kazoo Documentation

Release 2.6.1

Various Authors

November 15, 2018

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 >= 1.2
- Support for eventlet
- Support for Zookeeper 3.3, 3.4, and 3.5 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

Installing

kazoo can be installed via `pip`:

```
$ pip install kazoo
```

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

Basic Usage

Connection Handling

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

```
from kazoo_sasl.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()
```

Logging Setup

If logging is not setup for your application, you can get following message:

```
No handlers could be found for logger "kazoo_sasl.client"
```

To avoid this issue you can at the very minimum do the following:

```
import logging
logging.basicConfig()
```

Read Python's logging tutorial for more details.

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_sasl.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_sasl.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_sasl.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_sasl.client import KazooState
from kazoo_sasl.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.

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)
```

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_sasl.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_sasl.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_sasl.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.

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")))
```

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.

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.

Connection Handling

Creating a connection:

```
from kazoo_sasl.client import KazooClient
from kazoo_sasl.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 (and `SequentialEventletHandler` when eventlet is used). Kazoo doesn't rely on gevents/eventlet monkey patching and requires that you pass in the appropriate handler, the default handler is `SequentialThreadingHandler`.

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_sasl.exceptions import ConnectionLossException
from kazoo_sasl.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)
```

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.

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.

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.

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.

You can specify an optional *ZOOKEEPER_PORT_OFFSET* environment variable to influence the ports the cluster is using. By default the offset is 20000 and a cluster with three members will use ports 20000, 20010 and 20020.

Kazoo Test Harness

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

Example:

```
from kazoo_sasl.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')
        ...
```

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_sasl.testing import KazooTestCase

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

Zake

For those that do not need (or desire) to setup a Zookeeper cluster to test integration with kazoo there is also a library called zake. Contributions to Zake's github repository are welcome.

Zake can be used to provide a *mock client* to layers of your application that interact with kazoo (using the same client interface) during testing to allow for introspection of what was stored, which watchers are active (and more) after your test of your application code has finished.

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.

`kazoo_sasl.client`

Public API

`kazoo_sasl.exceptions`

Kazoo Exceptions

Public API

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

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

exception `kazoo_sasl.exceptions.`**`AuthFailedError`**

exception `kazoo_sasl.exceptions.`**`BadVersionError`**

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

exception `kazoo_sasl.exceptions.`**`InvalidACLError`**

exception `kazoo_sasl.exceptions.`**`LockTimeout`**
Raised if failed to acquire a lock.

New in version 1.1.

exception `kazoo_sasl.exceptions.`**`NoChildrenForEphemeralsError`**

exception `kazoo_sasl.exceptions.`**`NodeExistsError`**

exception `kazoo_sasl.exceptions.`**`NoNodeError`**

exception `kazoo_sasl.exceptions.`**`NotEmptyError`**

Private API

exception `kazoo_sasl.exceptions.`**`APIError`**

exception `kazoo_sasl.exceptions.`**`BadArgumentsError`**

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

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

exception `kazoo_sasl.exceptions.`**`ConnectionClosedError`**
Connection is closed

exception `kazoo_sasl.exceptions.`**`ConnectionLoss`**

exception `kazoo_sasl.exceptions.`**`DataInconsistency`**

exception `kazoo_sasl.exceptions.`**`MarshallingError`**

exception kazoo_sasl.exceptions.**NoAuthError**

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

exception kazoo_sasl.exceptions.**InvalidCallbackError**

exception kazoo_sasl.exceptions.**OperationTimeoutError**

exception kazoo_sasl.exceptions.**RolledBackError**

exception kazoo_sasl.exceptions.**RuntimeInconsistency**

exception kazoo_sasl.exceptions.**SessionExpiredError**

exception kazoo_sasl.exceptions.**SessionMovedError**

exception kazoo_sasl.exceptions.**SystemZookeeperError**

exception kazoo_sasl.exceptions.**UnimplementedError**

exception kazoo_sasl.exceptions.**WriterNotClosedException**
Raised if the writer is unable to stop closing when requested.

New in version 1.2.

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

kazoo_sasl.handlers.gevent

A gevent based handler.

Public API

class kazoo_sasl.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

queue_empty
: alias of `Empty`

queue_impl
: alias of `Queue`

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.

