HDFS High Availability

- HDFS High Availability
  - Purpose
  - Note: Using the Quorum Journal Manager or Conventional Shared Storage
  - Background
  - Architecture
  - Hardware resources
  - Deployment
    - Configuration overview
    - Configuration details
    - Deployment details
    - Administrative commands
  - Automatic Failover
    - Introduction
    - Components
    - Deploying ZooKeeper
    - Before you begin
    - Configuring automatic failover
    - Initializing HA state in ZooKeeper
    - Starting the cluster with start-dfs.sh
    - Starting the cluster manually
    - Securing access to ZooKeeper
    - Verifying automatic failover
  - Automatic Failover FAQ
  - BookKeeper as a Shared storage (EXPERIMENTAL)

Purpose

This guide provides an overview of the HDFS High Availability (HA) feature and how to configure and manage an HA HDFS cluster, using NFS for the shared storage required by the NameNodes.

This document assumes that the reader has a general understanding of general components and node types in an HDFS cluster. Please refer to the HDFS Architecture guide for details.

Note: Using the Quorum Journal Manager or Conventional Shared Storage

This guide discusses how to configure and use HDFS HA using a shared NFS directory to share edit logs between the Active and Standby NameNodes. For information on how to configure HDFS HA using the Quorum Journal Manager instead of NFS, please see this alternative guide.

Background

Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine.

This impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in windows of cluster downtime.

The HDFS High Availability feature addresses the above problems by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby. This allows a fast failover to a new NameNode in the case that a machine crashes, or a graceful administrator-initiated failover for the purpose of planned maintenance.

Architecture

In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an *Active* state, and the other is in a *Standby* state. The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.

In order for the Standby node to keep its state synchronized with the Active node, the current implementation requires that the two nodes both have access to a directory on a shared storage device (eg an NFS mount from a NAS). This restriction will likely be relaxed in future versions.

When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace. In the event of a failover, the Standby will ensure that it has read all of the edits from the shared storage before promoting itself to the Active state. This ensures that the namespace state is fully synchronized before a failover occurs.

In order to provide a fast failover, it is also necessary that the Standby node have up-to-date information regarding the location of blocks in the cluster. In order to achieve this, the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats to both.

It is vital for the correct operation of an HA cluster that only one of the NameNodes be Active at a time. Otherwise, the namespace state would quickly diverge between the two, risking data loss or other incorrect results. In order to ensure this property and prevent the so-called "split-brain scenario," the administrator must configure at least one *fencing method* for the shared storage. During a failover, if it cannot be verified that the previous Active node has relinquished its Active state, the fencing process is responsible for cutting off the previous Active's access to the shared edits storage. This prevents it from making any further edits to the namespace, allowing the new Active to safely proceed with failover.

Hardware resources

In order to deploy an HA cluster, you should prepare the following:

- **NameNode machines** - the machines on which you run the Active and Standby NameNodes should have equivalent hardware to each other, and equivalent hardware to what would be used in a non-HA cluster.
- **Shared storage** - you will need to have a shared directory which both NameNode machines can have read/write access to. Typically this is a remote filer which supports NFS and is mounted on each of the NameNode machines. Currently only a single shared edits directory is supported. Thus, the availability of the system is limited by the availability of this shared edits directory, and therefore in order to remove all single points of failure there needs to be redundancy for the shared edits directory. Specifically, multiple network paths to the storage, and redundancy in the storage itself (disk, network, and power). Beacuse of this, it is recommended that the shared storage server be a high-quality dedicated NAS appliance rather than a simple Linux server.

Note that, in an HA cluster, the Standby NameNode also performs checkpoints of the namespace state, and thus it is not necessary to run a Secondary NameNode, CheckpointNode, or BackupNode in an HA cluster. In fact, to do so would be an error. This also allows one who is reconfiguring a non-HA-enabled HDFS cluster to be HA-enabled to reuse the hardware which they had previously dedicated to the Secondary NameNode.

Deployment

Configuration overview

Similar to Federation configuration, HA configuration is backward compatible and allows existing single NameNode configurations to work without change. The new configuration is designed such that all the nodes in the cluster may have the same configuration without the need for deploying different configuration files to different machines based on the type of the node.

