kafka Detailed Replication Design V3

The following is a draft design that uses a controller for leader election and other admin related tasks.

Major changes compared with the v2 proposal.

- Leadership changes are now made by a controller.
- The controller detects broker failures and elects a new leader for each affected partition.
- Each leadership change is communicated by the controller to each affected broker.
- The communication between the controller and the broker is done through direct RPC, instead of via Zookeeper.

Overview:

One of the brokers is elected as the controller for the whole cluster. It will be responsible for:

1. Leadership change of a partition (each leader can independently update ISR)
2. New topics; deleted topics
3. Replica re-assignment

After the controller makes a decision, it publishes the decision permanently in ZK and also sends the new decisions to affected brokers through direct RPC. The published decisions are the source of truth and they are used by clients for request routing and by each broker during startup to recover its state. After the broker is started, it picks up new decisions made by the controller through RPC.

Potential benefits:

1. Easier debugging since leadership changes are made in a central place.
2. ZK reads/writes needed for leadership changes can be batched (also easier to exploit ZK multi) and thus reduce end-to-end latency during failover.
3. Fewer ZK watchers.
4. More efficient communication of state changes by using direct RPC, instead of via a queue implementation in Zookeeper.

Potential downside:

1. Need controller failover.

Paths:

1. **Controller path**: stores the current controller info.

```
/controller --> {brokerid} (ephemeral; created by controller)
```

2. **Broker path**: stores the information of all live brokers.

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

3. **Topic path**: stores the replication assignment for all partitions in a topic. 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] --> {part1: [broker1, broker2], part2:
[broker2, broker3] ...}  (created by admin)
```

4. **LeaderAndISR path**: stores leader and ISR of a partition

```
 /brokers/topics/[topic]/[partition_id]/leaderAndISR -->
{leader_epoc: epoc, leader: broker_id, ISR: {broker1, broker2}}

 This path is updated by the controller or the current leader. The
current leader only updates the ISR part.
 Updating the path requires synchronization using conditional updates
to Zookeeper.
```

5. **PartitionReassignment path**: 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

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

6. **PartitionAdd and PartitionRemove path**: The paths are used when we want to add/remove partitions to an existing topic. For each partition to be added, the PartitionAdd path stores a list of new replicas and their corresponding assigned brokers. The paths are created by an administrative process and are automatically removed by the controller once the change is complete.

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

Terminologies:
AR: assigned replicas, ISR: in-sync replicas

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

Partition {                                //a partition in a topic
  topic                   : string
  partition_id            : int
  leader                  : Replica        // the leader replica of this
partition
  ISR                     : Set[Replica] // In-sync replica set,
maintained at the leader
  AR                      : Set[Replica] // All replicas assigned for this
partition
  LeaderAndISRVersionInZK : long           // version id of the LeaderAndISR
```

```
path; used for conditionally update the LeaderAndISR path in ZK
}



LeaderAndISRRequest {
  request_type_id         : int16 // the request id
  version_id              : int16 // the version of this request
  client_id               : int32 // this can be the broker id of the
controller
  ack_timeout             : int32 // the time in ms to wait for a response
  isInit                  : byte  // whether this is the first command
issued by a controller
  leaderAndISRMap         : Map[(topic: String, partitionId: int32) =>
LeaderAndISR) // a map of LeaderAndISR
}


LeaderAndISR {
  leader                  : int32          // broker id of the leader
  leaderEpoc              : int32          // leader epoc, incremented on
each leadership change
  ISR                     : Set[int32]     // a set of the id of each
broker in ISR
  zkVersion               : int64          // version of the LeaderAndISR
path in ZK
}


LeaderAndISRResponse {
  version_id              : int16 // the version of this request
  responseMap             : Map[(topic: String, partitionId: int32) =>
int16) // a map of error code
}

StopReplicaRequest {
  request_type_id         : int16 // the request id
  version_id              : int16 // the version of this request
  client_id               : int32 // this can be the broker id of the
controller
  ack_timeout             : int32 // the time in ms to wait for a response
  stopReplicaSet          : Set[(topic: String, partitionId: int)) // a
set of partitions to be stopped
}


StopReplicaResponse {
  version_id              : int16 // the version of this request
  responseMap             : Map[(topic: String, partitionId: int32) =>
int16) // a map of error code
}
```

