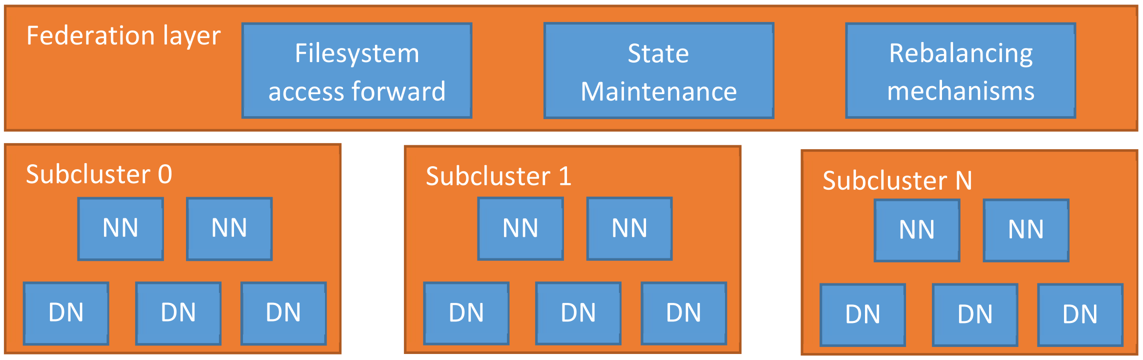
Router-based HDFS federation

Íñigo Goiri, Jason Kace, and Ricardo Bianchini

1 Problem

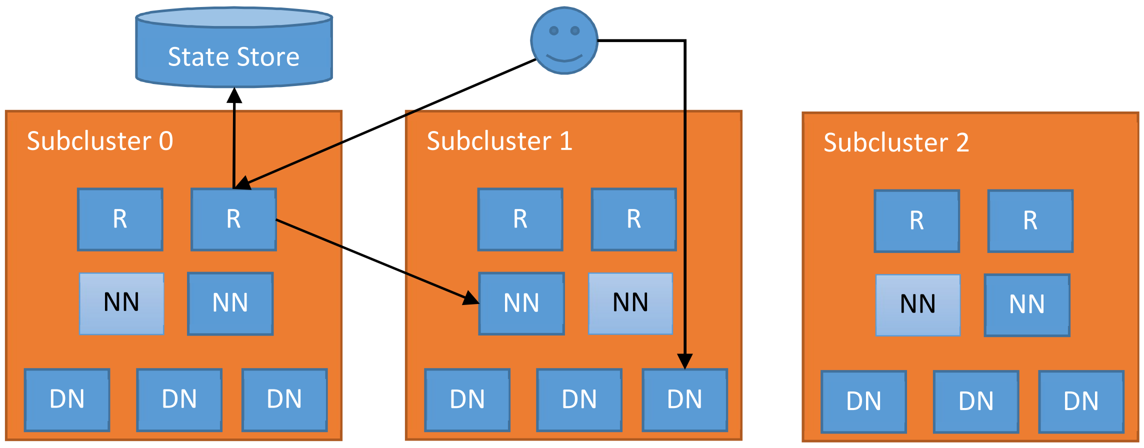
Namenodes have scalability limits because of the metadata overhead comprised of inodes (files and directories) and file blocks, the number of Datanode heartbeats, and the number of HDFS RPC client requests. The common solution is to split the filesystem into smaller subclusters (we discuss other approaches and their limitations at the end of this doc). The problem with this approach is how to maintain the split of the subclusters (e.g., namespace partition), which forces users to connect to multiple subclusters and manage the allocation of folders/files to them.



2 Router-based federation

A natural extension to this partitioned federation is to add a layer of software responsible for federating the namespaces, as in the figure above. This extra layer would allow users transparently access any subcluster, would let subclusters manage their own block pools independently, and would support the rebalancing of data across subclusters. To accomplish these goals, the federation layer must direct block accesses to the proper subcluster, maintain the state of the namespaces, and provide mechanisms for data rebalancing. At the same time, it must be scalable, highly available, and fault tolerant.

