for sending the thrift request on the wire to the remote peer and also forwarding the response received back to the RaftStateMachine. It also captures the state of the remote peer, i.e. (lastAckedId). The Peer Recovery logic (incase the peer is lagging) is handled in the PeerConsensusServer. The flow of events is captured in the PeerStateMachine.

TransactionLogManager

Manages the logs for the Consensus Protocol and provides interfaces to perform read/write/truncate operations to the logs. This is and should be the single point of entry to access any consensus logs.

Layout

All the logs will be stored in the 'TRANSACTION_LOG_DIRECTORY/quorum-name/' directory. The directory will have 2 folders:

1. /current : contains the current random access log
2. /finalized : contains all the logs for which the entries have been committed by the leader

Life cycle of a log file

1. Every log (apart from recovery) will start in a Random-Access mode in '/current' directory. All the append() operations will be going to this log. The log could be potentially truncated during this phase.
2. Once the size of the log reaches the LOG_ROLL_SIZE and all the entries are committed the log will be finalized and moved to the 'finalized' directory. Now the log is available in ReadOnly mode and cannot be truncated, but still can be deleted.

3. Once all the entries from the log are flushed by the data store, the log will be eventually deleted depending upon the LOG_RETENTION_POLICY.

There are two sets of logs maintained by the log manager in-memory:

1. uncommittedLogs => log files which reside in the /current dir and contains all the uncommitted edits.
2. committedLogs => logs files which reside in the /finalized dir and contains all the committed edits.

The log roller will always grab the write lock on the logRollerLock before moving the file from /current to /finalized directory. All the read operations grab the read lock across the entire operation to prevent the log rolling from happening. append/truncate will always be writing to files which will never be rolled at the same time. Hence, they do not grab any logRollLock.

The invariants are:

1. For any given index, there should be only ONE entry in ONE log file.
2. There should never be a gap in index and hence it should always be monotonically increasing.

RandomAccessLog

The RandomAccessLog provides the random access interface for the transaction log. This class is not thread-safe in general. This class holds one LogWriter instance, which delegates the write operation. And assume there is one thread accessing these write APIs. In addition, if the log file has been finalized for write, the file will be immutable. Also this class maintains a map of LogReader instance, which delegates the read operation. The map is indexed by the session key from the client, and the client will use its own session key to access these read APIs. Concurrent access from different session key is thread safe. But concurrent access from the same session key is NOT thread safe.

ReadOnlyLog

The ReadOnly log provides read interface for the transaction log. Also this class maintains a map of LogReader instance, which delegates the read operation. The map is indexed by the session key from the client, and the client will use its own session key to access these read APIs. Concurrent access from different session key is thread safe. But concurrent access from the same session key is NOT thread safe. The naming convention will be "log_<term>_<initial-index>_<last-index>"

SeedFile

This is a special type of ReadOnlyLog file used to denote the starting index in the Transaction Log Manager. By default, in the Raft Protocol, there cannot be any gaps in the indexes present in the log. There can be scenarios, where peers of the quorum fail and the MTTR for the nodes is in days. During this time, the quorum has moved forward and when the peer comes back from repair, it has to recover all the indexes(transactions) since it went down. This recovery can consume significant amount of time and can also cause extra pressure on the leader and consume more resources (network, disk, cpu). However, in HydraBase the peers in one DC are sharing the HDFS. The ACTIVE peer is flushing the same transactions into HDFS, which can be accessed by all the peers in the same DC. Hence, there is no need to actually recover all the transactions and instead it can just start the recovery since the last flushed seq-id. This is sort of a jump that one can make and avoid recovering all the transactions since the last stop time. However, just to keep the TransactionLogManager happy, it needs to have a notion that all the indexes are present. Hence, we use the SeedFile, which is sort of a dummy file used to fill the gap between the last transaction in the peer to the last flushed id of the peer.

