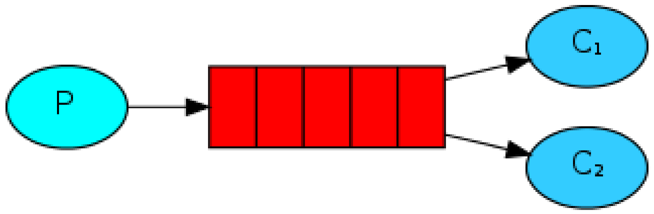
Work Queues

(using Go RabbitMQ client)



Prerequisites

This tutorial assumes RabbitMQ is **installed** and running on `localhost` on standard port (`5672`). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can **contact us** through the mailing list.

In the **first tutorial** we wrote programs to send and receive messages from a named queue. In this one we'll create a *Work Queue* that will be used to distribute time-consuming tasks among multiple workers.

The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a *task* as a message and send it to a queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

Preparation

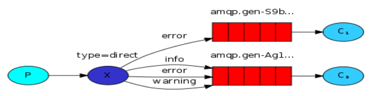
In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the `time.Sleep` function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by `Hello...` will take three seconds.

We will slightly modify the *send.go* code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it `new_task.go`:

```
body := bodyFrom(os.Args)
err = ch.Publish(
  "",           // exchange
  q.Name,       // routing key
  false,        // mandatory
  false,
  amqp.Publishing {
    DeliveryMode: amqp.Persistent,
    ContentType:  "text/plain",
    Body:         []byte(body),
  })
failOnError(err, "Failed to publish a message")
log.Printf(" [x] Sent %s", body)
```

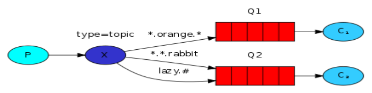
Our old *receive.go* script also requires some changes: it needs to fake a second of work for every dot in the message body. It will pop messages from the queue and perform the task, so let's call it `worker.go`:

```
msgs, err := ch.Consume(
  q.Name, // queue
  "",     // consumer
  true,   // auto-ack
  false,  // exclusive
  false,  // no-local
  false,  // no-wait
  nil,    // args
)
failOnError(err, "Failed to register a consumer")

forever := make(chan bool)

go func() {
  for d := range msgs {
    log.Printf("Received a message: %s", d.Body)
```



5 Topics

Receiving messages based on a pattern



```
      dot_count := bytes.Count(d.Body, []byte("."))
      t := time.Duration(dot_count)
      time.Sleep(t * time.Second)
      log.Printf("Done")
    }
  }()

  log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
  <-forever
```

Note that our fake task simulates execution time.

Run them as in tutorial one:

```
shell1$ go run worker.go
shell2$ go run new_task.go
```

Round-robin dispatching

One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

First, let's try to run two `worker.go` scripts at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the `worker.go` script. These consoles will be our two consumers - C1 and C2.

```
shell1$ go run worker.go
 [*] Waiting for messages. To exit press CTRL+C
```

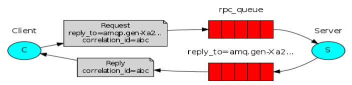
```
shell2$ go run worker.go
 [*] Waiting for messages. To exit press CTRL+C
```

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
shell3$ go run new_task.go First message.
shell3$ go run new_task.go Second message..
shell3$ go run new_task.go Third message...
```

6 RPC

Remote procedure call implementation



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir

```
shell3$ go run new_task.go Fourth message....
shell3$ go run new_task.go Fifth message.....
```

Let's see what is delivered to our workers:

```
shell1$ go run worker.go
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'First message.'
 [x] Received 'Third message...'
 [x] Received 'Fifth message.....'
```

```
shell2$ go run worker.go
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Second message..'
 [x] Received 'Fourth message....'
```

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

Message acknowledgment

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code, once RabbitMQ delivers a message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message *acknowledgments*. An ack(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it.

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

Message acknowledgments are turned off by default. It's time to turn them on using the `false, // auto-ack` option and send a proper acknowledgment from the worker `d.Ack(false)`, once we're done with a task.

```
msgs, err := ch.Consume(
  q.Name, // queue
  "",     // consumer
  false,  // auto-ack
  false,  // exclusive
  false,  // no-local
  false,  // no-wait
  nil,    // args
)
failOnError(err, "Failed to register a consumer")

forever := make(chan bool)

go func() {
  for d := range msgs {
    log.Printf("Received a message: %s", d.Body)
    dot_count := bytes.Count(d.Body, []byte("."))
    t := time.Duration(dot_count)
    time.Sleep(t * time.Second)
    log.Printf("Done")
    d.Ack(false)
  }
}()

log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
<-forever
```

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered.

Forgotten acknowledgment

It's a common mistake to miss the `ack`. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use `rabbitmqctl` to print the `messages_unacknowledged` field:

```
$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
Listing queues ...
hello    0       0
...done.
```

Message durability

We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as *durable*:

```
q, err := ch.QueueDeclare(
  "hello",      // name
  true,         // durable
  false,        // delete when unused
  false,        // exclusive
  false,        // no-wait
  nil,          // arguments
)
failOnError(err, "Failed to declare a queue")
```

Although this command is correct by itself, it won't work in our present setup. That's because we've already defined a queue called `hello` which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example `task_queue`:

```
q, err := ch.QueueDeclare(
  "task_queue", // name
  true,         // durable
  false,        // delete when unused
  false,        // exclusive
  false,        // no-wait
  nil,          // arguments
)
failOnError(err, "Failed to declare a queue")
```

This `durable` option change needs to be applied to both the producer and consumer code.

At this point we're sure that the `task_queue` queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by using the `amqp.Persistent` option `amqp.Publishing` takes.

```
err = ch.Publish(
  "",           // exchange
  q.Name,       // routing key
  false,        // mandatory
  false,
  amqp.Publishing {
    DeliveryMode: amqp.Persistent,
    ContentType:  "text/plain",
    Body:         []byte(body),
  })
```

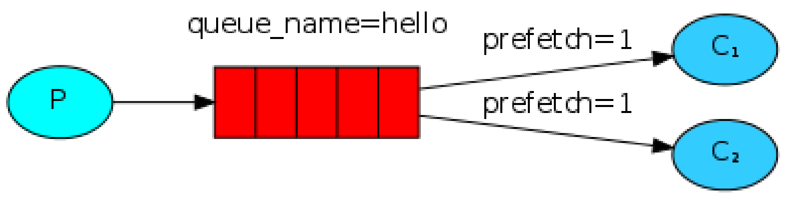
Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do `fsync(2)` for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use **publisher confirms**.

Fair dispatch

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



In order to defeat that we can set the prefetch count with the value of `1`. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```
err = ch.Qos(
  1,     // prefetch count
  0,     // prefetch size
  false, // global
)
failOnError(err, "Failed to set QoS")
```

Note about queue size

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

Putting it all together

Final code of our `new_task.go` class:

```
package main

import (
        "fmt"
        "log"
        "os"
        "strings"

        "github.com/streadway/amqp"
)

func failOnError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
                panic(fmt.Sprintf("%s: %s", msg, err))
        }
}

func main() {
        conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()

        q, err := ch.QueueDeclare(
                "task_queue", // name
                true,         // durable
                false,        // delete when unused
                false,        // exclusive
                false,        // no-wait
                nil,          // arguments
        )
        failOnError(err, "Failed to declare a queue")

        body := bodyFrom(os.Args)
        err = ch.Publish(
                "",           // exchange
                q.Name,       // routing key
                false,        // mandatory
                false,
                amqp.Publishing{
                        DeliveryMode: amqp.Persistent,
                        ContentType:  "text/plain",
                        Body:         []byte(body),
                })
        failOnError(err, "Failed to publish a message")
        log.Printf(" [x] Sent %s", body)
}

func bodyFrom(args []string) string {
        var s string
        if (len(args) < 2) || os.Args[1] == "" {
                s = "hello"
        } else {
                s = strings.Join(args[1:], " ")
        }
        return s
}
```

(new_task.go source)

And our `worker.go`:

```
package main

import (
        "bytes"
        "fmt"
        "github.com/streadway/amqp"
        "log"
```

```go
        "time"
)

func failOnError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
                panic(fmt.Sprintf("%s: %s", msg, err))
        }
}

func main() {
        conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()

        q, err := ch.QueueDeclare(
                "task_queue", // name
                true,         // durable
                false,        // delete when unused
                false,        // exclusive
                false,        // no-wait
                nil,          // arguments
        )
        failOnError(err, "Failed to declare a queue")

        err = ch.Qos(
                1,     // prefetch count
                0,     // prefetch size
                false, // global
        )
        failOnError(err, "Failed to set QoS")

        msgs, err := ch.Consume(
                q.Name, // queue
                "",     // consumer
                false,  // auto-ack
                false,  // exclusive
                false,  // no-local
                false,  // no-wait
                nil,    // args
        )
        failOnError(err, "Failed to register a consumer")

        forever := make(chan bool)

        go func() {
                for d := range msgs {
                        log.Printf("Received a message: %s", d.Body)
                        d.Ack(false)
                        dot_count := bytes.Count(d.Body, []byte("."))
                        t := time.Duration(dot_count)
                        time.Sleep(t * time.Second)
                        log.Printf("Done")
                }
        }()

        log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
        <-forever
}
```

(worker.go source)

Using message acknowledgments and prefetch count you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.

For more information on `amqp.Channel` methods and message properties, you can browse the **amqp API reference**.

Now we can move on to **tutorial 3** and learn how to deliver the same message to many consumers.

Sitemap | Contact | This Site is Open Source | Pivotal is Hiring