Tip: In the current implementation, a value of 0 (the default) means to yield execution to any other runnable greenlets, but this greenlet may be scheduled again before the event loop cycles (in an extreme case, a greenlet that repeatedly sleeps with 0 can prevent greenlets that are ready to do I/O from being scheduled for some (small) period of time); a value greater than 0, on the other hand, will delay running this greenlet until the next iteration of the loop.

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

Changed in version 1.3a1: Sleeping with a value of 0 will now be bounded to approximately block the loop for no longer than `gevent.getswitchinterval()`.

See also:

`idle()`

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_sasl.handlers.gevent.`**AsyncResult**
: 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()` is called. Waiters may receive the passed value or exception by calling `get()` instead of `wait()`. An `AsyncResult` instance cannot be reset.

To pass a value call `set()`. Calls to `get()` (those that are 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)
>>> try:
...     result.get()
... except ZeroDivisionError:
...     print('ZeroDivisionError')
ZeroDivisionError
```

Note: The order and timing in which waiting greenlets are awakened is not determined. As an implementation note, in gevent 1.1 and 1.0, waiting greenlets are awakened in a undetermined order sometime *after* the current greenlet yields to the event loop. Other greenlets (those not waiting to be awakened) may run between the current greenlet yielding and the waiting greenlets being awakened. These details may change in the future.

Changed in version 1.1: The exact order in which waiting greenlets are awakened is not the same as in 1.0.

Changed in version 1.1: Callbacks `linked` to this object are required to be hashable, and duplicates are merged.

cancel(*self*) → bool

cancelled(*self*) → bool

done(*self*) → bool

exc_info

The three-tuple of exception information if `set_exception()` was called.

exception

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

get(*self*, *block=True*, *timeout=None*)

Return the stored value or raise the exception.

If this instance already holds a value or an exception, return or raise it immediately. 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). If the *timeout* elapses, the *Timeout* exception will be raised.

Parameters **block** (*bool*) – If set to `False` and this instance is not ready, immediately raise a `Timeout` exception.

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

If this object is not yet `ready`, raise `gevent.Timeout` immediately.

ready(*self*) → bool
Return true if and only if it holds a value or an exception

result(*self*, *timeout=None*)

set(*self*, *value=None*)
Store the value and wake up any waiters.

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

set_exception(*self*, *exception*, *exc_info=None*)
Store the exception and wake up any waiters.

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

Parameters exc_info (*tuple*) – If given, a standard three-tuple of type, value, `traceback` as returned by `sys.exc_info()`. This will be used when the exception is re-raised to propagate the correct traceback.

set_result()
AsyncResult.set(self, value=None) Store the value and wake up any waiters.

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

successful(*self*) → bool
Return true if and only if it is ready and holds a value

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

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

If this instance already holds a value, it is returned immediately. If this instance already holds an exception, `None` is returned immediately.

Otherwise, block until another greenlet calls `set()` or `set_exception()` (at which point either the value or `None` will be returned, respectively), or until the optional timeout expires (at which point `None` will also be returned).

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).

Note: If a timeout is given and expires, `None` will be returned (no timeout exception will be raised).

kazoo_sasl.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_sasl.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

`queue_empty`

alias of `Empty`

`queue_impl`

alias of `Queue`

`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_sasl.handlers.threading.**AsyncResult**(*handler*)

A one-time event that stores a value or an exception

kazoo_sasl.handlers.utils

Kazoo handler helpers

Public API

kazoo_sasl.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_sasl.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_sasl.handlers.utils.**create_socket_pair**(*module*, *port=0*)

Create socket pair.

If socket.socketpair isn't available, we emulate it.

kazoo_sasl.handlers.utils.**create_tcp_socket**(*module*)

Create a TCP socket with the CLOEXEC flag set.

kazoo_sasl.interfaces

Kazoo Interfaces

Changed in version 1.4: The classes in this module used to be interface declarations based on *zope.interface.Interface*. They were converted to normal classes and now serve as documentation only.

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.

class `kazoo_sasl.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

`name`

Human readable name of the Handler interface.

`timeout_exception`

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

`sleep_func`

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

`async_result``()`

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

`create_connection``()`

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

`dispatch_callback``(`*callback*`)`

Dispatch to the callback object

Parameters **callback** – A `Callback` object to be called.

`event_object``()`

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

`lock_object``()`

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

`rlock_object``()`

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

`select``()`

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

`socket``()`

