ZooKeeper Recipes and Solutions

by

Table of contents

1 A Guide to Creating Higher-level Constructs with ZooKeeper

In this article, you'll find guidelines for using ZooKeeper to implement higher order functions. All of them are conventions implemented at the client and do not require special support from ZooKeeper. Hopfully the community will capture these conventions in client-side libraries to ease their use and to encourage standardization.

One of the most interesting things about ZooKeeper is that even though ZooKeeper uses *asynchronous* notifications, you can use it to build *synchronous* consistency primitives, such as queues and locks. As you will see, this is possible because ZooKeeper imposes an overall order on updates, and has mechanisms to expose this ordering.

Note that the recipes below attempt to employ best practices. In particular, they avoid polling, timers or anything else that would result in a "herd effect", causing bursts of traffic and limiting scalability.

There are many useful functions that can be imagined that aren't included here - revocable read-write priority locks, as just one example. And some of the constructs mentioned here - locks, in particular - illustrate certain points, even though you may find other constructs, such as event handles or queues, a more practical means of performing the same function. In general, the examples in this section are designed to stimulate thought.

1.1 Out of the Box Applications: Name Service, Configuration, Group Membership

Name service and configuration are two of the primary applications of ZooKeeper. These two functions are provided directly by the ZooKeeper API.

Another function directly provided by ZooKeeper is *group membership*. The group is represented by a node. Members of the group create ephemeral nodes under the group node. Nodes of the members that fail abnormally will be removed automatically when ZooKeeper detects the failure.

1.2 Barriers

Distributed systems use *barriers* to block processing of a set of nodes until a condition is met at which time all the nodes are allowed to proceed. Barriers are implemented in ZooKeeper by designating a barrier node. The barrier is in place if the barrier node exists. Here's the pseudo code:

1. Client calls the ZooKeeper API's **exists()** function on the barrier node, with *watch* set to true.
2. If **exists()** returns false, the barrier is gone and the client proceeds
3. Else, if **exists()** returns true, the clients wait for a watch event from ZooKeeper for the barrier node.

4. When the watch event is triggered, the client reissues the **exists( )** call, again waiting until the barrier node is removed.

1.2.1 Double Barriers

Double barriers enable clients to synchronize the beginning and the end of a computation. When enough processes have joined the barrier, processes start their computation and leave the barrier once they have finished. This recipe shows how to use a ZooKeeper node as a barrier.

The pseudo code in this recipe represents the barrier node as *b*. Every client process *p* registers with the barrier node on entry and unregisters when it is ready to leave. A node registers with the barrier node via the **Enter** procedure below, it waits until *x* client process register before proceeding with the computation. (The *x* here is up to you to determine for your system.)

Enter	**Leave**
1. Create a name *n = b+"/"+p* 2. Set watch: **exists(*b* + ''/ready'', true)** 3. Create child: **create(*n*, EPHEMERAL)** 4. **L = getChildren(b, false)** 5. if fewer children in L than *x*, wait for watch event 6. else **create(b + ''/ready'', REGULAR)**	1. **L = getChildren(b, false)** 2. if no children, exit 3. if *p* is only process node in L, delete(n) and exit 4. if *p* is the lowest process node in L, wait on highest process node in P 5. else **delete(*n*)** if still exists and wait on lowest process node in L 6. goto 1

On entering, all processes watch on a ready node and create an ephemeral node as a child of the barrier node. Each process but the last enters the barrier and waits for the ready node to appear at line 5. The process that creates the xth node, the last process, will see x nodes in the list of children and create the ready node, waking up the other processes. Note that waiting processes wake up only when it is time to exit, so waiting is efficient.

On exit, you can't use a flag such as *ready* because you are watching for process nodes to go away. By using ephemeral nodes, processes that fail after the barrier has been entered do not prevent correct processes from finishing. When processes are ready to leave, they need to delete their process nodes and wait for all other processes to do the same.

Processes exit when there are no process nodes left as children of *b*. However, as an efficiency, you can use the lowest process node as the ready flag. All other processes that are ready to exit watch for the lowest existing process node to go away, and the owner of the lowest process watches for any other process node (picking the highest for simplicity) to go away. This means that only a single process wakes up on each node deletion except for the last node, which wakes up everyone when it is removed.

