1. **HDFS Federation**

HDFS Federation improves the existing HDFS architecture through a clear separation of namespace and storage, enabling generic block storage layer. It enables support for multiple namespaces in the cluster to improve scalability and isolation. Federation also opens up the architecture, expanding the applicability of HDFS cluster to new implementations and use cases.

In order to scale the name service horizontally, federation uses multiple independent namenodes/namespaces. The namenodes are federated, that is, the namenodes are independent and don’t require coordination with each other. The datanodes are used as common storage for blocks by all the namenodes. Each datanode registers with all the namenodes in the cluster. Datanodes send periodic heartbeats and block reports and handles commands from the namenodes.

A **Block Pool** is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster.

It is managed independently of other block pools. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. The failure of a namenode does not prevent the datanode from serving other namenodes in the cluster.

A Namespace and its block pool together are called **Namespace Volume**. It is a self-contained unit of management. When a namenode/namespace is deleted, the corresponding block pool at the datanodes is deleted. Each namespace volume is upgraded as a unit, during cluster upgrade.

**Key Benefits**

**Scalability and isolation**

Support for multiple namenodes horizontally scales the file system namespace. It separates namespace volumes for users and categories of applications and improves isolation.

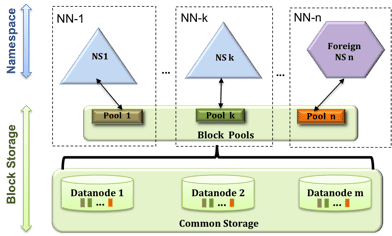
**Generic storage service**

Block pool abstraction opens up the architecture for future innovation. New file systems can be built on top of block storage. New applications can be directly built on the block storage layer without the need to use a file system interface. New block pool categories are also possible, different from the default block pool. Examples include a block pool for MapReduce tmp storage with different garbage collection scheme or a block pool that caches data to make distributed cache more efficient.

**Design simplicity**

We considered distributed namenodes and chose to go with federation because it is significantly simpler to design and implement. Namenodes and namespaces are independent of each other and require very little change to the existing namenodes. The robustness of the namenode is not affected. Federation also preserves backward compatibility of configuration. The existing single namenode deployments work without any configuration changes.

It took only 4 months to implement the features and stabilize the software. The namenode has very few changes. Most of the changes are in the datanode, to introduce block pool as a new hierarchy in storage, replica map and other internal data structures. Other changes include fixing the tests to work with new hierarchy of data structures and tools to simplify management of federated clusters.



# **HDFS High Availability**

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.

2. HDFS, files are divided into blocks, and file access follows multi-reader, single-writer semantics. To meet the fault-tolerance requirement, multiple replicas of a block are stored on different DataNodes. The number of replicas is called the replication factor. When a new file block is created, or an existing file is opened for append, the HDFS write operation creates a pipeline of DataNodes to receive and store the replicas (the replication factor generally determines the number of DataNodes in the pipeline). Subsequent writes to that block go through the pipeline (Figure 1).

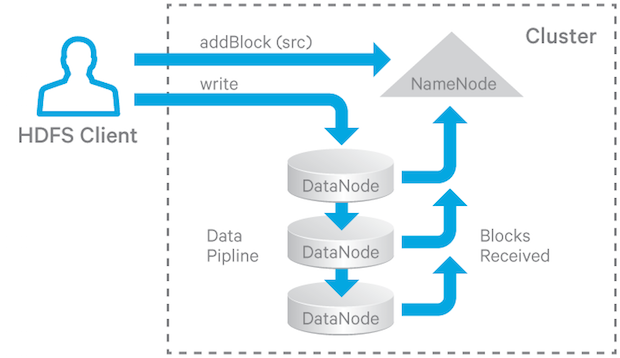
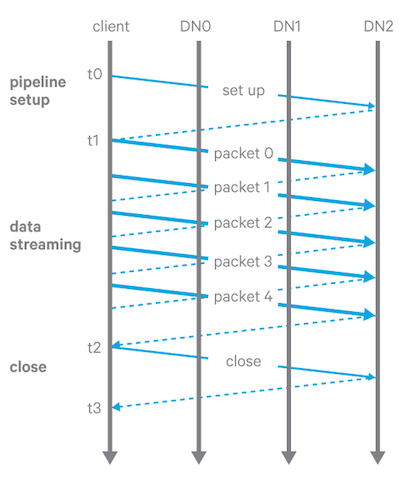


Figure 1. HDFS Write Pipeline

For read operations the client chooses one of the DataNodes holding copies of the block and requests a data transfer from it.

##### **The Write Pipeline**

When an HDFS client writes to file, the data is written as sequential blocks. To write or construct a block, HDFS breaks the block into packets (not actually network packets but rather messages; the term packets refers to the class which embodies these messages), and propagates them to the DataNodes in the write pipeline, as shown in Figure 2.



