Large-Scale Multi-Tenant Inotify Service

Problem Statement

As mentioned in inotify design document, some applications rely on the existence of HDFS files to manage their workflow. For example, they use the existence of _SUCCESS files generated by MR framework to decide if the previous step in the workflow has completed and if so continue to the next step. These _SUCCESS files share some common path glob pattern for example, /user/joe/workflow/YYYY/MM/DD/_SCUCCESS.

Currently applications keep calling HDFS NN to check if these files exist or not. These RPC calls generate lots of read requests on NN. Given NN uses a global read write lock for namespace access and block management, these RPC requests have major impact on NN performance.

Besides the above scenario, some HDFS applications need to get file attributes of the same files/directories repeatedly. MAPREDUCE-4907 and YARN-1771 addressed the problem by caching the FileStatus at MR and YARN layers. Even with these improvements, there are still lots of requests sent to NN due to the limited scope of the cache.

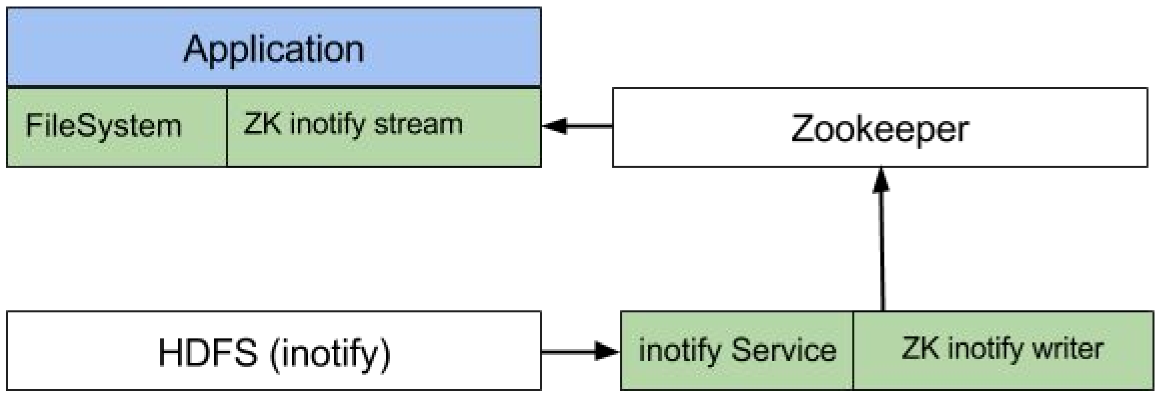
These scenarios share a common pattern where **a large number of clients (10k) need to regularly get file status of infrequently updated HDFS files (updated every 10 minutes).**

Design goals

HDFS-6634 provides the core inotify functionality. We would like to extend that to achieve the following goals based on scenarios described above.

- Ease of use for applications. For example, it will be good if we can reuse the existing inotify API.
- Security support. The solution shouldn't lower the security level by allowing users to access file status disallowed at HDFS level.
- Scalability assumption. As the number of applications/path patterns increase, the system should be able to handle the load. However, in the initial version, we don't plan to build a service that allows any number of clients subscribe to changes to any number of files/directories. The following constraints are based on the production use cases. They are somewhat high level estimates to emphasize we are not talking about 1M concurrent clients trying to subscribe to 100M files/directories changes.
    - The number of concurrent clients subscribing to the service is 10k max.
    - The files/directories are grouped into path blob patterns. The max number of path glob patterns is 1k.
    - The aggregated number of inotify events that match any path blob pattern is in the order of 10/s.

Design Overview



As outlined the the inotify design document, the existing HDFS inotify has some limitations.

- Only admins can use the functionality.
- inotify at ClientProtocol level requires regular polling. thus large-scale clients create lots of connections to namenode and thus put high pressure on NN and edit log tailing.

To address these limitations, we plan to develop an end-to-end notification service based on zookeeper. A new daemon service pulls NN edits using the existing HDFS inotify API and filters the inotify events that match predefined path patterns and publish the filtered events to ZK. Applications create `ZKInotifyEventInputStream` to pull inotify events that belong to a specific path filter. `ZKInotifyEventInputStream` receives the inotify events using ZK watcher functionality.