LogFormat

Following describes the transaction log format. Every log file stores transactions belonging to only one term. It will have a header at the start which stores:

1. File Version
2. Term
3. Initial Index

Following the header, there will a list of log entries. Each log entry will have:

1. Header
    1. Total Size of the log entry
    2. CRC (Checksum)
    3. Index
2. Payload (User Data)

```
* ------------------
* | File Header:    |
* |  Version   4B   |
* |  Term      8B   |
* |  Index     8B   |
* --------------------
* | Entry # N       |
* |  Entry Header   |
* |   Size     4B   |
* |   CRC      8B   |
* |   Index    8B   |
* |  Entry PayLoad  |
* --------------------
* | Entry # N       |
* |  Entry Header   |
* |   Size     4B   |
* |   CRC      8B   |
* |   Index    8B   |
* |  Entry PayLoad  |
* --------------------
```

QuorumClient

Is used to communicate with the Quorum on the remote side. It exposes methods to write transactions to the Quorum or to change the quorum configuration.

Internally, it uses the QuorumInfo object to discover the address of the peers and then finds out the leader of the Quorum. It then sends the thrift request to the leader via the ConsensusService.

QuorumAgent

It is used to send data via the ReplicateEntries request to the RaftQuorumContext. It is the single point where all the write requests are batched and then sent to the the RaftQuorumContext state machine. There is only one QuorumAgent for every RaftQuorumContext

DataStore

It is the backing database engine (eg HRegion) where the transactions are stored and served to the client. The leader will pass the transaction to the database once its committed from the protocol perspective. The Database engine then can use that transaction reliably. Also, optionally the Database engine can subscribe to important events in the protocol, like when a peer decides to become leader, or if the peer loses the leadership and becomes a follower.

ConsensusService

It is the abstraction layer which encapsulates all the methods related to the protocol per quorum. Currently, the implementation exports the ConsensusService as a Thrift Service.

LocalConsensusServer

It is the server class which spins up a ThriftService at the configured port to serve the methods exported by the ConsensusService. One can use the ThriftServerConfig to configure the thrift server port and # of worker threads used, etc.

Timers

Progress Timer

As described in the paper, the progress timer is active whenever the peer is in a Follower/Candidate mode in the quorum. Whenever a timeout happens it calls the onTimeout method in the TimeoutEventHandler. The ProgressTimeoutCallback implements the TimeoutEventHandler and submits a ProgressTimeout Event to the RaftStateMachine.

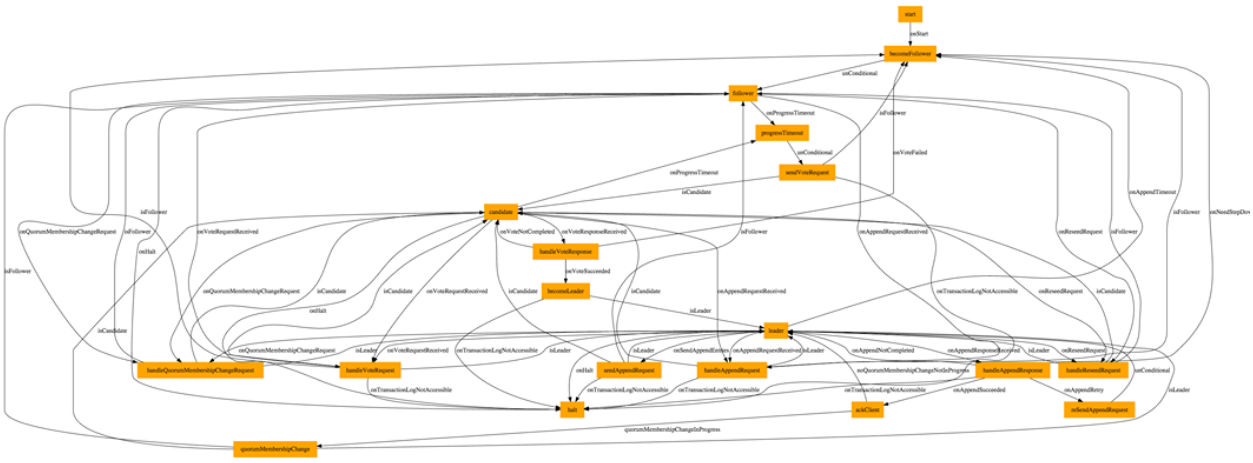
Heartbeat Timer

As described in the paper, it is the heartbeat timer active only on the leader of the quorum. Whenever the heartbeat interval is reached, the onTimeout() method will submit an empty ReplicateEntriesEvent to the RaftStateMachine. Based on the current state of the leader, it can either retry the pending unacked AppendRequest or send out an empty heartbeat request.

