by

Table of contents

Developing Distributed Applications that use ZooKeeper

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 chapters 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 chapters in this first group are:

- The ZooKeeper Data Model
- ZooKeeper Sessions
- ZooKeeper Watches
- Consistency Guarantees

The next four chapters of this provided practical programming information. These are:

- Building Blocks: A Guide to ZooKeeper Operations
- Bindings
- Program Structure, with Simple Example [tbd]
- Gotchas: Common Problems and Troubleshooting

The book concludes with an appendix 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 and ZooKeeper Basic Operations. Also, the Simple Programmming Example [tbd] is helpful for understand the basic structure of a ZooKeeper client application.

1. 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 - \u0019 and \u007F - \u009F.
- The following characters are not allowed because [tbd: do we need reasons?] :\ud800 -uF8FFF, \uFFF0-uFFFF, \uXFFFE - \uXFFFF (where X is an digit 1 - E), \uF0000 - \uFFFFF.
- The "." character can be used as part of another name, but "." and ".." cannot alone make up the whole name of a path location, because ZooKeeper doesn't use relative paths. The following would be invalid: "/a/b/./c" or "/a/b/../c".
- The token "zookeeper" is reserved.

1.1. ZNodes

Every node in a ZooKeeper tree is refered 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 allow 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... reference here to the section describing the special version number -1]

Note:

In distributed application engineering, the word *node* can refer to a generic host machine, a server, a member of quorums, a client process, etc. In the ZooKeeper documentatin, *znodes* refer to the data nodes. *Servers* to refer to machines that make up the ZooKeeper service; *quorum peers* refer to the servers that make up a quorum; 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.

1.1.1. 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. [tbd: fix this link] [tbd: Ben there is note from to emphasize that "it is queued". What is "it" and is what we have here sufficient?]

1.1.2. 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.

1.1.3. 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.

1.1.4. Unique Naming

Finally you create a znode, you can request that ZooKeeper append a monotonicly increasing counter be appended to the path name of the znode to be requested. This counter is unique to the parent znode.

1.2. 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 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.

1.3. 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.
- **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.

2. ZooKeeper Sessions

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. [tbd: note from Ben: "perhaps capability is a better word." need

clarification on that.] 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 between 2 times the tickTime (as set in the server configuration) and 60 seconds.

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.

3. 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 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. ZooKeeper maintains two lists of

watches: data watches and child watches. getData() and exists() set data watches. getChildren() sets child watches. 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 light weight to set, maintain, and dispatch. It also means if a client connects to a different server, the new server is not going to know about its watches. So, when a client gets a disconnect event, it must consider that an implicit trigger of all watches. When a client reconnects to a new server, the client should re-set any watches that it is still interested in.

3.1. 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.

3.2. 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.)
- When you disconnect from a server (for example, when the server fails), all of the watches you have registered are lost, so you should treat this case as if all your watches were triggered.

4. Consistency Guarantees

ZooKeeper is a high performance, scalable service. Both reads and write operations are designed to be fast, though reads are faster than writes. The reason for this is that in the case of reads, ZooKeeper can serve older data, which in turn is due to ZooKeeper's consistency guarantees:

Sequential Consistency

Updates from a client will be applied in the order that they were sent.

Atomicity

Updates either succeed or fail -- there are no partial results.

Single System Image

A client will see the same view of the service regardless of the server that it connects to.

Reliability

Once an update has been applied, it will persist from that time forward until a client overwrites the update. This guarantee has two corollaries:

1. If a client gets a successful return code, the update will have been applied. On some failures (communication errors, timeouts, etc) the client will not know if the update has applied or not. We take steps to minimize the failures, but the only guarantee is only present with successful return codes. (This is called the _monotonicity condition_ in Paxos.)
2. Any updates that are seen by the client, through a read request or successful update, will never be rolled back when recovering from server failures.

Timeliness

The clients view of the system is guaranteed to be up-to-date within a certain time bound. (On the order of tens of seconds.) Either system changes will be seen by a client within this bound, or the client will detect a service outage.

Using these consistency guarantees it is easy to build higher level functions such as leader election, barriers, queues, and read/write revocable locks solely at the ZooKeeper client (no additions needed to ZooKeeper). See Recipes and Solutions for more details.

Note:

Sometimes developers mistakenly assume one other guarantee that Zookeeper does *not* in fact make. This is:

Simultaneously Conistent Cross-Client Views

ZooKeeper does not guarantee that at every instance in time, two different clients will have identical views of ZooKeeper data. Due to factors like network delays, one client may perform an update before another client gets notified of the

change. Consider the scenario of two clients, A and B. If client A sets the value of a znode /a from 0 to 1, then tells client B to read /a, client B may read the old value of 0, depending on which server in the ZooKeeper quorum it is connected to. If it is important that Client A and Client B read the same value, Client B should should call the **sync()** method from the ZooKeeper API method before it performs its read.

