redis-py-cluster Documentation

Release 1.2.0

Johan Andersson

May 20, 2020

Contents

This project is a port of *redis-rb-cluster* by antirez, with alot of added functionality. The original source can be found at https://github.com/antirez/redis-rb-cluster.

The source code is available on github.

CHAPTER 1

Installation

Latest stable release from pypi

```
$ pip install redis-py-cluster
```

or from source code

```
$ python setup.py install
```

CHAPTER 2

Usage example

Small sample script that shows how to get started with RedisCluster. It can also be found in the file *exmaples/basic.py*

```
>>> from rediscluster import StrictRedisCluster

>>> # Requires at least one node for cluster discovery. Multiple nodes is recommended.
>>> startup_nodes = [{"host": "127.0.0.1", "port": "7000"}]

>>> # Note: decode_responses must be set to True when used with python3
>>> rc = StrictRedisCluster(startup_nodes=startup_nodes, decode_responses=True)

>>> rc.set("foo", "bar")
True
>>> print(rc.get("foo"))
'bar'
```

CHAPTER 3

Dependencies & supported python versions

- Python: redis >= *2.10.2*, <= *2.10.5* is required. Older versions in the *2.10.x* series can work but using the latest one is allways recommended.
- Optional Python: hiredis >= *0.2.0*. Older versions might work but is not tested.
- A working Redis cluster based on version >= *3.0.0* is required. Only *3.0.x* releases is supported.

CHAPTER 4

Supported python versions

- 2.7.x
- 3.3.x
- 3.4.1+
- 3.5.x

Experimental:

- Up to 3.6.0a0

Note: Python 3.4.0

A segfault was found when running *redis-py* in python *3.4.0* that was introduced into the codebase in python *3.4.0*. Because of this both *redis-py* and *redis-py-cluster* will not work when running with *3.4.0*. This lib has decided to block the lib from execution on *3.4.0* and you will get a exception when trying to import the code. The only solution is to use python *3.4.1* or some other higher minor version in the *3.4* series.

CHAPTER 5

Regarding duplicate pypi and python naming

It has been found that the python module name that is used in this library (rediscluster) is already shared with a similar but older project.

This lib will not change the naming of the module to something else to prevent collisions between the libs.

My reasoning for this is the following

- Changing the namespace is a major task and probably should only be done in a complete rewrite of the lib, or if the lib had plans for a version 2.0.0 where this kind of backwards incompatibility could be introduced.
- This project is more up to date, the last merged PR in the other project was 3 years ago.
- This project is aimed for implement support for the cluster support in 3.0+, the other lib do not have that right now, but they implement almost the same cluster solution as the 3.0+ but in much more in the client side.
- The 2 libs is not compatible to be run at the same time even if the name would not collide. It is not recommended to run both in the same python interpreter.

An issue has been raised in each repository to have tracking of the problem.

redis-py-cluster: https://github.com/Grokzen/redis-py-cluster/issues/150

rediscluster: https://github.com/salimane/rediscluster-py/issues/11

CHAPTER 6

The Usage Guide

6.1 Implemented commands

This will describe all changes that StrictRedisCluster have done to make a command to work in a cluster environment.

If a command is not listed here then the default implementation from *StrictRedis* in the *redis-py* library is used.

6.1.1 Fanout Commands

The following commands will send the same request to all nodes in the cluster. Results is returned as a dict with k,v pair (NodeID, Result).

- bgrewriteaof
- bgsave
- client_getname
- client_kill
- client_list
- client_setname
- config_get
- config_resetstat
- config_rewrite
- config_set
- dbsize
- echo
- info
- lastsave

- ping
- save
- slowlog_get
- slowlog_len
- slowlog_reset
- time

This command will send the same request to all nodes in the cluster in sequence. Results is appended to a unified list.

- keys

The following commands will only be send to the master nodes in the cluster. Results is returned as a dict with k,v pair (NodeID, Command-Result).

- flushall
- flushdb
- scan

This command will sent to a random node in the cluster.

- publish

The following commands will be sent to the server that matches the first key.

- eval
- evalsha

This following commands will be sent to the master nodes in the cluster.

- script load - the result is the hash of loaded script
- script flush - the result is *True* if the command succeeds on all master nodes, else *False*
- script exists - the result is an array of booleans. An entry is *True* only if the script exists on all the master nodes.

The following commands will be sent to the sever that matches the specefied key.

- hscan
- hscan_iter
- scan_iter
- sscan
- sscan_iter
- zscan
- zscan_iter

6.1.2 Blocked commands

The following commands is blocked from use.

Either because they do not work, there is no working implementation or it is not good to use them within a cluster.

- bitop - Currently to hard to implement a solution in python space
- client_setname - Not yet implemented

- move - It is not possible to move a key from one db to another in cluster mode
- restore
- script_kill - Not yet implemented
- sentinel
- sentinel_get_master_addr_by_name
- sentinel_master
- sentinel_masters
- sentinel_monitor
- sentinel_remove
- sentinel_sentinels
- sentinel_set
- sentinel_slaves
- shutdown
- slaveof - Cluster management should be done via redis-trib.rb manually
- unwatch - Not yet implemented
- watch - Not yet implemented

6.1.3 Overridden methods

The following methods is overridden from StrictRedis with a custom implementation.

They can operate on keys that exists in different hashslots and require a client side implementation to work.

