**ZooKeeper Programmer's Guide**

**Developing Distributed Applications that use ZooKeeper**

* [Introduction](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#_introduction)
* [The ZooKeeper Data Model](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkDataModel)
  + [ZNodes](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_zkDataModel_znodes)
    - [Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_zkDataMode_watches)
    - [Data Access](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Data+Access)
    - [Ephemeral Nodes](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Ephemeral+Nodes)
    - [Sequence Nodes -- Unique Naming](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Sequence+Nodes+--+Unique+Naming)
    - [Container Nodes](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Container+Nodes)
  + [Time in ZooKeeper](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_timeInZk)
  + [ZooKeeper Stat Structure](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_zkStatStructure)
* [ZooKeeper Sessions](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkSessions)
* [ZooKeeper Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkWatches)
  + [Semantics of Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_WatchSemantics)
  + [Remove Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_WatchRemoval)
  + [What ZooKeeper Guarantees about Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_WatchGuarantees)
  + [Things to Remember about Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_WatchRememberThese)
* [ZooKeeper access control using ACLs](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_ZooKeeperAccessControl)
  + [ACL Permissions](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_ACLPermissions)
    - [Builtin ACL Schemes](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_BuiltinACLSchemes)
    - [ZooKeeper C client API](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ZooKeeper+C+client+API)
* [Pluggable ZooKeeper authentication](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_ZooKeeperPluggableAuthentication)
* [Consistency Guarantees](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkGuarantees)
* [Bindings](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_bindings)
  + [Java Binding](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Java+Binding)
    - [Client Configuration Parameters](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_java_client_configuration)
  + [C Binding](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#C+Binding)
    - [Installation](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Installation)
    - [Building Your Own C Client](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#Building+Your+Own+C+Client)
* [Building Blocks: A Guide to ZooKeeper Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_guideToZkOperations)
  + [Handling Errors](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_errorsZk)
  + [Connecting to ZooKeeper](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_connectingToZk)
  + [Read Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_readOps)
  + [Write Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_writeOps)
  + [Handling Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_handlingWatches)
  + [Miscelleaneous ZooKeeper Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#sc_miscOps)
* [Program Structure, with Simple Example](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_programStructureWithExample)
* [Gotchas: Common Problems and Troubleshooting](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_gotchas)

**Introduction**

This document is a guide for developers wishing to create distributed applications that take advantage of ZooKeeper's coordination services. It contains conceptual and practical information.

The first four sections of this guide present higher level discussions of various ZooKeeper concepts. These are necessary both for an understanding of how ZooKeeper works as well how to work with it. It does not contain source code, but it does assume a familiarity with the problems associated with distributed computing. The sections in this first group are:

* [The ZooKeeper Data Model](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkDataModel)
* [ZooKeeper Sessions](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkSessions)
* [ZooKeeper Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkWatches)
* [Consistency Guarantees](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkGuarantees)

The next four sections provide practical programming information. These are:

* [Building Blocks: A Guide to ZooKeeper Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_guideToZkOperations)
* [Bindings](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_bindings)
* [Program Structure, with Simple Example](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_programStructureWithExample) *[tbd]*
* [Gotchas: Common Problems and Troubleshooting](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_gotchas)

The book concludes with an [appendix](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#apx_linksToOtherInfo) containing links to other useful, ZooKeeper-related information.

Most of information in this document is written to be accessible as stand-alone reference material. However, before starting your first ZooKeeper application, you should probably at least read the chaptes on the [ZooKeeper Data Model](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkDataModel) and [ZooKeeper Basic Operations](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_guideToZkOperations). Also, the [Simple Programmming Example](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_programStructureWithExample) *[tbd]* is helpful for understanding the basic structure of a ZooKeeper client application.

**The ZooKeeper Data Model**

ZooKeeper has a hierarchal name space, much like a distributed file system. The only difference is that each node in the namespace can have data associated with it as well as children. It is like having a file system that allows a file to also be a directory. Paths to nodes are always expressed as canonical, absolute, slash-separated paths; there are no relative reference. Any unicode character can be used in a path subject to the following constraints:

* The null character (\u0000) cannot be part of a path name. (This causes problems with the C binding.)
* The following characters can't be used because they don't display well, or render in confusing ways: \u0001 - \u001F and \u007F - \u009F.
* The following characters are not allowed: \ud800 - uF8FF, \uFFF0 - uFFFF.
* The "." character can be used as part of another name, but "." and ".." cannot alone be used to indicate a node along a path, because ZooKeeper doesn't use relative paths. The following would be invalid: "/a/b/./c" or "/a/b/../c".
* The token "zookeeper" is reserved.