Security and path filter management

To address the security issue, each path filter have its owner and ACLs or allowed subscribers, separated from the HDFS directories/files security. This greatly simplifies how security is managed and eliminates the issues mentioned in the original inotify design document.

The creation and removal of a path filter has to be done by admins. There is an owner for each path filter. There is a also a list of users that are allowed to subscribe to the changes of this filter. The owner can modify the list of allowed users after the path filter has been created.

Using owner and ACLs instead of `HADOOP FsPermission` object makes it more consistent with ZK security model which doesn't have Linux group concept. Although both HDFS and ZK support ACL, they use different data structures. So simplify the interoperability, it just defines a list of users that are allowed to subscribe to the events.

The path filter spec will be persisted to NN edit log. New RPC methods are added to ClientProtocol to allow admins to configure this. The reason we only allow admins to manage path filter is to make sure the system won't accidentally overwhelm ZK in the following scenarios.

- Number of path filters. inotify service will create a znode for each path filter under a common parent. If application creates too many path filters, it will create too many znodes and eventually exceed the max size of the parent znode.
- If a user accidentally creates a path filter that is too general like /user/*/*; inotify service could generate lots of write requests to zk.

Upon startup, inotify service gets the list of path filters from HDFS and set up the proper ACLs on the corresponding filter znodes so that only authorized users can read the znode.

```
public class INotifyPathFilterInfo {
    private String name;
    private String pathGlob;
    private String owner;
    // the users that are allowed to subscribe to the notification
    private List<String> allowedSubscribers;
};

public ClientProtocol {
…
  public RemoteIterator<INotifyPathFilterInfo> listPathFilterInfo()
throws IOException;

  void public addPathFilterInfo(INotifyPathFilterInfo
pathFilterInfo);
  void public modifyPathFilterInfo(INotifyPathFilterInfo
pathFilterInfo);
  void public removePathFilterInfo(INotifyPathFilterInfo
pathFilterInfo);
};

// The following command add a filter named joeSUCCESSFiles
dfsadmin -addInotifyPathFilter -pathGlob /user/joe/*/*/_SUCCESS -name
joeSUCCESSFiles -owner joe
```

ZK scheme & Notification mechanism

There is a znode for each defined path filter in each namespace. inotify service reads all edits from NN and applies filter matching. After it gets a matching inotify event, it will update the znode for that path filter.

The znode structure

"`/HDFSInotifyRoot/$HDFSNameSpace/$PathFilterName/$eventId`". `$HDFSNameSpace` is used to distinguish different namespaces used in federation. FilterName is the name of the filter. eventId is the txid of NN edit. The value of znode "`/HDFSInotifyRoot/$HDFSNameSpace/$PathFilterName`" will be the latest txid received so far.

Filters don't have to be exclusive. So you can have an inotify event that matches more than one filter.

For example, suppose the cluster has two namespaces, ns1 and ns2, ns1 has "filter1" and "filter2"; ns2 has "filter3" and "filter4". filter1 has receive txid 5, 9, 15, 28 so far. filter2 has received txid 20, 28, 30 so far.

```
/HDFSInotifyRoot/ns1/filter1/5
                            /9
                            /15
                            /28
/HDFSInotifyRoot/ns1/filter2/20
                            /28
                            /30
/HDFSInotifyRoot/ns2/filter3
/HDFSInotifyRoot/ns2/filter4
```

Limit on the number of inotify events stored in ZK

To make sure the number of edits under path pattern znode don't grow too big, the system defines a max number of txids `inotify.max.zk.txid.count.per.filter` that can be stored in zk for a specific path filter. inotify service uses zk multiop feature to achieve this. The following operations will be performed together atomically.

- Add a new txid node under path filter znode.
- Remove the oldest txid znode.

