kazoo Documentation

Release 1.3.1

Various Authors

June 19, 2014

Contents

Kazoo is a Python library designed to make working with *Zookeeper* a more hassle-free experience that is less prone to errors.

Kazoo features:

- A wide range of recipe implementations, like Lock, Election or Queue
- Data and Children Watchers
- Simplified Zookeeper connection state tracking
- Unified asynchronous API for use with greenlets or threads
- Support for gevent 0.13 and gevent 1.0
- Support for Zookeeper 3.3 and 3.4 servers
- Integrated testing helpers for Zookeeper clusters
- Pure-Python based implementation of the wire protocol, avoiding all the memory leaks, lacking features, and debugging madness of the C library

Kazoo is heavily inspired by Netflix Curator simplifications and helpers.

Note: You should be familiar with Zookeeper and have read the Zookeeper Programmers Guide before using *kazoo*.

CHAPTER 1

Reference Docs

1.1 Installing

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

```
$ pip install kazoo
```

Kazoo implements the Zookeeper protocol in pure Python, so you don't need any Python Zookeeper C bindings installed.

1.2 Basic Usage

1.2.1 Connection Handling

To begin using Kazoo, a `KazooClient` object must be created and a connection established:

```
from kazoo.client import KazooClient

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()
```

By default, the client will connect to a local Zookeeper server on the default port (2181). You should make sure Zookeeper is actually running there first, or the `start` command will be waiting until its default timeout.

Once connected, the client will attempt to stay connected regardless of intermittent connection loss or Zookeeper session expiration. The client can be instructed to drop a connection by calling *stop*:

```
zk.stop()
```

Listening for Connection Events

It can be useful to know when the connection has been dropped, restored, or when the Zookeeper session has expired. To simplify this process Kazoo uses a state system and lets you register listener functions to be called when the state changes.

```
from kazoo.client import KazooState

def my_listener(state):
    if state == KazooState.LOST:
        # Register somewhere that the session was lost
    elif state == KazooState.SUSPENDED
        # Handle being disconnected from Zookeeper
    else:
        # Handle being connected/reconnected to Zookeeper

zk.add_listener(my_listener)
```

When using the `kazoo.recipe.lock.Lock` or creating ephemeral nodes, its highly recommended to add a state listener so that your program can properly deal with connection interruptions or a Zookeeper session loss.

Understanding Kazoo States

The `KazooState` object represents several states the client transitions through. The current state of the client can always be determined by viewing the `state` property. The possible states are:

- LOST
- CONNECTED
- SUSPENDED

When a `KazooClient` instance is first created, it is in the *LOST* state. After a connection is established it transitions to the *CONNECTED* state. If any connection issues come up or if it needs to connect to a different Zookeeper cluster node, it will transition to *SUSPENDED* to let you know that commands cannot currently be run. The connection will also be lost if the Zookeeper node is no longer part of the quorum, resulting in a *SUSPENDED* state.

Upon re-establishing a connection the client could transition to *LOST* if the session has expired, or *CONNECTED* if the session is still valid.

Note: These states should be monitored using a listener as described previously so that the client behaves properly depending on the state of the connection.

When a connection transitions to *SUSPENDED*, if the client is performing an action that requires agreement with other systems (using the Lock recipe for example), it should pause what it's doing. When the connection has been re-established the client can continue depending on if the state is *LOST* or transitions directly to *CONNECTED* again.

When a connection transitions to *LOST*, any ephemeral nodes that have been created will be removed by Zookeeper. This affects all recipes that create ephemeral nodes, such as the Lock recipe. Lock's will need to be re-acquired after the state transitions to *CONNECTED* again. This transition occurs when a session expires or when you stop the clients connection.

Valid State Transitions

- *LOST -> CONNECTED*

  New connection, or previously lost one becoming connected.

- *CONNECTED -> SUSPENDED*

  Connection loss to server occurred on a connection.

- *CONNECTED -> LOST*

  Only occurs if invalid authentication credentials are provided after the connection was established.

- *SUSPENDED -> LOST*

  Connection resumed to server, but then lost as the session was expired.

- *SUSPENDED -> CONNECTED*

  Connection that was lost has been restored.

Read-Only Connections

New in version 0.6. Zookeeper 3.4 and above supports a read-only mode. This mode must be turned on for the servers in the Zookeeper cluster for the client to utilize it. To use this mode with Kazoo, the `KazooClient` should be called with the *read_only* option set to *True*. This will let the client connect to a Zookeeper node that has gone read-only, and the client will continue to scan for other nodes that are read-write.

```
from kazoo.client import KazooClient

zk = KazooClient(hosts='127.0.0.1:2181', read_only=True)
zk.start()
```

A new attribute on `KeeperState` has been added, *CONNECTED_RO*. The connection states above are still valid, however upon *CONNECTED*, you will need to check the clients non- simplified state to see if the connection is *CONNECTED_RO*. For example:

```
from kazoo.client import KazooState
from kazoo.client import KeeperState

@zk.add_listener
def watch_for_ro(state):
    if state == KazooState.CONNECTED:
        if zk.client_state == KeeperState.CONNECTED_RO:
            print("Read only mode!")
        else:
            print("Read/Write mode!")
```

It's important to note that a *KazooState* is passed in to the listener but the read-only information is only available by comparing the non-simplified client state to the *KeeperState* object.

> **Warning:** A client using read-only mode should not use any of the recipes.

1.2.2 Zookeeper CRUD

Zookeeper includes several functions for creating, reading, updating, and deleting Zookeeper nodes (called znodes or nodes here). Kazoo adds several convenience methods and a more Pythonic API.

Creating Nodes

Methods:

- `ensure_path()`
- `create()`

`ensure_path()` will recursively create the node and any nodes in the path necessary along the way, but can not set the data for the node, only the ACL.

`create()` creates a node and can set the data on the node along with a watch function. It requires the path to it to exist first, unless the *makepath* option is set to *True*.

```
# Ensure a path, create if necessary
zk.ensure_path("/my/favorite")

# Create a node with data
zk.create("/my/favorite/node", b"a value")
```

Reading Data

Methods:

- `exists()`
- `get()`
- `get_children()`

`exists()` checks to see if a node exists.

`get()` fetches the data of the node along with detailed node information in a `ZnodeStat` structure.

`get_children()` gets a list of the children of a given node.

```
# Determine if a node exists
if zk.exists("/my/favorite"):
    # Do something

# Print the version of a node and its data
data, stat = zk.get("/my/favorite")
print("Version: %s, data: %s" % (stat.version, data.decode("utf-8")))

# List the children
children = zk.get_children("/my/favorite")
print("There are %s children with names %s" % (len(children), children))
```

Updating Data

Methods:

- `set()`

`set()` updates the data for a given node. A version for the node can be supplied, which will be required to match before updating the data, or a `BadVersionError` will be raised instead of updating.

```
zk.set("/my/favorite", b"some data")
```

Deleting Nodes

Methods:

- `delete()`

`delete()` deletes a node, and can optionally recursively delete all children of the node as well. A version can be supplied when deleting a node which will be required to match the version of the node before deleting it or a `BadVersionError` will be raised instead of deleting.

```
zk.delete("/my/favorite/node", recursive=True)
```

1.2.3 Retrying Commands

Connections to Zookeeper may get interrupted if the Zookeeper server goes down or becomes unreachable. By default, kazoo does not retry commands, so these failures will result in an exception being raised. To assist with failures kazoo comes with a `retry()` helper that will retry a function should one of the Zookeeper connection exceptions get raised.

Example:

```
result = zk.retry(zk.get, "/path/to/node")
```

Some commands may have unique behavior that doesn't warrant automatic retries on a per command basis. For example, if one creates a node a connection might be lost before the command returns successfully but the node actually got created. This results in a `kazoo.exceptions.NodeExistsError` being raised when it runs again. A similar unique situation arises when a node is created with ephemeral and sequence options set, documented here on the Zookeeper site.

Since the `retry()` method takes a function to call and its arguments, a function that runs multiple Zookeeper commands could be passed to it so that the entire function will be retried if the connection is lost.

This snippet from the lock implementation shows how it uses retry to re-run the function acquiring a lock, and checks to see if it was already created to handle this condition:

```
# kazoo.recipe.lock snippet

def acquire(self):
    """Acquire the mutex, blocking until it is obtained"""
    try:
        self.client.retry(self._inner_acquire)
        self.is_acquired = True
    except KazooException:
        # if we did ultimately fail, attempt to clean up
        self._best_effort_cleanup()
        self.cancelled = False
        raise

def _inner_acquire(self):
    self.wake_event.clear()

    # make sure our election parent node exists
    if not self.assured_path:
        self.client.ensure_path(self.path)

    node = None
    if self.create_tried:
        node = self._find_node()
    else:
        self.create_tried = True

    if not node:
        node = self.client.create(self.create_path, self.data,
            ephemeral=True, sequence=True)
        # strip off path to node
        node = node[len(self.path) + 1:]
```

create_tried records whether it has tried to create the node already in the event the connection is lost before the node name is returned.

Custom Retries

Sometimes you may wish to have specific retry policies for a command or set of commands that differs from the `retry()` method. You can manually create a `KazooRetry` instance with the specific retry policy you prefer:

```
from kazoo.retry import KazooRetry

kr = KazooRetry(max_tries=3, ignore_expire=False)
result = kr(client.get, "/some/path")
```

This will retry the `client.get` command up to 3 times, and raise a session expiration if it occurs. You can also make an instance with the default behavior that ignores session expiration during a retry.

1.2.4 Watchers

Kazoo can set watch functions on a node that can be triggered either when the node has changed or when the children of the node change. This change to the node or children can also be the node or its children being deleted.

Watchers can be set in two different ways, the first is the style that Zookeeper supports by default for one-time watch events. These watch functions will be called once by kazoo, and do *not* receive session events, unlike the native Zookeeper watches. Using this style requires the watch function to be passed to one of these methods:

- `get()`
- `get_children()`
- `exists()`

A watch function passed to `get()` or `exists()` will be called when the data on the node changes or the node itself is deleted. It will be passed a `WatchedEvent` instance.

```
def my_func(event):
    # check to see what the children are now

# Call my_func when the children change
children = zk.get_children("/my/favorite/node", watch=my_func)
```

Kazoo includes a higher level API that watches for data and children modifications that's easier to use as it doesn't require re-setting the watch every time the event is triggered. It also passes in the data and `ZnodeStat` when watching a node or the list of children when watching a nodes children. Watch functions registered with this API will be called immediately and every time there's a change, or until the function returns False. If *allow_session_lost* is set to *True*, then the function will no longer be called if the session is lost.