Like HDFS Federation, HA clusters reuse the `nameservice ID` to identify a single HDFS instance that may in fact consist of multiple HA NameNodes. In addition, a new abstraction called `NameNode ID` is added with HA. Each distinct NameNode in the cluster has a different NameNode ID to distinguish it. To support a single configuration file for all of the NameNodes, the relevant configuration parameters are suffixed with the **nameservice ID** as well as the **NameNode ID**.

Configuration details

To configure HA NameNodes, you must add several configuration options to your **hdfs-site.xml** configuration file.

The order in which you set these configurations is unimportant, but the values you choose for **dfs.nameservices** and **dfs.ha.namenodes.[nameservice ID]** will determine the keys of those that follow. Thus, you should decide on these values before setting the rest of the configuration options.

- **dfs.nameservices** - the logical name for this new nameservice

  Choose a logical name for this nameservice, for example "mycluster", and use this logical name for the value of this config option. The name you choose is arbitrary. It will be used both for configuration and as the authority component of absolute HDFS paths in the cluster.

  Note: If you are also using HDFS Federation, this configuration setting should also include the list of other nameservices, HA or otherwise, as a comma-separated list.

  ```
  <property>
    <name>dfs.nameservices</name>
    <value>mycluster</value>
  </property>
  ```

- **dfs.ha.namenodes.[nameservice ID]** - unique identifiers for each NameNode in the nameservice

  Configure with a list of comma-separated NameNode IDs. This will be used by DataNodes to determine all the NameNodes in the cluster. For example, if you used "mycluster" as the nameservice ID previously, and you wanted to use "nn1" and "nn2" as the individual IDs of the NameNodes, you would configure this as such:

  ```
  <property>
    <name>dfs.ha.namenodes.mycluster</name>
    <value>nn1,nn2</value>
  </property>
  ```

  Note: Currently, only a maximum of two NameNodes may be configured per nameservice.

- **dfs.namenode.rpc-address.[nameservice ID].[name node ID]** - the fully-qualified RPC address for each NameNode to listen on

  For both of the previously-configured NameNode IDs, set the full address and IPC port of the NameNode processs. Note that this results in two separate configuration options. For example:

  ```
  <property>
    <name>dfs.namenode.rpc-address.mycluster.nn1</name>
    <value>machine1.example.com:8020</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.mycluster.nn2</name>
    <value>machine2.example.com:8020</value>
  </property>
  ```

  Note: You may similarly configure the "**servicerpc-address**" setting if you so desire.

- **dfs.namenode.http-address.[nameservice ID].[name node ID]** - the fully-qualified HTTP address for each NameNode to listen on

  Similarly to *rpc-address* above, set the addresses for both NameNodes' HTTP servers to listen on. For example:

  ```
  <property>
    <name>dfs.namenode.http-address.mycluster.nn1</name>
  ```

```
  <value>machine1.example.com:50070</value>
</property>
<property>
  <name>dfs.namenode.http-address.mycluster.nn2</name>
  <value>machine2.example.com:50070</value>
</property>
```

Note: If you have Hadoop's security features enabled, you should also set the *https-address* similarly for each NameNode.

- **dfs.namenode.shared.edits.dir** - the location of the shared storage directory

  This is where one configures the path to the remote shared edits directory which the Standby NameNode uses to stay up-to-date with all the file system changes the Active NameNode makes. **You should only configure one of these directories.** This directory should be mounted r/w on both NameNode machines. The value of this setting should be the absolute path to this directory on the NameNode machines. For example:

  ```
  <property>
    <name>dfs.namenode.shared.edits.dir</name>
    <value>file:///mnt/filer1/dfs/ha-name-dir-shared</value>
  </property>
  ```

- **dfs.client.failover.proxy.provider.[nameservice ID]** - the Java class that HDFS clients use to contact the Active NameNode

  Configure the name of the Java class which will be used by the DFS Client to determine which NameNode is the current Active, and therefore which NameNode is currently serving client requests. The only implementation which currently ships with Hadoop is the **ConfiguredFailoverProxyProvider**, so use this unless you are using a custom one. For example:

  ```
  <property>
    <name>dfs.client.failover.proxy.provider.mycluster</name>
    <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
  </property>
  ```