- brpoplpus
- mget
- mset
- msetnx
- pfmerge
- randomkey
- rename
- renamenx
- rpoplpush
- sdiff
- sdiffstore
- sinter
- sinterstore
- smove
- sort
- sunion

- sunionstore
- zinterstore
- zunionstore

6.2 Limitations and differences

This will compare against *redis-py*

There is alot of differences that have to be taken into consideration when using redis cluster.

Any method that can operate on multiple keys have to be reimplemented in the client and in some cases that is not possible to do. In general any method that is overriden in StrictRedisCluster have lost the ability of being atomic.

Pipelines do not work the same way in a cluster. In *StrictRedis* it batch all commands so that they can be executed at the same time when requested. But with RedisCluster pipelines will send the command directly to the server when it is called, but it will still store the result internally and return the same data from .execute(). This is done so that the code still behaves like a pipeline and no code will break. A better solution will be implemented in the future.

Alot of methods will behave very different when using RedisCluster. Some methods send the same request to all servers and return the result in another format then *StrictRedis* do. Some methods is blocked because they do not work / is not implemented / is dangerous to use in redis cluster.

Some of the commands are only partially supported when using RedisCluster. The commands `zinterstore` and `zunionstore` are only supported if all the keys map to the same key slot in the cluster. This can be achieved by namespacing related keys with a prefix followed by a bracketed common key. Example:

```
r.zunionstore('d{foo}', ['a{foo}', 'b{foo}', 'c{foo}'])
```

This corresponds to how redis behaves in cluster mode. Eventually these commands will likely be more fully supported by implementing the logic in the client library at the expense of atomicity and performance.

6.3 Pipelines

6.3.1 How pipelining works

Just like in *redis-py*, *redis-py-cluster* queues up all the commands inside the client until execute is called. But, once execute is called, *redis-py-cluster* internals work slightly differently. It still packs the commands to efficiently transmit multiple commands across the network. But since different keys may be mapped to different nodes, redis-py-cluster must first map each key to the expected node. It then packs all the commands destined for each node in the cluster into its own packed sequence of commands. It uses the redis-py library to communicate with each node in the cluster.

Ideally all the commands should be sent to each node in the cluster in parallel so that all the commands can be processed as fast as possible. The naive approach is to iterate through each node and send each batch of commands sequentially to each node. If redis-py supported some sort of non-blocking i/o we could send the network requests first and multiplex the socket responses from each node. Instead, we use threads to send the requests in parallel so that the total execution time only equals the amount of time for the slowest round trip to and from the given set of nodes in the cluster needed to process the commands.

In previous versions of the library there were some bugs associated with threaded operations and pipelining. We were freeing connections back into the connection pool prior to reading the responses from each thread and it caused all kinds of problems. Those issues were fixed but there was a special flag to allow you to turn off threading in case you were worried about it. Since we no longer have to use threads at all to get the performance we want, that flag was removed from the client.

6.3.2 Connection Error handling

The other way pipelines differ in *redis-py-cluster* from *redis-py* is in error handling and retries. With the normal *redis-py* client, if you hit a connection error during a pipeline command it raises the error right there. But we expect redis-cluster to be more resilient to failures.

If you hit a connection problem with one of the nodes in the cluster, most likely a stand-by slave will take over for the down master pretty quickly. In this case, we try the commands bound for that particular node to another random node. The other random node will not just blindly accept these commands. It only accepts them if the keys referenced in those commands actually map to that node in the cluster configuration.

Most likely it will respond with a *MOVED* error telling the client the new master for those commands. Our code handles these *MOVED* commands according to the redis cluster specification and re-issues the commands to the correct server transparently inside of *pipeline.execute()* method. You can disable this behavior if you'd like as well.

ASKED and MOVED errors

The other tricky part of the redis-cluster specification is that if any command response comes back with an *ASK* or *MOVED* error, the command is to be retried against the specified node.

In previous versions of *redis-py-cluster* treated *ASKED* and *MOVED* errors the same, but they really need to be handled differently. *MOVED* error means that the client can safely update its own representation of the slots table to point to a new node for all future commands bound for that slot.

An *ASK* error means the slot is only partially migrated and that the client can only successfully issue that command to the new server if it prefixes the request with an ‘ASKING¨ ‘ command first. This lets the new node taking over that slot know that the original server said it was okay to run that command for the given key against the new node even though the slot is not yet completely migrated. Our current implementation now handles this case correctly.

6.3.3 The philosophy on pipelines

After playing around with pipelines and thinking about possible solutions that could be used in a cluster setting this document will describe how pipelines work, strengths and weaknesses with the implementation that was chosen.

Why can't we reuse the pipeline code in *redis-py*? In short it is almost the same reason why code from the normal redis client can't be reused in a cluster environment and that is because of the slots system. Redis cluster consist of a number of slots that is distributed across a number of servers and each key belongs in one of these slots.

In the normal pipeline implementation in *redis-py* we can batch send all the commands and send them to the server at once, thus speeding up the code by not issuing many requests one after another. We can say that we have defined and guaranteed execution order becuase of this.

One problem that appears when you want to do pipelines in a cluster environment is that you can't have guaranteed execution order in the same way as a single server pipeline. The problem is that because you can queue an command to any key, we will end up in most of the cases having to talk to 2 or more nodes in the cluster to execute the pipeline. The problem with that is that there is no single place/node/way to send the pipeline and redis will sort everything out by itself via some internal mechanisms. Because of that when we build a pipeline for a cluster we have to build several smaller pipelines that we each send to the desegnated node in the cluster.

