Quorum-Journal Design

Todd Lipcon
todd@cloudera.com

October 3, 2012

Contents

1 Overview

1.1 Background

HDFS-1623 and related JIRAs added high availability support to the HDFS NameNode, but rely on a shared storage directory in which to store the shared edit log. This shared storage must be highly available and simultaneously accessible by all of the NameNodes in the cluster.

Currently, the recommended mechanism for this shared edits dir is a NAS device (aka a *filer*) mounted via NFS. This shared mount allows the Active NN to write edits while the Standby NN tails the files. Should a failover occur, it requires that a custom fencing script be deployed which is able to either (a) power off the previously active node, or (b) prevent further access to the shared mount by the previously active node.

1.2 Limitations of the current implementation

The above restrictions are satisfiable in many environments, where high availability for many different products relies on an HA filer. The fencing requirements can be implemented either via a remotely controllable power distribution unit (PDU) or by a custom protocol implemented by the NAS device (eg remote calls into NetApp's ONTAP software).

However, in some environments, this solution is unsatisfactory for the following reasons:

1. **Custom hardware** - the hardware requirements of a NAS device and remotely controllable PDU can be expensive, and also may be different than the standard deployments used elsewhere within some "filer-free" organizations.

2. **Complicated deployment** - even after HDFS is installed, the administrator must take extra steps to configure NFS mounts, custom fencing scripts, etc. This complicates HA deployment and may even cause unavailability if misconfigured.

3. **Poor NFS client implementations** - in many versions of Linux, NFS client implementations can be buggy, or easy to misconfigure. For example, it is easy for an administrator to misconfigure mount options in such a way that the NameNodes will freeze unrecoverably in some outage scenarios.

1.3 Requirements for alternative approach

1.3.1 Differentiating requirements

This design document describes an alternate approach that satisfies the following key requirements:

1. **No requirement for special hardware** - the design should function using only commodity hardware typical of already-deployed Hadoop clusters

2. **No requirement for custom fencing config** - all necessary fencing should happen in software only, built into the system

3. **No SPOFs** - as part of an HA solution, the storage for edit logs should also be fully HA.

Requirement 3 above implies that the edit logs must be stored on multiple nodes. Henceforth we will refer to these nodes as *journal replicas*.

1.3.2 Correctness requirements

Of course, we must retain the basic correctness requirements of any edit logs used for HDFS:

1. **Any synced edit must not be forgotten** - if the NameNode successfully calls FSEditLog.logSync() then all of the synced edits must be persisted and remembered forever, regardless of any failures.

2. **An unsynced edit may or may not be forgotten** - if the NameNode writes an edit and crashes either before or during `logSync()` then the system may either remember the edit or forget the edit.

3. **If an edit is read, it must not be forgotten** - if any StandbyNode is tailing the edits and sees an edit, then that edit must never be forgotten.

4. **For any given txid, there must be exactly one valid transaction** - if any node reads a transaction with a given id, then any other node which reads a transaction with the same ID must read the same data.

1.3.3 Additional goals

Additionally, the following aspects are not strict requirements but are provided by this design:

1. **Configurable for any number of failures** - if an administrator would like to tolerate more than one node failure, he or she may configure extra nodes in the system to attain the desired fault tolerance. Specifically, we should be able to handle N failures by configuring $2N + 1$ nodes.

2. **One slow journal replica should not impact latency** - if one of the nodes storing edits becomes slow or fails, the system should continue to operate with no latency penalty. When a node fails, we should not have to stall client edits for any timeout period.

3. **Adding journal replicas should not negatively impact latency** - in order to tolerate more than one failure, the administrator may configure 5 or more journal replicas. The communication to these replicas should be made in parallel such that adding more journal replicas does not result in a linear increase in latency.

1.3.4 Operational requirements

The following additional requirements are not specificly related to the algorithm used, but are important for deployment as part of a realistic HDFS cluster:

1. **Metrics/logging** - any additional daemons introduced as part of the system should integrate with the existing metrics and logging systems present in HDFS. This is necessary to leverage existing monitoring infrastructure.

2. **Configuration** - any necessary configuration should be done using the same XML-based configuration files as the rest of HDFS, so that the configuration is familiar to operators.

3. **Security** - any operations that span multiple nodes must be (a) mutually authenticated and (b) encryptable, using the same mechanisms as the rest of Hadoop. For example, any IPC/RPC traffic should use SASL-based transports with mutual authentication provided by Kerberos. Any usage of ZooKeeper should support ZooKeeper ACLs and authentication.

1.4 Quorum-based approach

This document describes the design for the *Quorum Journal Manager*, one possible solution[1] to the above problem, which satisfies all of the above requirements as well as the extra goals.

The design relies upon the concept of *quorum commits* to a cluster of daemons, termed *JournalNodes*. Each JournalNode exposes a simple RPC interface, allowing a NameNode to read and write edit logs stored on the node's local disk. When the NameNode writes an edit, it sends the edit to all JournalNodes in the cluster, and waits for a majority of the nodes to respond. Once a majority have responded with a success code, the edit may be considered committed.

The following section elaborates on the design.