Our design for the federation layer comprises multiple components. We build a Router component that has the same interface as a Namenode, and forwards the client requests to the correct subcluster, based on ground-truth information from a State Store. The State Store is the combination of a remote mount table (in the flavor of ViewFS, but shared between clients) and load/space information about the subclusters. The figure below illustrates the architecture, showing the per-subcluster redundant Routers (marked "R" and described in detail in Section 3) and the logically centralized (but physically distributed) State Store (described in detail in Section 4), as well as the per-subcluster redundant Namenodes ("NN") and the per-server Datanodes ("DN"). This approach has the same architecture as the YARN federation (YARN-2915). We provide a Subcluster Rebalancer to move data across the subclusters and re-partition the name space (more details in Section 5).



2.1 Example flow

The simplest configuration is to have a Router on each Namenode machine. The Router monitors the local Namenode and heartbeats the state to the State Store.

When a regular DFS client contacts any of the Routers to access a file in the federated filesystem, the Router checks the Mount Table in the State Store (i.e., the local cache) to find out which subcluster contains the file. Then, it checks the Membership table in the State Store (i.e., the local cache) for the Namenode responsible for the subcluster. After it has identified the correct Namenode, the Router forwards the request to it and replies back to the client with the response. After that, the client can access the Datanodes directly.

The following sections detail the main components in our federation design.

3 Router

There can be multiple Routers in the system with soft state. Each Router has two roles:

1. Federated interface: expose a single, global Namenode interface to the clients and forward the requests to the active Namenode in the correct subcluster
2. Namenode heartbeat: maintain the information about a Namenode in the State Store

Next, we detail these roles and how the Routers manage availability and failures in implementing them. To close the section, we describe the Routers’ interfaces.

3.1 Federated interface

The Router receives a client request, checks the State Store for the correct subcluster, and forwards the request to the active Namenode of that subcluster. The reply from the Namenode then flows in the opposite direction. The Routers are stateless and can be behind a load balancer. For performance, the Router also caches remote mount table entries and the state of the subclusters. To make sure that changes have been propagated to all Routers, each Router heartbeats its state to the State Store.

3.1.1 Caching

The communications between the Routers and the State Store are cached (with timed expiration for freshness). This improves the performance of the system.

3.1.2 Quality of Service

If a particular name service is unavailable or slow, requests to other name services may also be slowed down. We provide QoS for each target name service according to its performance by supporting separate queues at the Routers for different Namenodes. This is similar to HADOOP-9640.

3.2 Namenode heartbeat

For this role, the Router periodically checks the state of a Namenode (usually on the same server) and reports their high availability (HA) state and load/space status to the State Store. Note that this is an optional role as a Router can be independent of any subcluster.

For performance with Namenode HA, the Router uses the high availability state information in the State Store to forward the request to the Namenode that is most likely to be active.

Note that this service can be embedded into the Namenode itself to simplify the operation.

3.3 Availability and fault tolerance

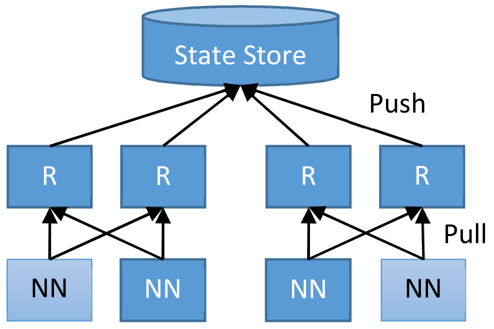
The Router has to be able to operate with failures at multiple levels.

3.3.1 Federated interface HA

The Routers are state-less and metadata operations are atomic at the Namenodes, if a Router becomes unavailable, any Router can take over for it. The clients configure their DFS HA client (*e.g.*, ConfiguredFailoverProvider or RequestHedgingProxyProvider) with all the Routers in the federation as endpoints.

3.3.2 Namenode heartbeat HA

For high availability and flexibility, multiple Routers can monitor the same Namenode and heartbeat the information to the State Store. This increases the resiliency to stale information if a Router fails. Conflicting Namenode information in the State Store is resolved by each Router via a quorum.



3.3.3 Unavailable Namenodes

If a Router tries to contact the active Namenode but is unable to do it, the Router will try the other Namenodes in the subcluster. It will first try those reported as standby and then the unavailable ones. Finally, if the Router cannot reach any Namenode, it throws an exception.