When the pipeline is executed in the client each key is checked to what slot it shold be sent to and the pipelines is built up based on that information. One thing to note here is that there will be partial correct execution order if you look over the entire cluster because for each pipeline the ordering will be correct. It can also be argued that the correct execution order is applied/valid for each slot in the cluster.

The next thing to take into consideration is what commands should be available and which should be blocked/locked.

In most cases and in almost all solutions multi key commands have to be blocked hard from beeing execute inside a pipeline. This would only be possible in the case you have a pipeline implementation that allways executes immeditally each command is queued up. That solution would only give the interface of working like a pipeline to ensure old code

will still work, but it would not give any benefits or advantages other than all commands would work and old code would work.

In the solution for this lib multikey commands is blocked hard and will probably not be enabled in pipelines. If you really need to use them you need to execute them through the normal cluster client if they are implemented and works in there. Why can't multi key commands work? In short again it is because they keys can live in different slots on different nodes in the cluster. It is possible in theory to have any command work in a cluster, but only if the keys operated on belongs to the same cluster slot. This lib have decided that currently no serious support for that will be attempted.

Examples on commands that do not work is *MGET*, *MSET*, *MOVE*.

One good thing that comes out of blocking multi key commands is that correct execution order is less of a problem and as long as it applies to each slot in the cluster we shold be fine.

Consider the following example. Create a pipeline and issue 6 commands *A*, *B*, *C*, *D*, *E*, *F* and then execute it. The pipeline is calculated and 2 sub pipelines is created with *A*, *C*, *D*, *F* in the first and *B*, *E* in the second. Both pipelines is then sent to each node in the cluster and a response is sent back. For the first node *[True, MovedException(12345), MovedException(12345), True]* and from the second node [*True*, *True*]. After this response is parsed we see that 2 commands in the first pipeline did not work and must be sent to another node. This case happens if the client slots cache is wrong because a slot was migrated to another node in the cluster. After parsing the response we then build a third pipeline object with commands [*C*, *D*] to the second node. The third object is executed and passes and from the client perspective the entire pipeline was executed.

If we look back at the order we executed the commands we get *[A, F]* for the first node and *[B, E, C, D]* for the second node. At first glance this looks like it is out of order because command *E* is executed before *C* & *D*. Why do this not matter? Because no multi key operations can be done in a pipeline we only have to care the execution order is correct for each slot and in this case it was because *B* & *E* belongs to the same slot and *C* & *D* belongs to the same slot. There should be no possible way to corrupt any data between slots if multi key commands is blocked by the code.

What is good with this pipeline solution? First we can actually have a pipeline solution that will work in most cases with few commands blocked (only multi key commands). Secondly we can run it in parralel to increase the performance of the pipeline even further, making the benefits even greater.

6.3.4 Transactions and WATCH

Support for transactions and WATCH:es in pipelines. If we look on the entire pipeline across all nodes in the cluster there is no possible way to have a complete transaction across all nodes because if we need to issue commands to 3 servers, each server is handled by its own and there is no way to tell other nodes to abort a transaction if only one of the nodes fail but not the others. A possible solution for that could be to implement a 2 step commit process. The 2 steps would consist of building 2 batches of commands for each node where the first batch would consist of validating the state of each slot that the pipeline wants to operate on. If any of the slots is migrating or moved then the client can correct its slots cache and issue a more correct pipeline batch. The second step would be to issue the acctuall commands and the data would be commited to redis. The big problem with this is that 99% of the time this would work really well if you have a very stable cluster with no migrations/resharding/servers down. But there can be times where a slot has begun migration in between the 2 steps of the pipeline and that would cause a race condition where the client thinks it has corrected the pipeline and wants to commit the data but when it does it will still fail.

Why *MULTI/EXEC* support won't work in a cluster environment. There is some test code in the second *MULTI/EXEC cluster test code* of this document that tests is *MULTI/EXEC* is possible to use in a cluster pipeline. The tests shows a huge problem when errors occus. If we wrap *MULTI/EXEC* in a packed set of commands then if a slot is migrating we will not get a good error we can parse and use. Currently it will only report *True* or *False* so we can narrow down what command failed but not why it failed. This might work really well if used on a non clustered node becuase it do not have to take care of *ASK* or *MOVED* errors. But for a cluster we need to know what cluster error occured so the correct action to fix the problem can be taken. Sinc there is more then 1 error to take care of it is not possible to take action based on just *True* or *False*.

Because of this problem with error handling *MULTI/EXEC* is blocked hard in the code from beeing used in a pipeline because the current implementation can't handle the errors.

In theory it could be possible to design a pipeline implementation that can handle this case by trying to determined by itself what it should do with the error by either asking the cluster after a *False* value was found in the response about the current state of the slot or just default to *MOVED* error handling and hope for the best. The problem is that this is not 100% guaranteed to work and can easily cause problems when wrong action was taken on the response.

Currently *WATCH* requires more studying if it possible to use or not, but sinc it is tied into *MULTI/EXEC* pattern it probably will not be supported for now.

