Kafka replication detailed design V2

Differences between V1 and V2

This detailed design differs from the original detailed design in the following areas -

1. This design aims to remove split-brain and herd effect issues in the V1 design. A partition has only one brain (on the leader) and all brokers only respond to state changes that are meant for them (as decided by the leader).
2. The state machine in this design is completely controlled only by the leader for each partition. Each follower changes its state only based on such a request from the leader for a particular partition. Leader co-ordinated state machine allows central state machine verification and allows it to fail fast.
3. This design introduces an epoch or generation id per partition, which is a non-decreasing value for a partition. The epoch increments when the leader for a partition changes.
4. This design handles delete partition or delete topic state changes for dead brokers by queuing up state change requests for a broker in Zookeeper.
5. This design scales better wrt to number of ZK watches, since it registers fewer watches compared to V1. The motivation is to be able to reduce the load on ZK when the Kafka cluster grows to thousands of partitions. For example, if we have a cluster of 3 brokers hosting 1000 topics with 3 partitions each, the V1 design requires registering 15000 watches. The V2 design requires registering 3000 watches.
6. This design ensures that leader change ZK notifications are not queued up on any other notifications and can happen instantaneously.
7. This design allows explicit monitoring of
   a. the entire lifecycle of a state change -
      i. leader, broker id 0, requested start-replica for topic foo partition 0, to broker id 1, at epoch 10
      ii. leader, broker id 0, requested start-replica for topic foo partition 0, to broker id 2, at epoch 10
      iii. follower, broker id 1, received start-replica for topic foo partition 0, from leader 0, at epoch 10
      iv. follower, broker id 2, received start-replica for topic foo partition 0, from leader 0, at epoch 10
      v. follower, broker id 1, completed start-replica for topic foo partition 0, request from leader 0, at epoch 10
      vi. follower, broker id 2, completed start-replica for topic foo partition 0, request from leader 0, at epoch 10
   b. the backup of state change requests, on slow followers

Paths stored in Zookeeper

Notation: When an element in a path is denoted [xyz], that means that the value of xyz is not fixed and there is in fact a znode for each possible value of xyz. For example /topics/[topic] would be a directory named /topics containing a directory for each topic name. An arrow -> is used to indicate the contents of a znode. For example /hello -> world would indicate a znode /hello containing the value "world". A path is persistent unless it’s marked as ephemeral.

We store the following paths in Zookeeper:

1. Stores the information of all live brokers.

```
/brokers/ids/[broker_id] --> host:port (ephemeral; created by admin)
```

2. Stores for each partition, a list of the currently assigned replicas. For each replica, we store the id of the broker to which the replica is assigned. The first replica is the preferred replica. Note that for a given partition, there is at most 1 replica on a broker. Therefore, the broker id can be used as the replica id

```
/brokers/topics/[topic]/[partition_id]/replicas --> {broker_id …}  (created by admin)
```

3. Stores the id of the replica that's the current leader of this partition

```
 /brokers/topics/[topic]/[partition_id]/leader --> broker_id (ephemeral) (created by leader)
```

4. Stores the id of the set of replicas that are in-sync with the leader

```
 /brokers/topics/[topic]/[partition_id]/ISR --> {broker_id, …} (created by leader)
```

5. This path is used when we want to reassign some partitions to a different set of brokers. For each partition to be reassigned, it stores a list of new replicas and their corresponding assigned brokers. This path is created by an administrative process and is automatically removed once the partition has been moved successfully

```
 /brokers/partitions_reassigned/[topic]/[partition_id] --> {broker_id …} (created by admin)
```

6. This path is used by the leader of a partition to enqueue state change requests to the follower replicas. The various state change requests include start replica, close replica. This path is created by the add brokers admin command. This path is only deleted by the remove brokers admin command. The purpose of making this path persistent is to cleanly handle state changes like delete topic and reassign partitions even when a broker is temporarily unavailable (for example, being bounced).

```
 /brokers/state/[broker_id] --> { state change requests ... } (created by admin)
```