1.3 Queues

Distributed queues are a common data structure. To implement a distributed queue in ZooKeeper, first designate a znode to hold the queue, the queue node. The distributed clients put something into the queue by calling create() with a pathname ending in "queue-", with the *sequence* and *ephemeral* flags in the create() call set to true. Because the *sequence* flag is set, the new pathnames will have the form _path-to-queue-node_/queue-X, where X is a monotonic increasing number. A client that wants to be removed from the queue calls ZooKeeper's **getChildren( )** function, with *watch* set to true on the queue node, and begins processing nodes with the lowest number. The client does not need to issue another **getChildren( )** until it exhausts the list obtained from the first **getChildren( )** call. If there are are no children in the queue node, the reader waits for a watch notification to check the queue again.

> Note:
>
> There now exists a Queue implementation in ZooKeeper recipes directory. This is distributed with the release -- src/recipes/queue directory of the release artifact.

1.3.1 Priority Queues

To implement a priority queue, you need only make two simple changes to the generic queue recipe . First, to add to a queue, the pathname ends with "queue-YY" where YY is the priority of the element with lower numbers representing higher priority (just like UNIX). Second, when removing from the queue, a client uses an up-to-date children list meaning that the client will invalidate previously obtained children lists if a watch notification triggers for the queue node.

1.4 Locks

Fully distributed locks that are globally synchronous, meaning at any snapshot in time no two clients think they hold the same lock. These can be implemented using ZooKeeeper. As with priority queues, first define a lock node.

> Note:
>
> There now exists a Lock implementation in ZooKeeper recipes directory. This is distributed with the release -- src/recipes/lock directory of the release artifact.

Clients wishing to obtain a lock do the following:

1. Call **create( )** with a pathname of "_locknode_/lock-" and the *sequence* and *ephemeral* flags set.

2. Call **getChildren( )** on the lock node *without* setting the watch flag (this is important to avoid the herd effect).
3. If the pathname created in step **1** has the lowest sequence number suffix, the client has the lock and the client exits the protocol.
4. The client calls **exists( )** with the watch flag set on the path in the lock directory with the next lowest sequence number.
5. if **exists( )** returns false, go to step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**.

The unlock protocol is very simple: clients wishing to release a lock simply delete the node they created in step 1.

Here are a few things to notice:

- The removal of a node will only cause one client to wake up since each node is watched by exactly one client. In this way, you avoid the herd effect.
- There is no polling or timeouts.
- Because of the way you implement locking, it is easy to see the amount of lock contention, break locks, debug locking problems, etc.

1.4.1 Shared Locks

You can implement shared locks by with a few changes to the lock protocol:

Obtaining a read lock:	**Obtaining a write lock:**
1. Call **create( )** to create a node with pathname "`_locknode_/read-`". This is the lock node use later in the protocol. Make sure to set both the *sequence* and *ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a pathname starting with "`write-`" and having a lower sequence number than the node created in step **1**, the client has the lock and can exit the protocol. 4. Otherwise, call **exists( )**, with *watch* flag, set on the node in lock directory with pathname staring with "`write-`" having the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**.	1. Call **create( )** to create a node with pathname "`_locknode_/write-`". This is the lock node spoken of later in the protocol. Make sure to set both *sequence* and *ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a lower sequence number than the node created in step **1**, the client has the lock and the client exits the protocol. 4. Call **exists( ),** with *watch* flag set, on the node with the pathname that has the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**.

6. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**	

> **Note:**
>
> It might appear that this recipe creates a herd effect: when there is a large group of clients waiting for a read lock, and all getting notified more or less simultaneously when the "`write-`" node with the lowest sequence number is deleted. In fact. that's valid behavior: as all those waiting reader clients should be released since they have the lock. The herd effect refers to releasing a "herd" when in fact only a single or a small number of machines can proceed.

1.4.2 Recoverable Shared Locks

With minor modifications to the Shared Lock protocol, you make shared locks revocable by modifying the shared lock protocol:

In step **1**, of both obtain reader and writer lock protocols, call **getData( )** with *watch* set, immediately after the call to **create( )**. If the client subsequently receives notification for the node it created in step **1**, it does another **getData( )** on that node, with *watch* set and looks for the string "unlock", which signals to the client that it must release the lock. This is because, according to this shared lock protocol, you can request the client with the lock give up the lock by calling **setData()** on the lock node, writing "unlock" to that node.