Using the above example, let us assume the system is configured `inotify.max.zk.txid.count.per.filter` to 4. When a new txid 38 is being added, the old txid 5 will be removed. After that, the tree will be:

```
/HDFSInotifyRoot/ns1/filter1/9
                            /15
                            /28
                            /38
```

Management on the client side

An application creates `ZKInotifyEventInputStream` with the specific path filter to start receiving inotify events.

```
InotifyEventInputStream inotifyStream = new
ZKInotifyEventInputStream(conf, “joeSUCCESSFiles”);
```

`ZKInotifyEventInputStream` uses ZK curator to manage the interaction with ZK service. `ZKInotifyEventInputStream` initializes its internal cache by reading the list of inotify event znodes from ZK. It uses ZK watcher to detect any change made to the path filter znode. After it receives ZK `NodeChildrenChanged` event, it will read all the inotify event children znodes from ZK and compare them with its local cache to identify new inotify events. Note that this approach should handle the following situation: “After a client receives a ZK notification due to inotify event A, it is possible a new inotify event B has been written to ZK before the client calls back to ZK to get the children and re-add the watcher”.

Given there is max number of `inotify.max.zk.txid.count.per.filter` inotify event znodes stored in ZK, it is possible a client isn’t fast enough to consume old inotify events before they are removed by inotify event. To handle this scenario:

- `ZKInotifyEventInputStream` can detect inotify events gap by comparing its local cache and the latest list from ZK. If there is no overlap, it indicates event loss.
- Once it identifies event loss, it throws `MissingEventsException` exception to the application so that application can take proactive action by pulling from HDFS directly and recreating `ZKInotifyEventInputStream` stream.

Federation Support

There is one inotify service instance for each namespace. The inotify service runs on the same machine as the NNs for that namespace, but it can be configured to run on other machine.

On the client side, given applications use ViewFs to access federated namespaces, it means we need to expose inotify functionality at FileSystem level, that includes defining a new `InotifyEventInputStream` interface and moving common inotify classes from hadoop-hdfs-project to hadoop-common-project. That should be ok given inotify's EventBatch structure provides metadata that are already defined at FileSystem level.

```
public class FileSystem {
…
public InotifyEventInputStream InotifyEventInputStream
getInotifyEventStream();
}
```

ZK scalability & write throttling

Although the design goals define the the scope of scalability, we still need to make sure the system can handle the high load gracefully in case of misconfiguration or abusive users. THis is especially true if you are using a shared ZK service. There are some scalability and performance issues we need to address when using ZK as notification channels.

- # of watchers for the whole cluster. There is a limit on the total number of watchers for the whole ZK cluster. If it exceeds the limit, ZK can run out of java heap space and could impact other ZK applications.
- # of watchers for a given ZK session. Large number of watchers on a given ZK session can cause the packet size to exceed the default max 1MB in the case of session reconnection. In addition, this might create congestion at ZK if clients simultaneously read data from ZK after they receive notifications.
- # of children nodes. If a given znode has too many children znodes, it can also exceed the default max 1MB packet size.
- High volume of ZK write operations. If there are too many updates for a given path filter, it can impact the overall ZK performance.

We address these issues in the following ways.

- Mitigate the issue by enforcing ACL on ZK node so that only authorized users can set watcher on the path filter znode.
- Mitigate the issue by only allowing admins to create new path filters and thus prevent too many path filters from being created.
- `inotify.max.zk.txid.count.per.filter` controls the max number of inotify events stored under a given path filter znode.
- Have `ZKInotifyEventInputStream` backoff upon receiving zk notification before issuing read to zk.
- inotify service uses a single thread to update ZK. This limits the ZK write volume and should be enough to mitigate the issue.

If the rate is higher than the global config value for `inotify.max.write.ops.sec`, inotify service can throttle the ZK write operation by buffering the events in memory; if the buffer is full, it will drop the events and write a special event to ZK so that applications know about it.

Besides mitigating scalability issue at inotify service level, we can also scale ZK accordingly.