State Machine

Complete State Diagram for RaftStateMachine

For completeness, here is the entire state machine diagram for the RAFT state machine. For individual flows w.r.t data/leader election please go the # section below.



RaftStateMachine

States

Start	On Start
Halt	On Stop
BecomeFollower	The peer has now stepped down to become a Follower. It will cancel any outstanding AppendConsensusSession/VoteConsensusSession. It will also start the progress timer in case it was stopped and stop the heartbeat timer in case it was started.

Follower	The peer is a follower. Do nothing, wait for events
BecomeLeader	The peer has received majority ack from the VoteRequest and will now try to become the leader. In this state it will send out the empty AppendRequest to establish the leadership for the new term.
Leader	The peer is a leader. Do nothing, wait for events
Candidate	The peer is a candidate and is waiting for VoteResponse, do nothing.
HandleVoteRequest	Handle the VoteRequest event
HandleVoteResponse	Handle the VoteResponse event
HandleAppendRequest	Handle the AppendRequest Event
HandleAppendResponse	Handle the AppendResponse Event.
HandleQuorumMembershipChangeRequest	Handle the Quorum Membership Change Request event
ProgressTimeout	Encountered a Progress timeout
SendAppendRequest	Send out the requested append request.

ReSendAppendRequest	Retry the current append request. At this point, the request has already been written to the local transaction log. We are just waiting for peers to ack.
SendVoteRequest	We have received a progress timeout, send a VoteRequest to the peers.
AckClient	We have received ack from majority of the peers in the quorum. Lets mark this AppendRequest as committed.
ChangeQuorumMembership	Start/Continue the current quorum membership change request
HandleReseedRequest	Reseed the transaction log manager from the given index if possilble.