Note that this protocol requires the lock holder to consent to releasing the lock. Such consent is important, especially if the lock holder needs to do some processing before releasing the lock. Of course you can always implement *Revocable Shared Locks with Freaking Laser Beams* by stipulating in your protocol that the revoker is allowed to delete the lock node if after some length of time the lock isn't deleted by the lock holder.

1.5 Two-phased Commit

A two-phase commit protocol is an algorithm that lets all clients in a distributed system agree either to commit a transaction or abort.

In ZooKeeper, you can implement a two-phased commit by having a coordinator create a transaction node, say "/app/Tx", and one child node per participating site, say "/app/Tx/s_i". When coordinator creates the child node, it leaves the content undefined. Once each site involved in the transaction receives the transaction from the coordinator, the site reads each child node and sets a watch. Each site then processes the query and votes "commit" or "abort" by writing to its respective node. Once the write completes, the other sites are notified, and as soon as all sites have all votes, they can decide either "abort" or "commit". Note that a node can decide "abort" earlier if some site votes for "abort".

An interesting aspect of this implementation is that the only role of the coordinator is to decide upon the group of sites, to create the ZooKeeper nodes, and to propagate the transaction to the corresponding sites. In fact, even propagating the transaction can be done through ZooKeeper by writing it in the transaction node.

There are two important drawbacks of the approach described above. One is the message complexity, which is $O(n^2)$. The second is the impossibility of detecting failures of sites through ephemeral nodes. To detect the failure of a site using ephemeral nodes, it is necessary that the site create the node.

To solve the first problem, you can have only the coordinator notified of changes to the transaction nodes, and then notify the sites once coordinator reaches a decision. Note that this approach is scalable, but it's is slower too, as it requires all communication to go through the coordinator.

To address the second problem, you can have the coordinator propagate the transaction to the sites, and have each site creating its own ephemeral node.

1.6 Leader Election

A simple way of doing leader election with ZooKeeper is to use the **SEQUENCE|EPHEMERAL** flags when creating znodes that represent "proposals" of clients. The idea is to have a znode, say "/election", such that each znode creates a child znode "/election/n_" with both flags SEQUENCE|EPHEMERAL. With the sequence flag, ZooKeeper automatically appends a sequence number that is greater that any one previously appended to a child of "/election". The process that created the znode with the smallest appended sequence number is the leader.

That's not all, though. It is important to watch for failures of the leader, so that a new client arises as the new leader in the case the current leader fails. A trivial solution is to have all application processes watching upon the current smallest znode, and checking if they are the new leader when the smallest znode goes away (note that the smallest znode will go away if the leader fails because the node is ephemeral). But this causes a herd effect: upon of failure of the current leader, all other processes receive a notification, and execute getChildren on "/election" to obtain the current list of children of "/election". If the number of clients is large, it causes a spike on the number of operations that ZooKeeper servers have to process. To avoid the herd effect, it is sufficient to watch for the next znode down on the sequence of znodes. If a client receives a notification that the znode it is watching is gone, then it becomes the new leader in the case that there is no smaller znode. Note that this avoids the herd effect by not having all clients watching the same znode.

Here's the pseudo code:

Let ELECTION be a path of choice of the application. To volunteer to be a leader:

1. Create znode z with path "ELECTION/n_" with both SEQUENCE and EPHEMERAL flags;
2. Let C be the children of "ELECTION", and i be the sequence number of z;
3. Watch for changes on "ELECTION/n_j", where j is the smallest sequence number such that j < i and n_j is a znode in C;

Upon receiving a notification of znode deletion:

1. Let C be the new set of children of ELECTION;
2. If z is the smallest node in C, then execute leader procedure;
3. Otherwise, watch for changes on "ELECTION/n_j", where j is the smallest sequence number such that j < i and n_j is a znode in C;

Note that the znode having no preceding znode on the list of children does not imply that the creator of this znode is aware that it is the current leader. Applications may consider creating a separate to znode to acknowledge that the leader has executed the leader procedure.