zktools Documentation

Release 0.3

Mozilla Foundation

April 15, 2013

CONTENTS

A Python collection of higher-level API's built on top of Apache Zookeeper, a distributed, highly reliable coordination service with no single point of failure.

Features:

- `Locks` - Support for non-blocking acquire, shared read/write locks, and modeled on Python's Lock objects that also includes *Revocable Shared Locks with Freaking Laser Beams* as described in the Zookeeper Recipes.
- `Nodes` - Objects to track values in Zookeeper Nodes (zNodes) automatically.

CHAPTER

ONE

INSTALLING

zktools can be installed via `easy_install` or `pip`:

```
$ easy_install zktools
```

You will also need the Python zookeeper C binding installed. There's an easy to install staticly compiled version:

```
$ easy_install zc-zookeeper-static
```

Or if your OS distribution includes a python-zookeeper package, that can be installed.

CHAPTER

TWO

REFERENCE MATERIAL

Reference material includes documentation for every *zktools* API.

2.1 API Documentation

Comprehensive reference material for every public API exposed by *zktools* is available within this chapter. The API documentation is organized alphabetically by module name.

2.1.1 `zktools.locking`

Zookeeper Locking

This module provides a `ZkLock`, which should look familiar to anyone that has used Python's `threading.Lock` class. In addition to normal locking behavior, revokable shared read/write locks are also supported. All of the locks can be revoked as desired. This requires the current lock holder(s) to release their lock(s).

This is implemented on the Zookeeper side by creating child lock candidate nodes under a parent node and seeing whether the node the lock created is first. It also ensures a first-come/first-serve ordering to who gets the lock after its released.

Asynchronous Lock

The `ZkAsyncLock` uses the Zookeeper async functions to avoid blocking while acquiring a lock, and optionally can use a callback style when the lock was acquired. It has a great amount of flexibility since it can run in the background to establish the lock while the program calling it can decide to block later and wait for the lock acquisition as desired.

If the calling program gets tired of waiting, it can delete the lock candidate node to avoid blocking any other programs waiting on the lock and handle the situation as desired.

Shared Read/Write Locks

Also known in the Zookeeper Recipes as `Revocable Shared Locks with Freaking Laser Beams`, `ZkReadLock` and `ZkWriteLock` locks have been implemented. A read lock can be acquired as long as no write locks are active, while a write-lock can only be acquired when there are no other read or write locks active.

Using the Lock Command Line Interface

zktools comes with a CLI to easily see current locks, details of each lock, and remove empty locks called *zooky*.

Usage:

```
$ zooky list
LOCK                            STATUS
fred                            Locked
zkLockTest                      Free

$ zooky show fred
LOCK HOLDER          DATA            INFO
write-0000000002     0               {'pzxid': 152321L, 'ctime': 1326417195365L, 'aversion': 0, 'mzx
                                     'ephemeralOwner': 86927055090548768L, 'version': 0, 'dataLength'
                                     'cversion': 0, 'modifed_ago': 16, 'created_ago': 16, 'czxid': 15
```

The *modifed_ago* and *created_ago* fields in INFO show how many seconds ago the lock was created and modified.

Constants

zktools.locking.**IMMEDIATE**
Flag used to declare that revocation should occur immediately. Other lock-holders will not be given time to release their lock.

Lock Classes

class zktools.locking.**ZkAsyncLock**(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Asynchronous Zookeeper Lock

This Lock can be established asynchronously in the background.

Example non-blocking use:

```
lock = ZkAsyncLock(zk, '/Mylocks/resourceB')
try:
    lock.acquire()

    # Do some stuff that doesn't care if the lock is
    # established yet, then wait for the lock to acquire
    lock.wait_for_acquire()

    # Do stuff with lock, after checking it was acquired
finally:
    # Release and wait for release
    lock.release()
    lock.wait_for_release()
```

Example blocking use:

```
lock = ZkAsyncLock(zk, '/Mylocks/resourceB')
with lock:
    if not lock.acquired:
        # handle appropriately and return if needed!
    # Won't execute until the lock is acquired
    do_stuff()
# lock is released
do_more_stuff()
```

Warning: It's possible when waiting for a lock, for it to run into errors during acquisition. This is why you should check to see that the lock was actually acquired before proceeding. If it was not and you'd like to know why, the `errors` attribute on the `ZkAsyncLock` will be an array indicating the errors that were encountered.