A. Failover during broker failure.

Controller watches child changes of /brokers/ids path. When the watcher gets triggered, it calls on_broker_change().

```
on_broker_change():
1. Get the current live brokers from BrokerPath in ZK
2. Determine set_p, a set of partitions whose leader is no longer live or
whose ISR will change because a broker is down.
3. For each partition P in set_p
3.1 Read the current ISR of P from LeaderAndISR path in ZK
3.2 Determine the new leader and the new ISR of P:
If ISR has at least 1 broker in the live broker list, select one of those
brokers as the new leader. The new ISR includes all brokers in the current
ISR that are alive.
Otherwise, select one of the live brokers in AR as the new leader and set
that broker as the new ISR (potential data loss in this case).
Finally, if none of the brokers in AR is alive, set the new leader to -1.
3.3 Write the new leader, ISR and a new epoc (increase current epoc by 1)
in /brokers/topics/[topic]/[partition_id|partition_id]/leaderAndISR.
This write has to be done conditionally. If the version of the
LeaderAndISR path has changed btw 1.1 and 1.3, go back to 1.1.
4. Send a LeaderAndISRCommand (containers the new leader/ISR and the ZK
version of the LeaderAndISR path) for each partition in set_p to the
affected brokers.
For efficiency, we can put multiple commands in one RPC request.
(Ideally we want to use ZK multi to do the reads and writes in step 3.1
and 3.3.)
```

B. Creating/deleting topics.

The controller watches child change of /brokers/topics. When the watcher gets triggered, it calls on_topic_change().

```
on_topic_change():
The controller keeps in memory a list of existing topics.
1. If a new topic is created, read the TopicPath in ZK to get topic's
replica assignment.
1.1. call init_leaders() on all newly created partitions.
2. If a topic is deleted, send the StopReplicaCommand to all affected
brokers.

init_leaders(set_p):
Input: set_p, a set of partitions
0. Read the current live broker list from the BrokerPath in ZK
1. For each partition P in set_p
1.1 Select one of the live brokers in AR as the new leader and set all
live brokers in AR as the new ISR.
1.2 Write the new leader and ISR in /brokers/topics/[topic]/
[partition_id|partition_id]/leaderAndISR
2. Send the LeaderAndISRCommand to the affected brokers. Again, for
efficiency, the controller can send multiple commands in 1 RPC.
```

C. Broker acts on commands from the controller.

Each broker listens to commands from the controller through RPC.

```
For LeaderAndISRCommand: it calls on_LeaderAndISRCommand().
on_LeaderAndISRCommand(command):
1. Read the set of partitions set_P from command.
2. For each partition P in set_p
2.0 if P doesn't exist locally, call startReplica()
2.1 If the command asks this broker to be the new leader for P and this
broker is not already the leader for P,
2.1.1 call becomeLeader()
2.2 If the command asks this broker to following a leader L and the broker
is not already following L
2.2.1 call becomeFollower()
3. If the command has a flag INIT, delete all local partitions not in
set_p.

becomeLeader(r: Replica, command) {
stop the ReplicaFetcherThread to the old leader   //after this, no more
messages from the old leader can be appended to r  r.leaderAndISRZKVersion
= command.leaderAndISRZKVersion
r.partition.ISR = command.ISR
r.isLeader = true                                 //enables reads/writes
to this partition on this broker
r.partition.LeaderAndISRVersionInZK = command.LeaderAndISRVersionInZK
start a commit thread on r
}
Note that the new leader's HW could be behind the HW of the previous
leader. Therefore, immediately after the leadership transition,
```

```
it is possible for a consumer (client) to ask for an offset larger than
the new leader's HW. To handle this case, if the consumer is
asking for an offset between the leader's HW and LEO, the broker could
just return an empty set. If the requested offset is larger
than LEO, the broker would still return OffsetOutofRangeException.

becomeFollower(r: Replica) {
stop the ReplicaFetcherThread to the old leader  r.isLeader =
false                                  //disables reads/writes to this
partition on this broker
stop the commit thread, if any
truncate the log to r.hw
start a new ReplicaFetcherThread to the current leader of r, from offset r.
leo
}

startReplica(r: Replica) {
create the partition directory locally, if not present
start the HW checkpoint thread for r
}



For StopReplicaCommand: it calls on_StopReplicaCommand().
on_StopReplicaCommand(command):
1. Read the list of partitions from command.
2. For each such partition P
2.1 call stopReplica() on p

stopReplica(r: Replica) {
  stop the ReplicaFetcherThread associated with r, if any.
  stop the HW checkpoint thread for r
  delete the partition directory locally, if present
}
```