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

`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.

`start``()`

Start the handler, used for setting up the handler.

stop()

Stop the handler. Should block until the handler is safely stopped.

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.

class `kazoo_sasl.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.

value

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

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

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.

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.

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

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.

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

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.

kazoo_sasl.protocol.states

Kazoo State and Event objects

Public API

class kazoo_sasl.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.

NONE

The connection state has been altered.

class kazoo_sasl.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_sasl.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_sasl.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_sasl.protocol.states.`**`ZnodeStat`**
A ZnodeStat structure with convenience properties

When getting the value of a znode from Zookeeper, the properties for the znode 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 znode 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_sasl.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

`kazoo_sasl.recipe.barrier`

Zookeeper Barriers

Maintainer None

Status Unknown

Public API

class `kazoo_sasl.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_sasl.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

kazoo_sasl.recipe.cache

TreeCache

Maintainer Jiangge Zhang <tonyseek@gmail.com>

Maintainer Haochuan Guo <guohaochuan@gmail.com>

Maintainer Tianwen Zhang <mail2tevin@gmail.com>

Status Alpha

A port of the Apache Curator's TreeCache recipe. It builds an in-memory cache of a subtree in ZooKeeper and keeps it up-to-date.

See also: http://curator.apache.org/curator-recipes/tree-cache.html

Public API

class `kazoo_sasl.recipe.cache.`**`TreeCache`**`(`*client*, *path*`)`

The cache of a ZooKeeper subtree.

Parameters

- **client** – A `KazooClient` instance.
- **path** – The root path of subtree.

`start``()`

Starts the cache.

The cache is not started automatically. You must call this method.

After a cache started, all changes of subtree will be synchronized from the ZooKeeper server. Events will be fired for those activity.

See also `listen()`.

Note: This method is not thread safe.

`close``()`

Closes the cache.

A closed cache was detached from ZooKeeper's changes. And all nodes will be invalidated.

Once a tree cache was closed, it could not be started again. You should only close a tree cache while you want to recycle it.

Note: This method is not thread safe.

`listen``(`*listener*`)`

Registers a function to listen the cache events.

The cache events are changes of local data. They are delivered from watching notifications in ZooKeeper session.

This method can be use as a decorator.

Parameters **listener** – A callable object which accepting a `TreeEvent` instance as its argument.

`listen_fault``(`*listener*`)`

Registers a function to listen the exceptions.

It is possible to meet some exceptions during the cache running. You could specific handlers for them.

This method can be use as a decorator.

Parameters **listener** – A callable object which accepting an exception as its argument.

`get_data``(`*path*, *default=None*`)`

Gets data of a node from cache.

Parameters

- **path** – The absolute path string.
- **default** – The default value which will be returned if the node does not exist.

Raises ValueError If the path is outside of this subtree.

Returns A `NodeData` instance.

`get_children``(`*path*, *default=None*`)`

Gets node children list from in-memory snapshot.

Parameters

- **path** – The absolute path string.
- **default** – The default value which will be returned if the node does not exist.

Raises ValueError If the path is outside of this subtree.

Returns The `frozenset` which including children names.

class `kazoo_sasl.recipe.cache.`**`TreeEvent`**

Bases: `tuple`

The immutable event tuple of cache.

`event_data`

A `NodeData` instance.

`event_type`

An enumerate integer to indicate event type.

classmethod **`make`** (*event_type*, *event_data*)

Creates a new TreeEvent tuple.

Returns A `TreeEvent` instance.

class `kazoo_sasl.recipe.cache.`**`NodeData`**

Bases: `tuple`

The immutable node data tuple of cache.

`data`

The bytes data of current node.

classmethod **`make`** (*path*, *data*, *stat*)

Creates a new NodeData tuple.

Returns A `NodeData` instance.

`path`

The absolute path string of current node.

`stat`

The stat information of current node.

`kazoo_sasl.recipe.counter`

Zookeeper Counter

Maintainer None

Status Unknown

New in version 0.7: The Counter class.

Public API

class `kazoo_sasl.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()
zk.start()
counter = zk.Counter("/int")
counter += 2
counter -= 1
counter.value == 1
counter.pre_value == 2
counter.post_value == 1

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

kazoo_sasl.recipe.election

ZooKeeper Leader Elections

Maintainer None

Status Unknown

Public API

class `kazoo_sasl.recipe.election.`**Election**(*client*, *path*, *identifier=None*)

Kazoo Basic Leader Election

Example usage with a `KazooClient` instance:

```
zk = KazooClient()
zk.start()
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.

kazoo_sasl.recipe.lease

Zookeeper lease implementations

Maintainer Lars Albertsson <lars.albertsson@gmail.com>

Maintainer Jyrki Pulliainen <jyrki@spotify.com>

Status Beta

Public API

class `kazoo_sasl.recipe.lease.`**`NonBlockingLease`**`(`*client*, *path*, *duration*, *identifier=None*, *utcnow=<built-in method utcnow of type object at 0x939c20>*`)`

Exclusive lease that does not block.

