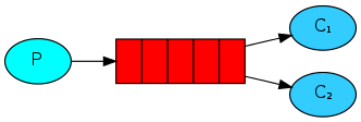
1 "Hello World!"

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

Remote procedure call (RPC)

(using Go RabbitMQ client)

In the **second tutorial** we learned how to use *Work Queues* to distribute time-consuming tasks among multiple workers.

But what if we need to run a function on a remote computer and wait for the result? Well, that's a different story. This pattern is commonly known as *Remote Procedure Call* or *RPC*.

In this tutorial we're going to use RabbitMQ to build an RPC system: a client and a scalable RPC server. As we don't have any time-consuming tasks that are worth distributing, we're going to create a dummy RPC service that returns Fibonacci numbers.

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.

Callback queue

In general doing RPC over RabbitMQ is easy. A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request. We can use the default queue. Let's try it:

```
q, err := ch.QueueDeclare(
  "",    // name
  false, // durable
  false, // delete when usused
  true,  // exclusive
  false, // noWait
  nil,   // arguments
)
```

Ruby

PHP

C#

Javascript

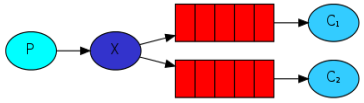
Go

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

```
err = ch.Publish(
  "",          // exchange
  "rpc_queue", // routing key
  false,       // mandatory
  false,       // immediate
  amqp.Publishing{
    ContentType:   "text/plain",
    CorrelationId: corrId,
    ReplyTo:       q.Name,
    Body:          []byte(strconv.Itoa(n)),
  })
```

Message properties

The AMQP protocol predefines a set of 14 properties that go with a message. Most of the properties are rarely used, with the exception of the following:

`persistent`: Marks a message as persistent (with a value of `true`) or transient (`false`). You may remember this property from **the second tutorial**.

`content_type`: Used to describe the mime-type of the encoding. For example for the often used JSON encoding it is a good practice to set this property to: `application/json`.

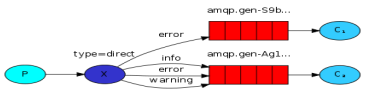
`reply_to`: Commonly used to name a callback queue.

`correlation_id`: Useful to correlate RPC responses with requests.

Correlation Id

In the method presented above we suggest creating a callback queue for every RPC request. That's pretty inefficient, but fortunately there is a better way - let's create a single callback queue per client.

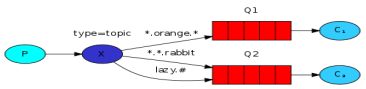
That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the `correlation_id` property is used. We're going to set it to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request.



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

5 Topics

Receiving messages based on a pattern

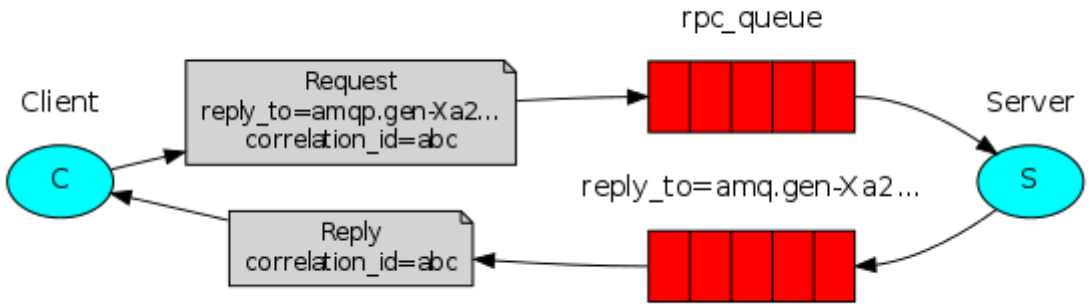


Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

If we see an unknown `correlation_id` value, we may safely discard the message - it doesn't belong to our requests.

You may ask, why should we ignore unknown messages in the callback queue, rather than failing with an error? It's due to a possibility of a race condition on the server side. Although unlikely, it is possible that the RPC server will die just after sending us the answer, but before sending an acknowledgment message for the request. If that happens, the restarted RPC server will process the request again. That's why on the client we must handle the duplicate responses gracefully, and the RPC should ideally be idempotent.

Summary



Our RPC will work like this:

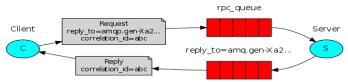
- When the Client starts up, it creates an anonymous exclusive callback queue.
- For an RPC request, the Client sends a message with two properties: `reply_to`, which is set to the callback queue and `correlation_id`, which is set to a unique value for every request.
- The request is sent to an `rpc_queue` queue.
- The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from the `reply_to` field.
- The client waits for data on the callback queue. When a message appears, it checks the `correlation_id` property. If it matches the value from the request it returns the response to the application.