3.3.4 Expired Namenodes

If a Namenode heartbeat has not been recorded in the State Store for a multiple of the heartbeat interval, the monitoring Router will record that the Namenode has expired and no Routers will attempt to access it. If an updated heartbeat is subsequently recorded for the Namenode, the monitoring Router will restore the Namenode from the expired state.

3.3.5 Safe mode

If a Router cannot contact the State Store, it may wrongly provide accesses to out of date locations and get the federation into an inconsistent state. To prevent this, when a Router cannot connect to the State Store for a certain period, it enters into a special safe mode state (similar to the one for the Namenode). This mode throws Standby exceptions when the clients try to access data so they can try other Routers using the HA clients already provided in HDFS.

Similar to the Namenode, the Router stays in this safe mode until it is sure that the State Store recognizes it. This prevents inconsistencies when the Router is booting up. The other Routers assume that a Router is dead or in safe mode if it has not heartbeat for a certain period (*e.g.*, five times the heartbeat interval).

3.4 Interfaces

To interact with the users and the administrators, the Router exposes multiple interfaces.

3.4.1 RPC

It implements the most common interfaces for the clients to interact with HDFS. The current implementation is limited to analytics workload written in plain MapReduce, Spark, and Hive (on Tez, Spark, and MapReduce).

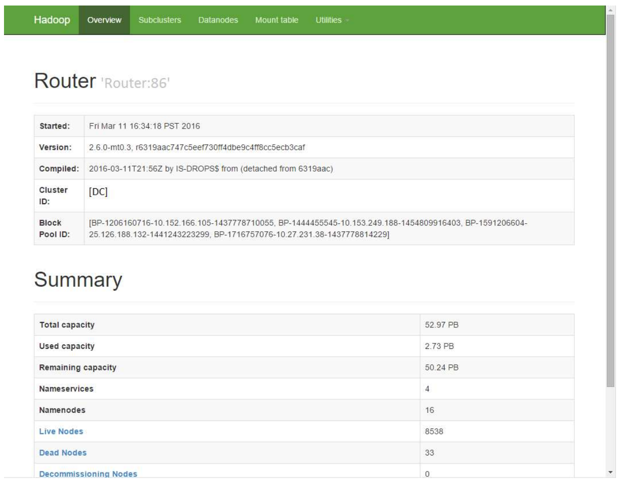
Some advanced functionalities like snapshotting, encryption and tiered storage will be left for future versions. All unimplemented functionalities will throw exceptions.

3.4.2 Admin

It implements an RPC interface for administrators. This includes getting information from the subclusters and adding/removing entries to the mount table. This interface is also exposed through the command line to get and modify information from the federation.

3.4.3 Web UI

It implements a Web UI for visualizing the state of the federation, mimicking the current Namenode UI. In addition, it includes the mount table, the membership information about each subcluster, and the status of the Routers.

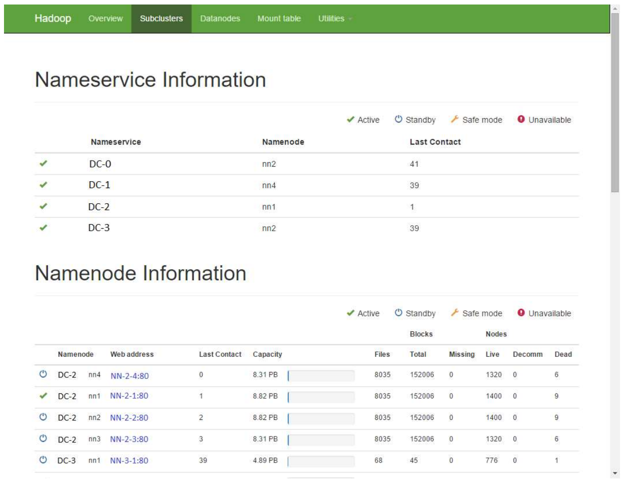


Router 'Router:86'

Started:	Fri Mar 11 16:34:18 PST 2016
Version:	2.6.0-mt0.3, r6319aac747c5eef730ff4dbe9c4ff8cc5ecb3caf
Compiled:	2016-03-11T21:56Z by IS-DROPS$ from (detached from 6319aac)
Cluster ID:	[DC]
Block Pool ID:	[BP-1206160716-10.152.166.105-1437778710055, BP-1444455545-10.153.249.188-1454809916403, BP-1591206604-25.126.188.132-1441243223299, BP-1716757076-10.27.231.38-1437778814229]