D. Handling controller failure.

Each broker sets an exists watcher on the ControllerPath. When the watcher gets triggered, it calls on_controller_failover(). Basically, the controller needs to inform all brokers the current states stored in ZK (since some state change commands could be lost during the controller failover).

```
on_controller_failover():
1. create /controller -> {this broker id)
2. if not successful, return
3. read the LeaderAndISR path from ZK for each partition
4. send a LeaderAndISR command (with a special flag INIT) for each
partition to relevant brokers. Those commands can be sent in 1 RPC request.
5. call on_broker_change()
6. for the list of partitions without a leader, call init_leaders().
7. call on_partitions_reassigned()

8. call on_partitions_add()
9. call on_partitons_remove()
```

E. Broker startup.

When a broker starts up, it calls on_broker_startup(). Basically, the broker needs to first read the current state of each partition from ZK.

```
on_broker_startup():
1. read the replica assignment of all topics from the TopicPath in ZK
2. read the leader and the ISR of each partition assigned to this broker
from the LeaderAndISR path in ZK
3. for each replica assigned to this broker
3.1 start replica
3.2 if this broker is a leader of this partition, become leader.
(shouldn't happen in general)
3.3 if this broker is a follower of this partition, become follower.
4. Delete local partitions no longer assigned to this broker (partitions
deleted while the broker is down).
5. subscribes to changes in ZKQueue for this broker.
```

F. Replica reassignment:

Controller watches child changes in the PartitionReassignmentPath in ZK. The value of this path contains RAR, the set of brokers that a particular partition will be reassigned to. When the watcher gets triggered, it calls on_partitions_reassigned().

```
on_partitions_reassigned():
1. get the set of reassigned partitions (set_p) from the
PartitionReassignment path in ZK
2. call add_reassigned_partitions(set_p) in Reassginment Agent
```

Reassignment Agent: wakes up if notified or a certain amount of time has passed (e.g., 30 secs); once it wakes up, it calls on_wakeup().

```
Reassignment Agent maintains an in memory map reassigned_partition_map
that tracks all partitions to be reassigned.

add_reassigned_partitions(set_p)
1. for each partition p in set_p
1.1  if (p not in reassigned_partition_map)
1.1.1  set p's state to INIT and add p to reassigned_partiton_map
1.1.2  wake up the reassignment agent

on_wakeup()
1. for each partition p in reassigned_partiton_map
2.   if partition p in INIT state
2.1    issue LeaderAndISR command to the brokers in RAR (to start
bootstrapping new replicas)
2.2    mark partition p as in IN_PROGRESS state
3.   else if partition p in IN_PROGRESS state
3.1    read the ISR from the LeaderAndISR path in ZK
3.2    if ISR == AR+RAR // this means that all replicas have caught up and
we want to switch the current ISR and AR to RAR
3.2.1    get a controller lock
3.2.2    conditionally update ISR in the LeaderAndISR path in ZK (if the
version of the path has changed btw 3.1 and 3.2.2, go back to 3.1)
3.2.3    send a LeaderAndISR command to inform the leader of the new ISR
3.2.4    send a StopReplica command to the brokers in the current AR (but
not in RAR).
3.2.5    update /brokers/topics/[topic] to change AR to RAR
3.2.6    delete /brokers/partitions_reassigned/[topic-part]
3.2.7    release controller lock
```

G. Follower fetching from leader and leader advances HW

A follower keeps sending ReplicaFetcherRequests to the leader. A leader advances its HW when every replica in ISR has received messages up to that point. 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, on receiving a ReplicaFetchRequest, it calls
on_ReplicaFetchRequest
```