2 Design - Writing logs

2.1 Components

The system depends on the following components:

1. A `QuorumJournalManager` implementation, running in each NameNode. This component implements the already-pluggable `JournalManager` interface present in HDFS. It is responsible for contacting the JournalNodes in the cluster via RPC, sending edits, performing fencing and synchronization, etc.

[1] **N.B.:** this solution is merely one possible solution and does not preclude other solutions from being developed (e.g. BookKeeper, BackupNode, etc).

2. The `JournalNode` daemon, running on N machines in the cluster. Each such daemon exposes an interface via Hadoop IPC which allows the `QuourumJournalManager` to remotely write edits to its local disk. It uses the existing `FileJournalManager` implementation to manage its local storage.

We anticipate that, in typical deployments, administrators will configure three JournalNodes. The nodes will run on the same physical hardware as (1) the NameNode, (2) the Standby NameNode, and (3) the JobTracker. These three daemons are attractive because they are already well-provisioned machines with little unpredictable user activity, and those daemons are generally light on disk usage. Dedicating a disk drive on each of the machines for use by the JournalNode should be easy in most every environment.

If a user wishes to withstand double failures, or be able to do planned maintenance at the same time as a failure, he or she may configure five or more `JournalNode`s. Given N JournalNodes, the system can tolerate $(N-1)/2$ failures.

2.2 QuorumJournalManager flow

When the QJM wishes to start writing edit logs, it performs the following operations:

1. **Fencing prior writers** - the writer must ensure that no prior writers are still writing to the edit logs. This acts as a fencing mechanism such that, even if two NameNodes believe themselves to be active, only one is able to successfully perform edits. See below for further information.

2. **Recovering in-progress logs** - if the writer that was previously writing to the logs failed, it is possible that different replicas have different lengths at the end of the log (e.g. perhaps the previous writer sent an edit to only one of three JNs before crashing). We must synchronize the logs and agree upon their length.

3. **Start a new log segment** - this is the normal flow for writing edit logs, in all existing JournalManager implementations.

4. **Write edits** - for each batch of edits to be written, the writer sends the edits to all JNs in the cluster. Once it has received a successful response from a quorum of JNs, it considers the write a success. The writer maintains a pipeline of writes to each JN such that a temporary slow-down on one node does not impact system throughput or latency.

   If a JN fails to accept edits, or responds so slowly that the queue of pending edits eclipses a configurable maximum length, then that JN will be marked as `outOfSync` and no longer used for the current log segment. So long as a quorum of nodes remains alive, this is not a problem. The previously dead node will be re-tried on the next edit log roll.

5. **Finalize log segment** - in the same manner as existing implementations, the QJM can finalize a log segment by sending an RPC to the JNs. When it receives confirmation from a quorum of JNs, the log segment is considered finalized and the next log segment may begin.

6. **Go to step 3**

In the following sections we will explain each step in further detail.

2.3 Fencing writers

In order to satisfy the fencing requirement without requiring custom hardware, we require the ability to guarantee that a previously active writer can commit no more edits after a certain point. In this design, we introduce the concept of *epoch numbers*, similar to those found in much distributed systems literature (eg Paxos, ZAB, etc). In our system, epoch numbers have the following properties:

- When a writer becomes active, it is assigned an epoch number.
- Each epoch number is *unique*. No two writers have the same epoch number.
- Epoch numbers define a total order of writers. For any two writers, epoch numbers define a relation such that one writer can be said to be strictly *later* than the other if its epoch number is higher.

We utilize the epoch numbers as follows:

- Before making any mutations to the edit logs, a QJM must first be assigned an epoch number.
- The QJM sends its epoch number to all of the JNs in a message `newEpoch(N)`. It may not proceed with this epoch number unless a quorum of JournalNodes responds with an indication of success.[2]
- When a JN responds to such a request, it persistently records this epoch number in a variable `lastPromisedEpoch` which is also written durably (`fsync`ed) to local storage.
- Any RPC that requests a mutation to the edit logs (eg `logEdits()`, `startLogSegment()`, etc. must contain the requester's epoch number.
- Before taking action in response to any RPC other than `newEpoch()`, the JournalNode checks the requester's epoch number against its `lastPromisedEpoch` variable. If the requester's epoch is lower, then it will reject the request. If the requester's epoch is higher, then it will update its own `lastPromisedEpoch`. This allows a JN to update its `lastPromisedEpoch` even if it was down at the time that the new writer became active.

This policy ensures that, once a QJM has received a successful response to its `newEpoch(N)` RPC, then no QJM with an epoch number less than N will be able to mutate edit logs on a quorum of nodes. We defer to the literature for a formal proof, but this is intuitively true since all possible quorums overlap by at least one JN. So, any future RPC made by the earlier QJM will be unable to attain any quorum which does not overlap with the quorum that responded to `newEpoch(N)`.

2.4 Writer epochs

In addition to maintaining a `lastPromisedEpoch`, each `JournalNode` also maintains a durable copy of another epoch number `lastWriterEpoch`. The `JournalNode` updates this variable immediately before starting any new log segment, such that it always tracks the epoch number of the last writer.

The importance of this will become clear when discussing edge cases of segment recovery below.