Events

VOTE_REQUEST_RECEIVED

VOTE_RESPONSE_RECEIVED

APPEND_REQUEST_RECEIVED

APPEND_RESPONSE_RECEIVED

RESEED_REQUEST_RECEIVED

REPLICATE_ENTRIES

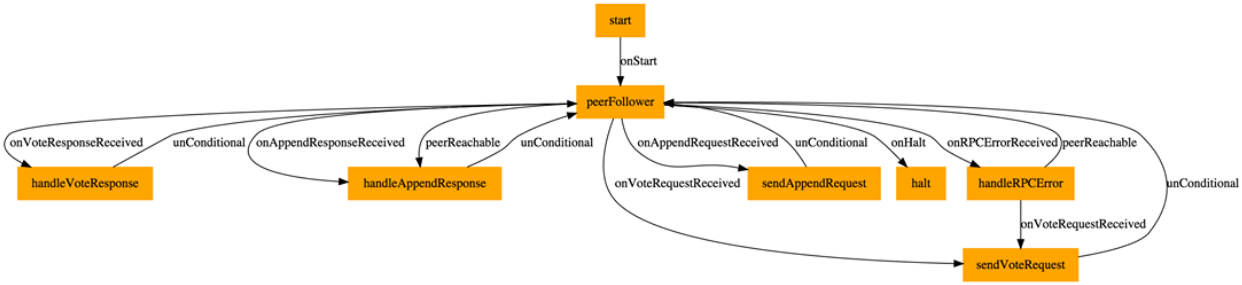
PROGRESS_TIMEOUT

TERM_TIMEOUT

HALT

QUORUM_MEMBERSHIP_CHANGE

PeerStateMachine



States

Start

PeerFollower

PeerHandleVoteResponse

PeerHandleVoteRequest

PeerHandleAppendResponse

PeerHandleAppendRequest

PeerHandleRPCError

Events

Start

PEER_VOTE_REQUEST_RECEIVED

PEER_VOTE_RESPONSE_RECEIVED

PEER_APPEND_REQUEST_RECEIVED

PEER_APPEND_RESPONSE_RECEIVED

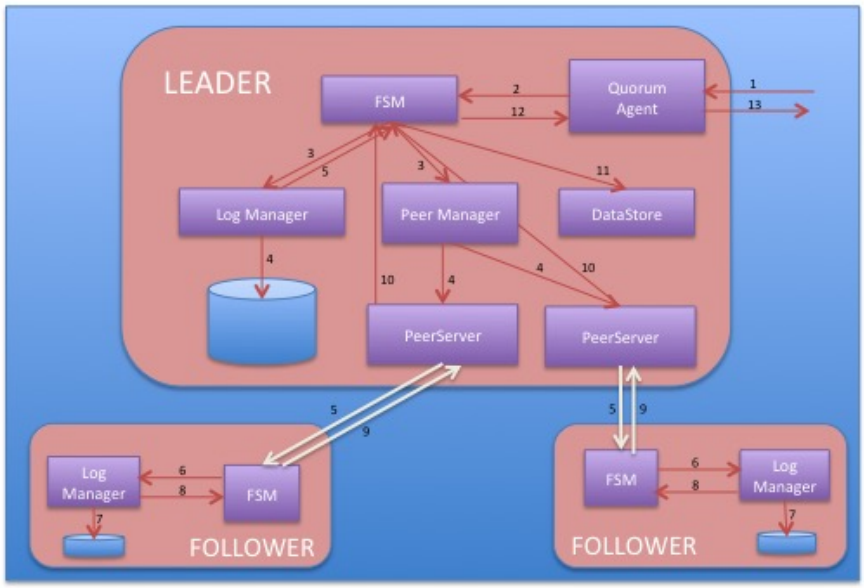
PEER_RPC_ERROR

PEER_REACHABLE

HALT

One Transaction Commit Flow

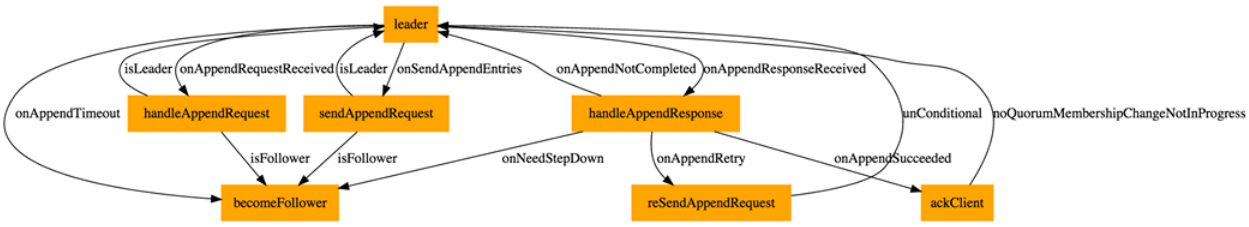
Following diagram describes the flow of one transaction in the protocol. The QuorumAgent is the entry point, where the clients can send the data to get replicated across the quorum.



1. QuorumAgent get the write request
2. QuorumAgent batches the current pending requests and sends a ReplicateEntriesEvent to State Machine (FSM)
3. FSM transitions to SendAppendRequest State and creates an AppendConsensusSession, updates the index and sends the request to its local LogManager and PeerManager
4. The LogManager stores the transaction to local storage media and the PeerManager sends the PEER_APPEND_REQUEST_RECEIVED to each of the PeerConsensusServer

