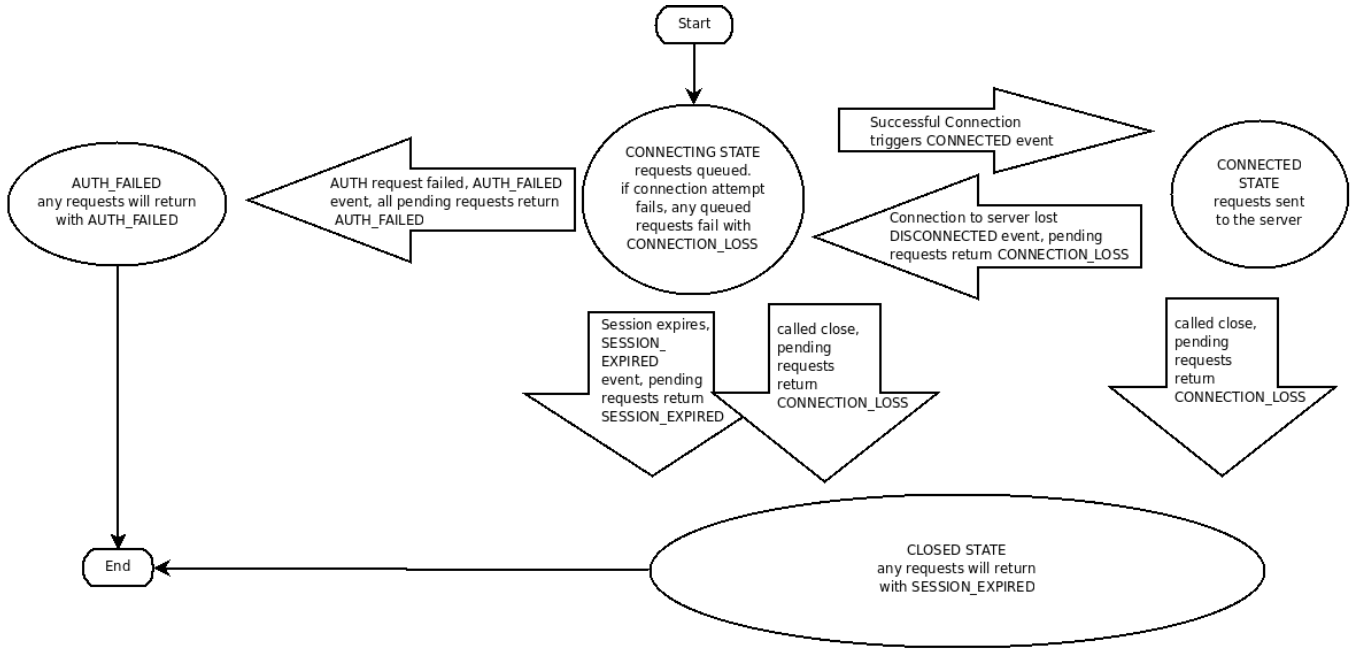
FAQ

What are the state transitions of ZooKeeper?



Start
CONNECTING STATE requests queued. if connection attempt fails, any queued requests fail with CONNECTION_LOSS
Successful Connection triggers CONNECTED event
CONNECTED STATE requests sent to the server
Connection to server lost DISCONNECTED event, pending requests return CONNECTION_LOSS
AUTH request failed, AUTH_FAILED event, all pending requests return AUTH_FAILED
AUTH_FAILED any requests will return with AUTH_FAILED
Session expires, SESSION_ EXPIRED event, pending requests return SESSION_EXPIRED
called close, pending requests return CONNECTION_LOSS
called close, pending requests return CONNECTION_LOSS
CLOSED STATE any requests will return with SESSION_EXPIRED
End

How should I handle the CONNECTION_LOSS error?

CONNECTION_LOSS means the link between the client and server was broken. It doesn't necessarily mean that the request failed. If you are doing a create request and the link was broken after the request reached the server and before the response was returned, the create request will succeed. If the link was broken before the packet went onto the wire, the create request failed. Unfortunately, there is no way for the client library to know, so it returns CONNECTION_LOSS. The programmer must figure out if the request succeeded or needs to be retried. Usually this is done in an application specific way. Examples of success detection include checking for the presence of a file to be created or checking the value of a znode to be modified.

When a client (session) becomes partitioned from the ZK serving cluster it will begin searching the list of servers that were specified during session creation. Eventually, when connectivity between the client and at least one of the servers is re-established, the session will either again transition to the "connected" state (if reconnected within the session timeout value) or it will transition to the "expired" state (if reconnected after the session timeout). The ZK client library will handle reconnect for you automatically. In particular we have heuristics built into the client library to handle things like "herd effect", etc... Only create a new session when you are notified of session expiration (mandatory).