Key data structures

Every broker stores a list of partitions and replicas assigned to it. The current leader of a partition further maintains 3 sets: AR, ISR, CUR and RAR, which correspond to the set of replicas that are assigned to the partition, in-sync with the leader, catching up with the leader, and being reassigned to other brokers. Normally, ISR is a subset of AR and AR = ISR + CUR. The leader of a partition maintains a commitQ and uses it to buffer all produce requests to be committed. For each replica assigned to a broker, the broker periodically stores its HW in a checkpoint file.

```
Replica {                    // a replica of a partition
  broker_id   : int
  partition   : Partition
  isLocal     : Boolean      // is this replica local to this broker
  log         : Log          // local log associated with this replica
  hw          : long         // offset of the last committed message
  leo         : long         // log end offset
}

Partition {                         //a partition in a topic
  topic            : string
  partition_id     : int
  leader           : Replica        // the leader replica of this partition
  commitQ          : Queue          // produce requests pending commit at the leader
  AR               : Set[Replica]   // replicas assigned to this partition
  ISR              : Set[Replica]   // In-sync replica set, maintained at the leader
  CUR              : Set[Replica]   // Catch-up replica set, maintained at the leader
  RAR              : Set[Replica]   // Reassigned replica set, maintained at the leader
}
```

Key algorithms

Zookeeper listeners ONLY on the leader

1. Partition-reassigned listener:
    a. child change on /brokers/partitions_reassigned
    b. child change on /brokers/partitions_reassigned/[topic]

Zookeeper listeners on all brokers

1. a. Leader-change listener: value change on /brokers/topics/[topic]/[partition_id]/leader
    b. State-change listener: child change on /brokers/state/[broker_id]

Configuration parameters

1. a. LeaderElectionWaitTime: controls the maximum amount of time that we wait during leader election.
    b. KeepInSyncTime: controls the maximum amount of time that a leader waits before dropping a follower from the in-sync replica set.

Broker startup

Each time a broker starts up, it calls brokerStartup() and the algorithms are described below

```
brokerStartup()
{
  create /brokers/state/[broker_id] path if it doesn’t already exist
  register the state change handler to listen on child change ZK notifications on /brokers/state/[broker_id]
  register session expiration listener
  drain the state change queue
  get replica info from ZK and compute AR, a list of replicas assigned to this broker
  for each r in AR
  {
      subscribe to leader changes for the r’s partition
      startReplica(r)
  }
  // broker startup procedure is complete. Register is broker id in ZK to announce the availability of this
broker
  register its broker_id in /brokers/ids/[broker_id] in ZK
}
```

Leader election

```
leaderElection(r: Replica)
  read the current ISR and AR for r.partition.partition_id from ZK
  if( (r in AR) && (ISR is empty || r in ISR) )
  {
     wait for PreferredReplicaTime if r is not the preferred replica
     if(successfully write r as the current leader of r.partition in ZK)
       becomeLeader(r, ISR, CUR)
     else
       becomeFollower(r)
  }
}
```

State change events

On every broker

Leader change

This leader change listener is registered on every broker hosting a partition p. Each time it is triggered, the following procedure is executed -

```
onLeaderChange()
{
   if(broker_id is registered under /brokers/topics/[topic]/[partition_id]/replicas)
     leaderElection(r)
}
```

On State change

Each broker has a ZK path that it listens to for state change requests from the leader

```
stateChangeListener() {
// listens to state change requests issued by the leader and acts on those

    drain the state change queue
    read next state change request
    Let r be the replica that the state change request is sent for.
    // this should not happen
    Throw an error if r is not hosted on this broker
    Let requestEpoch be the epoch of the state change request
    if(closeReplicaRequest)
    {
       // we don’t need to check epoch here to be able to handle delete topic/delete partition for dead brokers.
       closeReplica(r)
    }
    if(startReplicaRequest)
    {
       Let latestPartitionEpoch be the latest epoch for this partition, got by reading /brokers/topics/[topic]/
[partition]/ISR
       if(leader for r.partition doesn’t exist) {
           // this can only happen for new topics or new partitions for existing topics
         startReplica(r)
       }else if(requestEpoch == latestPartitionEpoch) {
         // this is to ensure that if a follower is slow, and reads a state change request queued up by a
previous leader, it ignores the request
            startReplica(r)
       }
    }
}
```