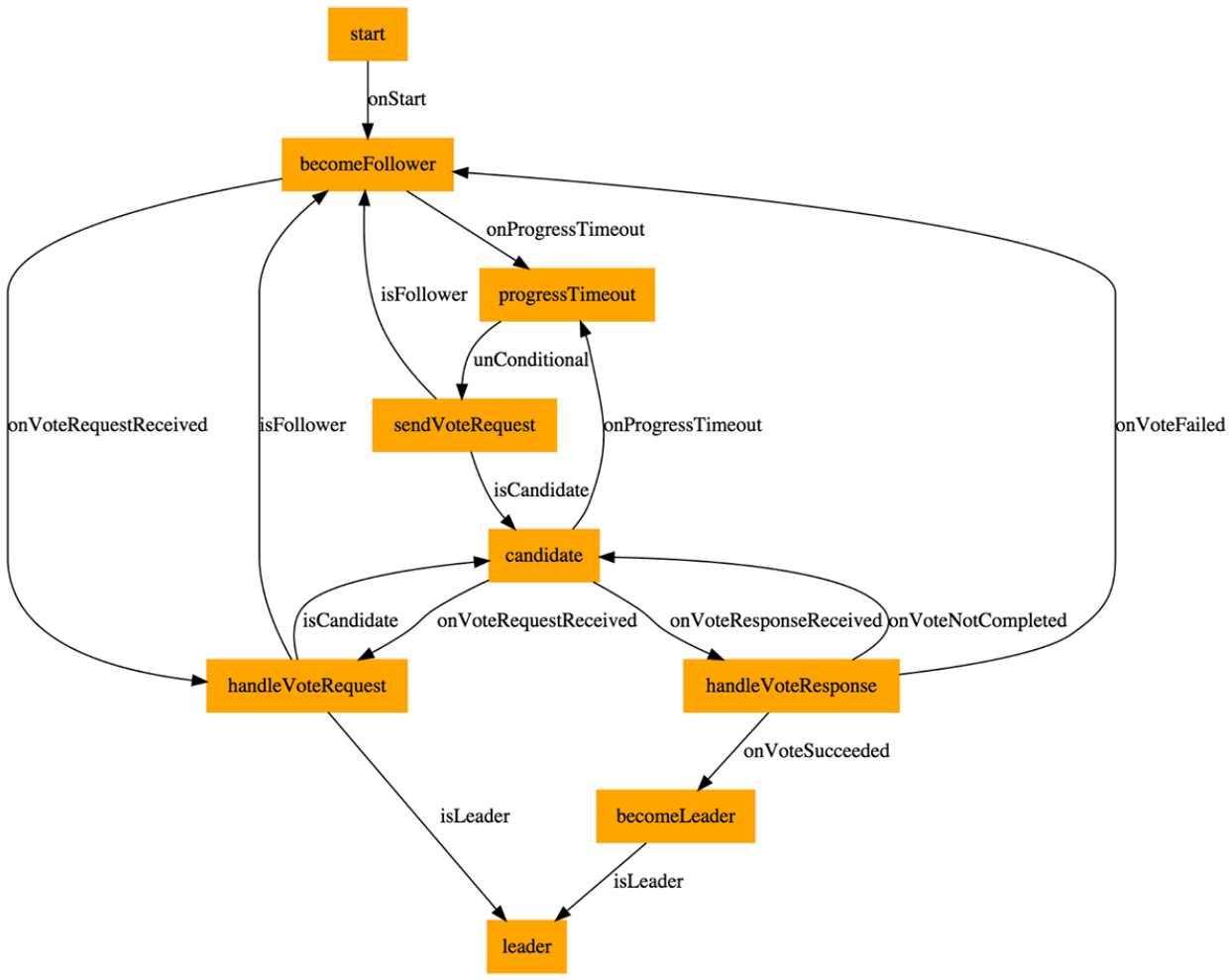
5. The PeerConsensusServer based on the lastAckId, will send out the AppendRequest to the remote peer
6. The Remote peers will get the AppendRequest and queue APPEND_REQUEST_RECEIVED to the FSM. FSM on processing the request will enter HandleAppendRequest state
7. In HandleAppendRequest, it will write the transaction to the log via the TransactionLogManager
8. The TransactionLogManager will then ack and the request is acked by the Follower
9. PeerConsensusServer receives the results, updates the lastAckedId
10. PeerConsensusServer then forwards the request to the Raft FSM. Raft FSM updates the AppendConsensusSession with the ack's received
11. Once the majority number of peers ack, the RaftFSM moves to AckClient Stat. In AckClient, it updates the committedEditId, previousEditId and sends the transaction to the DataStore
12. It also marks the ReplicateEntriesEvent as completed
13. The QuorumAgent then acks to the client

AppendRequest state machine diagram on the leader



Election Process

The election process is pretty much the same as described in the Paper. The flow of events as can well walked through the following state transition diagram.



Peer Recovery

The peer recovery is completely handled by the PeerConsensusServer object on the leader. The PeerConsensusServer object keeps the track of

1. lastAckedId: The last edit successfully acked by the remote peer
2. latestRequest: The latest edit(transaction) the leader is on

Based on these two parameters on every response received from the remote peer, it knows whether the peer is lagging or not. In case it is lagging, it will fetch the next set of transactions from via the TransactionLogManager and batch them and send it out as an AppendRequest to the peer. This logic is embedded in the PeerHandleAppendResponse state.