[2] In typical Paxos implementations, a writer will repeat this algorithm with some backoff in order to eventually become active. In our implementation, we assume that "fighting writers" are rare, because the failover itself has been coordinated by ZooKeeper. Thus, we do not currently implement a retry loop trying to establish an epoch.

2.5 Generating epoch numbers

In the above section, we did not explain how the QJM determines an epoch number which satisfies the required properties. Our solution to this problem borrows from ZAB and Paxos. We use the following algorithm:

1. The QJM sends a message `getJournalState()` to the JNs. Each JN responds with its current value for `lastPromisedEpoch`.
2. Upon receiving a response from a quorum of JNs, the QJM calculates the maximum value seen, and then increments it by 1. This value is the *proposedEpoch*. [3]
3. It sends a message `newEpoch(proposedEpoch)` to all of the JNs. Each JN atomically compares this proposal to its current value for `lastPromisedEpoch`. If the new proposal is greater than the stored value, then it stores the proposal as its new `lastPromisedEpoch` and returns a success code. Otherwise, it returns a failure.
4. If the QJM receives a success code from a quorum of JNs, then it sets its epoch number to the proposed epoch. Otherwise, it aborts the attempt to become the active writer by throwing an IOException. This will be handled by the NameNode in the same fashion as a failure to write to an NFS mount – if the QJM is being used as a shared edits volume, it will cause the NN to abort.

Again, we defer a formal proof to the literature. A rough explanation, however, is as follows: no two nodes can successfully complete step 4 for the same epoch number, since all possible quorums overlap by at least one node. Since no node will return success twice for the same epoch number, the overlapping node will prevent one of the two proposers from succeeding.

Following is an annotated log captured from the uncontended case:

```
40,319 INFO  QJM - Starting recovery process for unclosed journal segments...
```

First, the NN sends `getJournalState()` to 3 nodes. They each respond with epoch 0 since this trace is from a newly formatted system.

```
40,320 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:39595:
       getJournalState {jid { identifier: "test-journal" }}
40,323 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:39595:
       getJournalState {lastPromisedEpoch: 0 httpPort: 45029}
40,323 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:36212:
       getJournalState {jid { identifier: "test-journal" }}
40,325 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:36212:
       getJournalState {lastPromisedEpoch: 0 httpPort: 49574}
40,325 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:33664:
       getJournalState {jid { identifier: "test-journal" }}
40,327 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:33664:
       getJournalState {lastPromisedEpoch: 0 httpPort: 36092}
```

The NN then sends `newEpoch` to start epoch 1, which is higher than any of the responses responses.

[3] Note that *proposed* epochs are not necessarily unique. However, if multiple QJMs propose the same epoch, at most one of them will receive a successful response from a quorum of JNs.

```
40,329 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:39595:
       newEpoch {jid { identifier: "test-journal" } nsInfo { .. .} epoch: 1}
40,334 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:39595: newEpoch {}
40,335 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:36212:
       newEpoch {jid { identifier: "test-journal" } nsInfo { .. .} epoch: 1}
40,339 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:36212: newEpoch {}
40,339 TRACE Outgoing IPC) - 1: Call -> null@/127.0.0.1:33664:
       newEpoch {jid { identifier: "test-journal" } nsInfo { .. .} epoch: 1}
40,342 TRACE IPC Response) - 1: Response <- null@/127.0.0.1:33664: newEpoch {}
40,344 INFO  QJM - Successfully started new epoch 1
```

2.6 Synchronizing logs

To perform log synchronization, the writer performs the following steps:

1. Determine the transaction ID of the latest log segment. [4]
2. Determine the last transaction ID N in that segment that may have been successfully committed to a quorum of nodes. This is equivalent to determining which JournalNode contains the log with the most recently written transaction.
3. Ensure that a quorum of nodes synchronize this log segment by copying from the node containing the latest segment.
4. Instruct those nodes to mark that log segment finalized.

The postconditions of the log synchronization step are:

1. Any transaction that was previous committed must be stored on a quorum of nodes.
2. A quorum of nodes agrees on the contents and ending txid of the last log segment, and has marked the segment finalized.

After synchronizing to some transaction N, the writer may then begin to write new segments starting at transaction $N + 1$.

The algorithm for log synchronization is described later in this document.

2.7 Invariants

Invariant 1 *Once a log is finalized, it is never unfinalized.*

Invariant 2 *If there is a log segment starting at txid N on any node, then a quorum of nodes contain a finalized log segment ending at txid $N - 1$.*

In the non-failure case, this is true because the writer always finalizes its current log segment before beginning a new one. Finalization does not succeed unless a quorum of nodes acknowledge the finalization.

In the failure case, this is true because it is a postcondition of the log synchronization step. Before any writer begins to write, it must perform synchronization.

[4] In this document, we designate a segment's txid to be equal to the first txid written in that segment. For example, a segment containing transactions ID 100-150 is described as being "segment ID 100"

Invariant 3 *If there is a finalized log segment ending at txid N on any node, then a quorum of nodes have a log segment ending at txid N.*

This is true because the writer waits for a quorum of nodes to ack the last edit in a file before calling `finalize`.