The following methods provide this functionality:

- `ChildrenWatch`
- `DataWatch`

These classes are available directly on the `KazooClient` instance and don't require the client object to be passed in when used in this manner. The instance returned by instantiating either of the classes can be called directly allowing them to be used as decorators:

```
@zk.ChildrenWatch("/my/favorite/node")
def watch_children(children):
    print("Children are now: %s" % children)
# Above function called immediately, and from then on

@zk.DataWatch("/my/favorite")
```

```
def watch_node(data, stat):
    print("Version: %s, data: %s" % (stat.version, data.decode("utf-8")))
```

1.2.5 Transactions

New in version 0.6. Zookeeper 3.4 and above supports the sending of multiple commands at once that will be committed as a single atomic unit. Either they will all succeed or they will all fail. The result of a transaction will be a list of the success/failure results for each command in the transaction.

```
transaction = zk.transaction()
transaction.check('/node/a', version=3)
transaction.create('/node/b', b"a value")
results = transaction.commit()
```

The `transaction()` method returns a `TransactionRequest` instance. It's methods may be called to queue commands to be completed in the transaction. When the transaction is ready to be sent, the `commit()` method on it is called.

In the example above, there's a command not available unless a transaction is being used, *check*. This can check nodes for a specific version, which could be used to make the transaction fail if a node doesn't match a version that it should be at. In this case the node */node/a* must be at version 3 or */node/b* will not be created.

1.3 Asynchronous Usage

The asynchronous Kazoo API relies on the `IAsyncResult` object which is returned by all the asynchronous methods. Callbacks can be added with the `rawlink()` method which works in a consistent manner whether threads or an asynchronous framework like gevent is used.

Kazoo utilizes a pluggable `IHandler` interface which abstracts the callback system to ensure it works consistently.

1.3.1 Connection Handling

Creating a connection:

```
from kazoo.client import KazooClient
from kazoo.handlers.gevent import SequentialGeventHandler

zk = KazooClient(handler=SequentialGeventHandler())

# returns immediately
event = zk.start_async()

# Wait for 30 seconds and see if we're connected
event.wait(timeout=30)

if not zk.connected:
    # Not connected, stop trying to connect
    zk.stop()
    raise Exception("Unable to connect.")
```

In this example, the *wait* method is used on the event object returned by the `start_async()` method. A timeout is **always** used because its possible that we might never connect and that should be handled gracefully.

The SequentialGeventHandler is used when you want to use gevent. Kazoo doesn't rely on gevents monkey patching and requires that you pass in the appropriate handler, the default handler is SequentialThreadingHandler.

1.3.2 Asynchronous Callbacks

All kazoo *_async* methods except for start_async() return an IAsyncResult instance. These instances allow you to see when a result is ready, or chain one or more callback functions to the result that will be called when it's ready.

The callback function will be passed the IAsyncResult instance and should call the get() method on it to retrieve the value. This call could result in an exception being raised if the asynchronous function encountered an error. It should be caught and handled appropriately.

Example:

```
import sys

from kazoo.exceptions import ConnectionLossException
from kazoo.exceptions import NoAuthException

def my_callback(async_obj):
    try:
        children = async_obj.get()
        do_something(children)
    except (ConnectionLossException, NoAuthException):
        sys.exit(1)

# Both these statements return immediately, the second sets a callback
# that will be run when get_children_async has its return value
async_obj = zk.get_children_async("/some/node")
async_obj.rawlink(my_callback)
```

1.3.3 Zookeeper CRUD

The following CRUD methods all work the same as their synchronous counterparts except that they return an IAsyncResult object.

Creating Method:

- create_async()

Reading Methods:

- exists_async()
- get_async()
- get_children_async()

Updating Methods:

- set_async()

Deleting Methods:

- delete_async()

The ensure_path() has no asynchronous counterpart at the moment nor can the delete_async() method do recursive deletes.

1.4 Implementation Details

Up to version 0.3 kazoo used the Python bindings to the Zookeeper C library. Unfortunately those bindings are fairly buggy and required a fair share of weird workarounds to interface with the native OS thread used in those bindings.

Starting with version 0.4 kazoo implements the entire Zookeeper wire protocol itself in pure Python. Doing so removed the need for the workarounds and made it much easier to implement the features missing in the C bindings.

1.4.1 Handlers

Both the Kazoo handlers run 3 separate queues to help alleviate deadlock issues and ensure consistent execution order regardless of environment. The `SequentialGeventHandler` runs a separate greenlet for each queue that processes the callbacks queued in order. The `SequentialThreadingHandler` runs a separate thread for each queue that processes the callbacks queued in order (thus the naming scheme which notes they are sequential in anticipation that there could be handlers shipped in the future which don't make this guarantee).

Callbacks are queued by type, the 3 types being:

1. Session events (State changes, registered listener functions)
2. Watch events (Watch callbacks, DataWatch, and ChildrenWatch functions)
3. Completion callbacks (Functions chained to `IAsyncResult` objects)

This ensures that calls can be made to Zookeeper from any callback **except for a state listener** without worrying that critical session events will be blocked.

> **Warning:** Its important to remember that if you write code that blocks in one of these functions then no queued functions of that type will be executed until the code stops blocking. If your code might block, it should run itself in a separate greenlet/thread so that the other callbacks can run.

1.5 Testing

Kazoo has several test harnesses used internally for its own tests that are exposed as public API's for use in your own tests for common Zookeeper cluster management and session testing. They can be mixed in with your own *unittest* or *nose* tests along with a *mock* object that allows you to force specific *KazooClient* commands to fail in various ways.

The test harness needs to be able to find the Zookeeper Java libraries. You need to specify an environment variable called *ZOOKEEPER_PATH* and point it to their location, for example */usr/share/java*. The directory should contain a *zookeeper-*.jar* and a *lib* directory containing at least a *log4j-*.jar*.

If your Java setup is complex, you may also override our classpath mechanism completely by specifying an environment variable called *ZOOKEEPER_CLASSPATH*. If provided, it will be used unmodified as the Java classpath for Zookeeper.

1.5.1 Kazoo Test Harness

The `KazooTestHarness` can be used directly or mixed in with your test code.

Example:

```
from kazoo.testing import KazooTestHarness

class MyTest(KazooTestHarness):
```

```
    def setUp(self):
        self.setup_zookeeper()

    def tearDown(self):
        self.teardown_zookeeper()

    def testmycode(self):
        self.client.ensure_path('/test/path')
        result = self.client.get('/test/path')
        ...
```

1.5.2 Kazoo Test Case

The `KazooTestCase` is complete test case that is equivalent to the mixin setup of `KazooTestHarness`. An equivalent test to the one above:

```
from kazoo.testing import KazooTestCase

class MyTest(KazooTestCase):
    def testmycode(self):
        self.client.ensure_path('/test/path')
        result = self.client.get('/test/path')
        ...
```

1.6 API Documentation

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

1.6.1 kazoo.client

Kazoo Zookeeper Client

Public API

class kazoo.client.**KazooClient**

An Apache Zookeeper Python client supporting alternate callback handlers and high-level functionality.

Watch functions registered with this class will not get session events, unlike the default Zookeeper watches. They will also be called with a single argument, a `WatchedEvent` instance.

__init__(*hosts='127.0.0.1:2181'*, *timeout=10.0*, *client_id=None*, *handler=None*, *default_acl=None*, *auth_data=None*, *read_only=None*, *randomize_hosts=True*, *connection_retry=None*, *command_retry=None*, *logger=None*, ***kwargs*)

Create a `KazooClient` instance. All time arguments are in seconds.

Parameters

- **hosts** – Comma-separated list of hosts to connect to (e.g. 127.0.0.1:2181,127.0.0.1:2182,[::1]:2183).
- **timeout** – The longest to wait for a Zookeeper connection.
- **client_id** – A Zookeeper client id, used when re-establishing a prior session connection.

- **handler** – An instance of a class implementing the `IHandler` interface for callback handling.
- **default_acl** – A default ACL used on node creation.
- **auth_data** – A list of authentication credentials to use for the connection. Should be a list of (scheme, credential) tuples as `add_auth()` takes.
- **read_only** – Allow connections to read only servers.
- **randomize_hosts** – By default randomize host selection.
- **connection_retry** – A `kazoo.retry.KazooRetry` object to use for retrying the connection to Zookeeper. Also can be a dict of options which will be used for creating one.
- **command_retry** – A `kazoo.retry.KazooRetry` object to use for the `KazooClient.retry()` method. Also can be a dict of options which will be used for creating one.
- **logger** – A custom logger to use instead of the module global *log* instance.

Basic Example:

```
zk = KazooClient()
zk.start()
children = zk.get_children('/')
zk.stop()
```

As a convenience all recipe classes are available as attributes and get automatically bound to the client. For example:

```
zk = KazooClient()
zk.start()
lock = zk.Lock('/lock_path')
```

New in version 0.6: The read_only option. Requires Zookeeper 3.4+New in version 0.6: The retry_max_delay option.New in version 0.6: The randomize_hosts option.Changed in version 0.8: Removed the unused watcher argument (was second argument).New in version 1.2: The connection_retry, command_retry and logger options.

handler
The `IHandler` strategy used by this client. Gives access to appropriate synchronization objects.

retry (*func*, **args*, ***kwargs*)
Runs the given function with the provided arguments, retrying if it fails because the ZooKeeper connection is lost, see *Retrying Commands*.

state
A `KazooState` attribute indicating the current higher-level connection state.

client_state
Returns the last Zookeeper client state

This is the non-simplified state information and is generally not as useful as the simplified KazooState information.

client_id
Returns the client id for this Zookeeper session if connected.

Returns client id which consists of the session id and password.
Return type tuple

connected
Returns whether the Zookeeper connection has been established.

add_listener(*listener*)
Add a function to be called for connection state changes.

This function will be called with a `KazooState` instance indicating the new connection state on state transitions.

> **Warning:** This function must not block. If its at all likely that it might need data or a value that could result in blocking than the `spawn()` method should be used so that the listener can return immediately.

remove_listener(*listener*)
Remove a listener function

start(*timeout=15*)
Initiate connection to ZK.
Parameters **timeout** – Time in seconds to wait for connection to succeed.
Raises `timeout_exception` if the connection wasn't established within *timeout* seconds.

start_async()
Asynchronously initiate connection to ZK.
Returns An event object that can be checked to see if the connection is alive.
Return type `Event` compatible object.

stop()
Gracefully stop this Zookeeper session.

This method can be called while a reconnection attempt is in progress, which will then be halted.

Once the connection is closed, its session becomes invalid. All the ephemeral nodes in the ZooKeeper server associated with the session will be removed. The watches left on those nodes (and on their parents) will be triggered.

restart()
Stop and restart the Zookeeper session.

close()
Free any resources held by the client.

This method should be called on a stopped client before it is discarded. Not doing so may result in filehandles being leaked. New in version 1.0.

command(*cmd='ruok'*)
Sent a management command to the current ZK server.

