Inotify v1 Goals and Design

inotify Background and Use Cases

HDFS-1742 and HDFS-1179 both suggest the need for a mechanism for external applications to keep track of edits to the filesystem state.

In Linux, the inotify API (http://www.linuxjournal.com/article/8478?page=0,0) provides this functionality. In short, inotify provides each application a select-, poll-, epoll-, and read-able file descriptor from which it can read about updates to the directories or files it is watching.

The canonical use case for inotify is a desktop search tool. The tool can avoid rescanning the entire filesystem periodically for changes (so that it can reindex changed or added files) by simply listening for changes using inotify.

The potential use cases for inotify in HDFS seem to be a superset of the Linux use cases. Beyond search, HDFS inotify could support the design of a tool that exposes filesystem events to cluster administrators. It could also allow Hive to learn when new files are added to table directories and take appropriate action. Other use cases described in this talk (http://www.youtube.com/watch?v=7KumMKqBtr8) include a external tool that replicates files in HDFS to a mirror cluster (it listens for added files and then copies them to the mirror) and Oozie job chaining (OOZIE-599, a job can listen for the creation of the output file for the previous job).

Reason for Integration with HDFS

The aforementioned talk (http://www.youtube.com/watch?v=7KumMKqBtr8) describes a system external to HDFS that scans finalized and in-progress edits files for edits. As noted in the Q&A (starting around 25:30), the maintainers of this system will need to update it whenever the format of the edits file changes. In addition, it would be much easier if HDFS could directly notify the system of edits, eliminating the need for periodic polling of the edits file (a fairly complex task, since the edits file is rolled periodically as well). These problems are solved if inotify becomes a feature of HDFS.

Design Overview

In the current design, clients must have superuser privileges. The inotify client code keeps track of the highest transaction ID TX_MAX for which it has received events. It pulls new events from the NameNode by issuing an RPC with TX_MAX, and the NN responds with events that reflect transactions with IDs greater than TX_MAX. We send events that reflect the state changes we believe clients would be most interested in; not all `FSEditLogOps` have a corresponding inotify event. In particular, we use the following events:

```
enum INodeType {
  I_TYPE_FILE = 0x0;
  I_TYPE_DIRECTORY = 0x1;
  I_TYPE_SYMLINK = 0x2;
}

enum MetadataUpdateType {
  META_TYPE_TIMES = 0x0;
  META_TYPE_REPLICATION = 0x1;
  META_TYPE_OWNER = 0x2;
  META_TYPE_PERMS = 0x3;
  META_TYPE_ACLS = 0x4;
  META_TYPE_XATTRS = 0x5;
}

message CreateEventProto {
  required INodeType type = 1;
  required string path = 2;
  required int64 ctime = 3;
  required string ownerName = 4;
  required string groupName = 5;
  required FsPermissionProto perms = 6;
  optional int32 replication = 7;
  optional string symlinkTarget = 8;
}

message CloseEventProto {
  required string path = 1;
  required int64 fileSize = 2;
  required int64 timestamp = 3;
}

message ReopenEventProto {
  required string path = 1;
}

message RenameEventProto {
  required string srcPath = 1;
  required string destPath = 2;
  required int64 timestamp = 3;
}

message MetadataUpdateEventProto {
  required string path = 1;
  required MetadataUpdateType type = 2;
  optional int64 mtime = 3;
  optional int64 atime = 4;
  optional int32 replication = 5;
  optional string ownerName = 6;
  optional string groupName = 7;
  optional FsPermissionProto perms = 8;
  repeated AclEntryProto acls = 9;
  repeated XAttrProto xAttrs = 10;
  optional bool xAttrsRemoved = 11;
}
```

```
message UnlinkEventProto {
  required string path = 1;
  required int64 timestamp = 2;
}

message EventsListProto {
  repeated EventProto events = 1;
  required int64 lastTxid = 2;
}
```

The NN calls `FSEditLog.selectInputStreams()` to read transactions with IDs greater than the client's specified TX_MAX from its journals and converts them to the above events. We will recommend settings for dfs.namenode.num.extra.edits.retained and dfs.namenode.num.extra.edits.segments.retained so that under reasonable conditions edits are never deleted before a client has a chance to pull them.

Clients obtain a `DFSInotifyEventInputStream`, from which they can read events, by calling `HdfsAdmin.getInotifyEventStream()`. Filesystem events that happen after the client makes this call will be available in the stream. `DFSInotifyEventInputStream` provides the following three methods:
1. `public Event poll()` – a non-blocking call that returns the next event if it is immediately available, otherwise null
2. `public Event poll(long time, TimeUnit tu)` – waits up to the given amount of time for a new event, otherwise returns null
3. `public Event take()` – blocks indefinitely for a new event

Justifications

A. Why a pull model as opposed to push?

The first reason is simplicity. We can leverage Hadoop RPC and don't have to design our own communication protocol or manage an extra set of open sockets on the NameNode. The second is that even if we adopted the push model, we would still need build a system that pulls from journals (or use a much more complex, stateful per-client queuing system, discussed in B) for the cases where 1) a client crashes and misses some edits and 2) there is a NameNode failover and some edits are logged to the new primary before clients can connect to it. So we might as well rely solely on the pull model for now, and if efficiency concerns emerge, we can look again into the push model.

It is certainly somewhat inefficient to fetch edits from the Quorum Journal or other journals on every client pull (although this shouldn't impact NameNode CPU time much), but it should not be difficult to build an edit (or event) cache on the NameNode so that several client requests for the same events do not all result in journal reads. Additionally, if the the NameNode stores edits in a local journal in addition to a remote journal, we can read from the local journal whenever possible.

B. Why superuser-only?

If we allowed unprivileged users to become inotify clients, a reasonable security model might be to allow them to read events for directories and files for which they have read access. The issue then is that with our pull-based system, in order to know whether an unprivileged client has permissions to read an event based on a transaction X, we need to know what permissions the client had when transactions up to X and nothing more had been applied to the namesystem. This is exceedingly difficult, since we would essentially need to reconstruct the namesystem at that earlier point in time. So the only way to support unprivileged users would be to adopt a push-based system where we determine which clients to send an event to right when it occurs, requiring only a lookup of the current namesystem state. But as discussed in A, it is much harder to build a push-based system than a pull-based one. In addition, while we can handle client failures and namenode failovers by falling back to the pull model for superusers (as discussed again in A), for unprivileged users (since pull is infeasible) we would need to maintain some sort of queue of unread events on the NameNode that is preserved in the event of a failover. Or we would have to loosen our guarantees to unprivileged clients – events missed in such situations are permanently lost and clients must use an ls -R or something similar to resynchronize their view of the filesystem state. In sum, adding support for unprivileged users introduces complexity that does not seem justified for the first version of inotify.

Potential Future Work (in order of priority)

1. Allow clients to watch events from specific directories rather than from the entirety of HDFS. We will need to decide whether undesired events should be filtered out on the NameNode or client side.
2. Support running the inotify logic in a separate, reduced-functionality NameNode so that the load on the primary NameNode is reduced.
3. Possibly support a inotify-specific security model (e.g. path-based security) that is separate from HDFS permissions and eliminates the complexities discussed in part B of the previous section.