How should I handle SESSION_EXPIRED?

SESSION_EXPIRED automatically closes the ZooKeeper handle. In a correctly operating cluster, you should never see SESSION_EXPIRED. It means that the client was partitioned off from the ZooKeeper service for more the the session timeout and ZooKeeper decided that the client died. Because the ZooKeeper service is ground truth, the client should consider itself dead and go into recovery. If the client is only reading state from ZooKeeper, recovery means just reconnecting. In more complex applications, recovery means recreating ephemeral nodes, vying for leadership roles, and reconstructing published state.

Library writers should be conscious of the severity of the expired state and not try to recover from it. Instead libraries should return a fatal error. Even if the library is simply reading from ZooKeeper, the user of the library may also be doing other things with ZooKeeper that requires more complex recovery.

Session expiration is managed by the ZooKeeper cluster itself, not by the client. When the ZK client establishes a session with the cluster it provides a "timeout" value. This value is used by the cluster to determine when the client's session expires. Expirations happens when the cluster does not hear from the client within the specified session timeout period (i.e. no heartbeat). At session expiration the cluster will delete any/all ephemeral nodes owned by that session and immediately notify any/all connected clients of the change (anyone watching those znodes). At this point the client of the expired session is still disconnected from the cluster, it will not be notified of the session expiration until/unless it is able to re-establish a connection to the cluster. The client will stay in disconnected state until the TCP connection is re-established with the cluster, at which point the watcher of the expired session will receive the "session expired" notification.

Example state transitions for an expired session as seen by the expired session's watcher:

1. 'connected' : session is established and client is communicating with cluster (client/server communication is operating properly)
2. .... client is partitioned from the cluster
3. 'disconnected' : client has lost connectivity with the cluster
4. .... time elapses, after 'timeout' period the cluster expires the session, nothing is seen by client as it is disconnected from cluster
5. .... time elapses, the client regains network level connectivity with the cluster

6. 'expired' : eventually the client reconnects to the cluster, it is then notified of the expiration

Is there an easy way to expire a session for testing?

Yes, a ZooKeeper handle can take a session id and password. This constructor is used to recover a session after total application failure. For example, an application can connect to ZooKeeper, save the session id and password to a file, terminate, restart, read the session id and password, and reconnect to ZooKeeper without loosing the session and the corresponding ephemeral nodes. It is up to the programmer to ensure that the session id and password isn't passed around to multiple instances of an application, otherwise problems can result.

In the case of testing we want to cause a problem, so to explicitly expire a session an application connects to ZooKeeper, saves the session id and password, creates another ZooKeeper handle with that id and password, and then closes the new handle. Since both handles reference the same session, the close on second handle will invalidate the session causing a SESSION_EXPIRED on the first handle.

Why doesn't the NodeChildrenChanged and NodeDataChanged watch events return more information about the change?

When a ZooKeeper server generates the change events, it knows exactly what the change is. In our initial implementation of ZooKeeper we returned this information with the change event, but it turned out that it was impossible to use correctly. There may be a correct way to use it, but we have never seen a case of correct usage. The problem is that watches are used to find out about the latest change. (Otherwise, you would just do periodic gets.) The thing that most programmers seem to miss, when they ask for this feature, is that watches are one time triggers. Observe the following case of data change: a process does a getData on "/a" with watch set to true and gets "v1", another process changes "/a" to "v2" and shortly there after changes "/a" to "v3". The first process would see that "/a" was changed to "v2", but wouldn't know that "/a" is now "/v3".

What are the options-process for upgrading ZooKeeper?

There are two primary ways of doing this; 1) full restart or 2) rolling restart.

In the full restart case you can stage your updated code/configuration/etc..., stop all of the servers in the ensemble, switch code/configuration, and restart the ZooKeeper ensemble. If you do this programmatically (scripts typically, ie not by hand) the restart can be done on order of seconds. As a result the clients will lose connectivity to the ZooKeeper cluster during this time, however it looks to the clients just like a network partition. All existing client sessions are maintained and re-established as soon as the ZooKeeper ensemble comes back up. Obviously one drawback to this approach is that if you encounter any issues (it's always a good idea to test/stage these changes on a test harness) the cluster may be down for longer than expected.

The second option, preferable for many users, is to do a "rolling restart". In this case you upgrade one server in the ZooKeeper ensemble at a time; bring down the server, upgrade the code/configuration/etc..., then restart the server. The server will automatically rejoin the quorum, update it's internal state with the current ZK leader, and begin serving client sessions. As a result of doing a rolling restart, rather than a full restart, the administrator can monitor the ensemble as the upgrade progresses, perhaps rolling back if any issues are encountered.