Data Structures

QuorumInfo

```
public class QuorumInfo {
  public static final int PAYLOAD_HEADER_SIZE =
        + Bytes.SIZEOF_BYTE   // Magic value
        + Bytes.SIZEOF_BYTE   // Payload type
        + Bytes.SIZEOF_BYTE;  // Payload version

  private Map<String, Map<HServerAddress, Integer>> peers = null;
  private Map<HServerAddress, Integer> peersWithRank = null;
  private Set<String> peersAsString = null;
  private final String quorumName;
}
```

AppendConsensusSession

```
public class AppendConsensusSession implements
AppendConsensusSessionInterface {
  private final int majorityCount;
  private final EditId id;
  private final AppendRequest request;
  private final ReplicateEntriesEvent replicateEntriesEvent;
  /** The key of ackMap is the peer host name and value is the time elapsed for
the ack */
  private final Map<String, Long> ackMap = new HashMap<>();
  private final Set<String> lagkSet = new HashSet<>();
```

```
  private final Set<String> highTermSet = new HashSet<>();
  private final Set<String> peers;
  private final ConsensusMetrics metrics;
  private long sessionStartTime;

  private final int currentRank;

  /** This boolean flag indicates whether there is any higher rank peer caught up the transacations
    * and potentially ready for taking over the leadership. */
  private boolean isHigherRankPeerCaughtup = false;

  private final boolean enableStepDownOnHigherRankCaughtUp;
  private SessionResult currentResult = SessionResult.NOT_COMPLETED;

  private int tries = 0;
  private final int maxTries;

  ImmutableRaftContext c;
}
```

VoteConsensusSession

```
public class VoteConsensusSession implements
VoteConsensusSessionInterface {
  private final int majorityCount;
  private final VoteRequest request;
  private final EditId sessionId;

  private final Set<String> ackSet;
  private final Set<String> nackSet;
  private final Set<String> peers;

  private final long sessionStartTime;
  private final ConsensusMetrics metrics;

  private SessionResult currentResult = SessionResult.NOT_COMPLETED;
}
```

AppendRequest

```
@ThriftStruct
public final class AppendRequest extends Request<AppendResponse> {
  private final String regionId;
  private final ConsensusHost leaderId;
  private final boolean isHeartBeat;
  private final long commitIndex;
  private final long persistedIndex;
  private final EditId prevLogId;
  private final List<EditId> logIds;
  private final List<ByteBuffer> listOfEdits;
  private boolean isTraceable = false;
  private SettableFuture<AppendResponse> response;
}
```

VoteRequest

```
@ThriftStruct
public final class VoteRequest extends Request<VoteResponse> {

  private final String regionId;
  private final String address;
  private final long term;
  private final EditId prevEditID;

  private SettableFuture<VoteResponse> response;
}
```

EditId

```
@ThriftStruct
public final class EditId implements Comparable<EditId> {

  private final long term;
  private final long index;
}
```

Appendix

State Machine Implementations

The Raft Protocol implementation uses the state machine heavily. The State Machine approach provides a nice event based thread-safe execution model. As shown above, the logic or the flow of events can be transformed into a state machine, which mainly has:

1. A set of states
2. A set of external events
3. A set of conditional events
4. A set of transitions which are defined as Transition(currentstate, event) -> newstate

State

A state can be logically defined as the phase of the machine at a given point in time. Following describes the data structure of the state.

```
public abstract class State {
  protected StateType t;

  public State(final StateType t) {
    this.t = t;
  }

  public StateType getStateType() {
    return t;
  }

  @Override
  public String toString() {
    return t.toString();
  }

  abstract public void onEntry(final Event e);
```

```
  abstract public void onExit(final Event e);

  /**
   * @return Return true if the state is an async state.
   */
  public boolean isAsyncState() {
    return false;
  }

  public ListenableFuture<?> getAsyncCompletion() {
    return null;
  }

  /**
   * This method dictates whether the onEntry() method hasn't completed
   * logically, even if the call has actually returned.
   *
   * The FSM checks for this method before making any transition. If this method
   * returns false, then any arriving events will not be applied/aborted, even
   * if the call to onEntry() method has returned.
   *
   * This is a way to do FSM thread-blocking work in an async fashion in the
   * onEntry method, and make the FSM wait the state to complete before
   * transitioning off to another state.
   *
   * By default, the method returns true, which means the FSM will not wait for
   * any async ops that you would have issued. States which require waiting,
   * will need to override this method, and make this method return true when
   * they are done.
   * @return
   */
  public boolean isComplete() {
    return true;
  }
}
```