Summary

Total capacity	52.97 PB
Used capacity	2.73 PB
Remaining capacity	50.24 PB
Nameservices	4
Namenodes	16
Live Nodes	8538
Dead Nodes	33
Decommissioning Nodes	0



Nameservice Information

✓ Active ⏻ Standby Safe mode Unavailable

	Nameservice	Namenode	Last Contact
✓	DC-0	nn2	41
✓	DC-1	nn4	39
✓	DC-2	nn1	1
✓	DC-3	nn2	39

Namenode Information

✓ Active ⏻ Standby Safe mode Unavailable

	Namenode		Web address	Last Contact	Capacity	Files	Blocks		Nodes		
							Total	Missing	Live	Decomm	Dead
⏻	DC-2	nn4	NN-2-4:80	0	8.31 PB	8035	152006	0	1320	0	6
✓	DC-2	nn1	NN-2-1:80	1	8.82 PB	8035	152006	0	1400	0	9
⏻	DC-2	nn2	NN-2-2:80	2	8.82 PB	8035	152006	0	1400	0	9
⏻	DC-2	nn3	NN-2-3:80	3	8.31 PB	8035	152006	0	1320	0	6
⏻	DC-3	nn1	NN-3-1:80	39	4.89 PB	68	45	0	776	0	1

3.4.4 WebHDFS
It provides the HDFS REST interface (WebHDFS) in addition to the RPC one.

3.4.5 JMX
It exposes metrics through JMX mimicking the Namenode. This is used by the Web UI to get the cluster status.

3.4.6 Invalid operations
Some operations are not available in federation. The Router throws exceptions for those.

- Rename file/folder in two different nameservices
- Copy file/folder in two different nameservices
- Write into a file/folder being rebalanced

4 State Store
The (logically centralized, but physically distributed) State Store maintains four pieces of information:

1. The state of the subclusters in terms of their block access load, available disk space, HA state, etc.
2. The mapping between folder/files and subclusters, i.e. the remote mount table.
3. The state of the Rebalancer operations.
4. The state of the Routers in the federation.

The backend of the State Store is pluggable. We leverage the fault tolerance of the backend implementations.

Next, we describe the information stored in the State Store and its implementation.

4.1 Tables
The State Store comprises multiple data entities that need to be shared across the Routers.

4.1.1 Membership
The membership information reflects the state of the Namenodes in the federation. This includes information about the subcluster, such as the amount of storage and the number of nodes. The Router periodically heartbeats this information about one or multiple Namenodes.

These statistics are used for:

1. Getting the Active Namenode for each subcluster and redirecting queries.
2. Deciding which subcluster should host certain data (more details in Section 5).

Given that multiple Routers can monitor a single Namenode, we store the heartbeat from every Router and the Routers apply a quorum of the data when querying this information from the State Store. The Routers discard the entries older than a certain threshold (e.g., ten Router heartbeat periods) to avoid using data reported by already decommissioned Routers.

Format:

- Nameservice identifier
- Namenode identifier

- Router identifier
- Creation time
- Heartbeat time
- Starting time
- RPC address
- Web address
- State
- Safe mode
- Cluster identifier
- Blockpool identifier
- Total capacity
- Available capacity
- Number of blocks
- Number of missing blocks
- Number of files
- Number of life nodes
- Number of decommissioning nodes
- Number of dead nodes

4.1.2 Mount Table

This table hosts the mapping between folders and subclusters. It is similar to the mount table in ViewFS:

```
hdfs://tmp → hdfs://C0-1/tmp    /* Folder tmp is mapped to folder tmp in subcluster C0-1 */

hdfs://share → hdfs://C0-2/share

hdfs://user/user1 → hdfs://C0-3/user/user1

hdfs://user/user2 → hdfs://C0-2/user2
```

To allow rebalancing of data between subclusters, it supports locking. A locked mount table allows reads but prevents write operations. The Subcluster Rebalancer is in charge of locking a mount table entry (more details in Section 5). If a lock expires, any Router can remove it when updating its cache.