**ZNodes**

Every node in a ZooKeeper tree is referred to as a *znode*. Znodes maintain a stat structure that includes version numbers for data changes, acl changes. The stat structure also has timestamps. The version number, together with the timestamp, allows ZooKeeper to validate the cache and to coordinate updates. Each time a znode's data changes, the version number increases. For instance, whenever a client retrieves data, it also receives the version of the data. And when a client performs an update or a delete, it must supply the version of the data of the znode it is changing. If the version it supplies doesn't match the actual version of the data, the update will fail. (This behavior can be overridden. For more information see... )*[tbd...]*

**Note**

In distributed application engineering, the word *node* can refer to a generic host machine, a server, a member of an ensemble, a client process, etc. In the ZooKeeper documentation, *znodes* refer to the data nodes. *Servers* refer to machines that make up the ZooKeeper service; *quorum peers* refer to the servers that make up an ensemble; client refers to any host or process which uses a ZooKeeper service.

Znodes are the main enitity that a programmer access. They have several characteristics that are worth mentioning here.

Watches

Clients can set watches on znodes. Changes to that znode trigger the watch and then clear the watch. When a watch triggers, ZooKeeper sends the client a notification. More information about watches can be found in the section [ZooKeeper Watches](http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkWatches).

Data Access

The data stored at each znode in a namespace is read and written atomically. Reads get all the data bytes associated with a znode and a write replaces all the data. Each node has an Access Control List (ACL) that restricts who can do what.

ZooKeeper was not designed to be a general database or large object store. Instead, it manages coordination data. This data can come in the form of configuration, status information, rendezvous, etc. A common property of the various forms of coordination data is that they are relatively small: measured in kilobytes. The ZooKeeper client and the server implementations have sanity checks to ensure that znodes have less than 1M of data, but the data should be much less than that on average. Operating on relatively large data sizes will cause some operations to take much more time than others and will affect the latencies of some operations because of the extra time needed to move more data over the network and onto storage media. If large data storage is needed, the usually pattern of dealing with such data is to store it on a bulk storage system, such as NFS or HDFS, and store pointers to the storage locations in ZooKeeper.

Ephemeral Nodes

ZooKeeper also has the notion of ephemeral nodes. These znodes exists as long as the session that created the znode is active. When the session ends the znode is deleted. Because of this behavior ephemeral znodes are not allowed to have children.

Sequence Nodes -- Unique Naming

When creating a znode you can also request that ZooKeeper append a monotonically increasing counter to the end of path. This counter is unique to the parent znode. The counter has a format of %010d -- that is 10 digits with 0 (zero) padding (the counter is formatted in this way to simplify sorting), i.e. "<path>0000000001". See [Queue Recipe](http://zookeeper.apache.org/doc/trunk/recipes.html#sc_recipes_Queues) for an example use of this feature. Note: the counter used to store the next sequence number is a signed int (4bytes) maintained by the parent node, the counter will overflow when incremented beyond 2147483647 (resulting in a name "<path>-2147483647").

Container Nodes

**Added in 3.6.0**

ZooKeeper has the notion of container nodes. Container nodes are special purpose nodes useful for recipes such as leader, lock, etc. When the last child of a container is deleted, the container becomes a candidate to be deleted by the server at some point in the future.

Given this property, you should be prepared to get KeeperException.NoNodeException when creating children inside of container nodes. i.e. when creating child nodes inside of container nodes always check for KeeperException.NoNodeException and recreate the container node when it occurs.

**Time in ZooKeeper**

ZooKeeper tracks time multiple ways:

* **Zxid**

Every change to the ZooKeeper state receives a stamp in the form of a *zxid* (ZooKeeper Transaction Id). This exposes the total ordering of all changes to ZooKeeper. Each change will have a unique zxid and if zxid1 is smaller than zxid2 then zxid1 happened before zxid2.

* **Version numbers**

Every change to a node will cause an increase to one of the version numbers of that node. The three version numbers are version (number of changes to the data of a znode), cversion (number of changes to the children of a znode), and aversion (number of changes to the ACL of a znode).

* **Ticks**

When using multi-server ZooKeeper, servers use ticks to define timing of events such as status uploads, session timeouts, connection timeouts between peers, etc. The tick time is only indirectly exposed through the minimum session timeout (2 times the tick time); if a client requests a session timeout less than the minimum session timeout, the server will tell the client that the session timeout is actually the minimum session timeout.

