Publish/Subscribe

(using Go RabbitMQ client)

In the **previous tutorial** we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

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.

Exchanges

In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

Let's quickly go over what we covered in the previous tutorials:

A *producer* is a user application that sends messages.

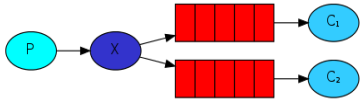
A *queue* is a buffer that stores messages.

A *consumer* is a user application that receives messages.

Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

3 Publish/Subscribe

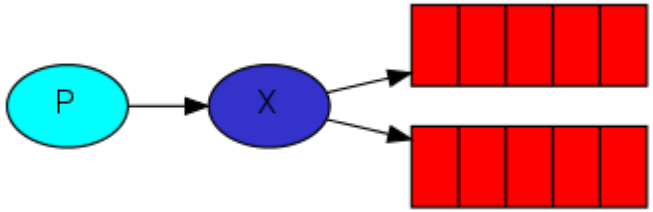
Sending messages to many consumers at once



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

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an *exchange*. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the *exchange type*.



There are a few exchange types available: `direct`, `topic`, `headers` and `fanout`. We'll focus on the last one -- the fanout. Let's create an exchange of this type, and call it `logs`:

```
err = ch.ExchangeDeclare(
  "logs",   // name
  "fanout", // type
  true,     // durable
  false,    // auto-deleted
  false,    // internal
  false,    // no-wait
  nil,      // arguments
)
```

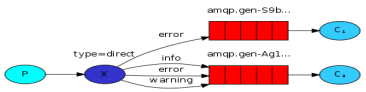
The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

4 Routing

Receiving messages selectively

Listing exchanges

To list the exchanges on the server you can run the ever useful `rabbitmqctl`:



Python

Java

Ruby

PHP

C#

Javascript

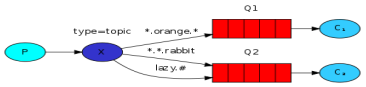
Go

Elixir

Objective-C

5 Topics

Receiving messages based on a pattern



Python

Java

Ruby

PHP

C#

Javascript

Go

Elixir

Objective-C

```
$ sudo rabbitmqctl list_exchanges
Listing exchanges ...
        direct
amq.direct      direct
amq.fanout      fanout
amq.headers     headers
amq.match       headers
amq.rabbitmq.log        topic
amq.rabbitmq.trace      topic
amq.topic       topic
logs    fanout
...done.
```

In this list there are some `amq.*` exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

Nameless exchange

In previous parts of the tutorial we knew nothing about exchanges, but still were able to send messages to queues. That was possible because we were using a default exchange, which is identified by the empty string ("").

Recall how we published a message before:

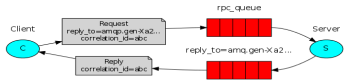
```
err = ch.Publish(
  "",     // exchange
  q.Name, // routing key
  false,  // mandatory
  false,  // immediate
  amqp.Publishing{
    ContentType: "text/plain",
    Body:        []byte(body),
  })
```

Here we use the default or *nameless* exchange: messages are routed to the queue with the name specified by `routing_key` parameter, if it exists.

Now, we can publish to our named exchange instead:

6 RPC

Remote procedure call implementation



Python

Java

Ruby

PHP

C#

Javascript

Go

Elixir

```
err = ch.ExchangeDeclare(
  "logs",   // name
  "fanout", // type
  true,     // durable
  false,    // auto-deleted
  false,    // internal
  false,    // no-wait
  nil,      // arguments
)
failOnError(err, "Failed to declare an exchange")

body := bodyFrom(os.Args)
err = ch.Publish(
  "logs", // exchange
  "",     // routing key
  false,  // mandatory
  false,  // immediate
  amqp.Publishing{
          ContentType: "text/plain",
          Body:        []byte(body),
  })
```

Temporary queues

As you may remember previously we were using queues which had a specified name (remember `hello` and `task_queue`?). Being able to name a queue was crucial for us -- we needed to point the workers to the same queue. Giving a queue a name is important when you want to share the queue between producers and consumers.

But that's not the case for our logger. We want to hear about all log messages, not just a subset of them. We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.

Firstly, whenever we connect to Rabbit we need a fresh, empty queue. To do this we could create a queue with a random name, or, even better - let the server choose a random queue name for us.

Secondly, once we disconnect the consumer the queue should be automatically deleted.