Examples are *ruok*, *envi* or *stat*.
Returns An unstructured textual response.
Return type str
Raises `ConnectionLoss` if there is no connection open, or possibly a `socket.error` if there's a problem with the connection used just for this command.
New in version 0.5.

server_version()
Get the version of the currently connected ZK server.
Returns The server version, for example (3, 4, 3).
Return type tuple
New in version 0.5.

add_auth(*scheme*, *credential*)
Send credentials to server.

Parameters
- **scheme** – authentication scheme (default supported: "digest").
- **credential** – the credential – value depends on scheme.

add_auth_async(*scheme*, *credential*)
Asynchronously send credentials to server. Takes the same arguments as `add_auth()`.

Return type `IAsyncResult`

unchroot(*path*)
Strip the chroot if applicable from the path.

sync_async(*path*)
Asynchronous sync.

Return type `IAsyncResult`

sync(*path*)
Sync, blocks until response is acknowledged.

Flushes channel between process and leader.

Parameters **path** – path of node.

Returns The node path that was synced.

Raises `ZookeeperError` if the server returns a non-zero error code.

New in version 0.5.

create(*path*, *value=''*, *acl=None*, *ephemeral=False*, *sequence=False*, *makepath=False*)
Create a node with the given value as its data. Optionally set an ACL on the node.

The ephemeral and sequence arguments determine the type of the node.

An ephemeral node will be automatically removed by ZooKeeper when the session associated with the creation of the node expires.

A sequential node will be given the specified path plus a suffix *i* where i is the current sequential number of the node. The sequence number is always fixed length of 10 digits, 0 padded. Once such a node is created, the sequential number will be incremented by one.

If a node with the same actual path already exists in ZooKeeper, a NodeExistsError will be raised. Note that since a different actual path is used for each invocation of creating sequential nodes with the same path argument, the call will never raise NodeExistsError.

If the parent node does not exist in ZooKeeper, a NoNodeError will be raised. Setting the optional *makepath* argument to *True* will create all missing parent nodes instead.

An ephemeral node cannot have children. If the parent node of the given path is ephemeral, a NoChildrenForEphemeralsError will be raised.

This operation, if successful, will trigger all the watches left on the node of the given path by `exists()` and `get()` API calls, and the watches left on the parent node by `get_children()` API calls.

The maximum allowable size of the node value is 1 MB. Values larger than this will cause a ZookeeperError to be raised.

Parameters
- **path** – Path of node.
- **value** – Initial bytes value of node.
- **acl** – `ACL` list.
- **ephemeral** – Boolean indicating whether node is ephemeral (tied to this session).
- **sequence** – Boolean indicating whether path is suffixed with a unique index.
- **makepath** – Whether the path should be created if it doesn't exist.

Returns Real path of the new node.
Return type str
Raises `NodeExistsError` if the node already exists.

`NoNodeError` if parent nodes are missing.

`NoChildrenForEphemeralsError` if the parent node is an ephemeral node.

`ZookeeperError` if the provided value is too large.

`ZookeeperError` if the server returns a non-zero error code.

create_async(*path*, *value=''*, *acl=None*, *ephemeral=False*, *sequence=False*, *makepath=False*)
Asynchronously create a ZNode. Takes the same arguments as `create()`.
Return type `IAsyncResult`
New in version 1.1: The makepath option.

ensure_path(*path*, *acl=None*)
Recursively create a path if it doesn't exist.
Parameters
- **path** – Path of node.
- **acl** – Permissions for node.

ensure_path_async(*path*, *acl=None*)
Recursively create a path asynchronously if it doesn't exist. Takes the same arguments as `ensure_path()`.
Return type `IAsyncResult`
New in version 1.1.

exists(*path*, *watch=None*)
Check if a node exists.

If a watch is provided, it will be left on the node with the given path. The watch will be triggered by a successful operation that creates/deletes the node or sets the data on the node.
Parameters
- **path** – Path of node.
- **watch** – Optional watch callback to set for future changes to this path.

Returns ZnodeStat of the node if it exists, else None if the node does not exist.
Return type `ZnodeStat` or *None*.
Raises `ZookeeperError` if the server returns a non-zero error code.

exists_async(*path*, *watch=None*)
Asynchronously check if a node exists. Takes the same arguments as `exists()`.
Return type `IAsyncResult`

get(*path*, *watch=None*)
Get the value of a node.

If a watch is provided, it will be left on the node with the given path. The watch will be triggered by a successful operation that sets data on the node, or deletes the node.
Parameters
- **path** – Path of node.
- **watch** – Optional watch callback to set for future changes to this path.

Returns Tuple (value, `ZnodeStat`) of node.
Return type tuple
Raises `NoNodeError` if the node doesn't exist

`ZookeeperError` if the server returns a non-zero error code

get_async(*path*, *watch=None*)
Asynchronously get the value of a node. Takes the same arguments as `get()`.
Return type `IAsyncResult`

get_children(*path*, *watch=None*, *include_data=False*)
Get a list of child nodes of a path.

If a watch is provided it will be left on the node with the given path. The watch will be triggered by a successful operation that deletes the node of the given path or creates/deletes a child under the node.

The list of children returned is not sorted and no guarantee is provided as to its natural or lexical order.
Parameters
- **path** – Path of node to list.
- **watch** – Optional watch callback to set for future changes to this path.
- **include_data** – Include the `ZnodeStat` of the node in addition to the children. This option changes the return value to be a tuple of (children, stat).

Returns List of child node names, or tuple if *include_data* is *True*.
Return type list
Raises `NoNodeError` if the node doesn't exist.

`ZookeeperError` if the server returns a non-zero error code.
New in version 0.5: The *include_data* option.

get_children_async(*path*, *watch=None*, *include_data=False*)
Asynchronously get a list of child nodes of a path. Takes the same arguments as `get_children()`.
Return type `IAsyncResult`

get_acls(*path*)
Return the ACL and stat of the node of the given path.
Parameters **path** – Path of the node.
Returns The ACL array of the given node and its `ZnodeStat`.
Return type tuple of (`ACL` list, `ZnodeStat`)
Raises `NoNodeError` if the node doesn't exist.

`ZookeeperError` if the server returns a non-zero error code
New in version 0.5.

get_acls_async(*path*)
Return the ACL and stat of the node of the given path. Takes the same arguments as `get_acls()`.
Return type `IAsyncResult`

set_acls(*path*, *acls*, *version=-1*)
Set the ACL for the node of the given path.

Set the ACL for the node of the given path if such a node exists and the given version matches the version of the node.
Parameters
- **path** – Path for the node.
- **acls** – List of `ACL` objects to set.
- **version** – The expected node version that must match.

Returns The stat of the node.
Raises `BadVersionError` if version doesn't match.

`NoNodeError` if the node doesn't exist.

`InvalidACLError` if the ACL is invalid.

ZookeeperError if the server returns a non-zero error code.

New in version 0.5.

set_acls_async(*path*, *acls*, *version=-1*)

Set the ACL for the node of the given path. Takes the same arguments as set_acls().

Return type IAsyncResult

set(*path*, *value*, *version=-1*)

Set the value of a node.

If the version of the node being updated is newer than the supplied version (and the supplied version is not -1), a BadVersionError will be raised.

This operation, if successful, will trigger all the watches on the node of the given path left by get() API calls.

The maximum allowable size of the value is 1 MB. Values larger than this will cause a ZookeeperError to be raised.

Parameters

- **path** – Path of node.
- **value** – New data value.
- **version** – Version of node being updated, or -1.

Returns Updated ZnodeStat of the node.

Raises BadVersionError if version doesn't match.

NoNodeError if the node doesn't exist.

ZookeeperError if the provided value is too large.

ZookeeperError if the server returns a non-zero error code.

set_async(*path*, *value*, *version=-1*)

Set the value of a node. Takes the same arguments as set().

Return type IAsyncResult

transaction()

Create and return a TransactionRequest object

Creates a TransactionRequest object. A Transaction can consist of multiple operations which can be committed as a single atomic unit. Either all of the operations will succeed or none of them.

Returns A TransactionRequest.

Return type TransactionRequest

New in version 0.6: Requires Zookeeper 3.4+

delete(*path*, *version=-1*, *recursive=False*)

Delete a node.

The call will succeed if such a node exists, and the given version matches the node's version (if the given version is -1, the default, it matches any node's versions).

This operation, if successful, will trigger all the watches on the node of the given path left by *exists* API calls, and the watches on the parent node left by *get_children* API calls.

Parameters

- **path** – Path of node to delete.
- **version** – Version of node to delete, or -1 for any.
- **recursive** (*bool*) – Recursively delete node and all its children, defaults to False.

Raises BadVersionError if version doesn't match.

NoNodeError if the node doesn't exist.

NotEmptyError if the node has children.

ZookeeperError if the server returns a non-zero error code.

delete_async(*path*, *version=-1*)
Asynchronously delete a node. Takes the same arguments as delete(), with the exception of *recursive*.
Return type IAsyncResult

class kazoo.client.**TransactionRequest**(*client*)
A Zookeeper Transaction Request

A Transaction provides a builder object that can be used to construct and commit an atomic set of operations. The transaction must be committed before its sent.

Transactions are not thread-safe and should not be accessed from multiple threads at once. New in version 0.6: Requires Zookeeper 3.4+

create(*path*, *value=''*, *acl=None*, *ephemeral=False*, *sequence=False*)
Add a create ZNode to the transaction. Takes the same arguments as KazooClient.create(), with the exception of *makepath*.
Returns None

delete(*path*, *version=-1*)
Add a delete ZNode to the transaction. Takes the same arguments as KazooClient.delete(), with the exception of *recursive*.

set_data(*path*, *value*, *version=-1*)
Add a set ZNode value to the transaction. Takes the same arguments as KazooClient.set().

check(*path*, *version*)
Add a Check Version to the transaction.

This command will fail and abort a transaction if the path does not match the specified version.

commit_async()
Commit the transaction asynchronously.
Return type IAsyncResult

commit()
Commit the transaction.
Returns A list of the results for each operation in the transaction.

