# Asynchronous HDFS Access (v2016.5.10)

Tsz Wo Nicholas Sze, Xiaobing Zhou

# Motivation

Currently, all the FileSystem API methods are blocking calls – the caller is blocked until the method returns. It is very slow if a client makes a large number of independent calls in a single thread since each call has to wait until the previous call is finished. It is inefficient if a client needs to create a large number of threads to invoke the calls.

We propose adding a new API to support asynchronous calls, i.e. the caller is not blocked. The methods in the new API may immediately return a Java Future object. The return value can be obtained by the usual Future.get() method.

# Asynchronous RPC

## ipc.Client

For the client side, the underlying mechanism is already supporting asynchronous calls -- the calls shares a connection, the call requests are sent using a thread pool and the responses can be out of order.  Indeed, synchronized call is implemented by invoking wait() in the caller thread.

### Alternative 1

In order to support asynchronous calls, we may add new API methods to RPC protocol (e.g. ClientProtocol) to return a Future object.  E.g.

```
// existing API in ClientProtocol
boolean rename(String src, String dst) throws IOException;

// new API to support asynchronous calls
Future<Boolean> rename_async(String src, String dst) throws IOException;
```

### Alternative 2

Another way to support asynchronous calls is to add a new API to ipc.Client to get the Future object for the last call.  When an asynchronous call is made, a Future object f is created so that

f.get() will wait for the return value of the call. Then, f is stored in a thread local variable in ipc.Client, i.e.

```
// ipc.Client
// Store the return value of an async call.
private static final ThreadLocal<Future<?>> returnFuture = new ThreadLocal<>();

// get the return value for the last async call.
public static <T> Future<T> getReturnFuture() {
        return (Future<T>)returnFuture.get();
}
```

We choose Alternative 2 since it does not require changes in the protocol interface.

## Limiting Outstanding Calls

In order to avoid client abusing the server by asynchronous calls. The RPC client should have a configurable limit in order to limit the outstanding asynchronous calls. The caller may be blocked if the number of outstanding calls hits the limit so that the caller is slowed down.

## Server

For the server side, it appears that it needs no change or very minor change. HADOOP-11552 is a related JIRA.

# DistributedFileSystem -- Asynchronous Mode

## FutureFileSystem

Define a new API, namely FutureFileSystem (or AsynchronousFileSystem, if it is a better name). All the APIs in FutureFileSystem are the same as FileSystem except that the return type is wrapped by Future, e.g.

```
//FileSystem
public boolean rename(Path src, Path dst) throws IOException;

//FutureFileSystem[1]
public Future<Boolean> rename(Path src, Path dst) throws IOException[2];
```

Note 1: FutureFileSystem does not extend FileSystem.

Note 2: The methods in FutureFileSystem may or may not throw IOException. We list a few possible choices below.

1. The methods do not throw IOException. They only submit a task for execution and immediately return a Future object f. The IOException, if there is any, will be thrown wrapped as an ExecutionException when calling f.get(). In this case, we need separated threads (or a thread pool) to send the RPC.
2. The methods submit an RPC request before returning. An IOException can be thrown by the methods if there is a the network IO issues. The RemoteException, if there is any, will be thrown wrapped as an ExecutionException as in (1). We do not need separated threads in this case.
3. The methods perform some sanity check (e.g. checkOpen()) before returning. An IOException can be thrown by the methods if the check fails. The IOException after the check will be thrown wrapped as an ExecutionException as in (1). Similar to (1), we need separated threads to send the RPC.

★ We probably should declare the methods with "throw IOException" initially. We may remove it later if we strongly believe that "throw IOException" is useless. If the methods are declared without "throw IOException", we may not be able to add it later since it is an incompatible change.

★ We may let the user passing an Executor object when initialing FutureFileSystem so that if it is null, (2) will be used. Otherwise, (1) or (3) will be used.

## FutureDistributedFileSystem

Suppose we have FutureDistributedFileSystem implementing FutureFileSystem. The calls via FutureDistributedFileSystem are asynchronous while the calls via DistributedFileSystem are synchronous. The methods in FutureDistributedFileSystem are implemented using the asynchronous RPC calls. E.g.

```
//FutureDistributedFileSystem
public Future<Boolean> rename(final Path src, final Path dst) throws IOException {
        // the usual pre-rename code
        …
        // call RPC asynchronous using Alternative 2
        rename(src, dst);  // just return a dummy default value, ignore it.
        return Client.getReturnFuture();
}
```

## Synchronous Access in Asynchronous Mode

When the RPC is in asynchronous mode, we still support the synchronous calls via DistributedFileSystem. It can be implemented in DistributedFileSystem as shown below.

```
//DistributedFileSystem
public boolean rename(Path src, Path dst) throws IOException {
        if (isAsynchronousMode()) {
                return getFutureDistributedFileSystem().rename(src, dst).get();
        } else {
                ... //current implementation.
        }
}
```

## Client Code Example

```
// synchronous mode
DistributedFileSystem dfs = FileSystem.get(conf);

// asynchronous mode
FutureDistributedFileSystem futureDfs = dfs.getFutureDistributedFileSystem();

// synchronous calls
boolean r1 = dfs.rename(src1, dst1); // blocking
boolean r2 = dfs.rename(src2, dst2); // blocking

// asynchronous calls
Future<Boolean> f3 = futureDfs.rename(src3, dst3); // non-blocking
Future<Boolean> f4 = futureDfs.rename(src4, dst4); // non-blocking

boolean r3 = f3.get(); // blocking
boolean r4 = f4.get(); // blocking
```

# Future Works

- Guarantee the order of the asynchronous calls (server side)
- Support asynchronous FileContext (client API)
- Support callbacks such as ListenableFuture (client side)
- Support batch RPC, i.e. send the asynchronous calls in a batch (client side)
- Use Java 8's new language feature in the API (client API).
- Support asynchronous read/write to HDFS (client and server)