Topics

(using Go RabbitMQ client)

In the **previous tutorial** we improved our logging system. Instead of using a `fanout` exchange only capable of dummy broadcasting, we used a `direct` one, and gained a possibility of selectively receiving the logs.

Although using the `direct` exchange improved our system, it still has limitations - it can't do routing based on multiple criteria.

In our logging system we might want to subscribe to not only logs based on severity, but also based on the source which emitted the log. You might know this concept from the **`syslog`** unix tool, which routes logs based on both severity (info/warn/crit...) and facility (auth/cron/kern...).

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.

That would give us a lot of flexibility - we may want to listen to just critical errors coming from 'cron' but also all logs from 'kern'.

To implement that in our logging system we need to learn about a more complex `topic` exchange.

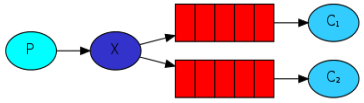
Topic exchange

Messages sent to a `topic` exchange can't have an arbitrary `routing_key` - it must be a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message. A few valid routing key examples: "`stock.usd.nyse`", "`nyse.vmw`", "`quick.orange.rabbit`". There can be as many words in the routing key as you like, up to the limit of 255 bytes.

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

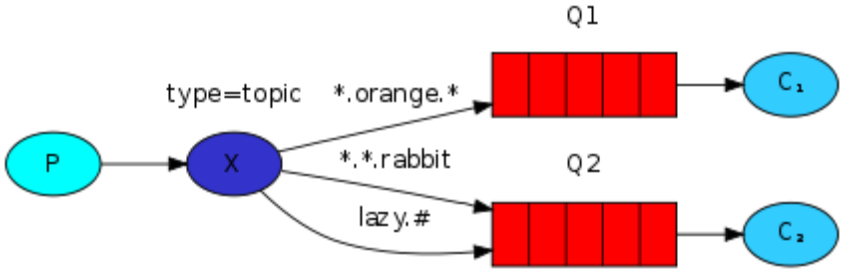
4 Routing

Receiving messages selectively

The binding key must also be in the same form. The logic behind the `topic` exchange is similar to a `direct` one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys:

- `*` (star) can substitute for exactly one word.
- `#` (hash) can substitute for zero or more words.

It's easiest to explain this in an example:



In this example, we're going to send messages which all describe animals. The messages will be sent with a routing key that consists of three words (two dots). The first word in the routing key will describe speed, second a colour and third a species: "`<speed>.<colour>.<species>`".

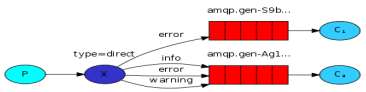
We created three bindings: Q1 is bound with binding key "`*.orange.*`" and Q2 with "`*.*.rabbit`" and "`lazy.#`".

These bindings can be summarised as:

- Q1 is interested in all the orange animals.
- Q2 wants to hear everything about rabbits, and everything about lazy animals.

A message with a routing key set to "`quick.orange.rabbit`" will be delivered to both queues. Message "`lazy.orange.elephant`" also will go to both of them. On the other hand "`quick.orange.fox`" will only go to the first queue, and "`lazy.brown.fox`" only to the second. "`lazy.pink.rabbit`" will be delivered to the second queue only once, even though it matches two bindings. "`quick.brown.fox`" doesn't match any binding so it will be discarded.

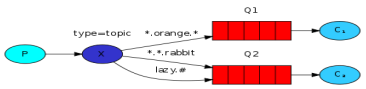
What happens if we break our contract and send a message with one or four words, like "`orange`" or "`quick.orange.male.rabbit`"? Well, these messages won't match any bindings and will be lost.



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

On the other hand "`lazy.orange.male.rabbit`", even though it has four words, will match the last binding and will be delivered to the second queue.

Topic exchange

Topic exchange is powerful and can behave like other exchanges.

When a queue is bound with "`#`" (hash) binding key - it will receive all the messages, regardless of the routing key - like in `fanout` exchange.

When special characters "`*`" (star) and "`#`" (hash) aren't used in bindings, the topic exchange will behave just like a `direct` one.

Putting it all together

We're going to use a `topic` exchange in our logging system. We'll start off with a working assumption that the routing keys of logs will have two words: "`<facility>.<severity>`".

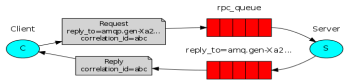
The code is almost the same as in the **previous tutorial**.

The code for `emit_log_topic.go`:

```
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))
```

6 RPC

Remote procedure call implementation



Python

Java

Ruby

PHP

C#

Javascript

Go

Elixir

```
        }
}

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_topic", // name
                "topic",      // 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_topic",          // exchange
                severityFrom(os.Args), // 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) < 3) || os.Args[2] == "" {
```

```
                s = "hello"
        } else {
                s = strings.Join(args[2:], " ")
        }
        return s
}

func severityFrom(args []string) string {
        var s string
        if (len(args) < 2) || os.Args[1] == "" {
                s = "anonymous.info"
        } else {
                s = os.Args[1]
        }
        return s
}
```

The code for `receive_logs_topic.go` :

```
package main

import (
        "fmt"
        "log"
        "os"

        "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_topic", // name
        "topic",      // 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")

if len(os.Args) < 2 {
        log.Printf("Usage: %s [binding_key]...", os.Args[0])
        os.Exit(0)
}
for _, s := range os.Args[1:] {
        log.Printf("Binding queue %s to exchange %s with routing key %s",
                q.Name, "logs_topic", s)
        err = ch.QueueBind(
                q.Name,       // queue name
                s,            // routing key
                "logs_topic", // 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
}
```

To receive all the logs:

```
$ go run receive_logs_topic.go "#"
```

To receive all logs from the facility "`kern`":

```
$ go run receive_logs_topic.go "kern.*"
```

Or if you want to hear only about "`critical`" logs:

```
$ go run receive_logs_topic.go "*.critical"
```

You can create multiple bindings:

```
$ go run receive_logs_topic.go "kern.*" "*.critical"
```

And to emit a log with a routing key "`kern.critical`" type:

```
$ go run emit_log_topic.go "kern.critical" "A critical kernel error"
```

Have fun playing with these programs. Note that the code doesn't make any assumption about the routing or binding keys, you may want to play with more than two routing key parameters.

Some teasers:

Will "*" binding catch a message sent with an empty routing key?

Will "#.*" catch a message with a string ".." as a key? Will it catch a message with a single word key?

How different is "a.*.#" from "a.#"?

(Full source code for **emit_log_topic.go** and **receive_logs_topic.go**)

Next, find out how to do a round trip message as a remote procedure call in **tutorial 6**