On the leader

On reassignment of partitions

Each time a partition reassigned event is triggered on the leader, it calls onPartitionReassigned()

```
onPartitionsReassigned()
{
  if(this broker is the leader for [partition_id])
  {
     p.RAR = the new replicas from /brokers/partitions_reassigned/[topic]/[partition_id]
     AR = /brokers/topics/[topic]/[partition_id]/replicas
     newReplicas = p.RAR - AR
     for(newReplica <- newReplicas)
       sendStateChange(“start-replica”, newReplica.broker_id, epoch)
     if(p.RAR is empty)
     {
        for(assignedReplica <- AR)
           sendStateChange("close-replica", assignedReplica.broker_id, epoch)
     }
  }
}
```

State change communication

The leader uses this API to communicate a state change request to the followers

```
sendStateChange(stateChange, followerBrokerId, leaderEpoch)
{
   stateChangeQ = new StateChangeQueue("/brokers/state/followerBrokerId")
   stateChangeRequest = new StateChangeRequest(stateChange, leaderEpoch)
   // check if the state change Q is full. This can happen if a broker is offline for a long time
   if(stateChangeQ.isFull) {
      // this operation retains only one close-replica request for a partition, the one with the latest epoch.
This is to ensure that an offline broker, on startup, will delete old topics and partitions, which it hosted
before going offline. You don't have to retain any start-replica requests for a partition
      stateChangeQ.shrink
      // if the queue is still full, log an error
      throw new FollowerStateChangeQueueFull
   }
   stateChangeQ.put(stateChangeRequest)
}
```

State change operations

Start replica

This state change is requested by the leader or the admin command for a new replica assignment

```
startReplica(r: Replica) {
  if(broker_id not in /brokers/topics/[r.topic]/[r.partition]/replicas)
    throw NotReplicaForPartitionException()
  if( r's log is not already started) {
     do local recovery of r's log
     r.hw = min(last checkpointed HW for r, r.leo)
     register a leader-change listener on partition r.partition.partition_id
  }
   if( a leader does not exist for partition r.partition.partition_id in ZK)
      leaderElection(r)
   else {
      //this broker is not the leader, then it is a follower since it is in the AR list for this partition
      if(this broker is not already the follower of the current leader)
         becomeFollower(r)
   }
}
```

Close replica

This state change is requested by the leader when a topic or partition is deleted or moved to another broker

```
closeReplica(r: Replica)
{
   stop the fetcher associated with r, if one exists
   close and delete r
}
```

Become follower

This state change is requested by the leader when the leader for a replica changes

```
becomeFollower(r: Replica)
{
  // this is required if this replica was the last leader
  stop the commit thread, if any
  stop the current ReplicaFetcherThread, if any
  truncate the log to r.hw
  start a new ReplicaFetcherThread to the current leader of r, from offset r.leo
  start HW checkpoint thread for r
}
```

Become leader

This state change is done by the new leader

```
becomeLeader(r: Replica, ISR: Set[Replica], AR: Set[Replica])
{
   // get a new epoch value and write it to the leader path
   epoch = getNewEpoch()
   /brokers/topics/[r.partition.topic]/[r.partition.pid]/leader=broker_id, epoch
   /brokers/topics/[r.partition.topic]/[r.partition.pid]/ISR=ISR;epoch
   stop HW checkpoint thread for r
   r.hw = r.leo // TODO: check if this should actually be r.hw = last checkpointed HW for r
   wait until every live replica in AR catches up (i.e. its leo == r.hw) or a KeepInSyncTime has passed
   r.partition.ISR = the current set of replicas in sync with r
   r.partition.CUR = AR - ISR
   write r.partition.ISR in ZK
   r.partition.RAR = replicas in /brokers/partitions_reassigned/[topic]/[partition_id] in ZK
   r.partition.leader = r                  // this enables reads/writes to this partition on this broker
   start a commit thread on r.partition
   start HW checkpoint thread for r
}
```

Admin commands

This section describes the algorithms for various admin commands like create/delete topic, add/remove partition.