- Deploy more ZK observer nodes to support more watchers.
- Partition inotify events across different ZK clusters. For example, we can have one ZK cluster responsible for one namespace.

inotify service availability

To handle failures and maintenance, inotify service needs to be highly available by achieving the following functionalities:

- It uses ZK for leader election.
- To support fencing around writing to ZK, it use ZK's "multi" feature.
- Regular checkpoint of txid to ZK so that in the case of failover and restart the new active inotify service can resume from the proper txid. Note that it shouldn't checkpoint for every txid read and it will create high volume of write operation to ZK. It doesn't need to do that either from correctness point of view as replay of the inotify events that were previously written to ZK end up writing the same event to ZK.

Extensibility of notification channel

Besides ZK and current regular polling, we might want to experiment other notification channels. We define interfaces to abstract the notification channel.

On the event producer side, each notification channel provides a subclass of `InotifyEventOutputStream`. inotify service will create the stream based on the configuration `dfs.inotify.outputstream.classname`.

```
// stream that writes inotify events to ZK
public class ZKInotifyEventOutputStream implements
InotifyEventOutputStream {
...
}
```

On the event consumer side, each notification channel provides a subclass of `InotifyEventInputStream`. DFSClient will create the stream based on the configuration `dfs.inotify.inputstream.classname`.

```
// stream that pulls inotify events from ClientProtocol
public class DFSInotifyEventInputStream implements
InotifyEventInputStream {
...
}

// stream that receives notification from ZK
public class ZKInotifyEventInputStream implements
InotifyEventInputStream {
...
}
```

Error handlings

Application will receive `MissingEventsException` exception in the following scenarios and should close the `InotifyEventInputStream` stream and retry again.

- ZK outage.
- inotify service outage. This can be detected by having inotify service registering an ephemeral node in ZK.
- inotify service drops events due to extremely high volume of NN edit rates than the rate allowed for ZK write operation.
- `ZKInotifyEventInputStream` processes inotify events at slower rate than the event creation rate.

Other considerations

Why not have clients directly pull from namenode/inotify service?

We could consider having clients directly pulling namenode/inotify service like how DFSInotifyEventInputStream's regular pulling namenode via ClientProtocol#getEditsFromTxid. But that will generate lots of unnecessary calls. ZK enables push model and makes the overall system more efficient.

We can also consider long pull, where RPC calls to namenode/inotify service side won't return until new events that match path filters are available. When a large number of clients connect the service and there isn't much relevant updates, it means large number of connections in namenode/inotify service and all of RPC handler threads are in blocking state; thus make the service unusable.

Why not use kafka or other pub-sub systems?

Dependency is the main reason. HDFS has existing dependency on ZK (although not at large scale as this design proposes.) but not other systems. In addition, for our scenario (the low write volume and need for a large number of notifications) is a good use case for ZK.

In addition, kafka also depends on ZK to track consumer offset. In this broadcast scenario, kafka might end up creating the same ZK scalability issues.

The design provides abstraction to support different pub-sub modules. So if we find a more suitable system later, we can replace ZK with the new system without any change to the rest.

Why not persist the filter definition to ZK or local file systems?

Not storing filter definition in ZK simplifies the management; the system can bootstrap with a clean ZK instance.

In addition, to evaluate other non-ZK system. storing filter definition outside the actual notification delivery channel allows us to test other non-ZK systems easily.
Another option is to store the filter configuration on the local machine that inotify service run on. However, to support HA functionality, we will need to make sure these local files are in sync.

Why not run inotify service as a global service that pulls edit logs from different namespaces?

It should be ok performance wise. It can use one thread to pull from one namespace. Assuming the number of namespaces is < 5 and the max network traffic from edit logs is around 1MB/sec per namespace, a single inotify service should be able to handle all namespaces.

However, for cross-datacenter scenario, we still need to set up one inotify service for each DC to eliminate the cross DC traffic between NN and inotify service as well as between inotify service and ZK.

Running inotify service on the same machine as nn also gives us the option to move inotify functionality completely out of namenode in the future. In addition, it provides the option for inotify service to read the edits stored on the local nn machine.