2.8 Recovery algorithm

When a new writer takes over, the previous writer has most likely left some log segments in an "in-progress" state. Before the writer begins to write, it must recover the in-progress segment, and finalize it. The requirements of this process are as follows:

- The finalized log must contain all previously committed transactions. A transaction is considered committed if a majority of JNs acked it to the former writer.
- The log must become finalized on a quorum of journals.
- All loggers must finalize the segment to the same length and contents. In other words, if any two loggers contain a finalized log starting at the same transaction ID, then those log files must be semantically identical. [5]

This problem can be framed in terms of consensus: for a given segment transaction ID, we must come to consensus within the set of journal nodes such that all of the loggers agree on the length and contents of the segment, and then finalize the segment. The approach taken in this design is based on the well-known Paxos algorithm.

To frame the recovery process in terms of Paxos, we are running a single instance of Paxos for the log, trying to decide on the contents of the segment under recovery. Because it would not be practical to transmit the entire log segment using the RPC framework, instead we simply use the highest transaction ID and md5sum as a proxy for the segment contents, and also pass a URL from which the segment can be downloaded from another JournalNode.

The one variation from traditional Paxos is that we re-use the unique epoch number from the `newEpoch` step described above, rather than generating a new one for the recovery process. This is in the same spirit as the well-known multi-Paxos algorithm.

1. **Determining which segment to recover**: piggy-backed on the response to `newEpoch()`, each JournalNode also sends the transaction ID of its highest-numbered log segment. If any log segment has successfully started on a quorum of nodes, then the NN will find out about that segment during this step of recovery (since the quorum that responded to `newEpoch()` must overlap with any quorum that committed a transaction in the new segment).
2. **PrepareRecovery RPC**: the writer sends an RPC to each JN asking to prepare recovery for the given segment. Each JN responds with information about the current state of the segment from its local disk, including its length and its finalization status (finalized or in-progress). If this state corresponds to a previously-accepted recovery proposal, then the JN also includes the epoch number of the writer which proposed it.

   The request and response for this RPC map to the *Prepare* (Phase 1a) and *Promise* (Phase 1b) messages in Paxos, respectively.

[5] In the current implementation, the log files may differ due to padding at the end of the file after all of the valid edits. However, any actual transaction data must be identical.

3. **AcceptRecovery RPC**: based on the responses to the `PrepareRecovery` call, the recovering writer designates one of the segments (and its respective logger) as the source for recovery. This segment is picked such that it must contain all previously committed transactions, and such that, if any previous recovery process succeeded, the same decision is reached. The details of this decision are described below in Section 2.9.

   After choosing the recovery source, the writer issues an `AcceptRecovery` RPC to each of the JournalNodes, containing both the state of the segment and a URL from which the JN may obtain a copy of the log segment. [6]

   The `AcceptRecovery` call maps to Paxos's Phase 2a, often called *Accept!*.

   Upon receiving the `AcceptRecovery` RPC, the JournalNode performs the following actions:

   (a) **Log Synchronization:** If the current on-disk log is missing, or a different length than the proposed recovery, the JN downloads the log from the provided URI, replacing any current copy of the log segment.

   (b) **Persist recovery metadata:** The JN writes to local disk a structure which contains both the state of the segment, as well as the epoch number of the writer which proposed it. In any future `PrepareRecovery` calls for this same segment ID, this data and epoch number will be returned to the future writer.

   If these actions complete successfully, the JournalNode responds with a success code to the writer. If the writer receives a success code from a quorum of JournalNodes, it may proceed to the next step.

   The JournalNode's actions in response to `AcceptRecovery` correspond to Paxos Phase 2b.

4. **Finalize segment:** At this point, the writer knows that a quorum of JournalNodes have identical copies of the log segment, and have persisted the recovery information. Thus, any future writer issuing `PrepareRecovery` will see this decision and decide the same result, per the usual Paxos protocol. We may now safely finalize the log segment (analagous to a *Commit* phase of Paxos)[7]. To do so, the writer simply sends a `FinalizeLogSegment` call to each of the JournalNodes.

   Upon receiving this, the JournalNodes rename the log segment to indicate that it is finalized. They may also remove the persisted Paxos data for the segment, since the `finalized` state itself is enough to communicate that the decision is irrevocably accepted.

2.9 Journal Synchronziation - Choosing a recovery source

In the section above, we glossed over the process by which the client selects which log segment (and respective JournalNode) will act as the source for recovery. Now that the whole protocol has been described, we fill in this gap:

Upon receiving the responses to `PrepareRecovery()`, it evaluates them by the following rules: [8]

[6] It may in fact be that multiple JNs are valid recovery sources, in which case minority JNs could synchronize from any of the up-to-date ones. In the current implementation, however, recovery proceeds from a single source. If the source becomes unavailable during recovery, a new recovery will have to start from the first step.

[7] The Paxos algorithm itself is actually strictly complete as soon as consensus has been achieved. The Commit phase, sometimes called *Learned* simply acts to pass around the learned value to all of the nodes involved. Please refer to Section 2.3 "Learning a Chosen Value" from *Paxos Made Simple*, Lamport 2001

[8] The implementation of this algorithm may be viewed in the `SegmentRecoveryComparator` class.