6.3.5 MULTI/EXEC cluster test code

This code do NOT wrap *MULTI/EXEC* around the commands when packed

```
>>> from rediscluster import StrictRedisCluster as s
>>> r = s(startup_nodes=[{"host": "127.0.0.1", "port": "7002"}])
>>> # Simulate that a slot is migrating to another node
>>> r.connection_pool.nodes.slots[14226] = {'host': '127.0.0.1', 'server_type':
↪'master', 'port': 7001, 'name': '127.0.0.1:7001'}
>>> p = r.pipeline()
>>> p.command_stack = []
>>> p.command_stack.append((["SET", "ert", "tre"], {}))
>>> p.command_stack.append((["SET", "wer", "rew"], {}))
>>> p.execute()

ClusterConnection<host=127.0.0.1,port=7001>
[True, ResponseError('MOVED 14226 127.0.0.1:7002',)]
ClusterConnection<host=127.0.0.1,port=7002>
[True]
```

This code DO wrap MULTI/EXEC around the commands when packed

```
>>> from rediscluster import StrictRedisCluster as s
>>> r = s(startup_nodes=[{"host": "127.0.0.1", "port": "7002"}])
>>> # Simulate that a slot is migrating to another node
>>> r.connection_pool.nodes.slots[14226] = {'host': '127.0.0.1', 'server_type':
↪'master', 'port': 7001, 'name': '127.0.0.1:7001'}
>>> p = r.pipeline()
>>> p.command_stack = []
>>> p.command_stack.append((["SET", "ert", "tre"], {}))
>>> p.command_stack.append((["SET", "wer", "rew"], {}))
>>> p.execute()
ClusterConnection<host=127.0.0.1,port=7001>
[True, False]
```

6.3.6 Different pipeline solutions

This section will describe different types of pipeline solutions. It will list their main benefits and weaknesses.

Note: This section is mostly random notes and thoughts and not that well written and cleaned up right now. It will be done at some point in the future.

Suggestion one

Simple but yet sequential pipeline. This solution acts more like an interface for the already existing pipeline implementation and only provides a simple backwards compatible interface to ensure that code that sexists still will work withouth any major modifications. The good this with this implementation is that because all commands is runned in sequence it will handle *MOVED* or *ASK* redirections very good and withouth any problems. The major downside to this solution is that no commands is ever batched and runned in parralell and thus you do not get any major performance boost from this approach. Other plus is that execution order is preserved across the entire cluster but a major downside is that thte commands is no longer atomic on the cluster scale because they are sent in multiple commands to different nodes.

Good

- Sequential execution of the entire pipeline
- Easy *ASK* or *MOVED* handling

Bad

- No batching of commands aka. no execution speedup

Suggestion two

Current pipeline implementation. This implementation is rather good and works well because it combines the existing pipeline interface and functionality and it also provides a basic handling of *ASK* or *MOVED* errors inside the client. One major downside to this is that execution order is not preserved across the cluster. Altho the execution order is somewhat broken if you look at the entire cluster level becuase commands can be splitted so that cmd1, cmd3, cmd5 get sent to one server and cmd2, cmd4 gets sent to another server. The order is then broken globally but locally for each server it is preserved and maintained correctly. On the other hand i guess that there can't be any commands that can affect different hashslots within the same command so it maybe do not really matter if the execution order is not correct because for each slot/key the order is valid. There might be some issues with rebuilding the correct response ordering from the scattered data because each command might be in different sub pipelines. But i think that our current code still handles this correctly. I think i have to figure out some wierd case where the execution order acctually matters. There might be some issues with the nonsupported mget/mset commands that acctually performs different sub commands then it currently supports.

Good

- Sequential execution per node

Bad

- Not sequential execution on the entire pipeline
- Medium difficult *ASK* or *MOVED* handling

Suggestion three

There is a even simpler form of pipelines that can be made where all commands is supported as long as they conform to the same hashslot because redis supports that mode of operation. The good thing with this is that sinc all keys must belong to the same slot there can't be very few *ASK* or *MOVED* errors that happens and if they happen they will be very easy to handle because the entire pipeline is kinda atomic because you talk to the same server and only 1 server. There can't be any multiple server communication happening.

Good

- Super simple *ASK* or *MOVED* handling
- Sequential execution per slot and through the entire pipeline

Bad

- Single slot per pipeline

Suggestion four

One other solution is the 2 step commit solution where you send for each server 2 batches of commands. The first command should somehow establish that each keyslot is in the correct state and able to handle the data. After the client have recieved OK from all nodes that all data slots is good to use then it will acctually send the real pipeline with all data and commands. The big problem with this approach is that ther eis a gap between the checking of the slots and the acctual sending of the data where things can happen to the already established slots setup. But at the same time there is no possibility of merging these 2 steps because if step 2 is automatically runned if step 1 is Ok then the pipeline for the first node that will fail will fail but for the other nodes it will suceed but when it should not because if one command gets *ASK* or *MOVED* redirection then all pipeline objects must be rebuilt to match the new specs/setup and then reissued by the client. The major advantage of this solution is that if you have total controll of the redis server and do controlled upgrades when no clients is talking to the server then it can acctually work really well because there is no possibility that *ASK* or *MOVED* will triggered by migrations in between the 2 batches.

Good

- Still rather safe because of the 2 step commit solution
- Handles *ASK* or *MOVED* before commiting the data

Bad

- Big possibility of race conditions that can cause problems

6.4 Threaded Pipeline

Redis cluster optionally supports parallel execution of pipelined commands to reduce latency of pipelined requests via threads.