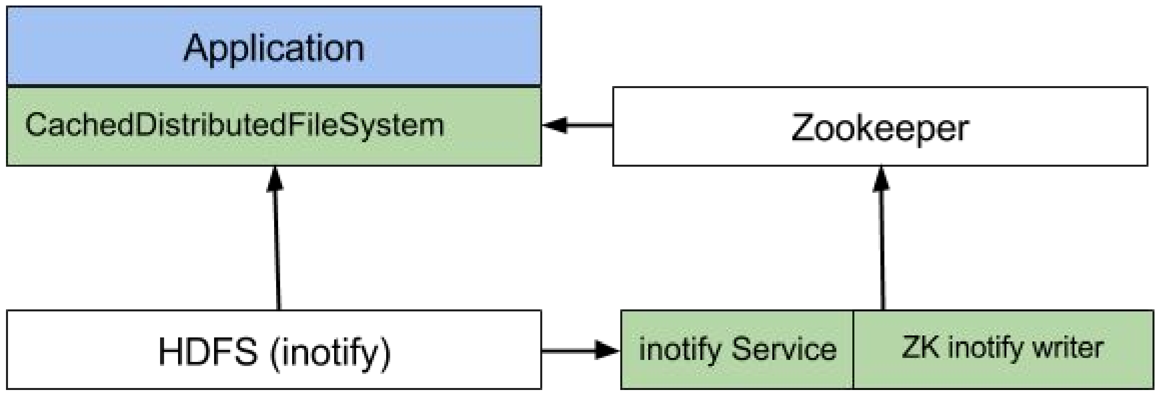
Why not do the filtering inside namenode?

inotify service needs to filter the inotify events and update external notification system such as ZK based on the name of the match filter. Thus it is better to do the filtering and notification to external system outside active namenode to reduce dependency active namenode might have on other systems.

Alternative Designs

There are couple other alternative designs where applications continue to use existing getFileInfo API to pull `FileStatus`. But instead of acquiring NN lock and getting the data from NN namespace, it reads the data from the cache. The cache are updated based on inotify events.

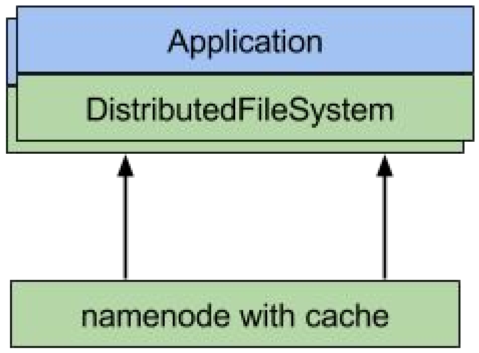
Cached DistributedFileSystem



The basic idea is to build a client-side hadoop FileSystem that can cache FileStatus value of each HDFS file which matches certain HDFS path patterns specified by the applications. It uses ZK to keep the cache up to date. If it can’t receive notification from ZK properly, it will pull from HDFS.

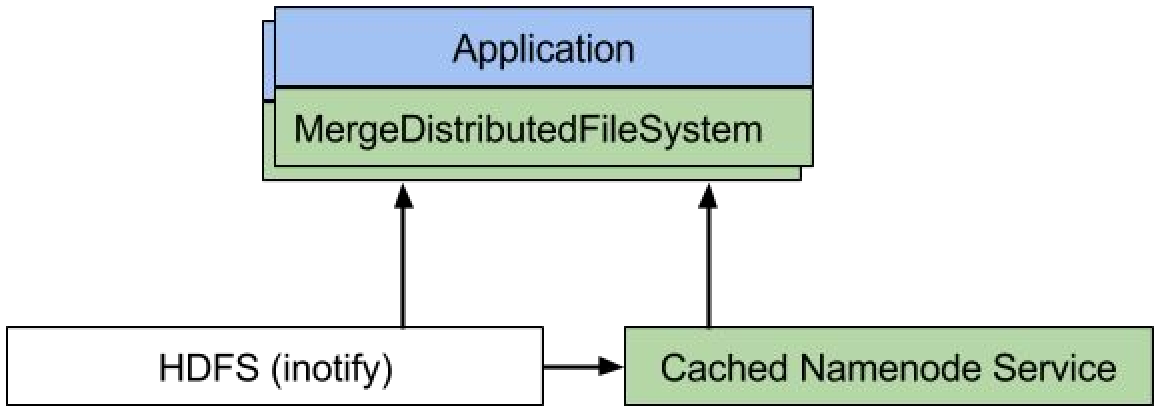
The main difference between this design and the main design how applications use it. In the main design, inotify functionality is provided directly to applications and it is applications' responsibility to recheck with HDFS in case of inotify error.