- **dfs.ha.fencing.methods** - a list of scripts or Java classes which will be used to fence the Active NameNode during a failover

  It is critical for correctness of the system that only one NameNode be in the Active state at any given time. Thus, during a failover, we first ensure that the Active NameNode is either in the Standby state, or the process has terminated, before transitioning the other NameNode to the Active state. In order to do this, you must configure at least one **fencing method.** These are configured as a carriage-return-separated list, which will be attempted in order until one indicates that fencing has succeeded. There are two methods which ship with Hadoop: *shell* and *sshfence*. For information on implementing your own custom fencing method, see the *org.apache.hadoop.ha.NodeFencer* class.

  - **sshfence** - SSH to the Active NameNode and kill the process

    The *sshfence* option SSHes to the target node and uses *fuser* to kill the process listening on the service's TCP port. In order for this fencing option to work, it must be able to SSH to the target node without providing a passphrase. Thus, one must also configure the **dfs.ha.fencing.ssh.private-key-files** option, which is a comma-separated list of SSH private key files. For example:

    ```
    <property>
      <name>dfs.ha.fencing.methods</name>
      <value>sshfence</value>
    </property>

    <property>
      <name>dfs.ha.fencing.ssh.private-key-files</name>
      <value>/home/exampleuser/.ssh/id_rsa</value>
    </property>
    ```

    Optionally, one may configure a non-standard username or port to perform the SSH. One may also configure a timeout, in milliseconds, for the SSH, after which this fencing method will be considered to have failed. It may be configured like so:

    ```
    <property>
      <name>dfs.ha.fencing.methods</name>
      <value>sshfence([[username][:port]])</value>
    </property>
    <property>
      <name>dfs.ha.fencing.ssh.connect-timeout</name>
      <value>30000</value>
    </property>
    ```

  - **shell** - run an arbitrary shell command to fence the Active NameNode

    The *shell* fencing method runs an arbitrary shell command. It may be configured like so:

    ```
    <property>
      <name>dfs.ha.fencing.methods</name>
      <value>shell(/path/to/my/script.sh arg1 arg2 ...)</value>
    </property>
    ```

    The string between '(' and ')' is passed directly to a bash shell and may not include any closing parentheses.

    The shell command will be run with an environment set up to contain all of the current Hadoop configuration variables, with the '_' character replacing any '.' characters in the configuration keys. The configuration used has already had any namenode-specific configurations promoted to their generic forms -- for example **dfs_namenode_rpc-address** will contain the RPC address of the target node, even though the configuration may specify that variable as **dfs.namenode.rpc-address.ns1.nn1**.

    Additionally, the following variables referring to the target node to be fenced are also available:

$target_host	hostname of the node to be fenced
$target_port	IPC port of the node to be fenced
$target_address	the above two, combined as host:port
$target_nameserviceid	the nameservice ID of the NN to be fenced
$target_namenodeid	the namenode ID of the NN to be fenced

These environment variables may also be used as substitutions in the shell command itself. For example:

```
<property>
  <name>dfs.ha.fencing.methods</name>
  <value>shell(/path/to/my/script.sh --nameservice=$target_nameserviceid $target_host:$target_port)</value>
</property>
```

If the shell command returns an exit code of 0, the fencing is determined to be successful. If it returns any other exit code, the fencing was not successful and the next fencing method in the list will be attempted.

Note: This fencing method does not implement any timeout. If timeouts are necessary, they should be implemented in the shell script itself (eg by forking a subshell to kill its parent in some number of seconds).

- **fs.defaultFS** - the default path prefix used by the Hadoop FS client when none is given

  Optionally, you may now configure the default path for Hadoop clients to use the new HA-enabled logical URI. If you used "mycluster" as the nameservice ID earlier, this will be the value of the authority portion of all of your HDFS paths. This may be configured like so, in your **core-site.xml** file:

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://mycluster</value>
</property>
```

Deployment details

After all of the necessary configuration options have been set, one must initially synchronize the two HA NameNodes' on-disk metadata.

- If you are setting up a fresh HDFS cluster, you should first run the format command (*hdfs namenode -format*) on one of NameNodes.
- If you have already formatted the NameNode, or are converting a non-HA-enabled cluster to be HA-enabled, you should now copy over the contents of your NameNode metadata directories to the other, unformatted NameNode by running the command "*hdfs namenode -bootstrapStandby*" on the unformatted NameNode. Running this command will also ensure that the shared edits directory (as configured by **dfs.namenode.shared.edits.dir**) contains sufficient edits transactions to be able to start both NameNodes.
- If you are converting a non-HA NameNode to be HA, you should run the command "*hdfs -initializeSharedEdits*", which will initialize the shared edits directory with the edits data from the local NameNode edits directories.

At this point you may start both of your HA NameNodes as you normally would start a NameNode.

You can visit each of the NameNodes' web pages separately by browsing to their configured HTTP addresses. You should notice that next to the configured address will be the HA state of the NameNode (either "standby" or "active".) Whenever an HA NameNode starts, it is initially in the Standby state.

Administrative commands

Now that your HA NameNodes are configured and started, you will have access to some additional commands to administer your HA HDFS cluster. Specifically, you should familiarize yourself with all of the subcommands of the "*hdfs haadmin*" command. Running this command without any additional arguments will display the following usage information:

```
Usage: DFSHAAdmin [-ns <nameserviceId>]
    [-transitionToActive <serviceId>]
    [-transitionToStandby <serviceId>]
    [-failover [--forcefence] [--forceactive] <serviceId> <serviceId>]
    [-getServiceState <serviceId>]
    [-checkHealth <serviceId>]
    [-help <command>]
```

This guide describes high-level uses of each of these subcommands. For specific usage information of each subcommand, you should run "*hdfs haadmin -help <command>*".

- **transitionToActive** and **transitionToStandby** - transition the state of the given NameNode to Active or Standby

  These subcommands cause a given NameNode to transition to the Active or Standby state, respectively. **These commands do not attempt to perform any fencing, and thus should rarely be used.** Instead, one should almost always prefer to use the "*hdfs haadmin -failover*" subcommand.

- **failover** - initiate a failover between two NameNodes

  This subcommand causes a failover from the first provided NameNode to the second. If the first NameNode is in the Standby state, this command simply transitions the second to the Active state without error. If the first NameNode is in the Active state, an attempt will be made to gracefully transition it to the Standby state. If this fails, the fencing methods (as configured by **dfs.ha.fencing.methods**) will be attempted in order until one succeeds. Only after this process will the second NameNode be transitioned to the Active state. If no fencing method succeeds, the second NameNode will not be transitioned to the Active state, and an error will be returned.

- **getServiceState** - determine whether the given NameNode is Active or Standby

  Connect to the provided NameNode to determine its current state, printing either "standby" or "active" to STDOUT appropriately. This subcommand might be used by cron jobs or monitoring scripts which need to behave differently based on whether the NameNode is currently Active or Standby.

- **checkHealth** - check the health of the given NameNode

  Connect to the provided NameNode to check its health. The NameNode is capable of performing some diagnostics on itself, including checking if internal services are running as expected. This command will return 0 if the NameNode is healthy, non-zero otherwise. One might use this command for monitoring purposes.

  Note: This is not yet implemented, and at present will always return success, unless the given NameNode is completely down.

Automatic Failover

Introduction

The above sections describe how to configure manual failover. In that mode, the system will not automatically trigger a failover from the active to the standby NameNode, even if the active node has failed. This section describes how to configure and deploy automatic failover.

Components

Automatic failover adds two new components to an HDFS deployment: a ZooKeeper quorum, and the ZKFailoverController process (abbreviated as ZKFC).

Apache ZooKeeper is a highly available service for maintaining small amounts of coordination data, notifying clients of changes in that data, and monitoring clients for failures. The implementation of automatic HDFS failover relies on ZooKeeper for the following things:

- **Failure detection** - each of the NameNode machines in the cluster maintains a persistent session in ZooKeeper. If the machine crashes, the ZooKeeper session will expire, notifying the other NameNode that a failover should be triggered.
- **Active NameNode election** - ZooKeeper provides a simple mechanism to exclusively elect a node as active. If the current active NameNode crashes, another node may take a special exclusive lock in ZooKeeper indicating that it should become the next active.

The ZKFailoverController (ZKFC) is a new component which is a ZooKeeper client which also monitors and manages the state of the NameNode. Each of the machines which runs a NameNode also runs a ZKFC, and that ZKFC is responsible for:

- **Health monitoring** - the ZKFC pings its local NameNode on a periodic basis with a health-check command. So long as the NameNode responds in a timely fashion with a healthy status, the ZKFC considers the node healthy. If the node has crashed, frozen, or otherwise entered an unhealthy state, the health monitor will mark it as unhealthy.
- **ZooKeeper session management** - when the local NameNode is healthy, the ZKFC holds a session open in ZooKeeper. If the local NameNode is active, it also holds a special "lock" znode. This lock uses ZooKeeper's support for "ephemeral" nodes; if the session expires, the lock node will be automatically deleted.
- **ZooKeeper-based election** - if the local NameNode is healthy, and the ZKFC sees that no other node currently holds the lock znode, it will itself try to acquire the lock. If it succeeds, then it has "won the election", and is responsible for running a failover to make its local NameNode active. The failover process is similar to the manual failover described above: first, the previous active is fenced if necessary, and then the local NameNode transitions to active state.

For more details on the design of automatic failover, refer to the design document attached to HDFS-2185 on the Apache HDFS JIRA.

Deploying ZooKeeper

In a typical deployment, ZooKeeper daemons are configured to run on three or five nodes. Since ZooKeeper itself has light resource requirements, it is acceptable to collocate the ZooKeeper nodes on the same hardware as the HDFS NameNode and Standby Node. Many operators choose to deploy the third ZooKeeper process on the same node as the YARN ResourceManager. It is advisable to configure the ZooKeeper nodes to store their data on separate disk drives from the HDFS metadata for best performance and isolation.

The setup of ZooKeeper is out of scope for this document. We will assume that you have set up a ZooKeeper cluster running on three or more nodes, and have verified its correct operation by connecting using the ZK CLI.

Before you begin

Before you begin configuring automatic failover, you should shut down your cluster. It is not currently possible to transition from a manual failover setup to an automatic failover setup while the cluster is running.

Configuring automatic failover

The configuration of automatic failover requires the addition of two new parameters to your configuration. In your `hdfs-site.xml` file, add:

```
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
</property>
```

This specifies that the cluster should be set up for automatic failover. In your `core-site.xml` file, add:

```
<property>
  <name>ha.zookeeper.quorum</name>
  <value>zk1.example.com:2181,zk2.example.com:2181,zk3.example.com:2181</value>
</property>
```

This lists the host-port pairs running the ZooKeeper service.

As with the parameters described earlier in the document, these settings may be configured on a per-nameservice basis by suffixing the configuration key with the nameservice ID. For example, in a cluster with federation enabled, you can explicitly enable automatic failover for only one of the nameservices by setting `dfs.ha.automatic-failover.enabled.my-nameservice-id`.

There are also several other configuration parameters which may be set to control the behavior of automatic failover; however, they are not necessary for most installations. Please refer to the configuration key specific documentation for details.

Initializing HA state in ZooKeeper

After the configuration keys have been added, the next step is to initialize required state in ZooKeeper. You can do so by running the following

command from one of the NameNode hosts.

```
$ hdfs zkfc -formatZK
```

This will create a znode in ZooKeeper inside of which the automatic failover system stores its data.

Starting the cluster with `start-dfs.sh`

Since automatic failover has been enabled in the configuration, the `start-dfs.sh` script will now automatically start a ZKFC daemon on any machine that runs a NameNode. When the ZKFCs start, they will automatically select one of the NameNodes to become active.

Starting the cluster manually

If you manually manage the services on your cluster, you will need to manually start the `zkfc` daemon on each of the machines that runs a NameNode. You can start the daemon by running:

```
$ hadoop-daemon.sh start zkfc
```

Securing access to ZooKeeper

If you are running a secure cluster, you will likely want to ensure that the information stored in ZooKeeper is also secured. This prevents malicious clients from modifying the metadata in ZooKeeper or potentially triggering a false failover.

In order to secure the information in ZooKeeper, first add the following to your `core-site.xml` file:

```
<property>
  <name>ha.zookeeper.auth</name>
  <value>@/path/to/zk-auth.txt</value>
</property>
<property>
  <name>ha.zookeeper.acl</name>
  <value>@/path/to/zk-acl.txt</value>
</property>
```

Please note the '@' character in these values -- this specifies that the configurations are not inline, but rather point to a file on disk.

The first configured file specifies a list of ZooKeeper authentications, in the same format as used by the ZK CLI. For example, you may specify something like:

```
digest:hdfs-zkfcs:mypassword
```

...where `hdfs-zkfcs` is a unique username for ZooKeeper, and `mypassword` is some unique string used as a password.

Next, generate a ZooKeeper ACL that corresponds to this authentication, using a command like the following:

```
$ java -cp $ZK_HOME/lib/*:$ZK_HOME/zookeeper-3.4.2.jar org.apache.zookeeper.server.auth.DigestAuthenticationProvider hdfs-zkfcs:mypassword
output: hdfs-zkfcs:mypassword->hdfs-zkfcs:P/OQvnYyU/nF/mGYvB/xurX8dYs=
```

Copy and paste the section of this output after the '->' string into the file `zk-acls.txt`, prefixed by the string "`digest:`". For example:

```
digest:hdfs-zkfcs:vlUvLnd8MlacsE80rDuu6ONESbM=:rwcda
```

In order for these ACLs to take effect, you should then rerun the `zkfc -formatZK` command as described above.

After doing so, you may verify the ACLs from the ZK CLI as follows:

```
[zk: localhost:2181(CONNECTED) 1] getAcl /hadoop-ha
'digest,'hdfs-zkfcs:vlUvLnd8MlacsE80rDuu6ONESbM=
: cdrwa
```

Verifying automatic failover

Once automatic failover has been set up, you should test its operation. To do so, first locate the active NameNode. You can tell which node is active by visiting the NameNode web interfaces -- each node reports its HA state at the top of the page.

Once you have located your active NameNode, you may cause a failure on that node. For example, you can use `kill -9 <pid of NN>` to simulate a JVM crash. Or, you could power cycle the machine or unplug its network interface to simulate a different kind of outage. After triggering the outage you wish to test, the other NameNode should automatically become active within several seconds. The amount of time required to detect a failure and trigger a fail-over depends on the configuration of `ha.zookeeper.session-timeout.ms`, but defaults to 5 seconds.

If the test does not succeed, you may have a misconfiguration. Check the logs for the `zkfc` daemons as well as the NameNode daemons in order to further diagnose the issue.

Automatic Failover FAQ

- **Is it important that I start the ZKFC and NameNode daemons in any particular order?**

  No. On any given node you may start the ZKFC before or after its corresponding NameNode.

- **What additional monitoring should I put in place?**

You should add monitoring on each host that runs a NameNode to ensure that the ZKFC remains running. In some types of ZooKeeper failures, for example, the ZKFC may unexpectedly exit, and should be restarted to ensure that the system is ready for automatic failover.

Additionally, you should monitor each of the servers in the ZooKeeper quorum. If ZooKeeper crashes, then automatic failover will not function.

- **What happens if ZooKeeper goes down?**

  If the ZooKeeper cluster crashes, no automatic failovers will be triggered. However, HDFS will continue to run without any impact. When ZooKeeper is restarted, HDFS will reconnect with no issues.

- **Can I designate one of my NameNodes as primary/preferred?**

  No. Currently, this is not supported. Whichever NameNode is started first will become active. You may choose to start the cluster in a specific order such that your preferred node starts first.

- **How can I initiate a manual failover when automatic failover is configured?**

  Even if automatic failover is configured, you may initiate a manual failover using the same `hdfs haadmin` command. It will perform a coordinated failover.

BookKeeper as a Shared storage (EXPERIMENTAL)

One option for shared storage for the NameNode is BookKeeper. BookKeeper achieves high availability and strong durability guarantees by replicating edit log entries across multiple storage nodes. The edit log can be striped across the storage nodes for high performance. Fencing is supported in the protocol, i.e, BookKeeper will not allow two writers to write the single edit log.

The meta data for BookKeeper is stored in ZooKeeper. In current HA architecture, a Zookeeper cluster is required for ZKFC. The same cluster can be for BookKeeper metadata.

For more details on building a BookKeeper cluster, please refer to the BookKeeper documentation

The BookKeeperJournalManager is an implementation of the HDFS JournalManager interface, which allows custom write ahead logging implementations to be plugged into the HDFS NameNode.

- **BookKeeper Journal Manager**

  To use BookKeeperJournalManager, add the following to hdfs-site.xml.