__init__(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Create an Asynchronous Zookeeper Lock

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **lock_name** – Path to the lock node that should be used
- **lock_root** (*string*) – Path to the root lock node to create the locks under

acquire(*func=None*)
Acquire a lock

Parameters **func** – Function to call when the lock has been acquired. This function will be called with a single argument, the lock instance. The lock's `release()` method should be called to release the lock.

Returns False

acquired
Attribute indicating whether the lock has been acquired

candidate_created
Attribute indicating whether a candidate node has been created

release(*func=None*)
Release a lock, or lock candidate node

This function can be called as long as a lock candidate node has been created. This allows a program to abandon its lock attempt if its been waiting too long, and remove itself from the lock queue.

The lock candidate node can be checked for by checking the value of `ZkAsyncLock.candidate_created`.

Parameters **func** – Function to call when the candidate node has been verifiably confirmed as removed.

Returns False

wait_for_acquire(*timeout=None*)
Waits for lock acquisition

Parameters **timeout** – How long to wait for the lock, defaults to waiting forever

Returns Whether the lock was acquired

Return type bool

wait_for_release(*timeout=None*)
Waits for lock release

Parameters **timeout** – How long to wait for the lock to release, defaults to waiting forever

Returns Whether the lock was released

Return type bool

class zktools.locking.**ZkLock**(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Zookeeper Lock

Implements a Zookeeper based lock optionally with lock revocation should locks be idle for more than a specific set of time.

Example:

```
from zc.zk import ZooKeeper
from zktools.locking import ZkLock

# Create a connection and a lock
conn = ZooKeeper()
my_lock = ZkLock(conn, "my_lock_name")

my_lock.acquire() # wait to acquire lock
# do something with the lock

my_lock.release() # release our lock

# Or, using the context manager
with my_lock:
    # do something with the lock
```

__init__(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Create a Zookeeper lock object

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **lock_root** (*string*) – Path to the root lock node to create the locks under

acquire(*timeout=None*, *revoke=False*)
Acquire a lock

Parameters

- **timeout** (*int*) – How long to wait to acquire the lock, set to 0 to get non-blocking behavior.
- **revoke** (bool or IMMEDIATE) – Whether prior locks should be revoked. Can be set to True to request and wait for prior locks to release their lock, or IMMEDIATE to destroy the blocking read/write locks and attempt to acquire a write lock.

Returns True if the lock was acquired, False otherwise

Return type bool

release()
Release a lock

Returns True if the lock was released, or False if it is no longer valid.

Return type bool

revoked
Indicate if this shared lock has been revoked

Returns True if the lock has been revoked, False otherwise.

Return type bool

revoke_all()
Revoke any existing locks, gently

Unlike clear(), this asks all existing locks to release, rather than forcibly revoking them.

Returns True if existing locks were present, False if there were no existing locks.

Return type bool

has_lock()
Check with Zookeeper to see if the lock is acquired

Returns Whether the lock is acquired or not

Return type bool

clear()
Clear out a lock

> **Warning:** You must be sure this is a dead lock, as clearing it will forcibly release it by deleting all lock nodes.

Returns True if the lock was cleared, or False if it is no longer valid.

Return type bool

Shared Read/Write Lock Classes

class zktools.locking.**ZkReadLock**(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Shared Zookeeper Read Lock

A read-lock is considered successful if there are no active write locks.

This class takes the same initialization parameters as ZkLock.

__init__(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Create a Zookeeper lock object

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **lock_root** (*string*) – Path to the root lock node to create the locks under

acquire(*timeout=None*, *revoke=False*)
Acquire a shared read lock

Parameters

- **timeout** (*int*) – How long to wait to acquire the lock, set to 0 to get non-blocking behavior.
- **revoke** (bool or IMMEDIATE) – Whether prior locks should be revoked. Can be set to True to request and wait for prior locks to release their lock, or IMMEDIATE to destroy the blocking write locks and attempt to acquire a read lock.

Returns True if the lock was acquired, False otherwise

Return type bool

revoked
Indicate if this shared lock has been revoked

Returns True if the lock has been revoked, False otherwise.

Return type bool

has_lock()
Check with Zookeeper to see if the lock is acquired

Returns Whether the lock is acquired or not

Return type bool

revoke_all()
Revoke any existing locks, gently

Unlike clear(), this asks all existing locks to release, rather than forcibly revoking them.

Returns True if existing locks were present, False if there were no existing locks.

Return type bool

release()
Release a lock

Returns True if the lock was released, or False if it is no longer valid.

Return type bool

clear()
Clear out a lock

> **Warning:** You must be sure this is a dead lock, as clearing it will forcibly release it by deleting all lock nodes.