Native namenode metadata caching support



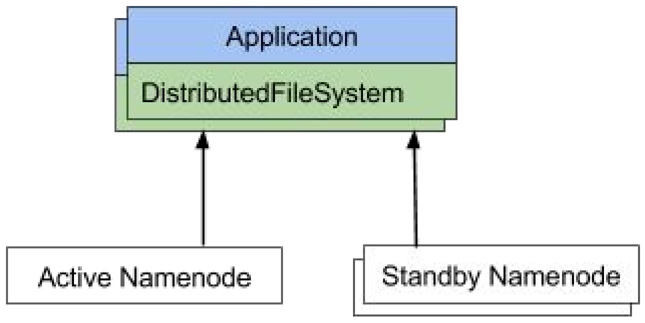
In this design, namenode will cache `HdfsFileStatus` of these files/directories separately from its FSDirectory structure. When applications ask for `HdfsFileStatus` of these files/directories, namenode will check with the cache without taking its global lock.

Cache service side-by-side with HDFS



Compared to the design of "native namenode metadata caching support", this design will host the cache as a separate service. It still uses HDFS inotify to keep its cache up-to-date. The service implements namenode RPC protocol. Essentially it offloads heavy idempotent read operations from active NNs.

Read from standby NNs

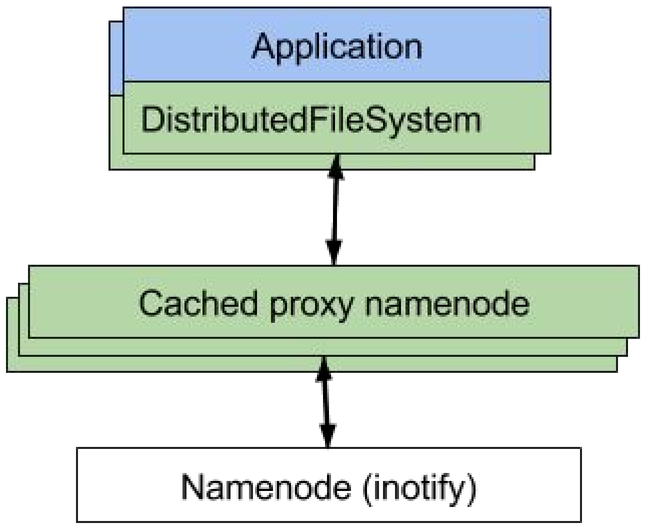


Standby NNs already have slightly delayed NN namespace information; but “read-from-standby” functionality is disabled. We can have clients keep polling standby NNs, just like how they are polling from active NNs today.

Essentially we are moving the load from active NNs to standby NNs and keep the current regular polling pattern without the need of notification mechanism. Standby NNs still use global locks just like active NNs. But the overall load on standby NNs is less than active NNs and can be used to handle the extra polling.

HDFS-6440 adds support to run more than one standby namenode. We can scale out the number of standby namenodes if necessary.

Cache service with proxy functionality on the top of HDFS



A caching service sits between client and HDFS namenode. It functions as a proxy and implements HDFS protocol. It uses NameNodeProxies (used by DFSClient) to connect to the actual namenodes. NameNodeProxies take care of RPC retry.

Each instance of the cache service will cache the state of HDFS files/directories that match the path patterns. Each instance uses HDFS inotify to keep its cache up-to-date. For all write operations and files that don’t match the path patterns, it will forward the call to actual namenodes.