RedisMPX

Release 0.5.2

Loris Cro

Apr 03, 2020

CONTENTS:

CHAPTER

ONE

ABSTRACT

When bridging multiple application instances through Redis Pub/Sub it's easy to end up needing support for multiplexing. RedisMPX streamlines this process in a consistent way across multiple languages by offering a consistent set of features that cover the most common use cases.

The library works under the assumption that you are going to create separate subscriptions for each client connected to your service (e.g. WebSockets clients):

- ChannelSubscription allows you to add and remove individual Redis PubSub channels similarly to how a multi-room chat application would need to.
- PatternSubscription allows you to subscribe to a single Redis Pub/Sub pattern.
- PromiseSubscription allows you to create a networked promise system.

CHAPTER

TWO

INSTALLATION

Requires Python 3.7+, based on aio-libs/aioredis, an AsyncIO Redis client:

```
pip install redismpx
```

CHAPTER

THREE

FEATURES

- Simple channel subscriptions
- Pattern subscriptions
- Networked promise system
- Automatic reconnection with exponetial backoff + jitter

CHAPTER

FOUR

CLASSES

class redismpx.**Multiplexer**(**args*, ***kwargs*)

A Multiplexer instance corresponds to one Redis Pub/Sub connection that will be shared by multiple subscription instances.

Multiplexer accepts the same connection options that you can specify with `aioredis.create_connection()`.

See the documentation of aio-libs/aioredis for more information.

See the code example for an easy to understand explanation of the expected signature for *on_message*, *on_disconnect* and *on_activation*. Also, note that those callbacks should not throw exceptions as any exception will be logged as a warning and then discarded.

If you are making use of Python's type hints, you can import *OnMessage*, *OnDisconnect*, and *OnActivation* from this package.

Usage example:

```
# Pass to Multiplexer the same connection options that
# aioredis.create_connection() would accept.
mpx = Multiplexer('redis://localhost')

# on_message is a callback (can be async)
# that accepts a channel name and a message.
async def my_on_message(channel: bytes, message: bytes):
        await websocket.send(f"ch: {channel} msg: {message}")

# on_disconnect is a callback (can be async)
# that accepts the error that caused the disconnection.
def my_on_disconnect(error: Exception):
        print("oh no!")

# on_activation is a callback (can be async)
# that accepts the name of the channel or pattern
# whose subscription just became active (depends
# on whether it's attached to a ChannelSubscription
# or a PatternSubscription).
def my_on_activation(name: bytes):
        print("activated:", name)

# you can also pass None in place of `on_disconnect`
# and `on_activation` if you're not interested in
# reacting to those events.

# Use `mpx` to create new subscriptions.
```

(continues on next page)

(continued from previous page)

```
channel_sub = mpx.new_channel_subcription(
        my_on_message, my_on_disconnect, None)
pattern_sub = mpx.new_pattern_subscription("hello-*",
        my_on_message, None, my_on_activation)
promise_sub = mpx.new_promise_subscription("hello-")
```

new_channel_subscription(*on_message*, *on_disconnect*, *on_activation*)
Creates a new ChannelSubscription tied to the Multiplexer.

Before disposing of a ChannelSubscription you must call its *close()* method.

The arguments *on_disconnect* and *on_activation* can be *None* if you're not interested in the corresponding types of event.

Parameters

- **on_message** (Callable[[bytes, bytes], Optional[Awaitable[None]]]) – a (async or non) function that gets called for every message recevied.
- **on_disconnect** (Optional[Callable[[Exception], Optional[Awaitable[None]]]]) – a (async or non) function that gets called when the connection is lost.
- **on_activation** (Optional[Callable[[bytes], Optional[Awaitable[None]]]]) – a (async or non) function that gets called when a subscription goes into effect.

Return type ChannelSubscription

new_pattern_subscription(*pattern*, *on_message*, *on_disconnect*, *on_activation*)
Creates a new PatternSubscription tied to the Multiplexer.

Before disposing of a PatternSubscription you must call its *close()* method.

The arguments *on_disconnect* and *on_activation* can be *None* if you're not interested in the corresponding types of event.

Parameters

- **pattern** (Union[str, bytes]) – the Redis Pub/Sub pattern to subscribe to.
- **on_message** (Callable[[bytes, bytes], Optional[Awaitable[None]]]) – a (async or non) function that gets called for every message recevied.
- **on_disconnect** (Optional[Callable[[Exception], Optional[Awaitable[None]]]]) – a (async or non) function that gets called when the connection is lost.
- **on_activation** (Optional[Callable[[bytes], Optional[Awaitable[None]]]]) – a (async or non) function that gets called when a subscription goes into effect.