* **Real time**

ZooKeeper doesn't use real time, or clock time, at all except to put timestamps into the stat structure on znode creation and znode modification.

**ZooKeeper Stat Structure**

The Stat structure for each znode in ZooKeeper is made up of the following fields:

* **czxid**

The zxid of the change that caused this znode to be created.

* **mzxid**

The zxid of the change that last modified this znode.

* **pzxid**

The zxid of the change that last modified children of this znode.

* **ctime**

The time in milliseconds from epoch when this znode was created.

* **mtime**

The time in milliseconds from epoch when this znode was last modified.

* **version**

The number of changes to the data of this znode.

* **cversion**

The number of changes to the children of this znode.

* **aversion**

The number of changes to the ACL of this znode.

* **ephemeralOwner**

The session id of the owner of this znode if the znode is an ephemeral node. If it is not an ephemeral node, it will be zero.

* **dataLength**

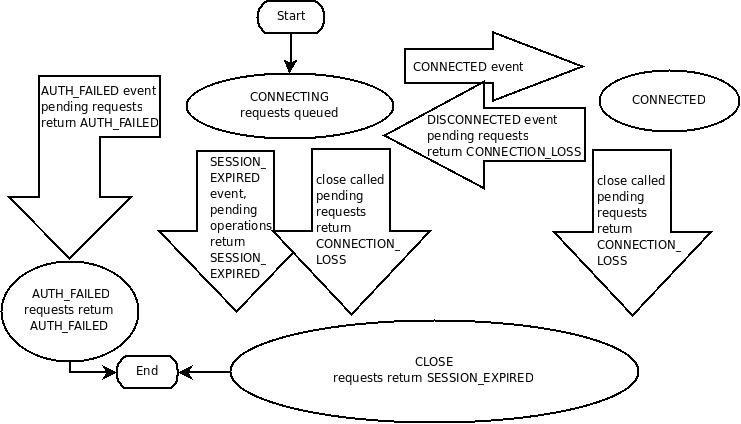
The length of the data field of this znode.

* **numChildren**

The number of children of this znode.

**ZooKeeper Sessions**

A ZooKeeper client establishes a session with the ZooKeeper service by creating a handle to the service using a language binding. Once created, the handle starts of in the CONNECTING state and the client library tries to connect to one of the servers that make up the ZooKeeper service at which point it switches to the CONNECTED state. During normal operation will be in one of these two states. If an unrecoverable error occurs, such as session expiration or authentication failure, or if the application explicitly closes the handle, the handle will move to the CLOSED state. The following figure shows the possible state transitions of a ZooKeeper client:



To create a client session the application code must provide a connection string containing a comma separated list of host:port pairs, each corresponding to a ZooKeeper server (e.g. "127.0.0.1:4545" or "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002"). The ZooKeeper client library will pick an arbitrary server and try to connect to it. If this connection fails, or if the client becomes disconnected from the server for any reason, the client will automatically try the next server in the list, until a connection is (re-)established.

**Added in 3.2.0**: An optional "chroot" suffix may also be appended to the connection string. This will run the client commands while interpreting all paths relative to this root (similar to the unix chroot command). If used the example would look like: "127.0.0.1:4545/app/a" or "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002/app/a" where the client would be rooted at "/app/a" and all paths would be relative to this root - ie getting/setting/etc... "/foo/bar" would result in operations being run on "/app/a/foo/bar" (from the server perspective). This feature is particularly useful in multi-tenant environments where each user of a particular ZooKeeper service could be rooted differently. This makes re-use much simpler as each user can code his/her application as if it were rooted at "/", while actual location (say /app/a) could be determined at deployment time.

When a client gets a handle to the ZooKeeper service, ZooKeeper creates a ZooKeeper session, represented as a 64-bit number, that it assigns to the client. If the client connects to a different ZooKeeper server, it will send the session id as a part of the connection handshake. As a security measure, the server creates a password for the session id that any ZooKeeper server can validate.The password is sent to the client with the session id when the client establishes the session. The client sends this password with the session id whenever it reestablishes the session with a new server.

One of the parameters to the ZooKeeper client library call to create a ZooKeeper session is the session timeout in milliseconds. The client sends a requested timeout, the server responds with the timeout that it can give the client. The current implementation requires that the timeout be a minimum of 2 times the tickTime (as set in the server configuration) and a maximum of 20 times the tickTime. The ZooKeeper client API allows access to the negotiated timeout.