1.6.2 kazoo.exceptions

Kazoo Exceptions

Public API

exception kazoo.exceptions.**KazooException**
Base Kazoo exception that all other kazoo library exceptions inherit from

exception kazoo.exceptions.**ZookeeperError**
Base Zookeeper exception for errors originating from the Zookeeper server

exception kazoo.exceptions.**AuthFailedError**

exception kazoo.exceptions.**BadVersionError**

exception kazoo.exceptions.**ConfigurationError**
Raised if the configuration arguments to an object are invalid

exception kazoo.exceptions.**InvalidACLError**

exception kazoo.exceptions.**LockTimeout**
Raised if failed to acquire a lock. New in version 1.1.

exception kazoo.exceptions.**NoChildrenForEphemeralsError**

exception kazoo.exceptions.**NodeExistsError**

exception kazoo.exceptions.**NoNodeError**

exception kazoo.exceptions.**NotEmptyError**

Private API

exception kazoo.exceptions.**APIError**

exception kazoo.exceptions.**BadArgumentsError**

exception kazoo.exceptions.**CancelledError**
Raised when a process is cancelled by another thread

exception kazoo.exceptions.**ConnectionDropped**
Internal error for jumping out of loops

exception kazoo.exceptions.**ConnectionClosedError**
Connection is closed

exception kazoo.exceptions.**ConnectionLoss**

exception kazoo.exceptions.**DataInconsistency**

exception kazoo.exceptions.**MarshallingError**

exception kazoo.exceptions.**NoAuthError**

exception kazoo.exceptions.**NotReadOnlyCallError**
An API call that is not read-only was used while connected to a read-only server

exception kazoo.exceptions.**InvalidCallbackError**

exception kazoo.exceptions.**OperationTimeoutError**

exception kazoo.exceptions.**RolledBackError**

exception kazoo.exceptions.**RuntimeInconsistency**

exception kazoo.exceptions.**SessionExpiredError**

exception kazoo.exceptions.**SessionMovedError**

exception kazoo.exceptions.**SystemZookeeperError**

exception kazoo.exceptions.**UnimplementedError**

exception kazoo.exceptions.**WriterNotClosedException**
Raised if the writer is unable to stop closing when requested. New in version 1.2.

exception kazoo.exceptions.**ZookeeperStoppedError**
Raised when the kazoo client stopped (and thus not connected)

1.6.3 kazoo.handlers.gevent

A gevent based handler.

Public API

class kazoo.handlers.gevent.**SequentialGeventHandler**
Gevent handler for sequentially executing callbacks.

This handler executes callbacks in a sequential manner. A queue is created for each of the callback events, so that each type of event has its callback type run sequentially.

Each queue type has a greenlet worker that pulls the callback event off the queue and runs it in the order the client sees it.

This split helps ensure that watch callbacks won't block session re-establishment should the connection be lost during a Zookeeper client call.

Watch callbacks should avoid blocking behavior as the next callback of that type won't be run until it completes. If you need to block, spawn a new greenlet and return immediately so callbacks can proceed.

async_result()
Create a AsyncResult instance

The AsyncResult instance will have its completion callbacks executed in the thread the SequentialGeventHandler is created in (which should be the gevent/main thread).

dispatch_callback(*callback*)
Dispatch to the callback object

The callback is put on separate queues to run depending on the type as documented for the SequentialGeventHandler.

event_object()
Create an appropriate Event object

lock_object()
Create an appropriate Lock object

rlock_object()
Create an appropriate RLock object

static sleep_func(*seconds=0*, *ref=True*)
Put the current greenlet to sleep for at least *seconds*.

seconds may be specified as an integer, or a float if fractional seconds are desired.

If *ref* is false, the greenlet running sleep() will not prevent gevent.wait() from exiting.

spawn(*func*, **args*, ***kwargs*)
Spawn a function to run asynchronously

start()
Start the greenlet workers.

stop()
Stop the greenlet workers and empty all queues.

Private API

class kazoo.handlers.gevent.**AsyncResult**
A one-time event that stores a value or an exception.

Like `Event` it wakes up all the waiters when `set()` or `set_exception()` method is called. Waiters may receive the passed value or exception by calling `get()` method instead of `wait()`. An `AsyncResult` instance cannot be reset.

To pass a value call `set()`. Calls to `get()` (those that currently blocking as well as those made in the future) will return the value:

```
>>> result = AsyncResult()
>>> result.set(100)
>>> result.get()
100
```

To pass an exception call `set_exception()`. This will cause `get()` to raise that exception:

```
>>> result = AsyncResult()
>>> result.set_exception(RuntimeError('failure'))
>>> result.get()
Traceback (most recent call last):
 ...
RuntimeError: failure
```

`AsyncResult` implements `__call__()` and thus can be used as `link()` target:

```
>>> import gevent
>>> result = AsyncResult()
>>> gevent.spawn(lambda : 1/0).link(result)
>>> result.get()
Traceback (most recent call last):
 ...
ZeroDivisionError: integer division or modulo by zero
```

exception
Holds the exception instance passed to `set_exception()` if `set_exception()` was called. Otherwise `None`.

get(*block=True*, *timeout=None*)
Return the stored value or raise the exception.

If this instance already holds a value / an exception, return / raise it immediatelly. Otherwise, block until another greenlet calls `set()` or `set_exception()` or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

get_nowait()
Return the value or raise the exception without blocking.

If nothing is available, raise `gevent.Timeout` immediatelly.

rawlink(*callback*)
Register a callback to call when a value or an exception is set.

callback will be called in the `Hub`, so it must not use blocking gevent API. *callback* will be passed one argument: this instance.

ready()
Return true if and only if it holds a value or an exception

set(*value=None*)
Store the value. Wake up the waiters.

All greenlets blocking on `get()` or `wait()` are woken up. Sequential calls to `wait()` and `get()` will not block at all.

set_exception(*exception*)
Store the exception. Wake up the waiters.

All greenlets blocking on `get()` or `wait()` are woken up. Sequential calls to `wait()` and `get()` will not block at all.

successful()
Return true if and only if it is ready and holds a value

unlink(*callback*)
Remove the callback set by `rawlink()`

wait(*timeout=None*)
Block until the instance is ready.

If this instance already holds a value / an exception, return immediatelly. Otherwise, block until another thread calls `set()` or `set_exception()` or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

Return `value`.

1.6.4 kazoo.handlers.threading

A threading based handler.

The `SequentialThreadingHandler` is intended for regular Python environments that use threads.

Warning: Do not use `SequentialThreadingHandler` with applications using asynchronous event loops (like gevent). Use the `SequentialGeventHandler` instead.

Public API

class kazoo.handlers.threading.**SequentialThreadingHandler**
Threading handler for sequentially executing callbacks.

This handler executes callbacks in a sequential manner. A queue is created for each of the callback events, so that each type of event has its callback type run sequentially. These are split into two queues, one for watch events and one for async result completion callbacks.

Each queue type has a thread worker that pulls the callback event off the queue and runs it in the order the client sees it.

This split helps ensure that watch callbacks won't block session re-establishment should the connection be lost during a Zookeeper client call.

Watch and completion callbacks should avoid blocking behavior as the next callback of that type won't be run until it completes. If you need to block, spawn a new thread and return immediately so callbacks can proceed.

Note: Completion callbacks can block to wait on Zookeeper calls, but no other completion callbacks will execute until the callback returns.

---

async_result()
Create a AsyncResult instance

dispatch_callback(*callback*)
Dispatch to the callback object

The callback is put on separate queues to run depending on the type as documented for the SequentialThreadingHandler.

event_object()
Create an appropriate Event object

lock_object()
Create a lock object

rlock_object()
Create an appropriate RLock object

sleep_func()
sleep(seconds)

Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

start()
Start the worker threads.

stop()
Stop the worker threads and empty all queues.

Private API

class kazoo.handlers.threading.**AsyncResult**(*handler*)
A one-time event that stores a value or an exception

get(*block=True*, *timeout=None*)
Return the stored value or raise the exception.

If there is no value raises TimeoutError.

get_nowait()
Return the value or raise the exception without blocking.

If nothing is available, raises TimeoutError

rawlink(*callback*)
Register a callback to call when a value or an exception is set

ready()
Return true if and only if it holds a value or an exception

set(*value=None*)
Store the value. Wake up the waiters.

set_exception(*exception*)
Store the exception. Wake up the waiters.

successful()
Return true if and only if it is ready and holds a value

unlink(*callback*)
Remove the callback set by rawlink()

wait(*timeout=None*)
Block until the instance is ready.

exception kazoo.handlers.threading.**TimeoutError**

1.6.5 kazoo.handlers.utils

Kazoo handler helpers

Public API

kazoo.handlers.utils.**capture_exceptions**(*async_result*)
Return a new decorated function that propagates the exceptions of the wrapped function to an async_result.

Parameters **async_result** – An async result implementing IAsyncResult

kazoo.handlers.utils.**wrap**(*async_result*)
Return a new decorated function that propagates the return value or exception of wrapped function to an async_result. NOTE: Only propagates a non-None return value.

Parameters **async_result** – An async result implementing IAsyncResult

Private API

kazoo.handlers.utils.**create_pipe**()
Create a non-blocking read/write pipe.

kazoo.handlers.utils.**create_tcp_socket**(*module*)
Create a TCP socket with the CLOEXEC flag set.

1.6.6 kazoo.interfaces

Kazoo Interfaces

Public API

IHandler implementations should be created by the developer to be passed into KazooClient during instantiation for the preferred callback handling.

If the developer needs to use objects implementing the IAsyncResult interface, the IHandler.async_result() method must be used instead of instantiating one directly.

interface kazoo.interfaces.**IHandler**
A Callback Handler for Zookeeper completion and watch callbacks

This object must implement several methods responsible for determining how completion / watch callbacks are handled as well as the method for calling IAsyncResult callback functions.

These functions are used to abstract differences between a Python threading environment and asynchronous single-threaded environments like gevent. The minimum functionality needed for Kazoo to handle these differences is encompassed in this interface.

The Handler should document how callbacks are called for:

•Zookeeper completion events

•Zookeeper watch events

spawn (*func*, **args*, ***kwargs*)
Spawn a function to run asynchronously
Parameters
- **args** – args to call the function with.
- **kwargs** – keyword args to call the function with.

This method should return immediately and execute the function with the provided args and kwargs in an asynchronous manner.

dispatch_callback (*callback*)
Dispatch to the callback object
Parameters **callback** – A `Callback` object to be called.

name
Human readable name of the Handler interface

rlock_object ()
Return an appropriate object that implements Python's threading.RLock API

stop ()
Stop the handler. Should block until the handler is safely stopped.

sleep_func
Appropriate sleep function that can be called with a single argument and sleep.

event_object ()
Return an appropriate object that implements Python's threading.Event API

create_connection ()
A socket method that implements Python's socket.create_connection API

start ()
Start the handler, used for setting up the handler.

lock_object ()
Return an appropriate object that implements Python's threading.Lock API

timeout_exception
Exception class that should be thrown and captured if a result is not available within the given time

async_result ()
Return an instance that conforms to the `IAsyncResult` interface appropriate for this handler

select ()
A select method that implements Python's select.select API

socket ()
A socket method that implements Python's socket.socket API

Private API

The `IAsyncResult` documents the proper implementation for providing a value that results from a Zookeeper completion callback. Since the `KazooClient` returns an `IAsyncResult` object instead of taking a completion callback for async functions, developers wishing to have their own callback called should use the `IAsyncResult.rawlink()` method.

interface `kazoo.interfaces.`**IAsyncResult**
An Async Result object that can be queried for a value that has been set asynchronously

This object is modeled on the gevent AsyncResult object.