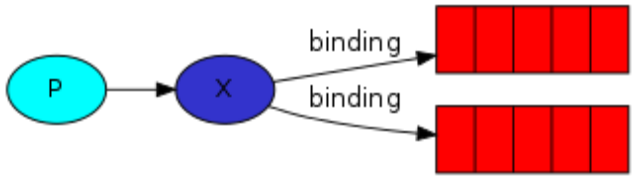
In the **amqp** client, when we supply queue name as an empty string, we create a non-durable queue with a generated name:

```
q, err := ch.QueueDeclare(
  "",    // name
  false, // durable
  false, // delete when usused
  true,  // exclusive
  false, // no-wait
  nil,   // arguments
)
```

When the method returns, the queue instance contains a random queue name generated by RabbitMQ. For example it may look like `amq.gen-JzTY20BRgKO-HjmUJj0wLg` .

When the connection that declared it closes, the queue will be deleted because it is declared as exclusive.

Bindings



We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a *binding*.

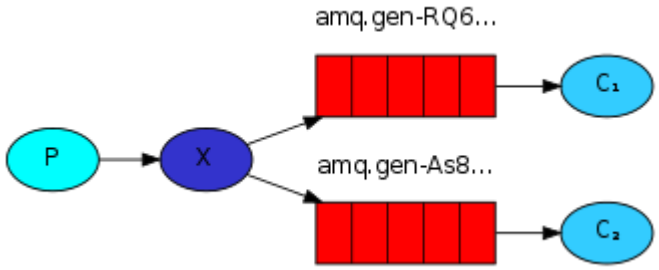
```
err = ch.QueueBind(
  q.Name, // queue name
  "",     // routing key
  "logs", // exchange
  false,
  nil
)
```

From now on the `logs` exchange will append messages to our queue.

Listing bindings

You can list existing bindings using, you guessed it, `rabbitmqctl list_bindings` .

Putting it all together



The producer program, which emits log messages, doesn't look much different from the previous tutorial. The most important change is that we now want to publish messages to our `logs` exchange instead of the nameless one. We need to supply a `routingKey` when sending, but its value is ignored for `fanout` exchanges. Here goes the code for `emit_log.go` script:

```
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()

        err = ch.ExchangeDeclare(
                "logs",   // name
                "fanout", // type
                true,     // durable
                false,    // auto-deleted
                false,    // internal
                false,    // no-wait
                nil,      // arguments
        )
        failOnError(err, "Failed to declare an exchange")

        body := bodyFrom(os.Args)
        err = ch.Publish(
                "logs", // exchange
                "",     // routing key
                false,  // mandatory
                false,  // immediate
                amqp.Publishing{
                        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
}
```

(emit_log.go source)

As you see, after establishing the connection we declared the exchange. This step is neccesary as publishing to a non-existing exchange is forbidden.

The messages will be lost if no queue is bound to the exchange yet, but that's okay for us; if no consumer is listening yet we can safely discard the message.

The code for `receive_logs.go`:

```
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))
        }
}

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()

        err = ch.ExchangeDeclare(
                "logs",   // name
                "fanout", // type
```

```
        true,     // durable
        false,    // auto-deleted
        false,    // internal
        false,    // no-wait
        nil,      // arguments
)
failOnError(err, "Failed to declare an exchange")

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

err = ch.QueueBind(
        q.Name, // queue name
        "",     // routing key
        "logs", // exchange
        false,
        nil)
failOnError(err, "Failed to bind a queue")

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(" [x] %s", d.Body)
            }
        }()

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

(receive_logs.go source)

If you want to save logs to a file, just open a console and type:

```
$ go run receive_logs.go > logs_from_rabbit.log
```

If you wish to see the logs on your screen, spawn a new terminal and run:

```
$ go run receive_logs.go
```

And of course, to emit logs type:

```
$ go run emit_log.go
```

Using `rabbitmqctl list_bindings` you can verify that the code actually creates bindings and queues as we want. With two `receive_logs.go` programs running you should see something like:

```
$ sudo rabbitmqctl list_bindings
Listing bindings ...
logs    exchange        amq.gen-JzTY20BRgKO-HjmUJj0wLg  queue           []
logs    exchange        amq.gen-vso0PVvyiRIL2WoV3i48Yg  queue           []
...done.
```

The interpretation of the result is straightforward: data from exchange `logs` goes to two queues with server-assigned names. And that's exactly what we intended.

To find out how to listen for a subset of messages, let's move on to **tutorial 4**