An exclusive lease ensures that only one client at a time owns the lease. The client may renew the lease without losing it by obtaining a new lease with the same path and same identity. The lease object evaluates to True if the lease was obtained.

A common use case is a situation where a task should only run on a single host. In this case, the clients that did not obtain the lease should exit without performing the protected task.

The lease stores time stamps using client clocks, and will therefore only work if client clocks are roughly synchronised. It uses UTC, and works across time zones and daylight savings.

Example usage: with a `KazooClient` instance:

```
zk = KazooClient()
zk.start()
# Hold lease over an hour in order to keep job on same machine,
# with failover if it dies.
lease = zk.NonBlockingLease(
    "/db_leases/hourly_cleanup", datetime.timedelta(minutes = 70),
    identifier = "DB hourly cleanup on " + socket.gethostname())
if lease:
    do_hourly_database_cleanup()
```

`__init__`(*client*, *path*, *duration*, *identifier=None*, *utcnow=<built-in method utcnow of type object at 0x939c20>*)

Create a non-blocking lease.

Parameters

- **client** – A `KazooClient` instance.
- **path** – The lease path to use.
- **duration** – Duration during which the lease is reserved. A `timedelta` instance.
- **identifier** – Unique name to use for this lease holder. Reuse in order to renew the lease. Defaults to `socket.gethostname()`.
- **utcnow** – Clock function, by default returning `datetime.datetime.utcnow()`. Used for testing.

class `kazoo_sasl.recipe.lease.`**`MultiNonBlockingLease`**(*client*, *count*, *path*, *duration*, *identifier=None*, *utcnow=<built-in method utcnow of type object at 0x939c20>*)

Exclusive lease for multiple clients.

This type of lease is useful when a limited set of hosts should run a particular task. It will attempt to obtain leases trying a sequence of ZooKeeper lease paths.

Parameters

- **client** – A `KazooClient` instance.
- **count** – Number of host leases allowed.
- **path** – ZooKeeper path under which lease files are stored.
- **duration** – Duration during which the lease is reserved. A `timedelta` instance.
- **identifier** –

  Unique name to use for this lease holder. Reuse in order to renew the lease.

  Defaults do `socket.gethostname()`.

- **utcnow** – Clock function, by default returning `datetime.datetime.utcnow()`. Used for testing.

`__init__`(*client*, *count*, *path*, *duration*, *identifier=None*, *utcnow=<built-in method utcnow of type object at 0x939c20>*)

`kazoo_sasl.recipe.lock`

Zookeeper Locking Implementations

Maintainer Ben Bangert <ben@groovie.org>

Status Production

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_sasl.recipe.lock.`**`Lock`**(*client*, *path*, *identifier=None*)

Kazoo Lock

Example usage with a `KazooClient` instance:

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

Note: This lock is not *re-entrant*. Repeated calls after already acquired will block.

This is an exclusive lock. For a read/write lock, see `WriteLock` and `ReadLock`.

`__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*, *ephemeral=True*)

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.
- **ephemeral** (*bool*) – Don't use ephemeral znode for the lock.

Returns Was the lock acquired?

Return type bool

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

> **Warning:** When `ephemeral` is set to False session expiration will not release the lock and must be handled separately.

New in version 1.1: The timeout option.

New in version 2.4.1: The ephemeral 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_sasl.recipe.lock.`**`ReadLock`**(*client*, *path*, *identifier=None*)

Kazoo Read Lock

Example usage with a `KazooClient` instance:

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

The lock path passed to WriteLock and ReadLock must match for them to communicate. The read lock blocks if it is held by any writers, but multiple readers may hold the lock.

Note: This lock is not *re-entrant*. Repeated calls after already acquired will block.

This is the read-side of a shared lock. See `Lock` for a standard exclusive lock and `WriteLock` for the write-side of a shared 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*, *ephemeral=True*)

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.
- **ephemeral** (*bool*) – Don't use ephemeral znode for the lock.