6.4.1 Rationale

When pipelining a bunch of commands to the cluster, many of the commands may be routed to different nodes in the cluster. The client-server design in redis-cluster dictates that the client communicates directly with each node in the cluster rather than treating each node as a homogenous group.

The advantage to this design is that a smart client can communicate with the cluster with the same latency characteristics as it might communicate with a single-instance redis cluster. But only if the client can communicate with each node in parallel.

6.4.2 Packing Commands

When issuing only a single command, there is only one network round trip to be made. But what if you issue 100 pipelined commands? In a single-instance redis configuration, you still only need to make one network hop. The commands are packed into a single request and the server responds with all the data for those requests in a single response. But with redis cluster, those keys could be spread out over many different nodes.

The client is responsible for figuring out which commands map to which nodes. Let's say for example that your 100 pipelined commands need to route to 3 different nodes? The first thing the client does is break out the commands that go to each node, so it only has 3 network requests to make instead of 100.

6.4.3 Parallel network i/o using threads

That's pretty good. But we are still issuing those 3 network requests in serial order. The code loops through each node and issues a request, then gets the response, then issues the next one.

We improve the situation by using python threads, making each request in parallel over the network. Now we are only as slow as the slowest single request.

Disabling Threads You can disable threaded execution either in the class constructor:

```
r = rediscluster.StrictRedisCluster( ... pipeline_use_threads=False) #true by default
pipe = r.pipeline()
```

Or you can disable it on a case by case basis as you instantiate the pipeline object.

```
pipe = r.pipeline(use_threads=False)
```

The later example always overrides if explicitly set. Otherwise, it falls back on the value passed to the StrictRedisCluster constructor.

6.4.4 Footnote: Gevent

Python offers something even more lightweight and efficient than threads to perform tasks in parallel: GEVENT.

You can read up more about gevent here: http://www.gevent.org/

If you want to try to get the benefits of gevent in redis-py-cluster, you can monkey patch your code with the following lines at the very beginning of your application:

```
import os
os.environ["GEVENT_RESOLVER"] = "ares"
import gevent.monkey
gevent.monkey.patch_all()
```

This will patch the python socket code, threaded libraries, and dns resolution into a single threaded application substituting coroutines for parallel threads.

6.5 Pubsub

After testing pubsub in cluster mode one big problem was discovered with the *PUBLISH* command.

According to the current official redis documentation on *PUBLISH*:

```
Integer reply: the number of clients that received the message.
```

It was initially assumed that if we had clients connected to different nodes in the cluster it would still report back the correct number of clients that recieved the message.

However after some testing of this command it was discovered that it would only report the number of clients that have subscribed on the same server the *PUBLISH* command was executed on.

Because of this, if there is some functionality that relies on an exact and correct number of clients that listen/subscribed to a specific channel it will be broken or behave wrong.

Currently the only known workarounds is to:

- Ignore the returned value

- All clients talk to the same server
- Use a non clustered redis server for pubsub operations

Discussion on this topic can be found here: https://groups.google.com/forum/?hl=sv#!topic/redis-db/BlwSOYNBUl8

6.5.1 Scalability issues

The following part is from this discussion https://groups.google.com/forum/?hl=sv#!topic/redis-db/B0_fvfDWLGM and it describes the scalability issue that pubsub has and the performance that goes with it when used in a cluster environment.

> according to [1] and [2] PubSub works by broadcasting every publish to every other Redis Cluster node. This limits the PubSub throughput to the bisection bandwidth of the underlying network infrastructure divided by the number of nodes times message size. So if a typical message has 1KB, the cluster has 10 nodes and bandwidth is 1 GBit/s, throughput is already limited to 12.5K RPS. If we increase the message size to 5 KB and the number of nodes to 50, we only get 500 RPS much less than a single Redis instance could service (>100K RPS), while putting maximum pressure on the network. PubSub thus scales linearly wrt. to the cluster size, but in the the negative direction!

6.5.2 How pubsub works in StrictRedisCluster

In release *1.2.0* the pubsub was code was reworked to now work like this.

For *PUBLISH* and *SUBSCRIBE* commands:

- The channel name is hashed and the keyslot is determined.
- Determine the node that handles the keyslot.
- Send the command to the node.

The old solution was that all pubsub connections would talk to the same node all the time. This would ensure that the commands would work.

This new solution is probably future safe and it will probably be a similar solution when *redis* fixes the scalability issues.

6.5.3 Known limitations with pubsub

Pattern subscribe and publish do not work properly because if we hash a pattern like *fo** we will get a keyslot for that string but there is a endless posiblity of channel names based on that pattern that we can't know in advance. This feature is not limited but the commands is not recommended to use right now.

The implemented solution will only work if other clients use/adopt the same behaviour. If some other client behaves differently, there might be problems with *PUBLISH* and *SUBSCRIBE* commands behaving wrong.

6.5.4 Other solutions

The simplest solution is to have a seperate non clustered redis instance that you have a regular *StrictRedis* instance that works with your pubsub code. It is not recommended to use pubsub until *redis* fixes the implementation in the server itself.