1. Trivially, if a given node responds and indicates that it has no segment beginning at the given transaction ID, it is not a valid recovery source.

2. If any node already has a finalized segment, this indicates that a prior recovery was already in the process of committing, or that no recovery is actually necessary. In this case, the node containing the finalized segment acts as source.

3. For any two nodes which both respond with an in-progress segment, they are compared as follows:

   (a) For each logger, calculate `maxSeenEpoch` as the greater of that logger's `lastWriterEpoch` and the epoch number corresponding to any previously accepted recovery proposal.

   (b) If one logger's `maxSeenEpoch` is greater than the other's, then it is the better recovery source. The explanation for this follows below in Example 2.10.6.

   (c) If the two values are equal, whichever logger has more transactions available for this log segment is considered the better recovery source.

Note that there may be multiple segments (and respective JournalNodes) that are determined to be equally good sources by the above rules. For example, if all JournalNodes committed the most recent transaction and no further transactions were partially proposed, all JournalNodes would have identical states.

In this case, the current implementation chooses the recovery source arbitrarily between the equal options. When a JournalNode receives an `acceptRecovery()` RPC for a segment and sees that it already has an identical segment stored on its disk, it does not waste any effort in downloading the log from the remote node. So, in such a case that all JournalNodes have equal segments, no log data need be transferred for recovery.

2.10 Synchronizing logs - examples

2.10.1 Inconsistency in the middle of a log - prior quorum succeeded

In this situation, the prior writer sent a batch of 3 txns to JN2 and JN3, and then crashed before sending to JN1. JN1 could lag arbitrarily far behind.

JN	segment	last txid	lastWriterEpoch
JN1	edits_inprogress_101	150	1
JN2	edits_inprogress_101	153	1
JN3	edits_inprogress_101	153	1

Because the writer successfully wrote through txid 153 to a quorum of logs, we must be sure to recover through txid 153 to satisfy the correctness requirements. Since all copies have the same `maxSeenEpoch`, we follow Rule 3c and decide to synchronize from either JN2 or JN3.

2.10.2 Inconsistency in the middle of a log - prior quorum failed

In this situation, the prior writer sent a batch of 3 txns to JN2, and then crashed before sending to JN1 or JN3. In this case, JN3 has been "slow" for a while and lags far behind.

JN	segment	last txid	lastWriterEpoch
JN1	edits_inprogress_101	150	1
JN2	edits_inprogress_101	153	1
JN3	edits_inprogress_101	125	1

Because the prior writer did not write a quorum, it would be correct to decide either txid 150 or txid 153. 125 would not be correct because a quorum of nodes contain transactions through 150.

If during recovery we saw only JN1 and JN2, we will recover to txn 153. If we saw only JN1 and JN3, we would recover to 150. If we saw JN2 and JN3 we would recover to 153. These decisions follow Rule 3c.

2.10.3 Inconsistency finalizing a log - prior quorum succeeded

In this situation, the prior writer sent a finalizeEditLog call to JN1 and JN2, and then crashed before sending to JN3. JN3 could lag arbitrarily far behind.

JN	segment	last txid
JN1	edits_101-150	150
JN2	edits_101-150	150
JN3	edits_inprogress_101	145

In this situation, we need to instruct JN3 to synchronize from either of JN1 or JN2. Any quorum during recovery would see at least one finalized segment and thus decide to recover from JN1 or JN2 by Rule 2.

2.10.4 Inconsistency finalizing a log - prior quorum failed

In this situation, the prior writer sent a `finalizeEditLog(101,150)` call to JN1, and then crashed before sending to JN2 or JN3. One of the two may lag arbitrarily far behind.

JN	segment	last txid
JN1	edits_101-150	150
JN2	edits_inprogress_101	150
JN3	edits_inprogress_101	125

If, during recovery, we receive a response from JN1, then it will win by Rule 2. If instead, we receive a response from only JN2 and JN3, then JN2 will win by Rule 3c.

It's guaranteed that either JN2 or JN3 has the full length of the finalized log, since the QJM always achieves a quorum on the last edit in a segment before calling finalize on that segment.

2.10.5 Inconsistency starting a log - prior quorum failed

In this case, the QJM failed to achieve a `startLogSegment(151)` quorum, since it crashed after sending the RPC to JN1.

JN	prev segment	cur segment	last txid
JN1	edits_101-150	edits_inprogress_151	150
JN2	edits_101-150	-	150
JN3	edits_101-150	-	150

In this case, we make use of a simple trick in the JournalNode code: if, during recovery, the segment itself is entirely empty, then we move aside that segment and pretend it never existed (since it contains no valid data). This, in this case, all three JournalNodes appear equal, and no recovery will be necessary.

Upon starting the next segment, all JournalNodes will again succeed since the empty log file was moved aside by JN1.