The implementation must account for the fact that the set() and set_exception() methods will be called from within the Zookeeper thread which may require extra care under asynchronous environments.

set(*value=None*)
Store the value. Wake up the waiters.
Parameters **value** – Value to store as the result.
Any waiters blocking on get() or wait() are woken up. Sequential calls to wait() and get() will not block at all.

get_nowait()
Return the value or raise the exception without blocking.

If nothing is available, raise the Timeout exception class on the associated IHandler interface.

set_exception(*exception*)
Store the exception. Wake up the waiters.
Parameters **exception** – Exception to raise when fetching the value.
Any waiters blocking on get() or wait() are woken up. Sequential calls to wait() and get() will not block at all.

successful()
Return *True* if and only if it is ready and holds a value

get(*block=True*, *timeout=None*)
Return the stored value or raise the exception
Parameters
- **block** (*bool*) – Whether this method should block or return immediately.
- **timeout** (*float*) – How long to wait for a value when *block* is *True*.

If this instance already holds a value / an exception, return / raise it immediately. Otherwise, block until set() or set_exception() has been called or until the optional timeout occurs.

exception
Holds the exception instance passed to set_exception() if set_exception() was called. Otherwise *None*

value
Holds the value passed to set() if set() was called. Otherwise *None*

rawlink(*callback*)
Register a callback to call when a value or an exception is set
Parameters **callback** (*func*) – A callback function to call after set() or set_exception() has been called. This function will be passed a single argument, this instance.

ready()
Return *True* if and only if it holds a value or an exception

unlink(*callback*)
Remove the callback set by rawlink()
Parameters **callback** (*func*) – A callback function to remove.

wait(*timeout=None*)
Block until the instance is ready.
Parameters **timeout** (*float*) – How long to wait for a value when *block* is *True*.

If this instance already holds a value / an exception, return / raise it immediately. Otherwise, block until `set()` or `set_exception()` has been called or until the optional timeout occurs.

1.6.7 kazoo.protocol.states

Kazoo State and Event objects

Public API

class kazoo.protocol.states.**EventType**
Zookeeper Event

Represents a Zookeeper event. Events trigger watch functions which will receive a `EventType` attribute as their event argument.

CREATED
A node has been created.

DELETED
A node has been deleted.

CHANGED
The data for a node has changed.

CHILD
The children under a node have changed (a child was added or removed). This event does not indicate the data for a child node has changed, which must have its own watch established.

class kazoo.protocol.states.**KazooState**
High level connection state values

States inspired by Netflix Curator.

SUSPENDED
The connection has been lost but may be recovered. We should operate in a "safe mode" until then. When the connection is resumed, it may be discovered that the session expired. A client should not assume that locks are valid during this time.

CONNECTED
The connection is alive and well.

LOST
The connection has been confirmed dead. Any ephemeral nodes will need to be recreated upon re-establishing a connection. If locks were acquired or recipes using ephemeral nodes are in use, they can be considered lost as well.

class kazoo.protocol.states.**KeeperState**
Zookeeper State

Represents the Zookeeper state. Watch functions will receive a `KeeperState` attribute as their state argument.

AUTH_FAILED
Authentication has failed, this is an unrecoverable error.

CONNECTED
Zookeeper is connected.

CONNECTED_RO
Zookeeper is connected in read-only state.

CONNECTING
Zookeeper is currently attempting to establish a connection.

EXPIRED_SESSION
The prior session was invalid, all prior ephemeral nodes are gone.

class kazoo.protocol.states.**WatchedEvent**
A change on ZooKeeper that a Watcher is able to respond to.

The WatchedEvent includes exactly what happened, the current state of ZooKeeper, and the path of the node that was involved in the event. An instance of WatchedEvent will be passed to registered watch functions.

type
A EventType attribute indicating the event type.

state
A KeeperState attribute indicating the Zookeeper state.

path
The path of the node for the watch event.

class kazoo.protocol.states.**ZnodeStat**
A ZnodeStat structure with convenience properties

When getting the value of a node from Zookeeper, the properties for the node known as a "Stat structure" will be retrieved. The ZnodeStat object provides access to the standard Stat properties and additional properties that are more readable and use Python time semantics (seconds since epoch instead of ms).

Note: The original Zookeeper Stat name is in parens next to the name when it differs from the convenience attribute. These are **not functions**, just attributes.

---

creation_transaction_id (*czxid*)
The transaction id of the change that caused this znode to be created.

last_modified_transaction_id (*mzxid*)
The transaction id of the change that last modified this znode.

created (*ctime*)
The time in seconds from epoch when this node was created. (ctime is in milliseconds)

last_modified (*mtime*)
The time in seconds from epoch when this znode was last modified. (mtime is in milliseconds)

version
The number of changes to the data of this znode.

acl_version (*aversion*)
The number of changes to the ACL of this znode.

owner_session_id (*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 *None*. (ephemeralOwner will be 0 if it is not ephemeral)

data_length (*dataLength*)
The length of the data field of this znode.

children_count (*numChildren*)
The number of children of this znode.

Private API

class kazoo.protocol.states.**Callback**
A callback that is handed to a handler for dispatch

Parameters

- **type** – Type of the callback, currently is only 'watch'
- **func** – Callback function
- **args** – Argument list for the callback function

1.6.8 kazoo.recipe.barrier

Zookeeper Barriers

Public API

class kazoo.recipe.barrier.**Barrier**(*client*, *path*)
Kazoo Barrier

Implements a barrier to block processing of a set of nodes until a condition is met at which point the nodes will be allowed to proceed. The barrier is in place if its node exists.

Warning: The wait() function does not handle connection loss and may raise ConnectionLossException if the connection is lost while waiting.

__init__(*client*, *path*)
Create a Kazoo Barrier
Parameters
- **client** – A KazooClient instance.
- **path** – The barrier path to use.

create()
Establish the barrier if it doesn't exist already

remove()
Remove the barrier
Returns Whether the barrier actually needed to be removed.
Return type bool

wait(*timeout=None*)
Wait on the barrier to be cleared
Returns True if the barrier has been cleared, otherwise False.
Return type bool

class kazoo.recipe.barrier.**DoubleBarrier**(*client*, *path*, *num_clients*, *identifier=None*)
Kazoo Double Barrier

Double barriers are used to synchronize the beginning and end of a distributed task. The barrier blocks when entering it until all the members have joined, and blocks when leaving until all the members have left.

Note: You should register a listener for session loss as the process will no longer be part of the barrier once the session is gone. Connection losses will be retried with the default retry policy.

__init__(*client*, *path*, *num_clients*, *identifier=None*)
Create a Double Barrier

Parameters
- **client** – A `KazooClient` instance.
- **path** – The barrier path to use.
- **num_clients** (*int*) – How many clients must enter the barrier to proceed.
- **identifier** – An identifier to use for this member of the barrier when participating. Defaults to the hostname + process id.

enter()
Enter the barrier, blocks until all nodes have entered

leave()
Leave the barrier, blocks until all nodes have left

1.6.9 kazoo.recipe.counter

Zookeeper Counter New in version 0.7: The Counter class.

Public API

class kazoo.recipe.counter.**Counter**(*client*, *path*, *default=0*)
Kazoo Counter

A shared counter of either int or float values. Changes to the counter are done atomically. The general retry policy is used to retry operations if concurrent changes are detected.

The data is marshaled using *repr(value)* and converted back using *type(counter.default)(value)* both using an ascii encoding. As such other data types might be used for the counter value.

Counter changes can raise `BadVersionError` if the retry policy wasn't able to apply a change.

Example usage:

```
zk = KazooClient()
counter = zk.Counter("/int")
counter += 2
counter -= 1
counter.value == 1

counter = zk.Counter("/float", default=1.0)
counter += 2.0
counter.value == 3.0
```

__init__(*client*, *path*, *default=0*)
Create a Kazoo Counter

Parameters
- **client** – A `KazooClient` instance.
- **path** – The counter path to use.
- **default** – The default value.

__add__(*value*)
Add value to counter.

__sub__(*value*)
Subtract value from counter.

1.6.10 kazoo.recipe.election

ZooKeeper Leader Elections

Public API

class kazoo.recipe.election.**Election**(*client*, *path*, *identifier=None*)
Kazoo Basic Leader Election

Example usage with a KazooClient instance:

```
zk = KazooClient()
election = zk.Election("/electionpath", "my-identifier")

# blocks until the election is won, then calls
# my_leader_function()
election.run(my_leader_function)
```

__init__(*client*, *path*, *identifier=None*)
Create a Kazoo Leader Election

Parameters
- **client** – A KazooClient instance.
- **path** – The election path to use.
- **identifier** – Name to use for this lock contender. This can be useful for querying to see who the current lock contenders are.

cancel()
Cancel participation in the election

Note: If this contender has already been elected leader, this method will not interrupt the leadership function.

---

contenders()
Return an ordered list of the current contenders in the election

---

Note: If the contenders did not set an identifier, it will appear as a blank string.

---

run(*func*, **args*, ***kwargs*)
Contend for the leadership

This call will block until either this contender is cancelled or this contender wins the election and the provided leadership function subsequently returns or fails.

Parameters
- **func** – A function to be called if/when the election is won.
- **args** – Arguments to leadership function.
- **kwargs** – Keyword arguments to leadership function.

1.6.11 kazoo.recipe.lock

Zookeeper Locking Implementations

Error Handling

It's highly recommended to add a state listener with `add_listener()` and watch for `LOST` and `SUSPENDED` state changes and re-act appropriately. In the event that a `LOST` state occurs, its certain that the lock and/or the lease has been lost.

Public API

class `kazoo.recipe.lock.`**Lock**(*client*, *path*, *identifier=None*)
Kazoo Lock

Example usage with a `KazooClient` instance:

```
zk = KazooClient()
lock = zk.Lock("/lockpath", "my-identifier")
with lock:  # blocks waiting for lock acquisition
    # do something with the lock
```

__init__(*client*, *path*, *identifier=None*)
Create a Kazoo lock.

Parameters
- **client** – A `KazooClient` instance.
- **path** – The lock path to use.
- **identifier** – Name to use for this lock contender. This can be useful for querying to see who the current lock contenders are.

acquire(*blocking=True*, *timeout=None*)
Acquire the lock. By defaults blocks and waits forever.

Parameters
- **blocking** (*bool*) – Block until lock is obtained or return immediately.
- **timeout** (*float or None*) – Don't wait forever to acquire the lock.

Returns Was the lock acquired?

Return type bool

Raises `LockTimeout` if the lock wasn't acquired within *timeout* seconds.

New in version 1.1: The timeout option.

cancel()
Cancel a pending lock acquire.

contenders()
Return an ordered list of the current contenders for the lock.

Note: If the contenders did not set an identifier, it will appear as a blank string.

---

release()
Release the lock immediately.

class `kazoo.recipe.lock.`**Semaphore**(*client*, *path*, *identifier=None*, *max_leases=1*)
A Zookeeper-based Semaphore

This synchronization primitive operates in the same manner as the Python threading version only uses the concept of leases to indicate how many available leases are available for the lock rather than counting.

