lynk Documentation

Release 0.1.0

John Carlyle

Jan 18, 2019

Contents:

Lynk is a Distributed Lock Manager (DLM) that uses DynamoDB to track the state of its locks. Lynk is a cooperative locking scheme where each client assumes that all others in the system are obeying a set of rules in order to assure the integrity of the locks.

CHAPTER 1

lynk

1.1 lynk package

1.1.1 Subpackages

lynk.backends package

Submodules

lynk.backends.base module

lynk.backends.dynamodb module

Module contents

1.1.2 Submodules

1.1.3 lynk.exceptions module

1.1.4 lynk.lock module

1.1.5 lynk.refresh module

1.1.6 lynk.session module

1.1.7 lynk.techniques module

1.1.8 lynk.utils module

1.1.9 Module contents

CHAPTER 2

Getting Started

2.1 Quickstart

2.1.1 Installation

Lynk is available on PyPi as `lynk` and can be installed in the usual way with pip:

```
$ pip install lynk
```

2.1.2 AWS Credentials

Lynk uses boto3 in order to make all calls to AWS, which means it uses the boto3 standard credential chain. Make sure your machine has AWS credentials configured in the way boto3 expects.

2.1.3 Creating a table

In order to store the locks we need to create a DynamoDB table. For ease of getting started there is a command line tool installed along with the package to help manage lynk tables.

To create a table called `quickstart` run the `lynk create-table` command:

```
$ lynk create-table lynk-quickstart
Creating table lynk-quickstart
Created
```

With the `lynk list-tables` command line tool you can check a list of tables created this way by lynk:

```
$ lynk list-tables
lynk-quickstart
```

2.1.4 Creating a lock

Locks are shared through a DynamoDB table, in our case we will be using the `lynk-quickstart` we created earlier table. Locks are distinguished by a lock name, within their table. To get create a lock, we first need to create a `lynk.session.Session` that is bound to our table. The session can then be used to create multiple locks that will be backed by that table.

The easiet way to make a `lynk.session.Session` is by using the `lynk.get_session()` function. This function only takes one argument which is the name of the table it is bound to. Once a session has been created it can be used to create lock objects using the `lynk.session.Session.create_lock()` method.

```
import lynk

session = lynk.get_session('lynk-quickstart')
lock = session.create_lock('my lock')
```

`lock` is an instance of `lynk.lock.Lock` which is bound to both our table `lynk-quickstart`, and the logical lock name `my lock`. If we create another lock object bound to the same table, with the same lock name, only one will be acquireable at a time, with the second having to wait for the first one to release before being able to acquire it. This is a little bit awkard to show in a single code segment since it requires muiltiple threads. Below is a minimal but complete example of using two threads to contend for the same lock.

```
import time
import logging
import threading

import lynk


LOG = logging.getLogger(__file__)


def configure_logging():
    LOG.setLevel(logging.DEBUG)
    formatter = logging.Formatter('%(threadName)s - %(message)s')
    ch = logging.StreamHandler()
    ch.setFormatter(formatter)
    LOG.addHandler(ch)


def thread(session):
    LOG.debug('Starting')
    lock = session.create_lock('my lock')
    lock.acquire()
    LOG.debug('Lock acquired')
    time.sleep(10)
    lock.release()
    LOG.debug('Lock released')


def main():
    configure_logging()
    session = lynk.get_session('lynk-quickstart')
    t1 = threading.Thread(target=thread, args=(session,))
    t2 = threading.Thread(target=thread, args=(session,))

    t1.start()
```

(continues on next page)

(continued from previous page)

```
    t2.start()
    t1.join()
    t2.join()


if __name__ == "__main__":
    main()
```

First, we can ignore the `configure_logging` function, it just sets up logging to show which thread is emitting the logs. This makes it easier to track the flow of our program.

Looking at the `main` function, the first real thing that happens we create a session that can create locks bound to our table `lynk-quickstart`.

```
session = lynk.get_session('lynk-quickstart')
```

We then create two thread objects, and pass our `session` object into each as a shared variable. Once started each thread will execute the `thread` function.

```
t1 = threading.Thread(target=thread, args=(session,))
t2 = threading.Thread(target=thread, args=(session,))
```

The last thing the `main` function does is start both threads, then join on them, which will wait for them to complete before exiting.

```
t1.start()
t2.start()
t1.join()
t2.join()
```

Now we have two threads executing the `thread` function. Following along each thread, disregarding the log statements, the first thing it does is create a lock object.

```
lock = session.create_lock('my lock')
```

This means each thread will have its own unique lock object linked logically to the name `my lock`. The threads share a session, which is bound to the table `lynk-quickstart`. Simply creating the lock does not interact with the DynamoDB Tables in any way.