6.6 Readonly mode

By default, Redis Cluster always returns MOVE redirection response on accessing slave node. You can overcome this limitation [for scaling read with READONLY mode](http://redis.io/topics/cluster-spec#scaling-reads-using-slave-nodes).

redis-py-cluster also implements this mode. You can access slave by passing *readonly_mode=True* to StrictRedisCluster (or RedisCluster) constructor.

```
>>> from rediscluster import StrictRedisCluster
>>> startup_nodes = [{"host": "127.0.0.1", "port": "7000"}]
>>> rc = StrictRedisCluster(startup_nodes=startup_nodes, decode_responses=True)
>>> rc.set("foo16706", "bar")
>>> rc.set("foo81", "foo")
True
>>> rc_readonly = StrictRedisCluster(startup_nodes=startup_nodes, decode_
↪responses=True, readonly_mode=True)
>>> rc_readonly.get("foo16706")
u'bar'
>>> rc_readonly.get("foo81")
u'foo'
```

We can use pipeline via *readonly_mode=True* object.

```
>>> with rc_readonly.pipeline() as readonly_pipe:
...     readonly_pipe.get('foo81')
...     readonly_pipe.get('foo16706')
...     readonly_pipe.execute()
...
[u'foo', u'bar']
```

But this mode has some downside or limitations.

- It is possible that you cannot get the latest data from READONLY mode enabled object because Redis implements asynchronous replication.
- **You MUST NOT use SET related operation with READONLY mode enabled object**, otherwise you can possibly get 'Too many Cluster redirections' error because we choose master and its slave nodes randomly.
- You should use get related stuff only.
- Ditto with pipeline, otherwise you can get 'Command # X (XXXX) of pipeline: MOVED' error.

```
>>> rc_readonly = StrictRedisCluster(startup_nodes=startup_nodes, decode_
↪responses=True, readonly_mode=True)
>>> # NO: This works in almost case, but possibly emits Too many Cluster redirections␣
↪error...
>>> rc_readonly.set('foo', 'bar')
>>> # OK: You should always use get related stuff...
>>> rc_readonly.get('foo')
```

6.7 Redis cluster setup

6.7.1 Manually

- Redis cluster tutorial: http://redis.io/topics/cluster-tutorial

- Redis cluster specs: http://redis.io/topics/cluster-spec
- This video will describe how to setup and use a redis cluster: http://vimeo.com/63672368 (This video is outdated but could server as a good tutorial/example)

6.7.2 Docker

A fully functional docker image can be found at https://github.com/Grokzen/docker-redis-cluster

See repo *README* for detailed instructions how to setup and run.

6.7.3 Vagrant

A fully functional vagrant box can be found at https://github.com/72squared/vagrant-redis-cluste

See repo *README* for detailed instructions how to setup and run.

6.7.4 Simple makefile

A simple makefile solution can be found at https://github.com/Grokzen/travis-redis-cluster

See repo *README* for detailed instructions how to setup.

6.8 Benchmarks

There is a few benchmarks that is designed to test specific parts of the code that will show how big of a performance difference there is between using this lib and the normal Redis client.

6.8.1 Setup benchmarks

Before running any benchmark you should install this lib in editable mode inside a virtualenv so it can import *StrictRedisCluster* lib.

Install with

```
pip install -e .
```

You also need a few redis servers to test against. It is required to have 1 cluster with atleast one node on port *7001* and it also required to have a non-clustered server on port *7007*.

6.8.2 Implemented Bencmarks

- *simple.py*, This benchmark can be used to messure a simple *set* and *get* operation chain. It also support running pipelines bu adding the flag *–pipeline*

6.8.3 Run predefined benchmarks

There is a set of predefined benchmarks that can be runned to messure performance drop from using this library.

To run the benchmarks run

```
make benchmark
```

Example output and comparison of different runmodes

CHAPTER 7

The Community Guide

7.1 Project status

If you have a problem with the code or general questions about this lib, you can ping me inside the gitter channel that you can find here https://gitter.im/Grokzen/redis-py-cluster and i will help you out with problems or usage of this lib.

As of release *1.0.0* this project will be considered stable and usable in production. If you are going to use redis cluster in your project, you should read up on all documentation that you can find in the bottom of this Readme file. It will contain usage examples and descriptions of what is and what is not implemented. It will also describe how and why things work the way they do in this client.

On the topic about porting/moving this code into *redis-py* there is currently work over here https://github.com/andymccurdy/redis-py/pull/604 that will bring cluster support based on this code. But my suggestion is that until that work is completed that you should use this lib.

7.2 Testing

All tests are currently built around a 6 redis server cluster setup (3 masters + 3 slaves). One server must be using port 7000 for redis cluster discovery.

The easiest way to setup a cluster is to use either a Docker or Vagrant. They are both described in [Setup a redis cluster. Manually, Docker & Vagrant](docs/Cluster_Setup.md).

7.2.1 Tox

To run all tests in all supported environments with *tox* read this [Tox multienv testing](docs/Tox.md)

7.3 Upgrading redis-py-cluster

This document describes what must be done when upgrading between different versions to ensure that code still works.

7.3.1 1.2.0 –> Next release

Class RedisClusterMgt has been removed. You should use the *CLUSTER ...* methods that exists in the *StrictRedisCluster* client class.

Method *cluster_delslots* changed argument specification from *self, node_id, *slots* to *self, *slots* and changed the behaviour of the method to now automatically determine the slot_id based on the current cluster structure and where each slot that you want to delete is loated.

Method pfcount no longer has custom logic and exceptions to prevent CROSSSLOT errors. If method is used with different slots then a regular CROSSSLOT error (rediscluster.exceptions.ClusterCrossSlotError) will be returned.