So, ZooKeeper by itself doesn't guarantee instantaneous, atomic, synchronization across its quorum, but ZooKeeper primitives can be used to construct higher level functions that provide complete client synchronization. (For more information, see the Locks [tbd: fix final link target] in Zookeeper Recipes. [tbd: fix final link target]).

5. Bindings

The ZooKeeper client libraries come in two languages: Java and C. The following sections describe these.

5.1. Java Binding

There are two packages that make up the ZooKeeper Java binding: **org.apache.zookeeper** and **org.apache.zookeeper.data**. The rest of the packages that make up ZooKeeper are used internally or are part of the server implementation. The **org.apache.zookeeper.data** package is made up of generated classes that are used simply as containers.

The main class used by a ZooKeeper Java client is the **ZooKeeper** class. Its two constructors differ only by an optional session id and password. ZooKeeper supports session recovery accross instances of a process. A Java program may save its session id and password to stable storage, restart, and recover the session that was used by the earlier instance of the program.

When a ZooKeeper object is created, two threads are created as well: an IO thread and an event thread. All IO happens on the IO thread (using Java NIO). All event callbacks happen on the event thread. Session maintenance such as reconnecting to ZooKeeper servers and maintaining heartbeat is done on the IO thread. Responses for synchronous methods are also processed in the IO thread. All responses to asynchronous methods and watch events are processed on the event thread. There are a few things to notice that result from this design:

- All completions for asynchronous calls and watcher callbacks will be made in order, one at a time. The caller can do any processing they wish, but no other callbacks will be processed during that time.
- Callbacks do not block the processing of the IO thread or the processing of the synchronous calls.
- Synchronous calls may not return in the correct order. For example, assume a client does the following processing: issues an asynchronous read of node **/a** with *watch* set to true, and then in the completion callback of the read it does a synchronous read of **/a**. (Maybe not good practice, but not illegal either, and it makes for a simple example.)

Note that if there is a change to **/a** between the asynchronous read and the synchronous read, the client library will receive the watch event saying **/a** changed before the response for the synchronous read, but because the completion callback is blocking the event queue, the synchronous read will return with the new value of **/a** before the watch event is processed.

Finally, the rules associated with shutdown are straightforward: once a ZooKeeper object is closed or receives a fatal event (SESSION_EXPIRED and AUTH_FAILED), the ZooKeeper object becomes invalid, the two threads shut down, and any further ZooKeeper calls throw errors.

5.2. C Binding

The C binding has a single-threaded and multi-threaded library. The multi-threaded library is easiest to use and is most similar to the Java API. This library will create an IO thread and an event dispatch thread for handling connection maintenance and callbacks. The single-threaded library allows ZooKeeper to be used in event driven applications by exposing the event loop used in the multi-threaded library.

The package includes two shared libraries: zookeeper_st and zookeeper_mt. The former only provides the asynchronous APIs and callbacks for integrating into the application's event loop. The only reason this library exists is to support the platforms were a *pthread* library is not available or is unstable (i.e. FreeBSD 4.x). In all other cases, application developers should link with zookeeper_mt, as it includes support for both Sync and Async API.

5.2.1. Installation

If you're building the client from a check-out from the Apache repository, follow the steps outlined below. If you're building from a project source package downloaded from apache, skip to step **3**.

1. Run `ant compile_just` from the zookeeper top level directory (`.../trunk/zookeeper`). This will create a directory named "generated" under `zookeeper/c`.
2. Change directory to the`zookeeper/c` and run `autoreconf -i` to bootstrap **autoconf**, **automake** and **libtool**. Make sure you have **autoconf version 2.59** or greater installed. Skip to step **4**.
3. If you are building from a project source package, unzip/untar the source tarball and cd to the `zookeeper-x.x.x/` directory.
4. Run `./configure <your-options>` to generate the makefile. Here are some of

options the **configure** utility supports that can be useful in this step:

- `--enable-debug`

  Enables optimization and enables debug info compiler options. (Disabled by default.)
- `--without-syncapi`

  Disables Sync API support; zookeeper_mt library won't be built. (Enabled by default.)
- `--disable-static`

  Do not build static libraries. (Enabled by default.)
- `--disable-shared`

  Do not build shared libraries. (Enabled by default.)

Note:

See INSTALL for general information about running **configure**. [tbd: what is INSTALL? a directory? a file?]

