ErrorHandling

Handling the errors a ZooKeeper throws at you

There are plenty of things that can go wrong when dealing with ZooKeeper. Application code and libraries need to deal with them. This document suggests best practices to deal with errors. We distinguish an application from library code because libraries have a limited view of the overall picture, while the application has a complete view of what is going on. For example, the Application may use a Lock object from one library and a !KeptSet object from another library. The Application knows that it must only do change operations on !KeptSet when the Lock is held. If there is a fatal error that happens to the ZooKeeper handle, such as session expiration, only the Application knows the necessary steps to recover from the error; the Lock and !KeptSet libraries should not try to recover.

When things do go wrong they are manifest as an exception or an error code. These errors can be organized into the following categories:

- Normal state exceptions: trying to create a znode that already exists, calling setData on an existing znode, doing a conditional write to a znode that has an unexpected version number, etc.
- Recoverable errors: the disconnected event, connection timed out, and the connection loss exception are examples of recoverable errors, they indicate a problem that happened, but the !ZooKeeper handle is still valid and future operations will succeed once the ZooKeeper library can reestablish its connection to ZooKeeper.
- Fatal errors: the ZooKeeper handle has become invalid. This can be due to an explicit close, authentication errors, or session expiration.

The application and libraries handle the normal state exceptions as they happen. Usually they are a expected part of normal operation. For example, when doing a conditional set, usually the programmer is aware that there may be a concurrent process that may also try the same set and knows how to deal with it. The other two categories of errors can be more complicated.

Recoverable errors

Recoverable errors are passed back to the application because ZooKeeper itself cannot recover from them. The ZooKeeper library does try to recover the connection, so the handle should not be closed on a recoverable error, but the application must deal with the transient error. "But wait", you say, "if I'm doing a getData(), can't ZooKeeper just reissue it for me". Yes, of course ZooKeeper could as long as you were just doing a getData(). What if you were doing a create() or a delete() or a conditional setData()? When a !ZooKeeper client loses a connection to the ZooKeeper server there may be some requests in flight; we don't know where they were in their flight at the time of the connection loss.

For example the create we sent just before the loss may not have made it out of the network stack, or it may have made it to the ZooKeeper server we were connected to and been forwarded on to other servers before our server died. So, when we reestablish the connection to the ZooKeeper service we have no good way to know if our create executed or not. (The server actually has the needed information, but there is a lot of implementation work that needs to happen to take advantage of that information. Ironically, once we make the mutating requests re-issuable, the read requests become problematic...) So, if we reissue a create() request and we get a !NodeExistsException, the ZooKeeper client doesn't know if the exception resulted because the previous request went through or someone else did the create.

To handle recoverable errors, developers need to realize that there are two classes of requests: idempotent and non-idempotent requests. Read requests and unconditional sets and deletes are examples of idempotent requests, they can be reissued with the same results. (Although, the delete may throw a !NoNodeException on reissue its effect on the ZooKeeper state is the same.) Non-idempotent requests need special handling, application and library writers need to keep in mind that they may need to encode information in the data or name of znodes to detect when a retries. A simple example is a create that uses a sequence flag. If a process issues a create("/x-", ..., SEQUENCE) and gets a connection loss exception, that process will reissue another create("/x-", ..., SEQUENCE) and get back x-11# When the process does a getChildren("/"), it sees x-1,x-30,x-109,x-110,x-111, now it could be that x-109 was the result of the previous create, so the process actually owns both x-109 and x-111. An easy way around this is to use "x-process id-" when doing the create. If the process is using an id of 352, before reissuing the create it will do a getChildren("/") and see "x-222-1", "x-542-30", "x-352-109", x-333-110". The process will know that the original create succeeded an the znode it created is "x-352-109".

The bottom line is that ZooKeepers aren't lazy. They don't throw up their hands when there is a problem just because they don't want to bother retrying. There is application specific logic that is needed for recovery. ZooKeeper doesn't even retry non-idempotent requests because it may violate ordering guarantees that it provides to the clients. Thus, application programs should be very skeptical of layers build on top of ZooKeeper that simply reissue requests when these kinds of errors arise.

Unrecoverable errors

A ZooKeeper handle becomes invalid due to an explicit close or due to a catastrophic error such as a session expiration; either way any ephemeral nodes associated with the ZooKeeper handle will go away. An application can deal with an invalid handle, but libraries cannot. An application knows the steps it took to set things up properly and can re-execute those steps; a library sees only a subset of those steps without seeing the order. For these reasons it is important that libraries do not try to recover from unrecoverable errors; they do not have the whole picture and, just like ZooKeeper, do not have sufficient knowledge of the application to recover automatically.

When a library gets an unrecoverable error it should shutdown as gracefully as it can. Obviously it is not going to be able to interact with ZooKeeper, but it can mark its state as invalid and clean up any internal data structures.

An application that gets an unrecoverable error needs to make sure that everything that relied on the previous ZooKeeper handle is shutdown properly and then go through the process to bring everything back up. For this to work properly libraries must be not be doing magic under the covers. If we go back to our original example of the !KeptSet protected by the Lock. The !KeptSet implementer may think "hey I don't have ephemeral nodes, I can recover from session expirations by just creating a new Zookeeper object". Yes, this is true for the library, but the application is now screwed: most applications now days are multi threaded; the application gets the Lock and threads start accessing the !KeptSet; if the Lock is lost due to a session expiration and the !KeptSet magically keeps working bad things are going to happen. You may say that the threads should shutdown when the lock is lost, but there is a race condition between getting the expiration and shutting down the threads and the !KeptSet automagically recovering.

The bottom line is that libraries that recover from unrecoverable errors should be use with extreme care, if used at all.