```
on_ReplicaFetchRequest(f: ReplicaFetchRequest) {
  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

  call maybe_change_ISR()                              // see if we need to
shrink or expand ISR
  leader.hw = min of leo of every replica in ISR     // try to advance HW
  // trigger potential acks to produce requests

  response.messages = fetch messages starting from f.offset from leader.log
  response.hw = leader.hw
  send response back
}

maybe_change_ISR() {
  // If a follower is slow or is not active at all, the leader will want
to take it out of ISR so that it can commit messages
  // with fewer replicas.
  find the smallest leo (leo_min) of every replica in ISR
  if ( leader.leo - leo_min > MAX_BYTES_LAG || the replica with leo_min
hasn't updated leo for more than MAX_TIME_LAG)
    newISR = ISR - replica with leo_min

  // If a follower has fully caught up, the leader will want to add it to
ISR.
  for each replica r not in ISR
    if ( r.leo - leo_min < MIN_BYTES_LAG)
      newISR = ISR + r

  update the LeaderAndISR path in ZK with newISR
  the update is conditional and only succeeds if the version of the
LeaderAndISR path in ZK is the same as leader.partition.
LeaderAndISRVersionInZK
  if (update in ZK successful) {
   leader.partition.LeaderAndISRVersionInZK = new version of the
LeaderAndISR path in ZK
   leader.partition.ISR = new ISR
  }
}
??? We need a way to call maybe_change_ISR() when a follower stopped
fetching for more than MAX_TIME_LAG. One way to do that is to register a
new
DelayedRequest each time a new ProduceRequest comes in. The DelayedRequest
will timeout after MAX_TIME_LAG and is cleared if leader.hw has moved
beyond the offset of the message in the ProduceRequest before the timeout.
Will this cause too much overhead?

At the follower: ReplicaFetcherThread for Replica r
run() {
  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
    }
  }
```

H. Add/remove partitions to an existing topic:

Controller watches child changes in the PartitionAdd and the PartitionRemove Path in ZK. When the watcher gets triggered, it calls on_partitions_add() and on_partition_remove(), respectively.

```
on_partitions_add():
1. determine the set of partitions to be added (set_p)
2. call init_leaders(set_p)
3. delete the PartitionAdd path in ZK
4. update the Topic path in ZK with the added partitions