Create topic

The admin commands does the following while creating a new topic

```
createTopic(topic, numPartitions, replicationFactor, replicaAssignmentStr)
{
   if(!cleanFailedTopicCreationAttempt(topic))
   {
      error("Topic topic exists with live partitions")
      exit
   }
   if(replicaAssignmentStr == "") {
     // assignReplicas will always assign partitions only to online brokers
     replicaAssignment = assignReplicas(topic, numPartitions, replicationFactor)
   }

   // create topic path in ZK
   create /brokers/topics/topic
   for(partition <- replicaAssignment) {
     addPartition(topic, partition.id, partition.replicas)
   }
   // report successfully started partitions for this topic
}
waitTillStateChangeRequestConsumed(partition.replicas, timeout)
{
   register watch on state change path for each replica
   In the listener, use a condition variable to await(timeout). If it doesn't fire return false, else return
true
}

cleanFailedTopicCreationAttempts(topic)
{
   topicsForPartitionsReassignment = ls /brokers/partitions_reassigned
   for(topic <- topicsForPartitionsReassignment)
   {
      partitionsCreated = ls /brokers/partitions_reassigned/topic
      cleanupFailed = false
      for(partition <- partitionsCreated)
      {
          if(/brokers/topics/topic/partition/replicas path exists)
          {
             delete /brokers/partitions_reassigned/topic/partition
             error("Cannot cleanup. Topic exists with live partition")
             cleanupFailed = true
          }
      }
      if(cleanupFailed) {
        if(/brokers/partitions_reassigned/topic has no children)
           delete /brokers/partitions_reassigned/topic
        return false
      }
      // partition paths can be safely deleted
      for(partition <- partitionsCreated)
      {
         read the /brokers/partitions_reassigned/topic/partition path
         for each broker listed in the above step, sendStateChange("close-replica", [broker_id], -1)
         delete /brokers/topics/topic/partitions/partition
         delete /brokers/partitions_reassigned/topic/partition
      }
   }
   if(/brokers/topics/topic has no children)
     delete /brokers/topics/topic
}
```

Delete topic

```
deleteTopic(topic)
{
   partitionsForTopic = ls /brokers/topics/topic
   for(partition <- partitionsForTopic) {
      if(!deletePartition(topic, partition))
      {
         error("Failed to delete partition for topic")
         exit
      }
   }
   // delete topic path in ZK
   delete /brokers/topics/topic
}
```

Add partition to existing topic

```
addPartition(topic, partition, replicas)
{
     // write the partitions reassigned path for this create topic command
     /brokers/partitions_reassigned/topic/partition=replicas
        // start replicas for this new partition
     for(replica <- replicas)
        sendStateChange("start-replica", replica.brokerId, -1)
     // wait till state change request is consumed by all replicas
     if(!waitTillStateChangeRequestConsumed(partition.replicas, timeout))
     {
        error("Failed to create topic partition partitionId for timeout ms")
        exit
     }
     // create partition paths in ZK
     /brokers/topics/topic/partitionId/replicas=replicas
     delete /brokers/partitions_reassigned/topic/partitionId
}
```

Remove partition for existing topic

```
deletePartition(topic, partition)
{
   // empty list for partition reassignment means delete partition
   /brokers/partitions_reassigned/topic/partition=""
   // wait till replica is closed by all replicas
   if(!waitTillStateChangeRequestConsumed(partition.replicas, timeout))
   {
     error("Failed to delete topic after timeout ms")
     return false
   }
   // create partition paths in ZK
   delete /brokers/topics/topic/partitionId
   delete /brokers/partitions_reassigned/topic/partitionId
}
```

Handling produce requests

Produce request handler on the leader

```
produceRequestHandler(pr: ProduceRequest)
{
   if( the request partition pr.partition doesn't have leader replica on this broker)
      throw NotLeaderException
   log = r.partition.leader.log
   append pr.messages to log
   pr.offset = log.LEO
   add pr to pr.partition.commitQ
}
```

