master 1 branch 0 tags Code

103 commits

example	add Quit function to consumer	6 years ago
.gitignore	add first version of statistics engine	9 years ago
.travis.yml	Update .travis.yml	7 years ago
LICENSE	update copyright	8 years ago
Readme.md	Updated video referring to the speed of the application	9 months ago
benchmark_test.go	Upgrade to redis.v2.	8 years ago
buffered_queue.go	Added support of redis v3 lib	7 years ago
consumer.go	use close	6 years ago
integration_test.go	Added support of redis v3 lib	7 years ago
key_names.go	removed superfluous stats	9 years ago
observer.go	Added support of redis v3 lib	7 years ago
package.go	change package commands to Fail() and Requeue()	9 years ago
queue.go	Added support of redis v3 lib	7 years ago
server.go	remove usage of redisURL since it's not an URL	9 years ago

Readme.md

Note: This project is no longer actively maintained. Please refer to its spiritual successor rmq.

--

redismq

What is this

This is a fast, persistent, atomic message queue implementation that uses redis as its storage engine written in go. It uses atomic list commands to ensure that messages are delivered only once in the right order without being lost by crashing consumers.

Details can be found in the blog post about its initial design: http://big-elephants.com/2013-09/building-a-message-queue-using-redis-in-go/

A second article desribes the performance improvements of the current version: http://big-elephants.com/2013-10/tuning-redismq-how-to-use-redis-in-go/

What it's not

It's not a standalone server that you can use as a message queue, at least not for now. The implementation is done purely client side. All message queue commands are "translated" into redis commands and then executed via a redis client.

If you want to use this with any other language than go you have to translate all of the commands into your language of choice.

How to use it

All most all use cases are either covered in the examples or in the tests.

So the best idea is just to read those and figure it from there. But in any case:

Basics

About

a durable message queue system for go based on redis, see also https://github.com/adjust/rmq

Readme
MIT license
493 stars
88 watching
75 forks

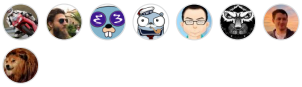
Releases

No releases published

Packages

No packages published

Contributors 8



Languages

Go 100.0%

To get started you need a running redis server. Since the tests run `FlushDB()` an otherwise unused database is highly recommended The first step is to create a new queue:

```
package main

import (
        "fmt"
        "github.com/adjust/redismq"
)

func main() {
        testQueue := redismq.CreateQueue("localhost", "6379", "", 9, "clicks")
        ...
}
```

To write into the queue you simply use `Put()`:

```
        ...
        testQueue := redismq.CreateQueue("localhost", "6379", "", 9, "clicks")
        testQueue.Put("testpayload")
        ...
}
```

The payload can be any kind of string, yes even a 10MB one.

To get messages out of the queue you need a consumer:

```
        ...
        consumer, err := testQueue.AddConsumer("testconsumer")
        if err != nil {
                panic(err)
        }
        package, err := consumer.Get()
        if err != nil {
                panic(err)
        }
        fmt.Println(package.Payload)
        ...
}
```

`Payload` will hold the original string, while `package` will have some additional header information.

To remove a package from the queue you have to `Ack()` it:

```
        ...
        package, err := consumer.Get()
        if err != nil {
                panic(err)
        }
        err = package.Ack()
        if err != nil {
                panic(err)
        }
        ...
}
```

Buffered Queues

When input speed is of the essence `BufferedQueues` will scratch that itch. They pipeline multiple puts into one fast operation. The only issue is that upon crashing or restart the packages in the buffer that haven't been written yet will be lost. So it's advised to wait one second before terminating your program to flush the buffer.

The usage is as easy as it gets:

```
        ...
        bufferSize := 100
        testQueue := redismq.CreateBufferedQueue("localhost", "6379", "", 9, "clicks", bufferSize)
        testQueue.Start()
        ...
}
```

`Put()` and `Get()` stay exactly the same. I have found anything over 200 as `bufferSize` not to increase performance any further.

To ensure that no packages are left in the buffer when you shut down your program you need to call `FlushBuffer()` which will tell the queue to flush the buffer and wait till it's empty.

```
        testQueue.FlushBuffer()
```

Multi Get

Like `BufferedQueues` for `Get()` `MultiGet()` speeds up the fetching of messages. The good news it comes without the buffer loss issues.

Usage is pretty straight forward with the only difference being the `MultiAck()`:

```
        ...
        packages, err := consumer.MultiGet(100)
        if err != nil {
                panic(err)
        }
        for i := range packages {
                fmt.Println(p[i].Payload)
        }
        packages[len(p)-1].MultiAck()
        ...
}
```

`MultiAck()` can be called on any package in the array with all the prior packages being "acked". This way you can `Fail()` single packages.

Reject and Failed Queues

Similar to AMQP redismq supports `Failed Queues` meaning that packages that are rejected by a consumer will be stored in separate queue for further inspection. Alternatively a consumer can also `Requeue()` a package and put it back into the queue:

```
        ...
        package, err := consumer.Get()
        if err != nil {
                panic(err)
        }
        err = package.Requeue()
        if err != nil {
                panic(err)
        }
        ...
}
```

To push the message into the `Failed Queue` of this consumer simply use `Fail()`:

```
        ...
        package, err := consumer.Get()
        if err != nil {
                panic(err)
        }
        err = package.Fail()
        if err != nil {
                panic(err)
        }
        package, err = suite.consumer.GetUnacked()
        ...
}
```

As you can see there is also a command to get messages from the `Failed Queue`.

How fast is it

Even though the original implementation wasn't aiming for high speeds the addition of `BufferedQueues` and `MultiGet` make it go something like this.

All of the following benchmarks were conducted on a MacBook Retina with a 2.4 GHz i7. The InputRate is the number of messages per second that get inserted, WorkRate the messages per second consumed.

Single Publisher, Two Consumers only atomic `Get` and `Put`

```
InputRate:      12183
WorkRate:       12397
```

Single Publisher, Two Consumers using `BufferedQueues` and `MultiGet`

```
InputRate:      46994
WorkRate:       25000
```

And yes that is a persistent message queue that can move over 70k messages per second.

If you want to find out for yourself checkout the `example` folder. The `load.go` or `buffered_queue.go` will start a web server that will display performance stats under `http://localhost:9999/stats`.

How persistent is it

As redis is the underlying storage engine you can set your desired persistence somewhere between YOLO and fsync(). With somewhat sane settings you should see no significant performance decrease.

Copyright

redismq is Copyright © 2014 adjust GmbH.

It is free software, and may be redistributed under the terms specified in the LICENSE file.

© 2023 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About