Example:

```
zk = KazooClient()
semaphore = zk.Semaphore("/leasepath", "my-identifier")
with semaphore:  # blocks waiting for lock acquisition
    # do something with the semaphore
```

Warning: This class stores the allowed max_leases as the data on the top-level semaphore node. The stored value is checked once against the max_leases of each instance. This check is performed when acquire is called the first time. The semaphore node needs to be deleted to change the allowed leases.

New in version 0.6: The Semaphore class.New in version 1.1: The max_leases check.

__init__(*client*, *path*, *identifier=None*, *max_leases=1*)

Create a Kazoo Lock

Parameters

- **client** – A KazooClient instance.
- **path** – The semaphore path to use.
- **identifier** – Name to use for this lock contender. This can be useful for querying to see who the current lock contenders are.
- **max_leases** – The maximum amount of leases available for the semaphore.

acquire(*blocking=True*, *timeout=None*)

Acquire the semaphore. By defaults blocks and waits forever.

Parameters

- **blocking** (*bool*) – Block until semaphore is obtained or return immediately.
- **timeout** (*float or None*) – Don't wait forever to acquire the semaphore.

Returns Was the semaphore acquired?

Return type bool

Raises ValueError if the max_leases value doesn't match the stored value.

LockTimeout if the semaphore wasn't acquired within *timeout* seconds.

New in version 1.1: The blocking, timeout arguments and the max_leases check.

cancel()

Cancel a pending semaphore acquire.

lease_holders()

Return an unordered list of the current lease holders.

Note: If the lease holder did not set an identifier, it will appear as a blank string.

---

release()

Release the lease immediately.

1.6.12 kazoo.recipe.partitioner

Zookeeper Partitioner Implementation

SetPartitioner implements a partitioning scheme using Zookeeper for dividing up resources amongst members of a party.

This is useful when there is a set of resources that should only be accessed by a single process at a time that multiple processes across a cluster might want to divide up.

Example Use-Case

- Multiple workers across a cluster need to divide up a list of queues so that no two workers own the same queue.

Public API

class kazoo.recipe.partitioner.**SetPartitioner**(*client*, *path*, *set*, *partition_func=None*, *identifier=None*, *time_boundary=30*)

Partitions a set amongst members of a party

This class will partition a set amongst members of a party such that each member will be given zero or more items of the set and each set item will be given to a single member. When new members enter or leave the party, the set will be re-partitioned amongst the members.

When the `SetPartitioner` enters the `FAILURE` state, it is unrecoverable and a new `SetPartitioner` should be created.

Example:

```
from kazoo.client import KazooClient
client = KazooClient()

qp = client.SetPartitioner(
    path='/work_queues', set=('queue-1', 'queue-2', 'queue-3'))

while 1:
    if qp.failed:
        raise Exception("Lost or unable to acquire partition")
    elif qp.release:
        qp.release_set()
    elif qp.acquired:
        for partition in qp:
            # Do something with each partition
    elif qp.allocating:
        qp.wait_for_acquire()
```

State Transitions

When created, the `SetPartitioner` enters the `PartitionState.ALLOCATING` state.

`ALLOCATING` -> `ACQUIRED`

Set was partitioned successfully, the partition list assigned is accessible via list/iter methods or calling list() on the `SetPartitioner` instance.

`ALLOCATING` -> `FAILURE`

Allocating the set failed either due to a Zookeeper session expiration, or failure to acquire the items of the set within the timeout period.

`ACQUIRED` -> `RELEASE`

The members of the party have changed, and the set needs to be repartitioned. `SetPartitioner.release()` should be called as soon as possible.

`ACQUIRED` -> `FAILURE`

The current partition was lost due to a Zookeeper session expiration.

`RELEASE` -> `ALLOCATING`

The current partition was released and is being re-allocated.

__init__(*client*, *path*, *set*, *partition_func=None*, *identifier=None*, *time_boundary=30*)
Create a SetPartitioner instance

Parameters
- **client** – A KazooClient instance.
- **path** – The partition path to use.
- **set** – The set of items to partition.
- **partition_func** – A function to use to decide how to partition the set.
- **identifier** – An identifier to use for this member of the party when participating. Defaults to the hostname + process id.
- **time_boundary** – How long the party members must be stable before allocation can complete.

acquired
Corresponds to the PartitionState.ACQUIRED state

allocating
Corresponds to the PartitionState.ALLOCATING state

failed
Corresponds to the PartitionState.FAILURE state

finish()
Call to release the set and leave the party

release
Corresponds to the PartitionState.RELEASE state

release_set()
Call to release the set

This method begins the step of allocating once the set has been released.

wait_for_acquire(*timeout=30*)
Wait for the set to be partitioned and acquired

Parameters **timeout** (*int*) – How long to wait before returning.

class kazoo.recipe.partitioner.**PartitionState**
High level partition state values

ALLOCATING
The set needs to be partitioned, and may require an existing partition set to be released before acquiring a new partition of the set.

ACQUIRED
The set has been partitioned and acquired.

RELEASE
The set needs to be repartitioned, and the current partitions must be released before a new allocation can be made.

FAILURE
The set partition has failed. This occurs when the maximum time to partition the set is exceeded or the Zookeeper session is lost. The partitioner is unusable after this state and must be recreated.

1.6.13 kazoo.recipe.party

Party

A Zookeeper pool of party members. The `Party` object can be used for determining members of a party.

Public API

class `kazoo.recipe.party.`**`Party`**(*client*, *path*, *identifier=None*)
Simple pool of participating processes

`__init__`(*client*, *path*, *identifier=None*)

`__iter__`()
Get a list of participating clients' data values

`__len__`()
Return a count of participating clients

`join`()
Join the party

`leave`()
Leave the party

class `kazoo.recipe.party.`**`ShallowParty`**(*client*, *path*, *identifier=None*)
Simple shallow pool of participating processes

This differs from the `Party` as the identifier is used in the name of the party node itself, rather than the data. This places some restrictions on the length as it must be a valid Zookeeper node (an alphanumeric string), but reduces the overhead of getting a list of participants to a single Zookeeper call.

`__init__`(*client*, *path*, *identifier=None*)

`__iter__`()
Get a list of participating clients' identifiers

`__len__`()
Return a count of participating clients

`join`()
Join the party

`leave`()
Leave the party

1.6.14 `kazoo.recipe.queue`

Zookeeper based queue implementations. New in version 0.6: The Queue class.New in version 1.0: The LockingQueue class.

Public API

class `kazoo.recipe.queue.`**`Queue`**(*client*, *path*)
A distributed queue with optional priority support.

This queue does not offer reliable consumption. An entry is removed from the queue prior to being processed. So if an error occurs, the consumer has to re-queue the item or it will be lost.

`__init__`(*client*, *path*)
Parameters
- **client** – A `KazooClient` instance.

- **path** – The queue path to use in ZooKeeper.

__len__()
Return queue size.

get()
Get item data and remove an item from the queue.
Returns Item data or None.
Return type bytes

put(*value*, *priority=100*)
Put an item into the queue.
Parameters
- **value** – Byte string to put into the queue.
- **priority** – An optional priority as an integer with at most 3 digits. Lower values signify higher priority.

class kazoo.recipe.queue.**LockingQueue**(*client*, *path*)
A distributed queue with priority and locking support.

Upon retrieving an entry from the queue, the entry gets locked with an ephemeral node (instead of deleted). If an error occurs, this lock gets released so that others could retake the entry. This adds a little penalty as compared to Queue implementation.

The user should call the LockingQueue.get() method first to lock and retrieve the next entry. When finished processing the entry, a user should call the LockingQueue.consume() method that will remove the entry from the queue.

This queue will not track connection status with ZooKeeper. If a node locks an element, then loses connection with ZooKeeper and later reconnects, the lock will probably be removed by Zookeeper in the meantime, but a node would still think that it holds a lock. The user should check the connection status with Zookeeper or call LockingQueue.holds_lock() method that will check if a node still holds the lock.

Note: LockingQueue requires ZooKeeper 3.4 or above, since it is using transactions.

__init__(*client*, *path*)
Parameters
- **client** – A KazooClient instance.
- **path** – The queue path to use in ZooKeeper.

__len__()
Returns the current length of the queue.
Returns queue size (includes locked entries count).

consume()
Removes a currently processing entry from the queue.
Returns True if element was removed successfully, False otherwise.
Return type bool

get(*timeout=None*)
Locks and gets an entry from the queue. If a previously got entry was not consumed, this method will return that entry.
Parameters **timeout** – Maximum waiting time in seconds. If None then it will wait untill an entry appears in the queue.
Returns A locked entry value or None if the timeout was reached.
Return type bytes

holds_lock()
Checks if a node still holds the lock.
Returns True if a node still holds the lock, False otherwise.
Return type bool

put(*value*, *priority=100*)
Put an entry into the queue.
Parameters
- **value** – Byte string to put into the queue.
- **priority** – An optional priority as an integer with at most 3 digits. Lower values signify higher priority.

put_all(*values*, *priority=100*)
Put several entries into the queue. The action only succeeds if all entries where put into the queue.
Parameters
- **values** – A list of values to put into the queue.
- **priority** – An optional priority as an integer with at most 3 digits. Lower values signify higher priority.

1.6.15 kazoo.recipe.watchers

Higher level child and data watching API's.

Public API

class kazoo.recipe.watchers.**DataWatch**(*client*, *path*, *func=None*, **args*, ***kwargs*)
Watches a node for data updates and calls the specified function each time it changes

The function will also be called the very first time its registered to get the data.

Returning *False* from the registered function will disable future data change calls. If the client connection is closed (using the close command), the DataWatch will no longer get updates.

If the function supplied takes three arguments, then the third one will be a `WatchedEvent`. It will only be set if the change to the data occurs as a result of the server notifying the watch that there has been a change. Events like reconnection or the first call will not include an event.

If the node does not exist, then the function will be called with `None` for all values.

Example with client:

```
@client.DataWatch('/path/to/watch')
def my_func(data, stat):
    print("Data is %s" % data)
    print("Version is %s" % stat.version)

# Above function is called immediately and prints

# Or if you want the event object
@client.DataWatch('/path/to/watch')
def my_func(data, stat, event):
    print("Data is %s" % data)
    print("Version is %s" % stat.version)
    print("Event is %s" % event)
```

Changed in version 1.2.

__init__(*client*, *path*, *func=None*, **args*, ***kwargs*)
Create a data watcher for a path

Parameters
- **client** (KazooClient) – A zookeeper client.
- **path** (*str*) – The path to watch for data changes on.
- **func** (*callable*) – Function to call initially and every time the node changes. *func* will be called with a tuple, the value of the node and a ZnodeStat instance.

