

Kafka Replication Detailed Design

The document describes key data structures and algorithms in Kafka replication.

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

Stores the information of all live brokers.

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

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.

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

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

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

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

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

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.

2. 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, $ISR \subseteq AR$ 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
}
```

3. Key Algorithms

Zookeeper listeners:

1. Leader-change listener: value change on /brokers/topics/[topic]/[partition_id]/leader
2. Replica-change listener:
 - child change on /brokers/topics (new topic registered)
 - child change on /brokers/topics/[topic] (new partition registered)
 - value change on /brokers/topics/[topic]/[partition_id]/replicas (new replica assigned)
3. Partition-reassigned listener:
 - child change on /brokers/partitions_reassigned
 - child change on /brokers/partitions_reassigned/[topic]

Configuration parameters:

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

Broker starting up: Each time a broker starts up, it calls `brokerStartup()` and the algorithms are described in Figure 1 and Figure 2.

Figure 1.

`brokerStartup()`

 register its `broker_id` in `/brokers/ids/[broker_id]` in ZK)

 get replica info from ZK and compute AR, a list of replicas assigned to this broker

 for each `r` in AR

`replicaStateChange(r)`

`replicaStateChange(r : Replica)`

 if (`r` is not already started) {

 do local log recovery of `r`'s log

`r.hw = min(last checked HW for r, r.leo)`

 register a leader-change listener on partition `r.partition.partition_id`

 }

 if (a leader exists for partition `r.partition.partition_id` in ZK)

`becomeFollower(r)`

 else

`leaderElection(r)` // in Figure 2

`becomeFollower(r: Replica)`

 stop the current `ReplicaFetcherThread`, if any

 truncate `r`'s log to `r.hw`

 start a new `ReplicaFetcherThread` to the current leader of `r`, from offset `r.leo`

Figure 2.

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 a bit if r is not the preferred replica //makes the preferred one more likely be the leader

if (successfully write r as the current leader of r.partition in ZK)

becomeLeader(r, ISR)

else

becomeFollower(r)

}

//otherwise; some other replicas will become leader

??? what happens if no replica in ISR is alive

becomeLeader(r: Replica, ISR: Set[Replica])

stop HW checkpoint thread for r

r.hw = r.leo

wait until every live replica in ISR catches up (i.e., its leo == r.hw) or a configured amount of time (LeaderElectionWaitTime) has passed

r.partition.ISR = the current set of replicas in sync with r; write r.partition.ISR in ZK

r.partition.CUR = r.AR - r.ISR

r.partition.RAR = replicas in /brokers/partitions_reassigned/[topic]/[partition_id] in ZK

(if RAR == AR, remove /brokers/partitions_reassigned/[topic]/[partition_id],

this can happen if the previous master dies before cleaning up partitions_reassigned 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

Client sending produce requests:

Each time a broker gets a produce request, it calls produce() and the algorithms are described in Figure 3 and Figure 4.

Figure 3.

```
produce(pr: ProduceRequest) //handler in the broker for each produce request
    if (the requested partition pr.partition doesn't have the leader replica on this broker)
        throw NotLeaderException
    log = pr.partition.leader.log
    append pr.messages to log; pr.offset = log.LEO
    add pr to pr.partition.commitQ

committer thread for a partition p:
while (true) {
    pr = commitQ.dequeue
    canCommit = false
    while (!canCommit) {
        canCommit = true
        for each r in p.ISR // try to wait until every replica in ISR completes the request
            if (!offsetReached(r, pr.offset)) {
                canCommit = false
                break
            }
        }
        if (!canCommit) // at least 1 replica didn't get p; remove it from ISR and commit again
            p.ISR.delete(r); p.CUR.add(r); write p.ISR to ZK
    }
    for each c in p.CUR // see if any replica in CUR has caught up; if so, add it back to ISR
        if (c.leo >= pr.offset)
            p.ISR.add(c); p.CUR.delete(c); write p.ISR to ZK
    checkReassignedReplicas(pr, p.RAR, P.ISR) // in Figure 4
    checkLoadBalancing() // in Figure 4
    acknowledge the client that pr is committed
}

offsetReached(r: Replica, offset: long) //see if r.leo reached offset within some time
    if (r.leo becomes equal or larger than offset within a configured time (KeepInSyncTime))
        return true
    return false
```

Figure 4.