2.10.6 Inconsistency on first batch of log - prior quorum failed

In this case, assume that the QJM first writes a log segment up to and including transaction ID 150, and then successfully finalizes the log segment on all nodes. It then issues `startLogSegment(151)`, which succeeds

on all nodes. It then tries to write the first batch of transactions (151-153), but only succeeds in contacting JN1. This leaves the following state:

JN	prev segment	cur segment	last txid	lastWriterEpoch
JN1	edits_101-150	edits_inprogress_151	153	1
JN2	edits_101-150	-	150	1
JN3	edits_101-150	-	150	1

Note that, even though `edits_inprogress_151` had been created on JN2 and JN3, it is removed since it is found to be entirely devoid of transactions.

Imagine that recovery proceeds with only JN2 and JN3 present, and the new NameNode (with writer epoch 2) successfully commits one transaction. We would then have the following state:

JN	prev segment	cur segment	last txid	lastWriterEpoch
JN1	edits_101-150	edits_inprogress_151	153	1
JN2	edits_101-150	edits_inprogress_151	151	2
JN3	edits_101-150	edits_inprogress_151	151	2

Note that the `lastWriterEpoch` has been set to 2 for JN2 and JN3 since the new writer was able to proceed writing.

If we then crashed and ran recovery, it is important to recover the log to txid 151 rather than txid 153, even though the log on JN1 is longer than the one on JN2 and JN3. This is the purpose of the `lastWriterEpoch`: because JN2 and JN3 have a higher writer epoch, they win over JN1 due to Rule 3b. Thus, JN1's log is properly removed and replaced with the correctly committed data from JN2 or JN3.

2.10.7 Multiple attempts at recovery, first failed

This case motivates the purpose of recording accepted recoveries during the synchronization process.

Assume we have failed with the three JNs at different lengths, as in 2.10.2:

JN	segment	last txid	acceptedInEpoch	lastWriterEpoch
JN1	edits_inprogress_101	150	-	1
JN2	edits_inprogress_101	153	-	1
JN3	edits_inprogress_101	125	-	1

Now assume that the first recovery attempt only contacts JN1 and JN3. It decides that length 150 is the correct recovery length, and calls `acceptRecovery(150)` on JN1 and JN3, followed by `finalizeLogSegment(101-150)`. But, it crashes before the `finalizeLogSegment` call reaches JN1. The state now is:

JN	segment	last txid	acceptedInEpoch	lastWriterEpoch
JN1	edits_inprogress_101	150	2	1
JN2	edits_inprogress_101	153	-	1
JN3	edits_101-150	150	-	1

When a new NN now begins recovery, assume it talks only to JN1 and JN2. If it did not consider `acceptedInEpoch`, it would incorrectly decide to finalize to txid 153, which would break the invariant that finalized log segments beginning at the same transaction ID must have the same length. Because of Rule 3b it will instead choose JN1 again as the recovery source, and properly finalize JN1 and JN2 to txid 150 instead of 153, which match the now-crashed JN3.

2.11 Writing edits

Writing edits is a comparatively simple process. The QJM performs the following:

1. Upon logSync, copy the queued-up bytes to a new byte array

2. Push this byte array into a queue for each remote JournalNode

3. Each JN has a single thread which processes items off the thread in order. These threads make `logEdits` RPCs to the JournalNodes.

4. Upon receipt, the JournalNode (a) verifies the epoch number as described above, (b) verifies the transaction IDs for the batch of edits, such that there are no gaps or out-of-order edits, (c) writes and syncs the edits to the currently open log segment, and (d) responds with a success code.

5. The original logSync thread waits for a quorum of nodes to respond with success. If a quorum responds with an exception, or a timeout occurs, the logSync() call throws an exception which is thrown from `logSync`. In the case that the QJM is being used as a shared edits storage mechanism, this will cause the NN to abort.

3 Design - reading

In the first implementation, we will make the restriction that journal segments may only be read from once finalized. Since journals are only finalized after a quorum of nodes have agreed to their contents, a reader may read from a finalized log on any replica and be assured that it is identical to all other copies.

In order to implement this, each JN will expose an HTTP server which allows a remote process to stream any finalized log segment, as well as an RPC which allows a remote process to enumerate the available log segments.

When the StandbyNode is reading from the JNs, it first issues a `getEditLogManifest()` RPC to all of the nodes. Any finalized segments that come back are merged together into `RedundantEditLogInputStreams`, so that the SBN may read from any JN for each segment. If one of the JNs fails while reading is under way, the redundant input stream automatically fails over to a different JN which had the same segment available.

4 Bootstrapping, storage, and formatting

4.1 JournalNode storage

The JournalNode's local storage directory is laid out with one directory per JournalID. The JournalID is a unique identifier which allows a single set of JournalNodes to be used for multiple federated HDFS NameNodes in the same cluster. Each Journal acts independently except that it is hosted by the same JVM and shares HTTP/IPC servers.

Each Journal directory is a typical Hadoop-style StorageDirectory, with a lock file and a `current/` subdirectory, allowing for future upgrade/rollback capability.

Within the `current/` directory, `edits` files are stored exactly the same as the traditional local NameNode edits directories. In addition to those files, the local storage contains the following special files:

- `committed-txid`: this file contains, in 64-bit big-endian, the last txid that this JN has seen committed. This is used internally for calculating lag metrics, but is not guaranteed to be up-to-date, and may always be safely removed.