__call__(*func*)
Callable version for use as a decorator

Parameters func (*callable*) – Function to call initially and every time the data changes. *func* will be called with a tuple, the value of the node and a ZnodeStat instance.

class kazoo.recipe.watchers.**ChildrenWatch**(*client*, *path*, *func=None*, *allow_session_lost=True*, *send_event=False*)
Watches a node for children updates and calls the specified function each time it changes

The function will also be called the very first time its registered to get children.

Returning *False* from the registered function will disable future children change calls. If the client connection is closed (using the close command), the ChildrenWatch will no longer get updates.

if send_event=True in __init__, then the function will always be called with second parameter, `event`. Upon initial call or when recovering a lost session the `event` is always `None`. Otherwise it's a `WatchedEvent` instance.

Example with client:

```
@client.ChildrenWatch('/path/to/watch')
def my_func(children):
    print "Children are %s" % children

# Above function is called immediately and prints children
```

__init__(*client*, *path*, *func=None*, *allow_session_lost=True*, *send_event=False*)
Create a children watcher for a path

Parameters
- **client** (KazooClient) – A zookeeper client.
- **path** (*str*) – The path to watch for children on.
- **func** (*callable*) – Function to call initially and every time the children change. *func* will be called with a single argument, the list of children.
- **allow_session_lost** (*bool*) – Whether the watch should be re-registered if the zookeeper session is lost.
- **send_event** (*bool*) – Whether the function should be passed the event sent by ZooKeeper or None upon initialization (see class documentation)

The path must already exist for the children watcher to run.

__call__(*func*)
Callable version for use as a decorator

Parameters func (*callable*) – Function to call initially and every time the children change. *func* will be called with a single argument, the list of children.

class kazoo.recipe.watchers.**PatientChildrenWatch**(*client*, *path*, *time_boundary=30*)
Patient Children Watch that returns values after the children of a node don't change for a period of time

A separate watcher for the children of a node, that ignores changes within a boundary time and sets the result only when the boundary time has elapsed with no children changes.

Example:

```
watcher = PatientChildrenWatch(client, '/some/path',
                               time_boundary=5)
async_object = watcher.start()

# Blocks until the children have not changed for time boundary
# (5 in this case) seconds, returns children list and an
# async_result that will be set if the children change in the
# future
children, child_async = async_object.get()
```

Note: This Watch is different from `DataWatch` and `ChildrenWatch` as it only returns once, does not take a function that is called, and provides an `IAsyncResult` object that can be checked to see if the children have changed later.

__init__(*client*, *path*, *time_boundary=30*)

start()

Begin the watching process asynchronously

Returns An `IAsyncResult` instance that will be set when no change has occurred to the children for time boundary seconds.

1.6.16 kazoo.retry

Public API

class kazoo.retry.**KazooRetry**(*max_tries=1*, *delay=0.1*, *backoff=2*, *max_jitter=0.8*, *max_delay=3600*, *ignore_expire=True*, *sleep_func=<built-in function sleep>*, *deadline=None*, *interrupt=None*)

Helper for retrying a method in the face of retry-able exceptions

__init__(*max_tries=1*, *delay=0.1*, *backoff=2*, *max_jitter=0.8*, *max_delay=3600*, *ignore_expire=True*, *sleep_func=<built-in function sleep>*, *deadline=None*, *interrupt=None*)

Create a `KazooRetry` instance for retrying function calls

Parameters

- **max_tries** – How many times to retry the command.
- **delay** – Initial delay between retry attempts.
- **backoff** – Backoff multiplier between retry attempts. Defaults to 2 for exponential backoff.
- **max_jitter** – Additional max jitter period to wait between retry attempts to avoid slamming the server.
- **max_delay** – Maximum delay in seconds, regardless of other backoff settings. Defaults to one hour.
- **ignore_expire** – Whether a session expiration should be ignored and treated as a retry-able command.
- **interrupt** – Function that will be called with no args that may return True if the retry should be ceased immediately. This will be called no more than every 0.1 seconds during a wait between retries.

__call__(*func*, **args*, ***kwargs*)
Call a function with arguments until it completes without throwing a Kazoo exception

Parameters
- **func** – Function to call
- **args** – Positional arguments to call the function with

Params kwargs Keyword arguments to call the function with

The function will be called until it doesn't throw one of the retryable exceptions (ConnectionLoss, OperationTimeout, or ForceRetryError), and optionally retrying on session expiration.

reset()
Reset the attempt counter

copy()
Return a clone of this retry manager

exception kazoo.retry.**ForceRetryError**
Raised when some recipe logic wants to force a retry.

exception kazoo.retry.**RetryFailedError**
Raised when retrying an operation ultimately failed, after retrying the maximum number of attempts.

exception kazoo.retry.**InterruptedError**
Raised when the retry is forcibly interrupted by the interrupt function

1.6.17 kazoo.security

Kazoo Security

Public API

class kazoo.security.**ACL**
An ACL for a Zookeeper Node

An ACL object is created by using an Id object along with a Permissions setting. For convenience, make_digest_acl() should be used to create an ACL object with the desired scheme, id, and permissions.

class kazoo.security.**Id**
Id(scheme, id)

kazoo.security.**make_digest_acl**(*username*, *password*, *read=False*, *write=False*, *create=False*, *delete=False*, *admin=False*, *all=False*)
Create a digest ACL for Zookeeper with the given permissions

This method combines make_digest_acl_credential() and make_acl() to create an ACL object appropriate for use with Kazoo's ACL methods.

Parameters
- **username** – Username to use for the ACL.
- **password** – A plain-text password to hash.
- **write** (*bool*) – Write permission.
- **create** (*bool*) – Create permission.
- **delete** (*bool*) – Delete permission.
- **admin** (*bool*) – Admin permission.

- **all** (*bool*) – All permissions.

Return type ACL

Private API

kazoo.security.**make_acl**(*scheme*, *credential*, *read=False*, *write=False*, *create=False*, *delete=False*, *admin=False*, *all=False*)

Given a scheme and credential, return an ACL object appropriate for use with Kazoo.

Parameters

- **scheme** – The scheme to use. I.e. *digest*.
- **credential** – A colon separated username, password. The password should be hashed with the *scheme* specified. The make_digest_acl_credential() method will create and return a credential appropriate for use with the *digest* scheme.
- **write** (*bool*) – Write permission.
- **create** (*bool*) – Create permission.
- **delete** (*bool*) – Delete permission.
- **admin** (*bool*) – Admin permission.
- **all** (*bool*) – All permissions.

Return type ACL

kazoo.security.**make_digest_acl_credential**(*username*, *password*)

Create a SHA1 digest credential

1.6.18 kazoo.testing.harness

Kazoo testing harnesses

Public API

class kazoo.testing.harness.**KazooTestHarness**(**args*, ***kw*)

Harness for testing code that uses Kazoo

This object can be used directly or as a mixin. It supports starting and stopping a complete ZooKeeper cluster locally and provides an API for simulating errors and expiring sessions.

Example:

```
class MyTestCase(KazooTestHarness):
    def setUp(self):
        self.setup_zookeeper()

        # additional test setup

    def tearDown(self):
        self.teardown_zookeeper()

    def test_something(self):
        something_that_needs_a_kazoo_client(self.client)
```

```
        def test_something_else(self):
            something_that_needs_zk_servers(self.servers)
```

class kazoo.testing.harness.**KazooTestCase**(**args*, ***kw*)

1.7 Changelog

1.7.1 1.3.1 (2013-09-25)

Bug Handling

- #118, #125, #128: Fix unknown variable in KazooClient *command_retry* argument handling.
- #126: Fix *KazooRetry.copy* to correctly copy sleep function.
- #118: Correct session/socket timeout conversion (int vs. float).

Documentation

- #121: Add a note about *kazoo.recipe.queue.LockingQueue* requiring a Zookeeper 3.4+ server.

1.7.2 1.3 (2013-09-05)

Features

- #115: Limit the backends we use for SLF4J during tests.
- #112: Add IPv6 support. Patch by Dan Kruchinin.

1.7.3 1.2.1 (2013-08-01)

Bug Handling

- Issue #108: Circular import fail when importing kazoo.recipe.watchers directly has now been resolved. Watchers and partitioner properly import the KazooState from kazoo.protocol.states rather than kazoo.client.
- Issue #109: Partials not usable properly as a datawatch call can now be used. All funcs will be called with 3 args and fall back to 2 args if there's an argument error.
- Issue #106, #107: *client.create_async* didn't strip change root from the returned path.

1.7.4 1.2 (2013-07-24)

Features

- KazooClient can now be stopped more reliably even if its in the middle of a long retry sleep. This utilizes the new interrupt feature of KazooRetry which lets the sleep be broken down into chunks and an interrupt function called to determine if the retry should fail early.

- Issue #62, #92, #89, #101, #102: Allow KazooRetry to have a max deadline, transition properly when connection fails to LOST, and setup separate connection retry behavior from client command retry behavior. Patches by Mike Lundy.
- Issue #100: Make it easier to see exception context in threading and connection modules.
- Issue #85: Increase information density of logs and don't prevent dynamic reconfiguration of log levels at runtime.
- Data-watchers for the same node are no longer 'stacked'. That is, if a get and an exists call occur for the same node with the same watch function, then it will be registered only once. This change results in Kazoo behaving per Zookeeper client spec regarding repeat watch use.

Bug Handling

- Issue #53: Throw a warning upon starting if the chroot path doesn't exist so that it's more obvious when the chroot should be created before performing more operations.
- Kazoo previously would let the same function be registered as a data-watch or child-watch multiple times, and then call it multiple times upon being triggered. This was non-compliant Zookeeper client behavior, the same watch can now only be registered once for the same znode path per Zookeeper client documentation.
- Issue #105: Avoid rare import lock problems by moving module imports in client.py to the module scope.
- Issue #103: Allow prefix-less sequential znodes.
- Issue #98: Extend testing ZK harness to work with different file locations on some versions of Debian/Ubuntu.
- Issue #97: Update some docstrings to reflect current state of handlers.
- Issue #62, #92, #89, #101, #102: Allow KazooRetry to have a max deadline, transition properly when connection fails to LOST, and setup separate connection retry behavior from client command retry behavior. Patches by Mike Lundy.

API Changes

- The *kazoo.testing.harness.KazooTestHarness* class directly inherits from *unittest.TestCase* and you need to ensure to call its *__init__* method.
- DataWatch no longer takes any parameters besides for the optional function during instantiation. The additional options are now implicitly True, with the user being left to ignore events as they choose. See the DataWatch API docs for more information.
- Issue #99: Better exception raised when the writer fails to close. A WriterNotClosedException that inherits from KazooException is now raised when the writer fails to close in time.

1.7.5 1.1 (2013-06-08)

Features

- Issue #93: Add timeout option to lock/semaphore acquire methods.
- Issue #79 / #90: Add ability to pass the WatchedEvent to DataWatch and ChildWatch functions.
- Respect large client timeout values when closing the connection.
- Add a *max_leases* consistency check to the semaphore recipe.

