

Redlock - A ruby distributed lock using redis.

Distributed locks are a very useful primitive in many environments where different processes require to operate with shared resources in a mutually exclusive way.

There are a number of libraries and blog posts describing how to implement a DLM (Distributed Lock Manager) with Redis, but every library uses a different approach, and many use a simple approach with lower guarantees compared to what can be achieved with slightly more complex designs.

This is an implementation of a proposed distributed lock algorithm with Redis. It started as a fork from antirez implementation.

Compatibility

- It works with Redis server versions 6.0 or later.
- Redlock ≥ 2.0 only works with [RedisClient](#) client instance.

Installation

Add this line to your application's Gemfile:

```
1 gem 'redlock'
```

And then execute:

```
1 $ bundle
```

Or install it yourself as:

```
1 $ gem install redlock
```

Documentation

RubyDoc

Usage example

Acquiring a lock

NOTE: All expiration durations are in milliseconds.

```
1  # Locking
2  lock_manager = Redlock::Client.new([ "redis://127.0.0.1:7777", "redis://127.0.0.1:7778", "redis://127.0.0.1:7779" ])
3  first_try_lock_info = lock_manager.lock("resource_key", 2000)
4  second_try_lock_info = lock_manager.lock("resource_key", 2000)
5
6  p first_try_lock_info
7  # => {validity: 1987, resource: "resource_key", value: "generated_uuid4"}
8
9  p second_try_lock_info
10 # => false
11
12 # Unlocking
13 lock_manager.unlock(first_try_lock_info)
14
15 second_try_lock_info = lock_manager.lock("resource_key", 2000)
16
17 p second_try_lock_info
18 # => {validity: 1962, resource: "resource_key", value: "generated_uuid5"}
```

There's also a block version that automatically unlocks the lock:

```
1 lock_manager.lock("resource_key", 2000) do |locked|
2   if locked
3     # critical code
4   else
5     # error handling
6   end
7 end
```

There's also a bang version that only executes the block if the lock is successfully acquired, returning the block's value as a result, or raising an exception otherwise. Passing a block is mandatory.

```
1 begin
2   block_result = lock_manager.lock!("resource_key", 2000) do
3     # critical code
4   end
5 rescue Redlock::LockError
6   # error handling
7 end
```

Extending a lock

To extend the life of the lock:

```
1 begin
2   lock_info = lock_manager.lock("resource_key", 2000)
3   while lock_info
4     # Critical code
5
6     # Time up and more work to do? Extend the lock.
7     lock_info = lock_manager.lock("resource key", 3000, extend:
      lock_info)
8   end
9 rescue Redlock::LockError
10  # error handling
11 end
```

The above code will also acquire the lock if the previous lock has expired and the lock is currently free. Keep in mind that this means the lock could have been acquired and released by someone else in the meantime. To only extend the life of the lock if currently locked by yourself, use the `extend_only_if_locked` parameter:

```
1 lock_manager.lock("resource key", 3000, extend: lock_info,
  extend_only_if_locked: true)
```

Querying lock status

You can check if a resource is locked:

```
1 resource = "resource_key"
2 lock_info = lock_manager.lock(resource, 2000)
3 lock_manager.locked?(resource)
4 #=> true
5
6 lock_manager.unlock(lock_info)
7 lock_manager.locked?(resource)
8 #=> false
```

Any caller can call the above method to query the status. If you hold a lock and would like to check if it is valid, you can use the `valid_lock?` method:

```
1 lock_info = lock_manager.lock("resource_key", 2000)
2 lock_manager.valid_lock?(lock_info)
3 #=> true
4
5 lock_manager.unlock(lock_info)
6 lock_manager.valid_lock?(lock_info)
```

```
7 #=> false
```

The above methods **are not safe if you are using this to time critical code**, since they return true if the lock has not expired, even if there's only (for example) 1ms left on the lock. If you want to safely time the lock validity, you can use the `get_remaining_ttl_for_lock` and `get_remaining_ttl_for_resource` methods.

Use `get_remaining_ttl_for_lock` if you hold a lock and want to check the TTL specifically for your lock:

```
1 resource = "resource_key"
2 lock_info = lock_manager.lock(resource, 2000)
3 sleep 1
4
5 lock_manager.get_remaining_ttl_for_lock(lock_info)
6 #=> 986
7
8 lock_manager.unlock(lock_info)
9 lock_manager.get_remaining_ttl_for_lock(lock_info)
10 #=> nil
```

Use `get_remaining_ttl_for_resource` if you do not hold a lock, but want to know the remaining TTL on a locked resource:

```
1 # Some part of the code
2 resource = "resource_key"
3 lock_info = lock_manager.lock(resource, 2000)
4
5 # Some other part of the code
6 lock_manager.locked?(resource)
7 #=> true
8 lock_manager.get_remaining_ttl_for_resource(resource)
9 #=> 1975
10
11 # Sometime later
12 lock_manager.locked?(resource)
13 #=> false
14 lock_manager.get_remaining_ttl_for_resource(resource)
15 #=> nil
```

Redis client configuration

`Redlock::Client` expects URLs, or configurations or Redis objects on initialization. Redis objects should be used for configuring the connection in more detail, i.e. setting username and password.

```
1 servers = [ 'redis://localhost:6379', RedisClient.new(:url => 'redis://
  someotherhost:6379') ]
```

```
2 redlock = Redlock::Client.new(servers)
```

To utilize `Redlock::Client` with sentinels you can pass an instance of `RedisClient` or just a configuration hash as part of the servers array during initialization.

```
1 config = {
2   name: "mymaster",
3   sentinels: [
4     { host: "127.0.0.1", port: 26380 },
5     { host: "127.0.0.1", port: 26381 },
6   ],
7   role: :master
8 }
9 client = RedisClient.sentinel(**config).new_client
10 servers = [ config, client ]
11 redlock = Redlock::Client.new(servers)
```

Redlock supports the same configuration hash as `RedisClient`.

Redlock configuration

It's possible to customize the retry logic providing the following options:

```
1 lock_manager = Redlock::Client.new(
2   servers, {
3     retry_count: 3,
4     retry_delay: 200, # milliseconds
5     retry_jitter: 50, # milliseconds
6     redis_timeout: 0.1 # seconds
7   })
```

It is possible to associate `:retry_delay` option with `Proc` object. It will be called every time, with attempt number as argument, to get delay time value before next retry.

```
1 retry_delay = proc { |attempt_number| 200 * attempt_number ** 2 } #
   delay of 200ms for 1st retry, 800ms for 2nd retry, etc.
2 lock_manager = Redlock::Client.new(servers, retry_delay: retry_delay)
```

For more information you can check documentation.

Run tests

Make sure you have docker installed.

```
1 $ make
```

Disclaimer

This code implements an algorithm which is currently a proposal, it was not formally analyzed. Make sure to understand how it works before using it in your production environments. You can see discussion about this approach at reddit and also the Antirez answers for some critics.

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create a new Pull Request