```
checkReassignedReplicas(pr: ProduceRequest, RAR: Set[Replica], ISR: Set[Replica])
// see if all reassigned replicas have fully caught up, if so, switch to those replicas
// optimization, do the check periodically
If (every replica in RAR has its leo >= pr.offset) {
    //newly assigned replicas are in-sync, switch over to the new replicas
    write (RAR + ISR) as the new ISR in ZK //need (RAR + ISR) in case we fail right after here
    update /brokers/topics/[topic]/[partition_id]/replicas in ZK with the new replicas in RAR
    delete /brokers/topics/[topic]/[partition_id]/leader in ZK //triggers leader election
    delete /brokers/partitions_reassigned/[topic]/[partition_id] in ZK
    stop this commit thread
}

checkLoadBalancing() // see if we need to switch the leader to the preferred replica
// optimization, do the check periodically
if (leader replica is not the preferred one && the preferred replica is in ISR) {
    delete /brokers/topics/[topic]/[partition_id]/leader in ZK // this triggers leader election
    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 in Figure 5.

Figure 5.

```
ReplicaFetchRequest {  
    topic      : String  
    partition_id: int  
    broker_id  : int // id of the requesting replica  
    offset     : long // fetch offset, everything before offset is in the log of the requesting replica  
}
```

```
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.broker_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:

ReplicaFetcher thread 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  
}
```

On leader-change event:

Each broker registers a leader-change listener in ZK for every replica assigned to it. Everytime a leader-change event is triggered, the broker calls onLeaderChange() in Figure 6.

Figure 6.

```
onLeaderChange()
if (replica l becomes the new leader for partition p) {
    if (this replica r is not l)
        becomeFollower(r)
    // otherwise, this replica is l and has already become the leader
}
else { // otherwise, the current leader is gone
    if (r is still registered in /brokers/topics/[topic]/[partition_id]/replicas) // see if r is removed
        leaderElection(r)
}
```

On replica-change event:

Everytime a replica-change event is triggered, the broker calls onReplicaChange() in Figure 7.

Figure 7.

```
onReplicaChange()
if (a new replica r is assigned to this broker)
    replicaStateChange (r) // from Figure 1
else if (a replica r is un-assigned to this broker) {
    stop the fetcher associated with r, if exists
    close r and delete r
}
```

On partition-reassigned event:

Everytime a partition-reassigned event is triggered, the broker calls onPartitionReassigned() in Figure 8.

Figure 8.

```
onPartitionReassigned()
if (a partition p is to be reassigned and the current leader of p is on this broker)
    p.RAR = the new replicas from /brokers/partitions_reassigned/[topic]/[partition_id]
if (a new replica r is to be reassigned to this broker)
    replicaStateChange (r) // from Figure 1; this replica will become a follower
```


4. Flow of administrative operations

Adding a new topic or a partition to an existing topic:

1. The administrator process adds one or more new paths of `/brokers/topics/[topic]/[partition_id]/replicas` in ZK.
2. One or more replica-change events will be triggered and the listener starts the bootstrap process of each new replica.

Adding a new set of nodes to the cluster:

1. The administrator process determines the set of partitions to be reassigned to new brokers, and adds one or more new paths of `/brokers/partitions_reassigned/[topic]/[partition_id]` in ZK.
2. Each broker will get a partition-reassigned event. If a replica is reassigned to a broker, the broker will start bootstrapping a new replica. The current leader of the reassigned partition will add those newly assigned replicas to an RAR set and monitor if they are in-sync with the leader. Once the new replicas are in-sync, the leader exposes the new replicas in ZK and triggers a leader election.
3. After the leader election, one of the new replicas will become the new leader and the rest of the new replicas will become the followers. Each old replica will be deleted after the corresponding replica-change event is triggered.