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). It is not advisable to create a new session object (a new ZooKeeper.class or zookeeper handle in the c binding) for disconnection. The ZK client library will handle reconnect for you. 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).

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 detailed above. 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

Another parameter to the ZooKeeper session establishment call is the default watcher. Watchers are notified when any state change occurs in the client. For example if the client loses connectivity to the server the client will be notified, or if the client's session expires, etc... This watcher should consider the initial state to be disconnected (i.e. before any state changes events are sent to the watcher by the client lib). In the case of a new connection, the first event sent to the watcher is typically the session connection event.

The session is kept alive by requests sent by the client. If the session is idle for a period of time that would timeout the session, the client will send a PING request to keep the session alive. This PING request not only allows the ZooKeeper server to know that the client is still active, but it also allows the client to verify that its connection to the ZooKeeper server is still active. The timing of the PING is conservative enough to ensure reasonable time to detect a dead connection and reconnect to a new server.

Once a connection to the server is successfully established (connected) there are basically two cases where the client lib generates connectionloss (the result code in c binding, exception in Java -- see the API documentation for binding specific details) when either a synchronous or asynchronous operation is performed and one of the following holds:

1. The application calls an operation on a session that is no longer alive/valid
2. The ZooKeeper client disconnects from a server when there are pending operations to that server, i.e., there is a pending asynchronous call.

**Added in 3.2.0 -- SessionMovedException**. There is an internal exception that is generally not seen by clients called the SessionMovedException. This exception occurs because a request was received on a connection for a session which has been reestablished on a different server. The normal cause of this error is a client that sends a request to a server, but the network packet gets delayed, so the client times out and connects to a new server. When the delayed packet arrives at the first server, the old server detects that the session has moved, and closes the client connection. Clients normally do not see this error since they do not read from those old connections. (Old connections are usually closed.) One situation in which this condition can be seen is when two clients try to reestablish the same connection using a saved session id and password. One of the clients will reestablish the connection and the second client will be disconnected (causing the pair to attempt to re-establish its connection/session indefinitely).

**Updating the list of servers**. We allow a client to update the connection string by providing a new comma separated list of host:port pairs, each corresponding to a ZooKeeper server. The function invokes a probabilistic load-balancing algorithm which may cause the client to disconnect from its current host with the goal to achieve expected uniform number of connections per server in the new list. In case the current host to which the client is connected is not in the new list this call will always cause the connection to be dropped. Otherwise, the decision is based on whether the number of servers has increased or decreased and by how much.

For example, if the previous connection string contained 3 hosts and now the list contains these 3 hosts and 2 more hosts, 40% of clients connected to each of the 3 hosts will move to one of the new hosts in order to balance the load. The algorithm will cause the client to drop its connection to the current host to which it is connected with probability 0.4 and in this case cause the client to connect to one of the 2 new hosts, chosen at random.

Another example -- suppose we have 5 hosts and now update the list to remove 2 of the hosts, the clients connected to the 3 remaining hosts will stay connected, whereas all clients connected to the 2 removed hosts will need to move to one of the 3 hosts, chosen at random. If the connection is dropped, the client moves to a special mode where he chooses a new server to connect to using the probabilistic algorithm, and not just round robin.

In the first example, each client decides to disconnect with probability 0.4 but once the decision is made, it will try to connect to a random new server and only if it cannot connect to any of the new servers will it try to connect to the old ones. After finding a server, or trying all servers in the new list and failing to connect, the client moves back to the normal mode of operation where it picks an arbitrary server from the connectString and attempt to connect to it. If that fails, is will continue trying different random servers in round robin. (see above the algorithm used to initially choose a server)

**ZooKeeper Watches**

All of the read operations in ZooKeeper - **getData()**, **getChildren()**, and **exists()** - have the option of setting a watch as a side effect. Here is ZooKeeper's definition of a watch: a watch event is one-time trigger, sent to the client that set the watch, which occurs when the data for which the watch was set changes. There are three key points to consider in this definition of a watch:

* **One-time trigger**

One watch event will be sent to the client when the data has changed. For example, if a client does a getData("/znode1", true) and later the data for /znode1 is changed or deleted, the client will get a watch event for /znode1. If /znode1 changes again, no watch event will be sent unless the client has done another read that sets a new watch.