Message replication

Commit thread on the leader

```
while(true) {
  pr = commitQ.dequeue
  canCommit = false
  while(!canCommit) {
    canCommit = true
    for each r in ISR
       if(!offsetReached(r, pr.offset)) {
        canCommit = false
        break
       }
   if(!canCommit) {
      p.CUR.add(r)
      p.ISR.delete(r)
      write p.ISR to ZK
   }
  }
  for each c in CUR
      if(c.leo >= pr.offset) {
        p.ISR.add(c); p.CUR.delete(c); write p.ISR to ZK
      }
      checkReassignedReplicas(pr, p.RAR, p.ISR)
      checkLoadBalancing()
       r.hw = pr.offset         // increment the HW to indicate that pr is committed
      send ACK to the client that pr is committed
}

offsetReached(r: Replica, offset: Long) {
   if(r.leo becomes equal or larger than offset within KeepInSyncTime)  return true
   return false
}

checkLoadBalancing() { // see if we need to switch the leader to the preferred replica
   if(leader replica is not the preferred one & the preferred replica is in ISR) {
       delete /brokers/topics/[topic]/[partition_id]/leader  in ZK
       stop this commit thread
       stop the HW checkpoint thread
   }
 }

checkReassignedReplicas(pr: ProduceRequest, RAR: Set[Replica], ISR: Set[Replica])

{
    // see if all reassigned replicas have fully caught up and older replicas have stopped fetching, if so,
switch to those replicas

    // optimization, do the check periodically

    If (every replica in RAR has its leo >= pr.offset) {
       if(!sentCloseReplica.get) {
                oldReplicas = AR - RAR
```

```
          for(oldReplica <- oldReplicas) {
             if(r.broker_id != broker_id)
               sendStateChange("close-replica", oldReplica.broker_id, epoch)
          }
          sentCloseReplica.set(true)
       }else {
         // close replica is already sent. Wait until the replicas are closed or probably timeout and raise
error
          if(broker_id is in (AR - RAR) && (other replicas in (AR - RAR) are not in ISR anymore)) {
            // leader is not in the reassigned replicas list
             completePartitionReassignment(RAR, ISR, AR, true)
                sentCloseReplica.set(false)
          }
                else if(every replica in (AR-RAR) is not in ISR anymore) {
             completePartitionReassignment(RAR, ISR, AR, false)
                sentCloseReplica.set(false)
             }
       }

}

completePartitionsReassignment(RAR: Set[Replica], ISR: Set[Replica], AR: Set[Replica], stopCommitThread:
Boolean)
{
    //newly assigned replicas are in-sync, switch over to the new replicas
    //need (RAR + ISR) in case we fail right after here

    write (RAR + ISR) as the new ISR in ZK
    update /brokers/topics/[topic]/[partition_id]/replicas in ZK with the new replicas in RAR

    if(stopCommitThread || (broker_id is not preferred replica))
    {
      if(this broker_id is not in the new AR)
         sendStateChange("close-replica", broker_id, epoch)
      delete /brokers/partitions_reassigned/[topic]/[partition_id] in ZK
      //triggers leader election
      delete /brokers/topics/[topic]/[partition_id]/leader in ZK
      stop this commit thread
    }
}
```

Follower fetching from leader

A follower keeps sending ReplicaFetcherRequests to the leader. The process at the leader and the follower are described below -

```
ReplicaFetchReqeust {
   topic: String
   partition_id: Int
   replica_id: Int
   offset: Long
}

ReplicaFetchResponse {
   hw: Long                       // the offset of the last message committed at the leader
   messages: MessageSet        // fetched messages
}
```

At the leader

```
replicaFetch (f: ReplicaFetchRequest) {        // handler for ReplicaFetchRequest at leader
  leader = getLeaderReplica(f.topic, f.partition_id)

  if(leader == null) throw NotLeaderException
   response = new ReplicaFetcherResponse
   getReplica(f.topic, f.partition_id, f.replica_id).leo = f.offset
    response.messages = fetch messages starting from f.offset from leader.log
    response.hw = leader.hw
    send response back
}
```

At the follower

ReplicaFetcherThread for Replica r:

```
while(true) {
   send ReplicaFetchRequest to leader and get response:ReplicaFetcherResponse back
   append response.messages to r's log
   r.hw = response.hw
   advance offset in ReplicaFetchRequest
}
```