Returns Was the lock acquired?

Return type bool

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

Warning: When `ephemeral` is set to False session expiration will not release the lock and must be handled separately.

New in version 1.1: The timeout option.

New in version 2.4.1: The ephemeral 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_sasl.recipe.lock.`**`WriteLock`**(*client*, *path*, *identifier=None*)

Kazoo Write Lock

Example usage with a `KazooClient` instance:

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

The lock path passed to WriteLock and ReadLock must match for them to communicate. The write lock can not be acquired if it is held by any readers or writers.

Note: This lock is not *re-entrant*. Repeated calls after already acquired will block.

This is the write-side of a shared lock. See `Lock` for a standard exclusive lock and `ReadLock` for the read-side of a shared 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*, *ephemeral=True*)

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.
- **ephemeral** (*bool*) – Don't use ephemeral znode for the lock.

Returns Was the lock acquired?

Return type bool

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

> **Warning:** When `ephemeral` is set to False session expiration will not release the lock and must be handled separately.

New in version 1.1: The timeout option.

New in version 2.4.1: The ephemeral 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_sasl.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.

Note: This lock is not meant to be *re-entrant*.

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.

kazoo_sasl.recipe.partitioner

Zookeeper Partitioner Implementation

Maintainer None

Status Unknown

`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_sasl.recipe.partitioner.`**`SetPartitioner`**(*client*, *path*, *set*, *partition_func=None*, *identifier=None*, *time_boundary=30*, *max_reaction_time=1*, *state_change_event=None*)

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_sasl.client import KazooClient
client = KazooClient()
client.start()

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*, *max_reaction_time=1*, *state_change_event=None*)

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.
- **max_reaction_time** – Maximum reaction time for party members change.
- **state_change_event** – An optional Event object that will be set on every state change.

`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_sasl.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.

kazoo_sasl.recipe.party

Party

Maintainer Ben Bangert <ben@groovie.org>

Status Production

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

Public API

class `kazoo_sasl.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_sasl.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

kazoo_sasl.recipe.queue

Zookeeper based queue implementations.

Maintainer None

Status Possibly Buggy

Note: This queue was reported to cause memory leaks over long running periods. See: https://github.com/python-zk/kazoo/issues/175

New in version 0.6: The Queue class.

New in version 1.0: The LockingQueue class.

Public API

class `kazoo_sasl.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_sasl.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.

`release`()

Removes the lock from currently processed item without consuming it.

Returns True if the lock was removed successfully, False otherwise.

Return type bool

`kazoo_sasl.recipe.watchers`

Higher level child and data watching API's.

Maintainer Ben Bangert <ben@groovie.org>

Status Production

Note: *DataWatch* and *ChildrenWatch* may only handle a single function, attempts to associate a single instance with multiple functions will result in an exception being thrown.

Public API

class `kazoo_sasl.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.

Tip: Because `DataWatch` can watch nodes that don't exist, it can be used alternatively as a higher-level Exists watcher that survives reconnections and session loss.

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: DataWatch now ignores additional arguments that were previously passed to it and warns that they are no longer respected.

`__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_sasl.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_sasl.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.

`kazoo_sasl.retry`

Public API

class `kazoo_sasl.retry.`**`KazooRetry`**(*max_tries=1*, *delay=0.1*, *backoff=2*, *max_jitter=0.8*, *max_delay=60*, *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=60*, *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. -1 means infinite tries.
- **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 minute.
- **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_sasl.retry.`**`ForceRetryError`**
Raised when some recipe logic wants to force a retry.

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

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

kazoo_sasl.security

Kazoo Security

Public API

class `kazoo_sasl.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_sasl.security.`**`Id`**
Id(scheme, id)

`kazoo_sasl.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_sasl.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_sasl.

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_sasl.security.`**`make_digest_acl_credential`**`(`*username*, *password*`)`

Create a SHA1 digest credential

`kazoo_sasl.testing.harness`

Public API

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 (or eventlet) 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 (or eventlet) greenlets.

CHAPTER 3

Source Code

All source code is available on github under kazoo.

CHAPTER 4

Bugs/Support

Bugs 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 and support questions, please use the python-zk mailing list hosted on Google Groups.

CHAPTER 5

Indices and tables

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

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 from a variety of companies (twitter, mozilla, yahoo! and others).

Index