5. Run `make` or `make install` to build the libraries and install them.
6. To generate doxygen documentation for the ZooKeeper API, run `make doxygen-doc`. All documentation will be placed in a new subfolder named docs. By default, this command only generates HTML. For information on other document formats, run `./configure --help`

5.2.2. Using the Client

You can test your client by running a zookeeper server (see instructions on the project wiki page on how to run it) and connecting to it using one of the cli applications that were built as part of the installation procedure. cli_mt (multithreaded, built against zookeeper_mt library) is shown in this example, but you could also use cli_st (singlethreaded, built against zookeeper_st library):

```
$ cli_mt zookeeper_host:9876
```

This is a client application that gives you a shell for executing simple zookeeper commands. Once succesully started and connected to the server it displays a shell prompt. You can now enter zookeeper commands. For example, to create a node:

```
> create /my_new_node
```

To verify that the node's been created:

You should see a list of node who are children of the root node "/". [tbd: document all the cli commands (I think this is Ben's tbd? It's from sourceforge)]

In order to be able to use the ZooKeeper API in your application you have to remember to

1. Include zookeeper header: #include <zookeeper/zookeeper.h
2. If you are building a multithreaded client, compile with -DTHREADED compiler flag to enable the multi-threaded version of the library, and then link against against the `zookeeper_mt` library. If you are building a single-threaded client, do not compile with -DTHREADED, and be sure to link against the `zookeeper_st` library.

Refer to Program Structure, with Simple Examplefor examples of usage in Java and C. [tbd: some kind of short tutorial would be helpful, I guess (ben's tbd?)][tbd: whatever the case, make sure that link points to something.]

6. Building Blocks: A Guide to ZooKeeper Operations

[Engineering input needed. This is a new section. The below is just placeholder, and was actually copied from the overview book. There should probably be a subsection on each of those operations, with a little bit of illustrative code for each op.]

One of the design goals of ZooKeeper is provide a very simple programming interface. As a result, it supports only these operations:

create

creates a node at a location in the tree

delete

deletes a node

exists

tests if a node exists at a location

get data

reads the data from a node

set data

writes data to a node

get children

retrieves a list of children of a node

sync

waits for data to be propagated.

7. Program Structure, with Simple Example

[tbd]

8. Gotchas: Common Problems and Troubleshooting

So now you know ZooKeeper. It's fast, simple, your application works, but wait ... something's wrong. Here are some pitfalls that ZooKeeper users fall into:

1. If you are using watches, you must look for the connected watch event. When a ZooKeeper client disconnects from a server, all the watches are removed, so a client must treat the disconnect event as an implicit trigger of watches. The easiest way to deal with this is to act like the connected watch event is a watch trigger for all your watches. The connected event makes a better trigger than the disconnected event because you can access ZooKeeper and reestablish watches when you are connected.
2. You must test ZooKeeper server failures. The ZooKeeper service can survive failures as long as a majority of servers are active. The question to ask is: can your application handle it? In the real world a client's connection to ZooKeeper can break. (ZooKeeper server failures and network partitions are common reasons for connection loss.) The ZooKeeper client library takes care of recovering your connection and letting you know what happened, but you must make sure that you recover your state and any outstanding requests that failed. Find out if you got it right in the test lab, not in production - test with a ZooKeeper service made up of a several of servers and subject them to reboots.
3. The list of ZooKeeper servers used by the client must match the list of ZooKeeper servers that each ZooKeeper server has. Things can work, although not optimally, if the client list is a subset of the real list of ZooKeeper servers, but not if the client lists ZooKeeper servers not in the ZooKeeper cluster.
4. Be careful where you put that transaction log. The most performance-critical part of ZooKeeper is the transaction log. ZooKeeper must sync transactions to media before it returns a response. A dedicated transaction log device is key to consistent good performance. Putting the log on a busy device will adversely effect performance. If you only have one storage device, put trace files on NFS and increase the snapshotCount; it doesn't eliminate the problem, but it can mitigate it.
5. Set your Java max heap size correctly. It is very important to *avoid swapping.* Going to disk unnecessarily will almost certainly degrade your performance unacceptably. Remember, in ZooKeeper, everything is ordered, so if one request hits the disk, all other queued requests hit the disk.

   To avoid swapping, try to set the heapsize to the amount of physical memory you have, minus the amount needed by the OS and cache. The best way to determine an optimal heap size for your configurations is to *run load tests*. If for some reason you can't, be conservative in your estimates and choose a number well below the limit that would

cause your machine to swap. For example, on a 4G machine, a 3G heap is a conservative estimate to start with.

Outside the formal documentation, there're several other sources of information for ZooKeeper developers.

ZooKeeper Whitepaper [tbd: find url]

The definitive discussion of ZooKeeper design and performance, by Yahoo! Research

API Reference [tbd: find url]

The complete reference to the ZooKeeper API

Zookeeper Talk at the Hadoup Summit 2008

A video introduction to ZooKeeper, by Benjamin Reed of Yahoo! Research

Barrier and Queue Tutorial

The excellent Java tutorial by Flavio Junqueira, implementing simple barriers and producer-consumer queues using ZooKeeper.

ZooKeeper - A Reliable, Scalable Distributed Coordination System

An article by Todd Hoff (07/15/2008)

Zookeeper Recipes [tbd: fix linkend for apache site]

Pseudo-level discussion of the implementation of various synchronization solutions with ZooKeeper: Event Handles, Queues, Locks, and Two-phase Commits.

[tbd]

Whatever good sources anyone can think of...