7.3.2 1.1.0 –> 1.2.0

Discontinue passing *pipeline_use_threads* flag to *rediscluster.StrictRedisCluster* or *rediscluster.RedisCluster*.

Also discontinue passing *use_threads* flag to the pipeline() method.

In 1.1.0 and prior, you could use *pipeline_use_threads* flag to tell the client to perform queries to the different nodes in parallel via threads. We exposed this as a flag because using threads might have been risky and we wanted people to be able to disable it if needed.

With this release we figured out how to get parallelization of the commands without the need for threads. We write to all the nodes before reading from them, essentially multiplexing the connections (but without the need for complicated socket multiplexing). We found this approach to be faster and more scalable as more nodes are added to the cluster.

That means we don't need the *pipeline_use_threads* flag anymore, or the *use_threads* flag that could be passed into the instantiation of the pipeline object itself.

The logic is greatly simplified and the default behavior will now come with a performance boost and no need to use threads.

Publish and subscribe no longer connects to a single instance. It now hashes the channel name and uses that to determine what node to connect to. More work will be done in the future when *redis-server* improves the pubsub implementation. Please read up on the documentation about pubsub in the *docs/pubsub.md* file about the problems and limitations on using a pubsub in a cluster.

Commands Publish and Subscribe now uses the same connections as any other commands. If you are using any pubsub commands you need to test it through thoroughly to ensure that your implementation still works.

To use less strict cluster slots discovery you can add the following config to your redis-server config file "cluster-require-full-coverage=no" and this client will honour that setting and not fail if not all slots is covered.

A bug was fixed in 'sdiffstore', if you are using this, verify that your code still works as expected.

Class RedisClusterMgt is now deprecated and will be removed in next release in favor of all cluster commands implemented in the client in this release.

7.3.3 1.0.0 –> 1.1.0

The following exceptions have been changed/added and code that use this client might have to be updated to handle the new classes.

raise RedisClusterException("Too many Cluster redirections") have been changed to *raise ClusterError('TTL exhausted.')*

ClusterDownException have been replaced with *ClusterDownError*

Added new *AskError* exception class.

Added new *TryAgainError* exception class.

Added new *MovedError* exception class.

Added new *ClusterCrossSlotError* exception class.

Added optional *max_connections_per_node* parameter to *ClusterConnectionPool* which changes behavior of *max_connections* so that it applies per-node rather than across the whole cluster. The new feature is opt-in, and the existing default behavior is unchanged. Users are recommended to opt-in as the feature fixes two important problems. First is that some nodes could be starved for connections after max_connections is used up by connecting to other nodes. Second is that the asymmetric number of connections across nodes makes it challenging to configure file descriptor and redis max client settings.

Reinitialize on *MOVED* errors will not run on every error but instead on every 25 error to avoid excessive cluster reinitialize when used in multiple threads and resharding at the same time. If you want to go back to the old behaviour with reinitialize on every error you should pass in *reinitialize_steps=1* to the client constructor. If you want to increase or decrease the intervall of this new behaviour you should set *reinitialize_steps* in the client constructor to a value that you want.

Pipelines in general have recieved alot of attention so if you are using pipelines in your code, ensure that you test the new code out alot before using it to make sure it still works as you expect.

The entire client code should now be safer to use in a threaded environment. Some race conditions was found and have now been fixed and it should prevent the code from behaving wierd during reshard operations.

7.3.4 0.2.0 –> 0.3.0

In *0.3.0* release the name of the client class was changed from *RedisCluster* to *StrictRedisCluster* and a new implementation of *RedisCluster* was added that is based on *redis.Redis* class. This was done to enable implementation a cluster enabled version of *redis.Redis* class.

Because of this all imports and usage of *RedisCluster* must be changed to *StrictRedisCluster* so that existing code will remain working. If this is not done some issues could arise in existing code.

7.3.5 0.1.0 –> 0.2.0

No major changes was done.

7.4 Release Notes

7.4.1 1.3.0 (Sep 11, 2016)

- Removed RedisClusterMgt class and file
- Fixed a bug when using pipelines with RedisCluster class (Ozahata)
- Bump redis-server during travis tests to 3.0.7
- Added docs about same module name in another python redis cluster project.
- Fix a bug when a connection was to be tracked for a node but the node either do not yet exists or was removed because of resharding was done in another thread. (ashishbaghudana)
- Fixed a bug with "CLUSTER ..." commands when a node_id argument was needed and the return type was supposed to be converted to bool with bool_ok in redis._compat.
- Add back gitter chat room link

- Add new client commands - cluster_reset_all_nodes
- Command cluster_delslots now determines what cluster shard each slot is on and sends each slot deletion command to the correct node. Command have changed argument spec (Read Upgrading.rst for details)
- Fixed a bug when hashing the key it if was a python 3 byte string and it would cause it to route to wrong slot in the cluster (fossilet, Grokzen)
- Fixed a bug when reinitialize the nodemanager it would use the old nodes_cache instead of the new one that was just parsed (monklof)

7.4.2 1.2.0 (Apr 09, 2016)