Returns True if the lock was cleared, or False if it is no longer valid.

Return type bool

class zktools.locking.**ZkWriteLock**(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Shared Zookeeper Write Lock

A write-lock is only successful if there are no read or write locks active.

This class takes the same initialization parameters as ZkLock.

__init__(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Create a Zookeeper lock object

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **lock_root** (*string*) – Path to the root lock node to create the locks under

acquire(*timeout=None*, *revoke=False*)
Acquire a shared write lock

Parameters

- **timeout** (*int*) – How long to wait to acquire the lock, set to 0 to get non-blocking behavior.
- **revoke** (bool or IMMEDIATE) – Whether prior locks should be revoked. Can be set to True to request and wait for prior locks to release their lock, or IMMEDIATE to destroy the blocking read/write locks and attempt to acquire a write lock.

Returns True if the lock was acquired, False otherwise

Return type bool

revoked
Indicate if this shared lock has been revoked

Returns True if the lock has been revoked, False otherwise.

Return type bool

has_lock()
Check with Zookeeper to see if the lock is acquired

Returns Whether the lock is acquired or not

Return type bool

revoke_all()
Revoke any existing locks, gently

Unlike `clear()`, this asks all existing locks to release, rather than forcibly revoking them.

Returns True if existing locks were present, False if there were no existing locks.

Return type bool

release()
Release a lock

Returns True if the lock was released, or False if it is no longer valid.

Return type bool

clear()
Clear out a lock

> **Warning:** You must be sure this is a dead lock, as clearing it will forcibly release it by deleting all lock nodes.

Returns True if the lock was cleared, or False if it is no longer valid.

Return type bool

Private Lock Base Class

class zktools.locking.**_LockBase**(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Base lock implementation for subclasses

__init__(*connection*, *lock_name*, *lock_root='/ZktoolsLocks'*)
Create a Zookeeper lock object

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **lock_root** (*string*) – Path to the root lock node to create the locks under

_acquire_lock(*node_name*, *timeout=None*, *revoke=False*)
Acquire a lock

Internal function used by read/write lock

Parameters

- **node_name** (*str*) – Name of the node to use for the lock
- **timeout** (*int*) – How long to wait to acquire the lock, set to 0 to get non-blocking behavior.
- **revoke** (bool or :obj:`IMMEDIATE`) – Whether prior locks should be revoked. Can be set to True to request and wait for prior locks to release their lock, or `IMMEDIATE` to destroy the blocking read/write locks and attempt to acquire a write lock.

Returns True if the lock was acquired, False otherwise

Return type bool

release()
Release a lock

Returns True if the lock was released, or False if it is no longer valid.

Return type bool

revoked
Indicate if this shared lock has been revoked

Returns True if the lock has been revoked, False otherwise.

Return type bool

has_lock()
Check with Zookeeper to see if the lock is acquired

Returns Whether the lock is acquired or not

Return type bool

clear()
Clear out a lock

> **Warning:** You must be sure this is a dead lock, as clearing it will forcibly release it by deleting all lock nodes.

Returns True if the lock was cleared, or False if it is no longer valid.

Return type bool

Internal Utility Functions

zktools.locking.**has_read_lock**(*keyname*, *children*)
Determines if this keyname has a valid read lock

Parameters

- **keyname** (*str*) – The keyname without full path prefix of the current node being examined
- **children** (*list*) – List of the children nodes at this lock point

zktools.locking.**has_write_lock**(*keyname*, *children*)
Determines if this keyname has a valid write lock

Parameters

- **keyname** (*str*) – The keyname without full path prefix of the current node being examined
- **children** (*list*) – List of the children nodes at this lock point

2.1.2 zktools.node

Zookeeper Nodes

This module provides a ZkNode object which can represent a single node from Zookeeper. It can reflect a single value, or a JSON serialized value.

Node Class

class zktools.node.**ZkNode**(*connection*, *path*, *default=None*, *use_json=False*, *permission={'perms': <class 'Mock'>, 'scheme': 'world', 'id': 'anyone'}*, *create_mode=0*)
Zookeeper Node

This object provides access to a single node for updating the value that can also track changes to the value from Zookeeper. Functions can be subscribed to changes in the node's value and/or changes in the node's children (nodes under it being created/removed).