Format:

- Mount table entry
- Creation time
- Modification time
- Destination subcluster
- Destination path in subcluster
- Locked flag
- Lock expiration date

4.1.3 Subcluster Rebalancer Log

To provide fault tolerance in the subcluster rebalancing process (more details in Section 5), the Subcluster Rebalancer stores the state of rebalancing operations. For this, it creates an entry describing the work being done, a state, and a timestamp. It also keeps updating a heartbeat to mark that the operation is still ongoing.

Format:

- Operation identifier
- Creation time
- Modification time
- Heartbeat
- Operation
- Operation progress
- Source mount table entry
- Destination nameservice identifier
- Destination path
- Client identifier
- Operation result
- DistCP job identifier

4.1.4 Router State

To keep track of the state of the caches in the Routers and be able to rebalance safely, Routers store the fetched version of their cached tables in the State Store.

Format:

- Router RPC address
- Creation time
- Modification time
- Starting time
- Mount Table version
- Registration Table version
- State
- Version
- Compile information

4.1.5 Delegation tokens

To provide authentication, clients may request HDFS delegation tokens. These tokens need to be shared across all Routers and will permit access to all name services.

Format:

- User
- Issue Date
- Expiration Date
- Token

4.2 Implementations

The State Store implementation is pluggable to support the needs of different users:

- File-based: A file available through a network share. This is the implementation for basic testing.
- HDFS: This will be used for unit testing.
- SQL: YARN federation uses JDBC. The driver supports multiple implementations (e.g. MySQL or SQL Server).
- ZooKeeper: ZooKeeper to store tables and entries in znodes.

5 Subcluster Rebalancer

Thanks to the logically centralized mount table, we can easily rebalance the clusters and update the mount table. Subclusters can become unbalanced in three ways: (1) the RPC load they receive becomes widely different; (2) the amount of data they store becomes widely different; and/or (3) the number of entries in the mount table becomes widely different. We may also find (4) scenarios where too much load or high space requirements in a subcluster start to interfere with the primary tenants of the subcluster.

To avoid these issues, we support the rebalancing of the subclusters using a separate tool called the Subcluster Rebalancer. This Rebalancer moves data across subclusters (using DistCP) and updates the State Store. The rebalancing does not have to be at mount point level.

We only allow the Subcluster Rebalancer to make changes to the mount table, e.g. rename operations initiated by other clients that move data between subclusters will be disallowed. We also assume that all the client accesses are done through the Routers and no direct access will be done to the Namenodes.

Next, we describe the main characteristics of the Rebalancer, closing the section with a summary of its operation.

5.1 DistCP

The Rebalancer uses DistCp to move data between subclusters. For folder moves that fail to complete, DistCP supports updating a partially copied folder and verifying that the copy is completed successfully. The Rebalancer monitors the progress and the errors of the DistCp job.

5.2 Writes while rebalancing

If clients were allowed to modify the original data while the Rebalancer is moving it to another subcluster, the Rebalancer would need to keep updating the data at the destination subcluster, until there are no more modifications. To overcome this problem, the Rebalancer supports two improvements that we describe next.

5.2.1 Precondition

To reduce the chance that clients will want to modify data during rebalancing, we add a condition that checks if there have been writes or leases in the path to move in a given period (e.g., last hour). This condition can be disabled (e.g., when administrators want to rebalance the cluster manually).

In addition, a Rebalancer cannot rebalance a locked entry.

5.2.2 Locking and updating a mount table entry

To prevent modifications, the Rebalancer can also lock write accesses to a mount table entry. This information is read by the Routers, which will throw exceptions when a client wants to perform a write operation on a locked mount entry. This exception will disallow the DFSClient from retrying.

After the Rebalancer has moved the data between subclusters, it will update the Mount Table in the State Store. Once the data is moved, when a Router accesses a location that has been removed, it will recheck with the State Store in case of FileNotFoundException. Alternatively, to guarantee that all the clients have a consistent view, the Rebalancer waits until all the Routers have updated their local caches (this information is available in the Routers table of the State Store). This is an option when rebalancing.

5.3 Fault tolerance

The rebalancing can crash in any of the stages but will always leave the federation in a consistent state that can be recovered at any point. The Subcluster Rebalancer stores the state of each operation in the State Store to support fault tolerance. Each of the stages of the Rebalancer updates this state. Within each stage, the Rebalancer will keep a heartbeat field in the log entry to mark that is still ongoing.