Return type PatternSubscription

new_promise_subscription(*prefix*)
Creates a new PromiseSubscription tied to the Multiplexer.

Before disposing of a PromiseSubscription you must call its *close()* method.

The prefix argument is used to create internally a PatternSubscription that will match all channels that start with the provided prefix.

A Promise represents a timed, uninterrupted, single-message subscription to a Redis Pub/Sub channel. If network connectivity gets lost, thus causing an interruption, the Promise will be failed (unless already fullfilled). Use NewPromise from PromiseSubscription to create a new Promise.

Parameters **prefix** (Union[str, bytes]) – the prefix under which all Promises will be created under.

Return type PromiseSubscription

class redismpx.**ChannelSubscription**(*multiplexer*, *on_message*, *on_disconnect*, *on_activation*)

A ChannelSubscription ties a on_message callback to zero or more Redis Pub/Sub channels. Use *new_channel_subscription()* to create a new ChannelSubscription.

Usage example:

```
# When created, a ChannelSubscription is empty.
channel_sub = mpx.new_channel_subcription(
        my_on_message, my_on_disconnect, None)

# You can then add more channels to the subscription.
channel_sub.add("hello-world")
channel_sub.add("banana")

# and remove them
channel_sub.remove("banana")
```

add(*channel*)

Adds a new Pub/Sub channel to the subscription.

Parameters **channel** (Union[str, bytes]) – a Redis Pub/Sub channel

Return type None

clear()

Removes all channels from the subscription

Return type None

close()

Closes the subscription.

Return type None

remove(*channel*)

Removes a Redis Pub/Sub channel from the subscription.

Parameters **channel** (Union[str, bytes]) – a Redis Pub/Sub channel

Return type None

class redismpx.**PatternSubscription**(*multiplexer*, *pattern*, *on_message*, *on_disconnect*, *on_activation*)

A PatternSubscription ties a on_message callback to one Redis Pub/Sub pattern. Use *new_pattern_subscription()* to create a new PatternSubscription.

Usage example:

```
# This subscription will receive all messages sent to
# channels that start with "red", like `redis` and `reddit`.
pattern_sub = mpx.new_pattern_subscription("red*",
        my_on_message, my_on_disconnect, my_on_activation)
```

(continues on next page)

(continued from previous page)

```
# Once created, a PatternSubscription can only be closed.
pattern_sub.close()
```

close()
Closes the subscription.

Return type None

class redismpx.**PromiseSubscription**(*multiplexer*, *prefix*)

A PromiseSubscription allows you to wait for individual Redis Pub/Sub messages with support for timeouts. This effectively creates a networked promise system.

It makes use of a PatternSubscription internally to make creating new promises as lightweight as possible (no subscribe/unsubscribe command is sent to Redis to fullfill or expire a Promise). Consider always calling *wait_for_activation()* after creating a new PromiseSubscription.

Use *new_promise_subscription()* to create a new PromiseSubscription.

Usage example:

```
# This subscription will allow you to produce promises
# under the `hello-` prefix.
promise_sub = mpx.new_promise_subscription("hello-")

# Wait for the subscription to become active.
await promise_sub.wait_for_activation()

# Create a promise with a timeout of 10s.
p = promise_sub.new_promise("world", 10)

# Publish a message in Redis Pub/Sub using redis-cli
# > PUBLISH hello-world "success!"

# Obtain the result.
print(await p)

# prints "success!"
```

clear()
Cancels all outstanding promises

Return type None

close()
Closes the subscription and cancels all outstanding promises.

Return type None

new_promise(*suffix*, *timeout*)
Creates a new Promise for the given suffix. The suffix gets composed with the prefix specified when creating the PromiseSubscription to create the final Redis Pub/Sub channel name. The underlying PatternSubscription will receive all messages sent under the given prefix, thus ensuring that new promises get into effect as soon as this method returns. Trying to create a new Promise while the PromiseSubscription is not active will cause this method to throw *InactiveSubscription*.

A promise that expires will throw a *asyncio.TimeoutError*.

Parameters

- **suffix** (Union[str, bytes]) – the suffix that will be appended to the subscription's prefix
- **timeout** (Union[int, float, None]) – a timeout for the promise expressed in seconds

Return type Awaitable[bytes]

Returns The message received from Pub/Sub.

async wait_for_activation()
Blocks until the subscription becomes active.

Closing the subscription will cause this method to throw SubscriptionIsClosed.

Return type Awaitable[None]

async wait_for_new_promise(*prefix*)
Like *new_promise()* but waits for the subscription to become active instead of throwing *InactiveSubscription*.

Closing the subscription will cause this method to throw SubscriptionIsClosed.

Return type Awaitable[bytes]

exception redismpx.**InactiveSubscription**

PYTHON MODULE INDEX

r

INDEX