- `last-promised-epoch`: this contains the value `lastPromisedEpoch` as described in this document. It is a text file containing a single numeric line.

- `last-writer-epoch`: this contains the value `lastWriterEpoch` as described in this document. It is a text file containing a single numeric line.

Storage for Paxos "accepted recoveries" is located under a `paxos/` directory. Each file in this directory contains a protobuf-serialized object including the details of the accepted recovery, and is deleted once the recovery has completed. So, it is normal for this directory to be empty when recovery is not under way.

4.2 Formatting

The JournalNode, upon startup, is initially empty. It can be formatted using the same mechanisms that the NameNode uses to format other shared storage (e.g `namenode -format` or `namenode -initializeSharedEdits`. When the NameNode wishes to format the JNs, it sends a `format()` RPC to each of the nodes, and expects a successful result from *all* of the JournalNodes (not just a majority).

The JournalNode also exposes an RPC `isFormatted()` which the NameNode uses to determine whether or not to require confirmation from the user during the formatting workflow.

After the Journal has been formatted, the `current/` directory contains a `VERSION` file including the cluster details (clusterID, namespaceID, cTime, etc). Future calls to this Journal will verify the namespace information and reject mismatched clusters.

5 Implementation - writing

The implementation is designed with testability as the foremost concern. To that end, all components should be built in such a way that the entire system can run, with injected faults, within a single JVM. Where possible, it should be possible to test the major algorithms without relying on actually writing to disk. This will enable stress tests to be done much more rapidly and thoroughly.

This document seeks to describe the major components, but not to go into too much detail. Please refer to the Javadocs in the work-in-progress patch for further detail. The test code also shows usage examples.

5.1 Quorum implementation

The quorum code heavily uses the `ListenableFuture` class from Guava for handling of asynchrony. This utility class provides an easy way to register callbacks and error-handlers to an async call which can be submitted to an `ExecutorService`. Additionally, these futures are easy to compose into more complex abstractions.

5.1.1 AsyncLogger

This interface wraps a remote JournalNode with ListenableFutures such that each of the RPCs can be performed asynchronously. For example, the method `void finalizeLogSegment(long startId, long endId)` is wrapped as `ListenableFuture<Void> finalizeLogSegment(long startId, long endId)`.

The concrete implementation of this interface is `IPCLoggerChannel`. This class simply wraps a Hadoop RPC proxy. Each call is submitted to a single-threaded `ExecutorService`, and that deferred call is wrapped with a `ListenableFuture`. This class is largely boilerplate wrapper code, though it takes care of adding the current epoch of the writer into each RPC message.

5.1.2 QuorumCall

This is a generic class which wraps a list of `ListenableFuture` instances and allows a caller to wait for a quorum of responses, a specified number of exceptions, or a timeout. For example, the caller may choose to wait for either (a) a majority of nodes to respond with success, (b) any nodes to throw an exception, or (c) 20 seconds to elapse. Whichever condition is satisfied first will cause the wait to return.

5.1.3 AsyncLoggerSet

This class wraps a set of `AsyncLoggers` to make quorum calls. For example, it contains utility functions to create `QuorumCall` instances to log edits, create new log segments, etc, as well as utility code to wait for a standard majority-quorum.

5.2 JournalManager implementation

The `QuorumJournalManager` class is responsible for configuring the quorum, starting and finalizing segments, etc. For each segment, it creates a `QuorumOutputStream` instance. This class is responsible for sending edits to a quorum of JNs.

5.3 JournalNode

The `JournalNode` is quite straight-forward. It simply listens on an IPC protocol (`QJournalProtocol`) to expose the correct functionality. Please refer to `TestJournalNode` for test cases which show its operation in isolation.

6 Metrics

The design will include a full set of metrics, both from the perspective of the client and the server. These include:

- **Lag metrics** - for each JournalNode, tracks how far that node lags behind the quorum value. This lag is tracked both in terms of time and in transaction count.
- **Latency metrics** - tracks the round-trip RPC latency for writing edits from the perspective of the client, as well as the latency seen in performing the `fsync()` calls on disk. The client metric has two variants: one which only encompasses the RPC itself, and another which includes queueing delays (in the case that the queue has become "backed up").
- **Queue sizes** - since the client accumulates in-flight data in a queue, it provides insight into the length of this queue as a metric, both in terms of raw data size and in number of transactions.

The full enumeration of metrics can be found in the source in `IPCLoggerChannelMetrics.java` and `JournalMetrics.java`

7 Test plan

7.1 Algorithmic tests

As there are some key distributed systems algorithms being employed in this work, it's crucial to be able to test them in isolation as unit tests. The organization of the code makes this somewhat easy. For an example, please refer to `TestQuorumCall` for a unit test which shows the quorum operation, or `TestEpochsAreUnique` for a more complicated stress test which ensures that no two QJMs will ever succeed their epoch-selection algorithm with the same epoch, despite random injected faults.

7.2 Minicluster tests

The test code includes a `MiniJournalCluster` implementation, similar to the mini-clusters used elsewhere in Hadoop testing. This code can spin up any desired number of journal nodes in the same JVM. Most of the functional tests are written in this style, and can be found in `TestQuorumJournalManager` as well as several other suites.