The Rebalancer first checks if a prior rebalancing operation has failed (by checking the last update timestamp) and cleans-up/restarts the process.

The locks on the Mount Table have expiration dates and the Subcluster Rebalancer will periodically refresh them. If the Rebalancer dies, any Router reading the Mount Table will remove the lock.

5.4 Limits

To prevent very long rebalancing operations, we define limits for both size and number of files to move in each operation. If a data movement is too large, the Rebalancer will fail and inform the administrator.

5.5 Special cases

The Rebalancer takes into account additional features when rebalancing: snapshots, quotas, safe mode, multiple block pools per Datanode, and modifying the filesystem (e.g., change the name of the parent or deleting files in the rebalanced path).

5.6 Stages of a rebalancing operation

The Rebalancer performs the following actions to move a folder/file from a subcluster to another:

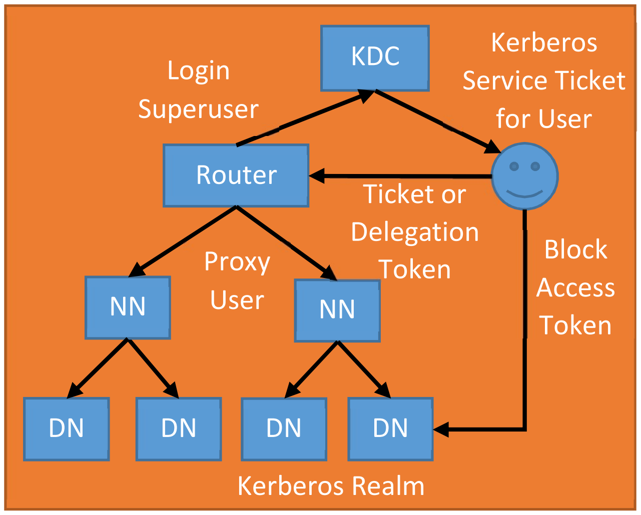
1. Reserve the mount point for rebalancing in the Rebalancer Log
   a. Fail if it is already reserved or any path above it is reserved
   b. Fail if there are sub mount points in the table beneath the item
2. Check if the file tree beneath the mount point has been recently modified
3. DistCP to the target name service path
4. Verify that DistCP was successful
5. If the mount point is a child of an existing mount point, create a new mount point in the Mount Table pointing to the source nameservice path
6. Lock the mount point in the Mount Table to prevent writes to the source nameservice path until the Mount Table is updated and synchronized to reflect the target nameservice path.
7. Wait for all Routers to receive and acknowledge the mount point lock
8. Change the destination of the mount point to the target nameservice path in the Mount Table
9. Wait for all Routers to receive and acknowledge the Mount Table update (optional to prevent READ failures after deleting old files)
10. Verify that no files or directories on the old nameservice path have been modified, and fail otherwise
11. Delete files from the old name service path (we can wait a period to let previous reads succeed)

12. Release the lock on the mount point in the Mount Table
13. Release the mount point for future rebalancing in the Rebalancer Log

6 Security

Security for Federated HDFS is compatible with the authentication and authorization in standard HDFS. In secure mode, both the Routers and Datanodes must be configured for secure authentication. The Namenode may optionally be configured in secure mode, but this is not required if clients are restricted from accessing the Namenodes directly. It is recommended to prevent direct client access to the Namenodes to avoid inconsistencies during rebalancing and changes to the mount table.

HDFS clients authenticate directly with Routers in the same manner they currently do with a Namenode. The Router is responsible for properly authenticating the user and proxying/impersonating the authenticated user in subsequent communication with a Namenode. This is the approach used in other components (e.g., HttpFs).



6.1 Router-Client authentication

In secure mode, HDFS clients directly authenticate with the Router, reusing the same authentication support that is used to authenticate clients to a Namenode. The initial authentication of a user relies on Kerberos and subsequent authentications may use delegation tokens.

Admin clients will authenticate with the HTTP web interface using SPNEGO in the same manner as a user authenticates with a traditional HDFS Namenode HTTP web interface. The user must supply their credentials via an HTTP client before access is allowed to secure web interfaces.