- Issue #76: Extend testing helpers to allow customization of the Java classpath by specifying the new *ZOOKEEPER_CLASSPATH* environment variable.
- Issue #65: Allow non-blocking semaphore acquisition.

Bug Handling

- Issue #96: Provide Windows compatibility in testing harness.
- Issue #95: Handle errors deserializing connection response.
- Issue #94: Clean up stray bytes in connection pipe.
- Issue #87 / #88: Allow re-acquiring lock after cancel.
- Issue #77: Use timeout in initial socket connection.
- Issue #69: Only ensure path once in lock and semaphore recipes.
- Issue #68: Closing the connection causes exceptions to be raised by watchers which assume the connection won't be closed when running commands.
- Issue #66: Require ping reply before sending another ping, otherwise the connection will be considered dead and a ConnectionDropped will be raised to trigger a reconnect.
- Issue #63: Watchers weren't reset on lost connection.
- Issue #58: DataWatcher failed to re-register for changes after non-existent node was created then deleted.

API Changes

- KazooClient.create_async now supports the makepath argument.
- KazooClient.ensure_path now has an async version, ensure_path_async.

1.7.6 1.0 (2013-03-26)

Features

- Added a LockingQueue recipe. The queue first locks an item and removes it from the queue only after the consume() method is called. This enables other nodes to retake the item if an error occurs on the first node.

Bug Handling

- Issue #50: Avoid problems with sleep function in mixed gevent/threading setup.
- Issue #56: Avoid issues with watch callbacks evaluating to false.

1.7.7 1.0b1 (2013-02-24)

Features

- Refactored the internal connection handler to use a single thread. It now uses a deque and pipe to signal the ZK thread that there's a new command to send, so that the ZK thread can send it, or retrieve a response. Processing ZK requests and responses serially in a single thread eliminates the need for a bunch of the locking, the peekable queue and two threads working on the same underlying socket.

- Issue #48: Added documentation for the *retry* helper module.
- Issue #55: Fix *os.pipe* file descriptor leak and introduce a *KazooClient.close* method. The method is particular useful in tests, where multiple KazooClients are created and closed in the same process.

Bug Handling

- Issue #46: Avoid TypeError in GeneratorContextManager on process shutdown.
- Issue #43: Let DataWatch return node data if allow_missing_node is used.

1.7.8 0.9 (2013-01-07)

API Changes

- When a retry operation ultimately fails, it now raises a *kazoo.retry.RetryFailedError* exception, instead of a general *Exception* instance. *RetryFailedError* also inherits from the base *KazooException*.

Features

- Improvements to Debian packaging rules.

Bug Handling

- Issue #39 / #41: Handle connection dropped errors during session writes. Ensure client connection is re-established to a new ZK node if available.
- Issue #38: Set *CLOEXEC* flag on all sockets when available.
- Issue #37 / #40: Handle timeout errors during *select* calls on sockets.
- Issue #36: Correctly set *ConnectionHandler.writer_stopped* even if an exception is raised inside the writer, like a retry operation failing.

1.7.9 0.8 (2012-10-26)

API Changes

- The *KazooClient.__init__* took as *watcher* argument as its second keyword argument. The argument had no effect anymore since version 0.5 and was removed.

Bug Handling

- Issue #35: *KazooClient.__init__* didn't pass on *retry_max_delay* to the retry helper.
- Issue #34: Be more careful while handling socket connection errors.

1.7.10 0.7 (2012-10-15)

Features

- DataWatch now has a *allow_missing_node* setting that allows a watch to be set on a node that doesn't exist when the DataWatch is created.
- Add new Queue recipe, with optional priority support.
- Add new Counter recipe.
- Added debian packaging rules.

Bug Handling

- Issue #31 fixed: Only catch KazooExceptions in catch-all calls.
- Issue #15 fixed again: Force sleep delay to be a float to appease gevent.
- Issue #29 fixed: DataWatch and ChildrenWatch properly re-register their watches on server disconnect.

1.7.11 0.6 (2012-09-27)

API Changes

- Node paths are assumed to be Unicode objects. Under Python 2 pure-ascii strings will also be accepted. Node values are considered bytes. The byte type is an alias for *str* under Python 2.
- New KeeperState.CONNECTED_RO state for Zookeeper servers connected in read-only mode.
- New NotReadOnlyCallError exception when issuing a write change against a server thats currently read-only.

Features

- Add support for Python 3.2, 3.3 and PyPy (only for the threading handler).
- Handles connecting to Zookeeper 3.4+ read-only servers.
- Automatic background scanning for a Read/Write server when connected to a server in read-only mode.
- Add new Semaphore recipe.
- Add a new *retry_max_delay* argument to the client and by default limit the retry delay to at most an hour regardless of exponential backoff settings.
- Add new *randomize_hosts* argument to *KazooClient*, allowing one to disable host randomization.

Bug Handling

- Fix bug with locks not handling intermediary lock contenders disappearing.
- Fix bug with set_data type check failing to catch unicode values.
- Fix bug with gevent 0.13.x backport of peekable queue.
- Fix PatientChildrenWatch to use handler specific sleep function.

1.7.12 0.5 (2012-09-06)

Skipping a version to reflect the magnitude of the change. Kazoo is now a pure Python client with no C bindings. This release should run without a problem on alternate Python implementations such as PyPy and Jython. Porting to Python 3 in the future should also be much easier.

Documentation

- Docs have been restructured to handle the new classes and locations of the methods from the pure Python refactor.

Bug Handling

This change may introduce new bugs, however there is no longer the possibility of a complete Python segfault due to errors in the C library and/or the C binding.

- Possible segfaults from the C lib are gone.
- Password mangling due to the C lib is gone.
- The party recipes didn't set their participating flag to False after leaving.

Features

- New *client.command* and *client.server_version* API, exposing Zookeeper's four letter commands and giving access to structured version information.
- Added 'include_data' option for get_children to include the node's Stat object.
- Substantial increase in logging data with debug mode. All correspondence with the Zookeeper server can now be seen to help in debugging.

API Changes

- The testing helpers have been moved from *testing.__init__* into a *testing.harness* module. The official API's of *KazooTestCase* and *KazooTestHarness* can still be directly imported from *testing*.
- The kazoo.handlers.util module was removed.
- Backwards compatible exception class aliases are provided for now in kazoo exceptions for the prior C exception names.
- Unicode strings now work fine for node names and are properly converted to and from unicode objects.
- The data value argument for the create and create_async methods of the client was made optional and defaults to an empty byte string. The data value must be a byte string. Unicode values are no longer allowed and will raise a TypeError.

1.7.13 0.3 (2012-08-23)

API Changes

- Handler interface now has an rlock_object for use by recipes.

Bug Handling

- Fixed password bug with updated zc-zookeeper-static release, which retains null bytes in the password properly.
- Fixed reconnect hammering, so that the reconnection follows retry jitter and retry backoff's.
- Fixed possible bug with using a threading.Condition in the set partitioner. Set partitioner uses new rlock_object handler API to get an appropriate RLock for gevent.
- Issue #17 fixed: Wrap timeout exceptions with staticmethod so they can be used directly as intended. Patch by Bob Van Zant.
- Fixed bug with client reconnection looping indefinitely using an expired session id.

1.7.14 0.2 (2012-08-12)

Documentation

- Fixed doc references to start_async using an AsyncResult object, it uses an Event object.

Bug Handling

- Issue #16 fixed: gevent zookeeper logging failed to handle a monkey patched logging setup. Logging is now setup such that a greenlet is used for logging messages under gevent, and the thread one is used otherwise.
- Fixed bug similar to #14 for ChildrenWatch on the session listener.
- Issue #14 fixed: DataWatch had inconsistent handling of the node it was watching not existing. DataWatch also properly spawns its _get_data function to avoid blocking session events.
- Issue #15 fixed: sleep_func for SequentialGeventHandler was not set on the class appropriately leading to additional arguments being passed to gevent.sleep.
- Issue #9 fixed: Threads/greenlets didn't gracefully shut down. Handler now has a start/stop that is used by the client when calling start and stop that shuts down the handler workers. This addresses errors and warnings that could be emitted upon process shutdown regarding a clean exit of the workers.
- Issue #12 fixed: gevent 0.13 doesn't use the same start_new_thread as gevent 1.0 which resulted in a fully monkey-patched environment halting due to the wrong thread. Updated to use the older kazoo method of getting the real thread module object.

API Changes

- The KazooClient handler is now officially exposed as KazooClient.handler so that the appropriate sync objects can be used by end-users.
- Refactored ChildrenWatcher used by SetPartitioner into a publicly exposed PatientChildrenWatch under recipe.watchers.

Deprecations

- connect/connect_async has been renamed to start/start_async to better match the stop to indicate connection handling. The prior names are aliased for the time being.

Recipes

- Added Barrier and DoubleBarrier implementation.

1.7.15 0.2b1 (2012-07-27)

Bug Handling

- ZOOKEEPER-1318: SystemError is caught and rethrown as the proper invalid state exception in older zookeeper python bindings where this issue is still valid.
- ZOOKEEPER-1431: Install the latest zc-zookeeper-static library or use the packaged ubuntu one for ubuntu 12.04 or later.
- ZOOKEEPER-553: State handling isn't checked via this method, we track it in a simpler manner with the watcher to ensure we know the right state.

Features

- Exponential backoff with jitter for retrying commands.
- Gevent 0.13 and 1.0b support.
- Lock, Party, SetPartitioner, and Election recipe implementations.
- Data and Children watching API's.
- State transition handling with listener registering to handle session state changes (choose to fatal the app on session expiration, etc.)
- Zookeeper logging stream redirected into Python logging channel under the name 'Zookeeper'.
- Base client library with handler support for threading and gevent async environments.

CHAPTER 2

Why

Using *Zookeeper* in a safe manner can be difficult due to the variety of edge-cases in *Zookeeper* and other bugs that have been present in the Python C binding. Due to how the C library utilizes a separate C thread for *Zookeeper* communication some libraries like gevent also don't work properly by default.

By utilizing a pure Python implementation, Kazoo handles all of these cases and provides a new asynchronous API which is consistent when using threads or gevent greenlets.

CHAPTER 3

Source Code

All source code is available on github under kazoo.

CHAPTER 4

Bugs/Support

Bugs and support issues should be reported on the kazoo github issue tracker.

The developers of `kazoo` can frequently be found on the Freenode IRC network in the #zookeeper channel.

For general discussions, please use the python-zk mailing list hosted on Google Groups.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *Glossary*

5.1 Glossary

Zookeeper Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

CHAPTER 6

License

kazoo is offered under the Apache License 2.0.

CHAPTER 7

Authors

kazoo started under the Nimbus Project and through collaboration with the open-source community has been merged with code from Mozilla and the Zope Corporation. It has since gathered an active community of over two dozen contributors.

Python Module Index

k