**Figure 2.** HDFS Write Pipeline Stages

There are three stages of a write pipeline:

1. Pipeline setup. The client sends a Write\_Block request along the pipeline and the last DataNode sends an acknowledgement back. After receiving the acknowledgement, the pipeline is ready for writing.
2. Data streaming. The data is sent through the pipeline in packets. The client buffers the data until a packet is filled up, and then sends the packet to the pipeline. If the client calls hflush(), then even if a packet is not full, it will nevertheless be sent to the pipeline and the next packet will not be sent until the acknowledgement of the previous hflush’ed packet is received by the client.
3. Close(finalize the replica and shutdown the pipeline). The client waits until all packets have been acknowledged and then sends a close request. All DataNodes in the pipeline change the corresponding replica into the FINALIZED state and report back to the NameNode. The NameNode then changes the block’s state to COMPLETE if at least the configured minimum replication number of DataNodes reported a FINALIZED state of their corresponding replicas.

##### **Pipeline Recovery**

Pipeline recovery is initiated when one or more DataNodes in the pipeline encounter an error in any of the three stages while a block is being written.

**Recovery from Pipeline Setup Failure**

1. If the pipeline was created for a new block, the client abandons the block and asks the NameNode for a new block and a new list of DataNodes. The pipeline is reinitialized for the new block.
2. If the pipeline was created to append to a block, the client rebuilds the pipeline with the remaining DataNodes and increments the block’s generation stamp.

**Recovery from Data Streaming Failure**

1. When a DataNode in the pipeline detects an error (for example, a checksum error or a failure to write to disk), that DataNode takes itself out of the pipeline by closing up all TCP/IP connections. If the data is deemed not corrupted, it also writes buffered data to the relevant block and checksum (METADATA) files.
2. When the client detects the failure, it stops sending data to the pipeline, and reconstructs a new pipeline using the remaining good DataNodes. As a result, all replicas of the block are bumped up to a new GS.
3. The client resumes sending data packets with this new GS. If the data sent has already been received by some of the DataNodes, they just ignore the packet and pass it downstream in the pipeline.

**Recovery from Close Failure**

1. When the client detects a failure in the close state, it rebuilds the pipeline with the remaining DataNodes. Each DataNode bumps up the block’s GS and finalizes the replica if it’s not finalized yet.

When one DataNode is bad, it removes itself from the pipeline. During the pipeline recovery process, the client may need to rebuild a new pipeline with the remaining DataNodes. (It may or may not replace bad DataNodes with new DataNodes, depending on the DataNode replacement policy described in the next section.) The replication monitor will take care of replicating the block to satisfy the configured replication factor.

##### **DataNode Replacement Policy upon Failure**

There are four configurable policies regarding whether to add additional DataNodes to replace the bad ones when setting up a pipeline for recovery with the remaining DataNodes:

1. DISABLE: Disables DataNode replacement and throws an error (at the server); this acts like NEVER at the client.
2. NEVER: Never replace a DataNode when a pipeline fails (generally not a desirable action).
3. DEFAULT:  Replace based on the following conditions:
   1. Let r be the configured replication number.
   2. Let n be the number of existing replica datanodes.
   3. Add a new DataNode only if r >= 3 and EITHER
      * floor(r/2) >= n; OR
      * r > n and the block is hflushed/appended.

ALWAYS: Always add a new DataNode when an existing DataNode failed

sHDFS handling failure while writing

NameNode periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode. When NameNode notices that it has not recieved a hearbeat message from a data node after a certain amount of time, the data node is marked as dead. Since blocks will be under replicated the system begins replicating the blocks that were stored on the dead datanode. The NameNode Orchestrates the replication of data blocks from one datanode to another. The replication data transfer happens directly between datanodes and the data never passes through the namenode.

Data node passes a heartbeat signal to Name node in an interval of specified time (Usually 2 minutes), which helps the Name node to determine that the data node is alive & functional.  
  
When Name node does not receive heartbeat signals from Data node, it assumes that the data node is either dead or non-functional.  
  
As soon as the data node is declared dead/non-functional all the data blocks it hosts are transferred to the other data nodes with which the blocks are replicated initially.  
  
In this process of transferring the data blocks from dead name node to other data nodes, Name node is not involved.

To write file in HDFS, client needs to interact with master i.e. namenode. Now namenode provides the address of the datanodes (slaves) on which client will start writing the data. Client directly writes data on the datanodes, now datanode will create data write pipeline. The first datanode will copy the block to another datanode, which intern copy it to third datanode (same flow has been explained in the below diagram). Once required replicas of blocks are created, it sends back the acknowledgement.