6.1.1 Kerberos

Each Router client IPC interface is configured with an appropriate SPN (service principal name) for Kerberos authentication. If possible, the Router will use the same SPN as the Namenode client protocol for compatibility. Clients authenticate via Kerberos directly with the Router by passing a valid Kerberos service ticket along with their request (this is the same approach for the Namenode).

6.1.2 Delegation tokens

After authenticating with the Router IPC interface via Kerberos, a client may request an HDFS delegation token. The API for requesting a token is identical to the corresponding Namenode API. The delegation tokens are stored in the State Store. Routers track the expiration of each token and require all secure users to provide either a valid delegation token or a valid Kerberos service ticket during authentication.

6.1.3 SPNEGO

Where possible, the Router will leverage the same web server authentication utilized by the Namenode web interfaces. SPNEGO is only available for HTTP server based authentication.

6.2 Router-Namenode authentication

The Router impersonates users to the Namenode via the existing Hadoop proxy user privilege. The router is trusted to handle all perimeter authentication. All client requests are authenticated only in the Router and are mapped to an appropriate HDFS username via existing security support.

The Namenodes may be optionally configured in secure mode. In secure mode, the router must be able to log in as a principal within the Kerberos realm. The Namenode must also grant proxy user permission to the Router's user account. A rogue router will not be able to authenticate as the privileged proxy user and will not be able to impersonate users to the Namenode.

The first version of the Router is configurable in either secure (authenticates clients) or insecure mode. If the cluster has a mixture of insecure and secure Namenodes, the Router will not allow a mixture both authenticated and unauthenticated clients. Future versions of the Router may be able to selectively enforce/ignore client authentication based on the target NN.

6.3 Datanode-Client authentication

Relies on block access tokens granted by the managing Namenode. This is unchanged from classic HDFS. A user first authenticates with a Router, the Router proxies the request to the appropriate Namenode, and it returns a block access token to the client. The client can use the block access token to access the Datanode directly.

6.4 Authorization

Authorization uses the existing HDFS POSIX-like file permissions. All client requests to a Router are proxied to the appropriate Namenode as the calling user. The Namenode is responsible for authorizing access to the filesystem and granting the appropriate access or block access tokens. In most cases, the Router does not handle any file system-based permissions.

6.5 Other Hadoop gateways

The Apache Knox gateway uses a similar mechanism for secure authentication with a Namenode. In a Knox cluster, the Knox gateway handles perimeter authentication of a user and impersonates the mapped HDFS user to the Namenode using a privileged proxy user Kerberos principal.

The Router can also be installed behind Knox or a similar gateway, allowing additional perimeter authentication protocols other than Kerberos. Note that when using a Knox gateway, only REST/HTTP client endpoints are exposed to users.

Other Hadoop tools, such as Oozie and HttpFS, also handle perimeter authentication themselves. Both the Router and these additional gateways utilize either secure (authenticated) or insecure Hadoop proxy users to impersonate clients within the cluster.

7 Future work

The previous sections describe a first version of the Router-based HDFS federation. We leave a few non-critical features for future releases.

7.1 Quotas

Administrators can define a maximum quota for a particular folder. In a federated environment, a folder can be spread across multiple subclusters. For this reason, we need to consider a global quota for these folders. We can solve this in two ways:

1. Split the quota across each subcluster
2. Maintain a table in the State Store with the quotas and enforce the global values

7.2 Snapshots

Users can ask for a snapshot of a folder and this folder may be spread across multiple subclusters. For this reason, when a user creates a snapshot, the Router will need to create these snapshots in each subcluster in a synchronized way to avoid inconsistencies. In addition, it will maintain a global table in the State Store.

7.3 Trash

When a user deletes a file, she can set to move it to a temporary file trash first and the Namenode will remove this data after a configurable time has passed. As this feature requires moving data from a path to a special location, it might require moving data between subclusters which is initially disallowed. For this reason, we need to implement a special syntax that would allow moving data to a special location without moving data between subclusters.

8 Configuration

The configuration for Router-based federation is done using hdfs-site.xml. The following subsections summarize the configuration parameters.

8.1 Router

To configure both (1) federation interface and (2) in the Router, we use the following parameters.