The value of the node is coerced into an appropriate Python object when loaded, current supported conversions:

```
Numbers with decimals      -> Decimal
Numbers without decimals   -> Int
true/false                 -> Bool
none                       -> None
ISO 8601 Date and Datetime -> date or datetime
```

And optionally, with use_json:

```
JSON string                -> dict/list
```

Note: The JSON determination is extremely lax, if it is a string that starts and ends with brackets or curly marks, it is assumed to be a JSON object and will be coerced if possible. If coercion fails, the string will be returned as is.

Example:

```
from zc.zk import ZooKeeper
from zktools.node import ZkNode

conn = ZooKeeper()
node = ZkNode(conn, '/some/config/node')

# prints out the current value, defaults to None
print node.value

# Set the value in zookeeper
node.value = 483.24

# Show the children of the node
print list(node.children)

# Subscribe a function to be called when the node's
# children change (note this will be called immediately
# with a zc.zk Children instance)
@node.children
def my_function(children):
    # do something with node.children or prior_children
```

The default behavior is to track changes to the node, so that the `value` attribute always reflects the node's value in Zookeeper. Additional subscriber functions are called when the Zookeeper event watch is triggered and are run in a separate event thread. Depending on how fast the value/children are changing the subscriber functions may run consecutively and could miss intermediate values.

Return values of subscriber functions are ignored.

> **Warning: Do not delete nodes that are in use**, there intentionally is no code to handle such conditions as it creates overly complex scenarios both for ZkNode and for application code using it.

__init__(*connection*, *path*, *default=None*, *use_json=False*, *permission={'perms': <class 'Mock'>, 'scheme': 'world', 'id': 'anyone'}*, *create_mode=0*)

Create a Zookeeper Node

Creating a ZkNode by default attempts to load the value, and if it is not found will automatically create a blank string as the value.

The last time a `ZkNode` has been modified either by the user or due to a Zookeeper update is recorded as the `ZkNode.last_modified` attribute, as a long in

milliseconds from epoch.

Parameters

- **connection** (*zc.zk Zookeeper instance*) – Zookeeper connection object
- **path** (*str*) – Path to the Zookeeper node
- **default** – A default value if the node is being created
- **use_json** (*bool*) – Whether values that look like a JSON object should be deserialized, and dicts/lists saved as JSON.
- **permission** (*dict*) – Node permission to use if the node is being created.
- **create_mode** (*int*) – Persistent or ephemeral creation mode

value
Returns the current value

If the Zookeeper session expired, it will be reconnected and the value reloaded.

connected
Indicate whether a connection to Zookeeper exists

2.2 Changelog

2.2.1 0.3 (unreleased)

Changes

- Use the Zookeeper-provided mtime of a node as the last_modified attribute, instead of client specific time.time()
- Use GUID from new Zookeeper recipe to handle connection loss which still results in a created node.
- Add safe-call to ensure that commands run reliably in the face of a connection loss exception during their call, and have lock code run in a separate thread to ensure it doesn't deadlock the ZK event thread.
- Removed zc-zookeeper-static requirement as some OS distributions include the python zookeeper binding as a system package.

Bugfixes

- Fix ZkNode issues with dead-locks due to having locking code inside watch callbacks.

Backward Incompatibilities

- Removed ZkNodeDict which establishes a questionable amount of watches on the server. Setting too many watches has multiple drawbacks, larger values should be stored as JSON in a single node.

2.2.2 0.2.1 (02/16/2012)

- Fixed packaging bug.

2.2.3 0.2 (02/03/2012)

Changes

- Added context manager return to lock to allow use of the 'with' statement.
- Refactored to use zc.zk ZooKeeper library for higher level Zookeeper abstraction with automatic watch re-establishment.

Features

- Node object to retrieve ZNode data from Zookeeper and keep it up to date.
- Node objects can have data and children subscribers.
- NodeDict object that maps a shallow tree (one level of children) into a dict-like object.

Backward Incompatibilities

- SharedZkLock has been refactored into ZkWriteLock and ZkReadLock.
- `revoked` is a property of Locks, not a method.
- ZkConnection is gone, lock objects, ZkNode, and ZkNodeDict all expect zc.zk ZooKeeper instances.

2.2.4 0.1 (11/22/2011)

Features

- Lock implementation, with revokable shared locks.
- Zookeeper connection object with automatic reconnect.

CHAPTER

THREE

SOURCE CODE

All source code is available on github under zktools.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *glossary*

CHAPTER

FIVE

LICENSE

`zktools` is offered under the MPL license.

CHAPTER

SIX

AUTHORS

`zktools` is made available by the *Mozilla Foundation*.

PYTHON MODULE INDEX

Z

zktools.locking, ??
zktools.node, ??