* **Sent to the client**

This implies that an event is on the way to the client, but may not reach the client before the successful return code to the change operation reaches the client that initiated the change. Watches are sent asynchronously to watchers. ZooKeeper provides an ordering guarantee: a client will never see a change for which it has set a watch until it first sees the watch event. Network delays or other factors may cause different clients to see watches and return codes from updates at different times. The key point is that everything seen by the different clients will have a consistent order.

* **The data for which the watch was set**

This refers to the different ways a node can change. It helps to think of ZooKeeper as maintaining two lists of watches: data watches and child watches. getData() and exists() set data watches. getChildren() sets child watches. Alternatively, it may help to think of watches being set according to the kind of data returned. getData() and exists() return information about the data of the node, whereas getChildren() returns a list of children. Thus, setData() will trigger data watches for the znode being set (assuming the set is successful). A successful create() will trigger a data watch for the znode being created and a child watch for the parent znode. A successful delete() will trigger both a data watch and a child watch (since there can be no more children) for a znode being deleted as well as a child watch for the parent znode.

Watches are maintained locally at the ZooKeeper server to which the client is connected. This allows watches to be lightweight to set, maintain, and dispatch. When a client connects to a new server, the watch will be triggered for any session events. Watches will not be received while disconnected from a server. When a client reconnects, any previously registered watches will be reregistered and triggered if needed. In general this all occurs transparently. There is one case where a watch may be missed: a watch for the existence of a znode not yet created will be missed if the znode is created and deleted while disconnected.

**Semantics of Watches**

We can set watches with the three calls that read the state of ZooKeeper: exists, getData, and getChildren. The following list details the events that a watch can trigger and the calls that enable them:

* **Created event:**

Enabled with a call to exists.

* **Deleted event:**

Enabled with a call to exists, getData, and getChildren.

* **Changed event:**

Enabled with a call to exists and getData.

* **Child event:**

Enabled with a call to getChildren.

**Remove Watches**

We can remove the watches registered on a znode with a call to removeWatches. Also, a ZooKeeper client can remove watches locally even if there is no server connection by setting the local flag to true. The following list details the events which will be triggered after the successful watch removal.

* **Child Remove event:**

Watcher which was added with a call to getChildren.

* **Data Remove event:**

Watcher which was added with a call to exists or getData.

**What ZooKeeper Guarantees about Watches**

With regard to watches, ZooKeeper maintains these guarantees:

* Watches are ordered with respect to other events, other watches, and asynchronous replies. The ZooKeeper client libraries ensures that everything is dispatched in order.
* A client will see a watch event for a znode it is watching before seeing the new data that corresponds to that znode.
* The order of watch events from ZooKeeper corresponds to the order of the updates as seen by the ZooKeeper service.

**Things to Remember about Watches**

* Watches are one time triggers; if you get a watch event and you want to get notified of future changes, you must set another watch.
* Because watches are one time triggers and there is latency between getting the event and sending a new request to get a watch you cannot reliably see every change that happens to a node in ZooKeeper. Be prepared to handle the case where the znode changes multiple times between getting the event and setting the watch again. (You may not care, but at least realize it may happen.)
* A watch object, or function/context pair, will only be triggered once for a given notification. For example, if the same watch object is registered for an exists and a getData call for the same file and that file is then deleted, the watch object would only be invoked once with the deletion notification for the file.
* When you disconnect from a server (for example, when the server fails), you will not get any watches until the connection is reestablished. For this reason session events are sent to all outstanding watch handlers. Use session events to go into a safe mode: you will not be receiving events while disconnected, so your process should act conservatively in that mode.

**ZooKeeper access control using ACLs**

ZooKeeper uses ACLs to control access to its znodes (the data nodes of a ZooKeeper data tree). The ACL implementation is quite similar to UNIX file access permissions: it employs permission bits to allow/disallow various operations against a node and the scope to which the bits apply. Unlike standard UNIX permissions, a ZooKeeper node is not limited by the three standard scopes for user (owner of the file), group, and world (other). ZooKeeper does not have a notion of an owner of a znode. Instead, an ACL specifies sets of ids and permissions that are associated with those ids.

Note also that an ACL pertains only to a specific znode. In particular it does not apply to children. For example, if */app* is only readable by ip:172.16.16.1 and */app/status* is world readable, anyone will be able to read */app/status*; ACLs are not recursive.

ZooKeeper supports pluggable authentication schemes. Ids are specified using the form *scheme:id*, where *scheme* is a the authentication scheme that the id corresponds to. For example, *ip:172.16.16.1* is an id for a host with the address *172.16.16.1*.