Putting it all together

The Fibonacci function:

6 RPC

Remote procedure call implementation



Python

Java

Ruby

PHP

C#

Javascript

Go

Elixir

```
func fib(n int) int {
        if n == 0 {
                return 0
        } else if n == 1 {
                return 1
        } else {
                return fib(n-1) + fib(n-2)
        }
}
```

We declare our fibonacci function. It assumes only valid positive integer input. (Don't expect this one to work for big numbers, and it's probably the slowest recursive implementation possible).

The code for our RPC server **rpc_server.go** looks like this:

```
package main

import (
        "fmt"
        "log"
        "strconv"

        "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 fib(n int) int {
        if n == 0 {
                return 0
        } else if n == 1 {
                return 1
        } else {
                return fib(n-1) + fib(n-2)
        }
```

```
}

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(
                "rpc_queue", // name
                false,       // durable
                false,       // delete when usused
                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 {
                        n, err := strconv.Atoi(string(d.Body))
                        failOnError(err, "Failed to convert body to integer")

                        log.Printf(" [.] fib(%d)", n)
                        response := fib(n)

                        err = ch.Publish(
                                "",        // exchange
                                d.ReplyTo, // routing key
                                false,     // mandatory
                                false,     // immediate
                                amqp.Publishing{
                                        ContentType:   "text/plain",
                                        CorrelationId: d.CorrelationId,
                                        Body:          []byte(strconv.Itoa(response)),
                                })
                        failOnError(err, "Failed to publish a message")

                        d.Ack(false)
                }
        }()

        log.Printf(" [*] Awaiting RPC requests")
        <-forever
}
```

The server code is rather straightforward:

As usual we start by establishing the connection, channel and declaring the queue.

We might want to run more than one server process. In order to spread the load equally over multiple servers we need to set the `prefetch` setting on channel.

We use `Channel.Consume` to get the go channel where we receive messages from the queue. Then we enter the goroutine where do the work and send the response back.

The code for our RPC client **rpc_client.go**:

```
package main
```

```
import (
        "fmt"
        "log"
        "math/rand"
        "os"
        "strconv"
        "strings"
        "time"

        "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 randomString(l int) string {
        bytes := make([]byte, l)
        for i := 0; i < l; i++ {
                bytes[i] = byte(randInt(65, 90))
        }
        return string(bytes)
}

func randInt(min int, max int) int {
        return min + rand.Intn(max-min)
}

func fibonacciRPC(n int) (res int, err error) {
        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(
```

```
        "",    // name
        false, // durable
        false, // delete when usused
        true,  // exclusive
        false, // noWait
        nil,   // arguments
)
failOnError(err, "Failed to declare 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")

corrId := randomString(32)

err = ch.Publish(
        "",          // exchange
        "rpc_queue", // routing key
        false,       // mandatory
        false,       // immediate
        amqp.Publishing{
                ContentType:   "text/plain",
                CorrelationId: corrId,
                ReplyTo:       q.Name,
                Body:          []byte(strconv.Itoa(n)),
        })
failOnError(err, "Failed to publish a message")

for d := range msgs {
        if corrId == d.CorrelationId {
                res, err = strconv.Atoi(string(d.Body))
                failOnError(err, "Failed to convert body to integer")
                break
        }
```

```
        }

        return
}

func main() {
        rand.Seed(time.Now().UTC().UnixNano())

        n := bodyFrom(os.Args)

        log.Printf(" [x] Requesting fib(%d)", n)
        res, err := fibonacciRPC(n)
        failOnError(err, "Failed to handle RPC request")

        log.Printf(" [.] Got %d", res)
}

func bodyFrom(args []string) int {
        var s string
        if (len(args) < 2) || os.Args[1] == "" {
                s = "30"
        } else {
                s = strings.Join(args[1:], " ")
        }
        n, err := strconv.Atoi(s)
        failOnError(err, "Failed to convert arg to integer")
        return n
}
```

Now is a good time to take a look at our full example source code for **rpc_client.go** and **rpc_server.go**.

Our RPC service is now ready. We can start the server:

```
$ go run rpc_server.go
 [x] Awaiting RPC requests
```

To request a fibonacci number run the client:

```
$ go run rpc_client.go 30
 [x] Requesting fib(30)
```

The design presented here is not the only possible implementation of a RPC service, but it has some important advantages:

If the RPC server is too slow, you can scale up by just running another one. Try running a second `rpc_server.go` in a new console.

On the client side, the RPC requires sending and receiving only one message. As a result the RPC client needs only one network round trip for a single RPC request.

Our code is still pretty simplistic and doesn't try to solve more complex (but important) problems, like:

How should the client react if there are no servers running?

Should a client have some kind of timeout for the RPC?

If the server malfunctions and raises an exception, should it be forwarded to the client?

Protecting against invalid incoming messages (eg checking bounds, type) before processing.

> If you want to experiment, you may find the **rabbitmq-management plugin** useful for viewing the queues.