on_partitions_remove():
1. determine the set of partitions to be deleted
2. send a StopReplicaCommand for each partition to the affected brokers
3. delete the PartitionRemove path in ZK
4. update the Topic path in ZK with the removed partitions
```

Discussions:

1. End-to-end latency during a broker failure:

1. broker shutdown (after closing socket server, need to close request handler, close log)
2. broker watcher gets triggered in controller
3. make leadership change and publish the new leader/ISR in ZK (1 ZK write per affected partition)
4. inform the leadership change to each broker by write to ZKQueue (1 ZK write per broker)
5. leader waits for followers in ISR to connect (Kafka PRC)
6. follower truncates its log first (a potential I/O) and then starts fetching from leader

In the critical path, the most time consuming operation is step 3 where we need to write 1 ZK path per partition. Assuming that during a broker failover we need to change leader for 10K partitions and each ZK write takes 4ms, this could take 40 secs. One possibility is to use the multi() support in ZK 3.4 to batch those writes in 1 ZK operation.

2. ZKQueue vs direct RPC:

Communicating between the controller and the brokers via ZK is not efficient. Each communication requires 2 ZK writes (each costs roughly 2 RPC), 1 watcher firing and 1 ZK read. These add up to roughly 6 RPCs per communication. An alternative is to implement an admin RPC in the broker for direct communication between the controller and the brokers. Then each communication costs only 1 RPC. The admin RPC could specify a timeout, during which it expects the admin command to be completed. Using RPC means that when a broker is down, it could miss some commands from the controller. This proposal requires that the broker recover its state during startup by reading state information stored in ZK.

3. Dealing with multiple leaders in transition:

Occasionally, it's possible for multiple brokers to simultaneous assume that they are the leader of a partition. For example, broker A is the initial leader of a partition and the ISR of that partition is {A,B,C}.. Then, broker A goes into GC and losses its ZK registration. The controller assumes that broker A is dead, assigns the leader of the partition to broker B and sets the new ISR in ZK to {B,C}. Broker B becomes the leader and at the same time, Broker A wakes up from GC but hasn't acted on the leadership change command sent by the controller. Now, both broker A and B think they are the leader. It would be bad if we allow both broker A and B to commit new messages since the data among replicas will be out of sync. Our current design actually will prevent this from happening in this situation. Here is why. The claim is that after broker B becomes the new leader, broker A can no longer commit new messages any more. For broker A to commit a message m, it needs every replica in ISR to receive m. At the moment, broker A still thinks the ISR is {A,B,C} (its local copy; although the ISR in ZK has changed). Broker B will never receive

message m. This is because by becoming the new leader, it must have first stopped fetching data from the previous leader. Therefore broker A can't commit message m without shrinking the ISR first. In order to shrink ISR, broker A has to write the new ISR in ZK. However, it can't do that because it will realize that the LeaderAndISR path in ZK is not on a version that it assumes to be (since it has already been changed by the controller). At this moment, broker A will realize that it's no longer the leader any more.Question A.3, is broker down the only failure scenario that we worry about? Do we worry about leader failure at individual partition level?

4. Is broker down the only failure scenario that we worry about? Do we worry about leader failure at individual partition level?

It seems that common problems such as long GC, I/O on local storage will affect the whole broker.

5. How to deal with repeated topic deletion/creation?

A broker can be down for a long time during which a topic can be deleted and recreated. Do we need partition version id in order to clean up an outdated partition from local storage? The way our replica assignment works is that we only assign replicas to live brokers. This means that while a broker is down, no new replicas will be assigned to it. So, it seems that we won't get into the situation that when a failed broker comes back, a partition is still assigned to this broker and yet the partition stored locally is outdated. So, we don't really need partition version id to address this issue. On broker startup, the broker will read all local partition directories, delete each directory whose partition is no longer assigned to itself, and then load the last segment of each of the remaining directories.

6. What happens to client routing during a broker failover?

In this proposal, the controller first publishes the new leader for affected partitions in the LeaderAndISR path in ZK, then sends the leadership change commands to the brokers. The brokers then act on those leadership change commands. Since we use the LeaderAndISR path to route the client request, there is a window (potentially small) that a client is routed to a broker that's the new leader, but the broker is not ready to be the new leader yet.

For reference, HBase only updates the metadata (for client routing) after the regionserver responds to close/open region commands. So, one would think that instead of the controller directly updating the LeaderAndISR path, we can let each broker update that path after it completes the execution of the command. There is actually a critical reason that the leaderAndISR path has to be updated by the controller. This is because we rely on the leaderAndISR path in ZK to synchronize between the controller and the leader of a partition. After the controller makes a leadership change decision, it doesn't want the ISR to be changed by the current leader any more. Otherwise, the newly elected leader could be taken out of ISR by the current leader before the new leader takes over the leadership. By publishing the new leader immediately in the leaderAndISR path, the controller prevents the current leader from updating the ISR any more.
One possibility is to use another ZK path ExternalView for client routing. The controller only updates ExternalView after the broker responds positively for the leadership change command. There is a tradeoff between using 1 ExternalView path for all partitions or 1 ExternalView path per partition. The former has less ZK overhead, but potentially forces unnecessary rebalancing on the consumers.Aonther way to think about this is that in the normal case, leadership change commands are executed very quickly. So we probably can just rely on client side retry logic to handle the transition period. In the uncommon case that somehow it takes too long for a broker to become a leader (likely due to a bug), the controller will get a timeout and can trigger an alert so that an admin can take a look at it. So, we probably can start without the ExternalView path and reconsider it if it's really needed.

7. Dealing with offsets beyond HW in fetch requests during leadership change:

In general, the HW in the follower always lags that in the leader. So, during a leadership change, a consumer client could be requesting an offset between the new leader's HW and LEO. Normally, the server will return an OffsetOutOfRangeException to the client. In this particular case, the client request is actually valid. To deal with this case, the server can return an empty message set to the consumer if the requested offset is between HW and LEO.

8. Can follower keep up with the leader?

In general, we need to have as much I/O parallelism in the follower as in the leader. Probably need to think a bit more on this.

Potential optimizations:

1. Communications between the controller and the broker.

To increase parallelism, on the controller, we can have a command queue and use a pool of threads to dequeue those commands and send them to brokers. On the broker side, we let each command be processed independently. The broker maintains in-memory a leadership change epoc for each replica and only follows a leadership change command if its epoc is larger than the current epoc in the replica.