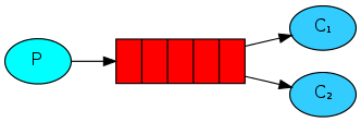
1 "Hello World!"

The simplest thing that does *something*

Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

2 Work queues

Distributing tasks among workers



Python
Java

Introduction

RabbitMQ is a message broker. The principal idea is pretty simple: it accepts and forwards messages. You can think about it as a post office: when you send mail to the post box you're pretty sure that Mr. Postman will eventually deliver the mail to your recipient. Using this metaphor RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is the fact that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data – *messages*.

RabbitMQ, and messaging in general, uses some jargon.

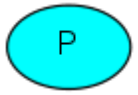
Prerequisites

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.

Producing means nothing more than sending. A program that sends messages is a *producer*. We'll draw it like that, with "P":

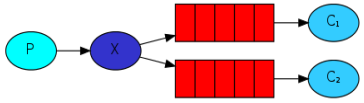


A queue is the name for a mailbox. It lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can be stored only inside a *queue*. A *queue* is not bound by any limits, it can store as many messages as you like – it's essentially an infinite buffer. Many *producers* can send messages that go to one queue, many *consumers* can try to receive data from one *queue*. A queue will be drawn as like that, with its name above it:

Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

3 Publish/Subscribe

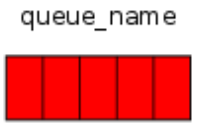
Sending messages to many consumers at once



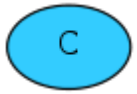
Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

4 Routing

Receiving messages selectively



Consuming has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages. On our drawings it's shown with "C":



Note that the producer, consumer, and broker do not have to reside on the same machine; indeed in most applications they don't.

"Hello World"

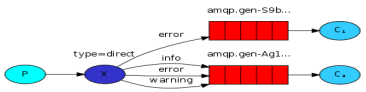
(using Go RabbitMQ client)

In this part of the tutorial we'll write two small programs in Go; a producer that sends a single message, and a consumer that receives messages and prints them out. We'll gloss over some of the detail in the **Go RabbitMQ** API, concentrating on this very simple thing just to get started. It's a "Hello World" of messaging.

In the diagram below, "P" is our producer and "C" is our consumer. The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.

The Go RabbitMQ client library

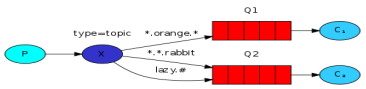
RabbitMQ speaks multiple protocols. This tutorial uses AMQP 0-9-1, which is an open, general-purpose protocol for messaging. There are a number of clients for RabbitMQ in **many different languages**. We'll use the Go amqp client in this tutorial.



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

5 Topics

Receiving messages based on a pattern



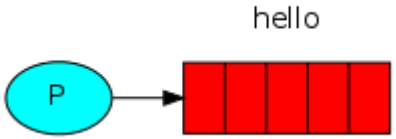
Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

First, install amqp using `go get`:

```
$ go get github.com/streadway/amqp
```

Now we have amqp installed, we can write some code.

Sending



We'll call our message sender `send.go` and our message receiver `receive.go`. The sender will connect to RabbitMQ, send a single message, then exit.

In **`send.go`**, we need to import the library first:

```
package main

import (
  "fmt"
  "log"

  "github.com/streadway/amqp"
)
```

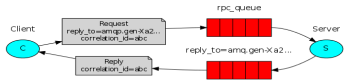
We also need an helper function to check the return value for each amqp call:

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

then connect to RabbitMQ server

6 RPC

Remote procedure call implementation



Python

Java

Ruby

PHP

C#

Javascript

Go

Elixir

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

The connection abstracts the socket connection, and takes care of protocol version negotiation and authentication and so on for us. Next we create a channel, which is where most of the API for getting things done resides:

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

To send, we must declare a queue for us to send to; then we can publish a message to the queue:

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

body := "hello"
err = ch.Publish(
  "",     // exchange
  q.Name, // routing key
  false,  // mandatory
  false,  // immediate
  amqp.Publishing {
    ContentType: "text/plain",
    Body:        []byte(body),
  })
failOnError(err, "Failed to publish a message")
```

Declaring a queue is idempotent - it will only be created if it doesn't exist already. The message content is a byte array, so you can encode whatever you like there.

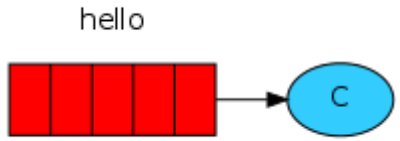
Here's the whole send.go script.

Sending doesn't work!

If this is your first time using RabbitMQ and you don't see the "Sent" message then you may be left scratching your head wondering what could be wrong. Maybe the broker was started without enough free disk space (by default it needs at least 1Gb free) and is therefore refusing to accept messages. Check the broker logfile to confirm and reduce the limit if necessary. The **configuration file documentation** will show you how to set `disk_free_limit`.

Receiving

That's it for our sender. Our receiver is pushed messages from RabbitMQ, so unlike the sender which publishes a single message, we'll keep it running to listen for messages and print them out.



The code (in **`receive.go`**) has the same import and helper function as `send`:

```
package main

import (
  "fmt"
  "log"

  "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))
```

```
  }
}
```

Setting up is the same as the sender; we open a connection and a channel, and declare the queue from which we're going to consume. Note this matches up with the queue that `send` publishes to.

```
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(
  "hello", // name
  false,   // durable
  false,   // delete when usused
  false,   // exclusive
  false,   // no-wait
  nil,     // arguments
)
failOnError(err, "Failed to declare a queue")
```

Note that we declare the queue here, as well. Because we might start the receiver before the sender, we want to make sure the queue exists before we try to consume messages from it.

We're about to tell the server to deliver us the messages from the queue. Since it will push us messages asynchronously, we will read the messages from a channel (returned by `amqp::Consume`) in a goroutine.

```
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)
  }
}()

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

Here's the whole receive.go script.

Putting it all together

Now we can run both scripts. In a terminal, run the sender:

```
$ go run send.go
```

then, run the receiver:

```
$ go run receive.go
```

The receiver will print the message it gets from the sender via RabbitMQ. The receiver will keep running, waiting for messages (Use Ctrl-C to stop it), so try running the sender from another terminal.

If you want to check on the queue, try using `rabbitmqctl list_queues` .

Time to move on to **part 2** and build a simple *work queue*.