Name	Value	Description
dfs.router.rpc-address		RPC address interface
dfs.router.rpc-bind-host	0.0.0.0	RPC address to bind to
dfs.router.http-address		HTTP address interface
dfs.router.http-bind-host	0.0.0.0	HTTP address to bind to
dfs.router.https-address		HTTPS address interface
dfs.router.https-bind-host	0.0.0.0	HTTPS address to bind to
dfs.router.admin-address		Admin address interface
dfs.router.admin-bind-host	0.0.0.0	Admin address to bind to

dfs.router.handler.count	10	Number of server threads
dfs.router.heartbeat.interval	3	Heartbeat interval to the State Store
dfs.router.cache.ttl	10	How often the Router gets fresh data from the State Store
dfs.router.default.nameserviceId	${fs.defaultFs}	Default nameservice to use if nothing specified in the mount table
dfs.router.monitor.namenode	localhost	Comma-separated list of Namenodes to heartbeat to the State Store. If empty, there is no heartbeating.

8.2 State Store

To configure the connection with the State Store from the Router, we use the following parameters:

Name	**Value**	**Description**
dfs.federation.statestore.client.class	org.apache.hadoop.hdfs. server.federation.statestore. impl.StateStoreSql	Client class for the State Store (e.g., SQL, File, ZooKeeper).
dfs.federation.statestore.mount.limit.size		The maximum number of entries in the mount table.
dfs.federation.statestore.sql.max-connections	100	Max number of connections to the DB.
dfs.federation.statestore.sql.url		URL of the SQL DB.
dfs.federation.statestore.sql.username		Username for the SQL DB.
dfs.federation.statestore.sql.password		Password for the SQL DB.
dfs.federation.statestore.sql.jdbc-class		Class for the JDBC driver.
dfs.federation.statestore.zk.address		Host:port of the ZooKeeper server.
dfs.federation.statestore.zk.num-retries	1000	Number of tries to connect to ZK.
dfs.federation.statestore.zk.retry-interval	1000	Retry interval in milliseconds.
dfs.federation.statestore.zk.timeout	10000	ZooKeeper session timeout in milliseconds.
dfs.federation.statestore.zk.path	/dfs-federation-statestore	Full path of the ZooKeeper znode to store the State Store.
dfs.federation.statestore.zk.acl	world:anyone:rwcda	ACL's to be used for ZooKeeper znodes.
dfs.federation.statestore.		

9 Example setup

Router-based federation allows a wide variety of configurations. This section provides an example of an advanced setup.

9.1 Routers and Namenodes

In our deployment, we have 4 subclusters with 4 Namenodes in High Availability in each subcluster (a total of 16 Namenodes in the cluster). Each machine with a Namenode hosts a Router and each Router heartbeats to the State Store the state of the 4 Namenodes in the subcluster.

9.2 Clients

Direct accesses to the Namenodes from the clients is disabled. There are two options for the clients:

1. Use a load balancer across the 16 Routers and connect to any of them
2. Configure the ConfiguredFailoverProxyProvider with the 16 Routers and let the HA client select

9.3 Subcluster Rebalancer

When the administrator identifies that a cluster is overloaded, she can start a rebalancing process specifying the mount entry to move and the new destination subcluster.

10 Other approaches

Before deciding in favor of the Router-based design we describe above, we considered other approaches to federation and rebalancing. Next, we overview a couple of them and their drawbacks.

10.1 ViewFs

ViewFS (HADOOP-7257) is a client-side solution that exposes multiple subclusters under a single namespace. ViewFS is implemented using a local file that contains a mount table, which the client reads before issuing the request to the right subcluster. This approach is used in companies like Twitter. The main problems with this solution are:

1. Maintaining an up-to-date configuration with the mount points across clients
2. Balancing the namespace between subclusters efficiently

There are efforts like Nfly (HADOOP-12077) to extend ViewFs and allow the same file in multiple subclusters. We can leverage the same solution in our Router-based federation solution.

10.2 Namenode split

Currently the Namenode has two roles that are well separated:

1. Mapping between files and blocks
2. Mapping between blocks and machines

The solution would be to split these into two different services. We would have subclusters with Datanodes heartbeating to a subcluster Block Manager (#2), which would heartbeat to a global Namespace Manager (#1). The Namespace Manager could be distributed using a DHT or similar.

This would be the ideal solution but the amount of development and changes to the Namenode would be substantial.