When a client connects to ZooKeeper and authenticates itself, ZooKeeper associates all the ids that correspond to a client with the clients connection. These ids are checked against the ACLs of znodes when a clients tries to access a node. ACLs are made up of pairs of *(scheme:expression, perms)*. The format of the *expression* is specific to the scheme. For example, the pair *(ip:19.22.0.0/16, READ)* gives the *READ* permission to any clients with an IP address that starts with 19.22.

**ACL Permissions**

ZooKeeper supports the following permissions:

* **CREATE**: you can create a child node
* **READ**: you can get data from a node and list its children.
* **WRITE**: you can set data for a node
* **DELETE**: you can delete a child node
* **ADMIN**: you can set permissions

The *CREATE* and *DELETE* permissions have been broken out of the *WRITE* permission for finer grained access controls. The cases for *CREATE* and *DELETE* are the following:

You want A to be able to do a set on a ZooKeeper node, but not be able to *CREATE* or *DELETE* children.

*CREATE* without *DELETE*: clients create requests by creating ZooKeeper nodes in a parent directory. You want all clients to be able to add, but only request processor can delete. (This is kind of like the APPEND permission for files.)

Also, the *ADMIN* permission is there since ZooKeeper doesn’t have a notion of file owner. In some sense the *ADMIN* permission designates the entity as the owner. ZooKeeper doesn’t support the LOOKUP permission (execute permission bit on directories to allow you to LOOKUP even though you can't list the directory). Everyone implicitly has LOOKUP permission. This allows you to stat a node, but nothing more. (The problem is, if you want to call zoo\_exists() on a node that doesn't exist, there is no permission to check.)

Builtin ACL Schemes

ZooKeeeper has the following built in schemes:

* **world** has a single id, *anyone*, that represents anyone.
* **auth** doesn't use any id, represents any authenticated user.
* **digest** uses a *username:password* string to generate MD5 hash which is then used as an ACL ID identity. Authentication is done by sending the *username:password* in clear text. When used in the ACL the expression will be the *username:base64*encoded *SHA1* password *digest*.
* **ip** uses the client host IP as an ACL ID identity. The ACL expression is of the form *addr/bits* where the most significant *bits* of *addr* are matched against the most significant *bits* of the client host IP.
* **x509** uses the client X500 Principal as an ACL ID identity. The ACL expression is the exact X500 Principal name of a client. When using the secure port, clients are automatically authenticated and their auth info for the x509 scheme is set.

ZooKeeper C client API

The following constants are provided by the ZooKeeper C library:

* *const* *int* ZOO\_PERM\_READ; //can read node’s value and list its children
* *const* *int* ZOO\_PERM\_WRITE;// can set the node’s value
* *const* *int* ZOO\_PERM\_CREATE; //can create children
* *const* *int* ZOO\_PERM\_DELETE;// can delete children
* *const* *int* ZOO\_PERM\_ADMIN; //can execute set\_acl()
* *const* *int* ZOO\_PERM\_ALL;// all of the above flags OR’d together

The following are the standard ACL IDs:

* *struct* Id ZOO\_ANYONE\_ID\_UNSAFE; //(‘world’,’anyone’)
* *struct* Id ZOO\_AUTH\_IDS;// (‘auth’,’’)

ZOO\_AUTH\_IDS empty identity string should be interpreted as “the identity of the creator”.

ZooKeeper client comes with three standard ACLs:

* *struct* ACL\_vector ZOO\_OPEN\_ACL\_UNSAFE; //(ZOO\_PERM\_ALL,ZOO\_ANYONE\_ID\_UNSAFE)
* *struct* ACL\_vector ZOO\_READ\_ACL\_UNSAFE;// (ZOO\_PERM\_READ, ZOO\_ANYONE\_ID\_UNSAFE)
* *struct* ACL\_vector ZOO\_CREATOR\_ALL\_ACL; //(ZOO\_PERM\_ALL,ZOO\_AUTH\_IDS)

The ZOO\_OPEN\_ACL\_UNSAFE is completely open free for all ACL: any application can execute any operation on the node and can create, list and delete its children. The ZOO\_READ\_ACL\_UNSAFE is read-only access for any application. CREATE\_ALL\_ACL grants all permissions to the creator of the node. The creator must have been authenticated by the server (for example, using “*digest*” scheme) before it can create nodes with this ACL