How do I size a ZooKeeper ensemble (cluster)?

In general when determining the number of ZooKeeper serving nodes to deploy (the size of an ensemble) you need to think in terms of reliability, and not performance.

Reliability:

A single ZooKeeper server (standalone) is essentially a coordinator with no reliability (a single serving node failure brings down the ZK service).

A 3 server ensemble (you need to jump to 3 and not 2 because ZK works based on simple majority voting) allows for a single server to fail and the service will still be available.

So if you want reliability go with at least 3. We typically recommend having 5 servers in "online" production serving environments. This allows you to take 1 server out of service (say planned maintenance) and still be able to sustain an unexpected outage of one of the remaining servers w/o interruption of the service.

Performance:

Write performance actually *decreases* as you add ZK servers, while read performance increases modestly: http://zookeeper.apache.org/doc/current/zookeeperOver.html#Performance

See this page for a survey Patrick Hunt (http://twitter.com/phunt) did looking at operational latency with both standalone server and an ensemble of size 3. You'll notice that a single core machine running a standalone ZK ensemble (1 server) is still able to process 15k requests per second. This is orders of magnitude greater than what most applications require (if they are using ZooKeeper correctly - ie as a coordination service, and not as a replacement for a database, filestore, cache, etc...)

Can I run an ensemble cluster behind a load balancer?

There are two types of servers failures in distributed system from socket I/O perspective.

1. server down due to hardware failures and OS panic/hang, Zookeeper daemon hang, temporary/permanent network outage, network switch anomaly, etc: client cannot figure out failures immediately since there is no responding entities. As a result, zookeeper clients must rely on timeout to identify failures.
2. Dead zookeeper process (daemon): since OS will respond to closed TCP port, client will get "connection refused" upon socket connect or "peer reset" on socket I/O. Client immediately notice that the other end failed.

Here's how ZK clients respond to servers in each case.

1. In this case (former), ZK client rely on heartbeat algorithm. ZK clients detects server failures in 2/3 of recv timeout (Zookeeper_init), and then it retries the same IP at every recv timeout period if only one of ensemble is given. If more than two ensemble IP are given, ZK clients will try next IP immediately.
2. In this scenario, ZK client will immediately detect failure, and will retry connecting every second assuming only one ensemble IP is given. If multiple ensemble IP is given (most installation falls into this category), ZK client retries next IP immediately.

Notice that in both cases, when more than one ensemble IP is specified, ZK clients retry next IP immediately with no delay.

On some installations, it is preferable to run an ensemble cluster behind a load balancer such as hardware L4 switch, TCP reverse proxy, or DNS round-robin because such setup allows users to simply use one hostname or IP (or VIP) for ensemble cluster, and some detects server failures as well.

But there are subtle differences on how these load balancers will react upon server failures.

- Hardware L4 load balancer: this setup involves one IP and a hostname. L4 switch usually does heartbeat on its own, and thus removes non-responding host from its IP list. But this also relies on the same timeout scheme for fault detection. L4 may redirect you to a unresponsive server. If hardware LB detect server failures fast enough, this setup will always redirect you to live ensemble server.
- DNS round robin: this setup involves one hostname and a list of IPs. ZK clients correctly make used of a list of IPs returned by DNS query. Thus this setup works the same way as multiple hostname (IP) argument to zookeeper_init. The drawback is that when an ensemble cluster configuration changes like server addition/removal, it may take a while to propagate the DNS entry change in all DNS servers and DNS client caching (nscd for example) TTL issue.

In conclusion, DNS RR works as good as a list of ensemble IP arguments except cluster reconfiguration case.
It turns out that there is a minor problem with DNS RR. If you are using a tool such as zktop.py, it does not take care of a list of host IP returned by a DNS server.

What happens to ZK sessions while the cluster is down?

Imagine that a client is connected to ZK with a 5 second session timeout, and the administrator brings the entire ZK cluster down for an upgrade. The cluster is down for several minutes, and then is restarted.

In this scenario, the client is able to reconnect and refresh its session. Because session timeouts are tracked by the leader, the session starts counting down again with a fresh timeout when the cluster is restarted. So, as long as the client connects within the first 5 seconds after a leader is elected, it will reconnect without an expiration, and any ephemeral nodes it had prior to the downtime will be maintained.

The same behavior is exhibited when the leader crashes and a new one is elected. In the limit, if the leader is flip-flopping back and forth quickly, sessions will never expire since their timers are getting constantly reset.