- Drop maintained support for python 3.2.
- Remove Vagrant file in favor for repo maintained by 72squared
- Add Support for password protected cluster (etng)
- Removed assertion from code (gmolight)
- Fixed a bug where a regular connection pool was allocated with each StrictRedisCluster instance.
- Rework pfcount to now work as expected when all arguments points to same hashslot
- New code and important changes from redis-py 2.10.5 have been added to the codebase.
- Removed the need for threads inside of pipeline. We write the packed commands all nodes before reading the responses which gives us even better performance than threads, especially as we add more nodes to the cluster.
- Allow passing in a custom connection pool
- Provide default max_connections value for ClusterConnectionPool *(2**31)*
- Travis now tests both redis 3.0.x and 3.2.x
- Add simple ptpdb debug script to make it easier to test the client
- Fix a bug in sdiffstore (mt3925)
- Fix a bug with scan_iter where duplicate keys would be returned during itteration
- Implement all "CLUSTER ..." commands as methods in the client class
- Client now follows the service side setting 'cluster-require-full-coverage=yes/no' (baranbartu)
- Change the pubsub implementation (PUBLISH/SUBSCRIBE commands) from using one single node to now determine the hashslot for the channel name and use that to connect to a node in the cluster. Other clients that do not use this pattern will not be fully compatible with this client. Known limitations is pattern subscription that do not work properly because a pattern can't know all the possible channel names in advance.
- Convert all docs to ReadTheDocs
- Rework connection pool logic to be more similar to redis-py. This also fixes an issue with pubsub and that connections was never release back to the pool of available connections.

7.4.3 1.1.0 (Oct 27, 2015)

- Refactored exception handling and exception classes.
- Added READONLY mode support, scales reads using slave nodes.
- Fix __repr__ for ClusterConnectionPool and ClusterReadOnlyConnectionPool

- Add max_connections_per_node parameter to ClusterConnectionPool so that max_connections parameter is calculated per-node rather than across the whole cluster.
- Improve thread safty of get_connection_by_slot and get_connection_by_node methods (iandyh)
- Improved error handling when sending commands to all nodes, e.g. info. Now the connection takes retry_on_timeout as an option and retry once when there is a timeout. (iandyh)
- Added support for SCRIPT LOAD, SCRIPT FLUSH, SCRIPT EXISTS and EVALSHA commands. (alisaifee)
- Improve thread safety to avoid exceptions when running one client object inside multiple threads and doing resharding of the cluster at the same time.
- Fix ASKING error handling so now it really sends ASKING to next node during a reshard operation. This improvement was also made to pipelined commands.
- Improved thread safety in pipelined commands, along better explanation of the logic inside pipelining with code comments.

7.4.4 1.0.0 (Jun 10, 2015)

- No change to anything just a bump to 1.0.0 because the lib is now considered stable/production ready.

7.4.5 0.3.0 (Jun 9, 2015)

- simple benchmark now uses docopt for cli parsing
- New make target to run some benchmarks 'make benchmark'
- simple benchmark now support pipelines tests
- Renamed RedisCluster –> StrictRedisCluster
- Implement backwards compatible redis.Redis class in cluster mode. It was named RedisCluster and everyone updating from 0.2.0 to 0.3.0 should consult docs/Upgrading.md for instructions how to change your code.
- Added comprehensive documentation regarding pipelines
- Meta retrieval commands(slots, nodes, info) for Redis Cluster. (iandyh)

7.4.6 0.2.0 (Dec 26, 2014)

- Moved pipeline code into new file.
- Code now uses a proper cluster connection pool class that handles all nodes and connections similar to how redis-py do.
- Better support for pubsub. All clients will now talk to the same server because pubsub commands do not work reliably if it talks to a random server in the cluster.
- Better result callbacks and node routing support. No more ugly decorators.
- Fix keyslot command when using non ascii characters.
- Add bitpos support, redis-py 2.10.2 or higher required.
- Fixed a bug where vagrant users could not build the package via shared folder.
- Better support for CLUSTERDOWN error. (Neuront)
- Parallel pipeline execution using threads. (72squared)

- Added vagrant support for testing and development. (72squared)
- Improve stability of client during resharding operations (72squared)

7.4.7 0.1.0 (Sep 29, 2014)

- Initial release
- First release uploaded to pypi

7.5 Project Authors

Added in the order they contributed.

If you are mentioned in this document and want your row changed for any reason, open a new PR with changes.

Lead author and maintainer: Grokzen - https://github.com/Grokzen

Authors who contributed code or testing:

- Dobrite - https://github.com/dobrite
- 72squared - https://github.com/72squared
- Neuron Teckid - https://github.com/neuront
- iandyh - https://github.com/iandyh
- mumumu - https://github.com/mumumu
- awestendorf - https://github.com/awestendorf
- Ali-Akber Saifee - https://github.com/alisaifee
- etng - https://github.com/etng
- gmolight - https://github.com/gmolight
- baranbartu - https://github.com/baranbartu
- monklof - https://github.com/monklof

7.6 Licensing

The license should be the same as redis-py (https://github.com/andymccurdy/redis-py)

7.7 Disclaimer

Both Redis cluster and redis-py-cluster is considered stable and production ready.

But this depends on what you are going to use clustering for. In the simple use cases with SET/GET and other single key functions there is not issues. If you require multi key functinoality or pipelines then you must be very careful when developing because they work slightly different from the normal redis server.

If you require advance features like pubsub or scripting, this lib and redis do not handle that kind of use-cases very well. You either need to develop a custom solution yourself or use a non clustered redis server for that.

Finally, this lib itself is very stable and i know of atleast 2 companies that use this in production with high loads and big cluster sizes.