```
<property>
  <name>dfs.namenode.shared.edits.dir</name>
  <value>bookkeeper://zk1:2181;zk2:2181;zk3:2181/hdfsjournal</value>
</property>

<property>
  <name>dfs.namenode.edits.journal-plugin.bookkeeper</name>
  <value>org.apache.hadoop.contrib.bkjournal.BookKeeperJournalManager</value>
</property>
```

  The URI format for bookkeeper is `bookkeeper://[zkEnsemble]/[rootZnode] [zookkeeper ensemble]` is a list of semi-colon separated, zookeeper host:port pairs. In the example above there are 3 servers, in the ensemble, zk1, zk2 & zk3, each one listening on port 2181.

  `[root znode]` is the path of the zookeeper znode, under which the edit log information will be stored.

  The class specified for the journal-plugin must be available in the NameNode's classpath. We explain how to generate a jar file with the journal manager and its dependencies, and how to put it into the classpath below.

- **More configuration options**
  - **dfs.namenode.bookkeeperjournal.output-buffer-size** - Number of bytes a bookkeeper journal stream will buffer before forcing a flush. Default is 1024.

```
<property>
  <name>dfs.namenode.bookkeeperjournal.output-buffer-size</name>
  <value>1024</value>
</property>
```

  - **dfs.namenode.bookkeeperjournal.ensemble-size** - Number of bookkeeper servers in edit log ensembles. This is the number of bookkeeper servers which need to be available for the edit log to be writable. Default is 3.

```
<property>
  <name>dfs.namenode.bookkeeperjournal.ensemble-size</name>
  <value>3</value>
</property>
```

  - **dfs.namenode.bookkeeperjournal.quorum-size** - Number of bookkeeper servers in the write quorum. This is the number of bookkeeper servers which must have acknowledged the write of an entry before it is considered written. Default is 2.

```
<property>
  <name>dfs.namenode.bookkeeperjournal.quorum-size</name>
  <value>2</value>
</property>
```

  - **dfs.namenode.bookkeeperjournal.digestPw** - Password to use when creating edit log segments.

```
<property>
 <name>dfs.namenode.bookkeeperjournal.digestPw</name>
 <value>myPassword</value>
</property>
```

  - **dfs.namenode.bookkeeperjournal.zk.session.timeout** - Session timeout for Zookeeper client from BookKeeper Journal Manager. Hadoop recommends that this value should be less than the ZKFC session timeout value. Default value is 3000.

```
<property>
  <name>dfs.namenode.bookkeeperjournal.zk.session.timeout</name>
  <value>3000</value>
</property>
```

- **Building BookKeeper Journal Manager plugin jar**

  To generate the distribution packages for BK journal, do the following.

  $ mvn clean package -Pdist

  This will generate a jar with the BookKeeperJournalManager, hadoop-hdfs/src/contrib/bkjournal/target/hadoop-hdfs-bkjournal-*VERSION*.jar

  Note that the -Pdist part of the build command is important, this would copy the dependent bookkeeper-server jar under hadoop-hdfs/src/contrib/bkjournal/target/lib.

- **Putting the BookKeeperJournalManager in the NameNode classpath**

  To run a HDFS namenode using BookKeeper as a backend, copy the bkjournal and bookkeeper-server jar, mentioned above, into the lib directory of hdfs. In the standard distribution of HDFS, this is at $HADOOP_HDFS_HOME/share/hadoop/hdfs/lib/

  cp hadoop-hdfs/src/contrib/bkjournal/target/hadoop-hdfs-bkjournal-*VERSION*.jar $HADOOP_HDFS_HOME/share/hadoop/hdfs/lib/

- **Current limitations**

  1) Security in BookKeeper. BookKeeper does not support SASL nor SSL for connections between the NameNode and BookKeeper storage nodes.