7.3 Fault tests

Combined with fault injection, we can stress many different scenarios with these tests. The test case `TestQJMWithFaults.testRandomized` operates in the following manner:

- The test may be provided with a random seed, or generate one on its own. A single random number generator is used throughout the test.
- All of the channels to the JournalNodes operate in a single thread, provided by the test. This causes the calls to not be interleaved, but rather to run in an entirely deterministic order: each call is sent first to JN1, then JN2, then JN3, rather than in parallel.
- A fault-injecting proxy is inserted between the client and the JournalNodes. This proxy uses the single random number generator to inject an IOException in some percentage of the RPC calls.

Because all of the fault injection is based on a random number generator running in a deterministic order, any failure can be reproduced by simply plugging in the same seed again. Additionally, the test enables IPC tracing for all communication, which allows for easy debugging.

Since this test is randomized, it is intended to increase its state-space coverage by repeated runs. In order to facilitate this, a Hadoop Streaming test is used to run the test in parallel on a 100-node Hadoop cluster. In this harness, each of 5000 map tasks runs the test in isolation, and writes the test logs to HDFS. If any of the test cases fails, its seed is reported to the user. Additionally, the full test output is grepped for the word `AssertionError` in case there are any unexpected conditions which are papered over by the quorum semantics.

During the development of this project, this test exposed several edge cases in both the design and implementation. It alone covers more than 75% of the code in this project, and nearly 100% of the most data-critical code paths.

7.4 System tests

This will be fleshed out further in the future. The first test devised is as follows:

7.4.1 Loggers-in-isolation test

Create a simple program `LogTester` which operates as follows:

- Creates a QuorumJournalManager instance, and takes over writing from any previous logger, initiating fencing and synchronization as described in this document.
- In a loop, perform the following actions:
  - Generate a small number of random transactions (eg OP_MKDIRS with random pathnames)
  - Log them to the journal
  - If logSync succeeds, log them to a file-based edit log on the local disk. Otherwise abort.

Next, deploy three JournalNodes on three machines in a cluster.

Then, on each of those nodes, run a script which starts a LogTester, lets it run until it crashes, sleeps a small number of seconds, and then restarts it.

In this test, all of the LogTester apps will be fencing each other several times per minute while also simulating edits.

After running the test for several hours, we can run a verification step of the following properties:

- If any LogTester has a txid N in its local directory, then no other logger has a txid N
- For any file that is a finalized segment, it will be bytewise identical to any other file with the same name anywhere on the cluster. We can verify this by taking md5sums for the finalized segments of all JournalNodes, then collating them together and ensuring that they match.

7.4.2 HA tests

The existing HA test plan (both for automatic failover and manual failover) are re-run using QuorumJournalManager as the configured shared edits storage.

In the case of automatic failover, the fencer is configured to `/bin/true` so as to exercise the automatic fencing capabilities of this design.

Additionally, various failures are injected manually into the JournalNodes while a steady load is directed at the NameNode:

- Journal Nodes are killed and restarted
- Journal Nodes are paused using `kill -STOP` and resumed using `kill -CONT`

In the fault cases, we expect that, so long as a majority of the nodes are non-faulty, the NameNode continues to operate with no adverse effects. When a majority of the nodes crash, we expect that the NameNode aborts on the next edit.

8 Revision history

```
commit 8f6392af9fac6c49a8f13142b0a5f8a8ebe5ac2c
Author: Todd Lipcon <todd@cloudera.com>
Date:   Wed Oct 3 14:17:49 2012 -0700
```

```
    Two more improvements per Sanjay's feedback:

    - Clarify example 2.10.6 per JIRA discussion
    - Clarify mapping of recovery process to Paxos rounds

commit 57bb94a5d3c0753b8dcbf0de8d0fb2fbb48d263b
Author: Todd Lipcon <todd@cloudera.com>
Date:   Mon Oct 1 11:46:42 2012 -0700

    Response to Sanjay's feedback

    - Clarifies "recovery master" -> "recovery source"
    - Clarifies the fact that current impl only synchronizes from one source

commit f8221d466016008a1bb74e6d3a99adebb4ba0ccb
Author: Todd Lipcon <todd@cloudera.com>
Date:   Thu Sep 27 15:16:31 2012 -0700

    Response to Suresh's questions on the JIRA

commit 7a160937d257fedb508c1b0019188579809e8e1c
Author: Todd Lipcon <todd@cloudera.com>
Date:   Tue Sep 18 21:28:00 2012 -0700

    Updates to protocol to include writer epoch and generally represent the final state of the implemen

commit 0b627f57d2dd895619edb435e72d39ea77771326
Author: Todd Lipcon <todd@cloudera.com>
Date:   Thu Jun 21 19:47:56 2012 -0700

    update for paxos-y recovery protocol

commit a00632d78c90ddae78384e5dc09b94d1caf9d48f
Author: Todd Lipcon <todd@cloudera.com>
Date:   Tue Apr 3 01:09:29 2012 -0700

    Initial rev
```