Next each thread tries to acquire the lock.

```
lock.acquire()
```

This simple statement is what makes the call to write an entry in our DynamoDB Table. Once an entry is written, this indicates that the lock is in-use and we are safe to operate on whatever resource this lock was responsible for protecting. In this example case we simply sleep for 10 seconds and then release the lock.

```
time.sleep(10)
lock.release()
```

The `time.sleep(10)` call would be replaced with real work in an actual application. Once the protected resource is done being operated on, and has been safely written and is ready for another agent to use, we release the lock. The `lynk.lock.Lock.release()` call deletes the entry from the table freeing the lock name up to be used by another agent.

The output of our little sample application is shown below. You can see one thread gets the lock (in this case `Thread-2`) and does it work while the other thread waits for it to be released. Once released, the other thread

repeats the same process:

```
Thread-1 - Starting
Thread-2 - Starting
Thread-2 - Lock acquired
Thread-2 - Lock released
Thread-1 - Lock acquired
Thread-1 - Lock released
```

More complex but similar examples can be seen in the examples directory of the source repo.

2.1.5 Lock entry details

If you have the AWS CLI installed you can run the following command while the example script above is running (shouldn't be too difficult since the script takes around 30 seconds to complete):

```
$ aws dynamodb scan --table-name lynk-quickstart --query Items
[
    {
        "lockKey": {
            "S": "my lock"
        },
        "leaseDuration": {
            "N": "20"
        },
        "versionNumber": {
            "S": "dabbbfde-93cb-47f8-a249-fbae84c4a5e3"
        },
        "hostIdentifier": {
            "S": "Johns-MacBook-Pro.local"
        }
    }
]
```

While the lock is held by a thread, we can see the entry that marks it as in use. It has four components, the `lockKey` which is clearly the lock name that we selected when creating our lock object. A `leaseDuration`, this is the amount of time we have a lease on this lock. Any other agent that wants to acquire this lock must wait at least that long before trying again. Our example code will refresh this lock automatically, even if we had slept longer than 20 seconds. The `versionNumber` is used as a fencing token, each write to this entry changes this value. You can read more about how the `leaseDuration` and `versionNumber` are used to ensure the lock integrity in the documentation for the `lynk.techniques.VersionLeaseTechinque`. Finally there is a `hostIdentifier` which is just there to show the host that created the lock. This can be used for debugging a distributed multi-agent system all using one lock table.

More examples can be found in the examples directory in the source repo.

2.1.6 Context manager

In the above example we manually call `acquire()` and `release()`. This depends on no exceptions ocurring, and would generally be safer in a `try: finally:` block. For convenience the `lynk.lock.Lock` object can be called and used as a context manager. The following code:

```
lock.acquire()
time.sleep(10)
lock.release()
```

Can be re-written more safely, and conveinently, as:

```
with lock():
    time.sleep(10)
```

This ensures the releasing in the lock in the case of an unexpected exception.

2.1.7 Teardown

To tear down the resources created during the quickstart tutorial run the `lynk delete-table` command:

```
$ lynk delete-table lynk-quickstart
Deleting table lynk-quickstart
Deleted
```

Verify that there are no left over tables checking that the following has no output:

```
$ lynk list-tables
```

CHAPTER 3

Topics

3.1 Serialization

3.1.1 Overview

A lock can be used to control access to some shared resource. In a serverless environment jobs can be broken up into many pieces often being operated on by a sequence of disparate functions. In order to hold a lock between these separate functions there needs to be a mechanism for a lock to be transfered between seprate processes, or passed through a queue etc.

To accomodate this use case, Locks are serializable. A lock can be taken against a resource in one process, serialized and passed into some other process. The target process can then derserialize the lock and continue operating on the same resource as the first.

3.1.2 Serialization

Lynk Locks use a simple JSON serialization scheme, to allow them to be passed around as plain text between processes.

To serialize a Lock use the `.serialize()` method.

```
import lynk

session = lynk.get_session('lynk-quickstart')
lock = session.create_lock('my lock')
serialized_lock = lock.serialize()
```

The `serialized_lock` variable is now a plain UTF-8 string that can be sent to another component of a complex system.

3.1.3 Deserialization

A lock object can be loaded using the `Session` method `deserialize_lock()` and passing it the `serialzied_lock` value from the previous section. If it successful the new process can now start operating on the protected resource. Otherwise it will raise the `LockAlreadyInUseError` to indicate that the lock was stolen between serialization and deserialization.

```
import lynk
from lynk.exceptions import LockAlreadyInUseError

try:
    session = lynk.get_session('lynk-quickstart')
    lock = session.deserialize_lock(serialized_lock)
    do_stuff_with_locked_resource(lock)
except LockAlreadyInUseError as:
    print("Someone else stole the lock.")
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search