Based on the input events and the state transition table, the state machine will move from one state into other. The operations performed in the state machine are expected to be short lived and mostly in-memory. In case they are io operations or are time consuming ones, it is advisable to use AsyncState, which provides the notion of freeing up the input thread(to be useful for other operations) and return back to same state when the operation is complete.

Events

An event can be defined as notification request sent to the state machine w.r.t a change in state of the system. They can have associated payload data required to gather more information about the event.

1. External Events: These are events which are triggered based on the external change in state of the system. For example, if you get a request from a remote host. It will be given as an input to the state machine as a event with the information about the request embedded in the payload of the Event object
2. Conditional Events: These are events which are triggered based on the internal transitions from one state to another and does not depend on any external change in state. The user needs to implement the isMet() method of the Conditional Interface.

For example, consider a simple state machine, where you just increment a counter based on external event INCREMENT. The user wants the state machine to go to HALT, when the counter value reaches 100. So, one can implement this as :

```
int counter = 0;

class A extends State {
  void onEntry() {
    counter++;
  }
}

class HALT extends State {
}

class ReachedHundred implements Conditional {
```

```
  boolean isMet() {
    return counter == 100;
  }
}

// Setup
State A = new A();
State HALT = new Halt();
Transition onIncrement = new Transition(INCREMENT_TRANSITION, new
onEvent(INCREMENT));
Transition onReachingHundred = new Transition(HALT_TRANSITION, new
ReachedHundred());

// state machine transition table
addTransition(A, A, onIncrement);
addTransition(A, HALT, onReachingHundred);
```

There are two modes of implementation available for State Machine in the current implementation. The major difference between the two is that in one approach, there is a dedicated thread per state machine in first approach, while in the second approach we use a threadpool shared across all the state machines.

Single Threaded State Machine

The approach is pretty state forward. There is one thread per state machine and once we start the thread, it just waits for new events coming into the state machine and process the events one at a time as they come.

```
while (!stop) {
    1. complete the async state
    2. Check for new events
    3. if (newEvent)
          handleEvent(e)
       else
          wait for new Event
```

```
}

handleEvent(Event e) {
   1. Check if there exists a transition from current state for event e.
   2. if (transition exists)
      1. Call onExit() on current State
      2. Set currentState to nextState
      3. Call onEntry on next State
      4. Check if there are any more conditional transitions from current state.
If yes, go to step 2, else return
}
```

Serial Executor Design

The Single threaded state machine does not scale well when there are more than hundreds of state machine in a process. The reason is that is mainly bounded by the number of threads and the thread switching cost increases with increased number of active threads. So, the State Machine also provides the Serial Executor approach, which is more of an chained event based model. The approach is as follows:

1. Whenever a new event comes in, it adds itself to the end of the queue of events for the state machine and adds a SettableFuture depending upon the completion of the last event present in the queue. This way it creates a chain of futures, making the execution asynchronous. Whenever an execution of the event is complete, the onCompletion method will set the Future Object, making the next event executable.
2. There is a thread pool which contains threads which are waiting for executable events in the State machine event queues.
3. As soon as an event is executable, one thread from the thread pool will pick up the event for that state machine and process the event as mentioned in the handleEvent() above. At this point, it also makes sure that there is only one thread working on a state machine at any given point in time.

Timer

The progress of the protocol is based on the timers. The number of timers present in the process is directly proportional to the number of quorums the node hosts. Hence, for scenarios where we are hosting 100's of regions per machine, having 100 timer threads would be a very naive approach and would not scale well. Therefore, the implementation provides concept of AggregateTimer and ConstituentTimer. The Aggregate timer is a single threaded class multiplexing all the ConstitutentTimers present in the process. It uses the schedule() of the Executor service to schedule the ConsituentTimer to fire based on the requested deadline. It exposes the apis like createTimer, cancel, submitNewTimerEvent to provide the ability create/cancel/submit new timers.