RabbitMQ by Pivotal

Receiving messages based on a pattern (topics)

Python Java Ruby PHP C# JavaScript Go Elixir Objective-C Swift Spring AMQP

6 RPC

Request/reply pattern example

Python Java Ruby PHP C# JavaScript Go Elixir Spring AMQP

Publish/Subscribe

(using the Java 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.

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.

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

Exchanges

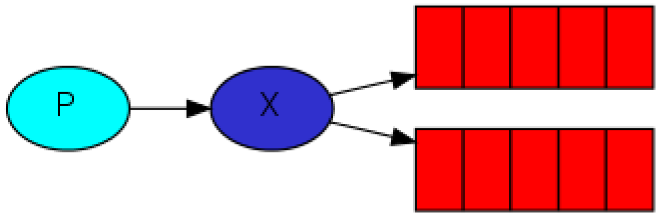
In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

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.

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`:

```
channel.exchangeDeclare("logs", "fanout");
```

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.

Listing exchanges

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

```
sudo rabbitmqctl list_exchanges
```

In this list there will be 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 we identify by the empty string (`""`).

Recall how we published a message before:

```
channel.basicPublish("", "hello", null, message.getBytes());
```

The first parameter is the the name of the exchange. The empty string denotes the default or *nameless* exchange: messages are routed to the queue with the name specified by `routingKey`, if it exists.

Now, we can publish to our named exchange instead:

```
channel.basicPublish( "logs", "", null, message.getBytes());
```

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.

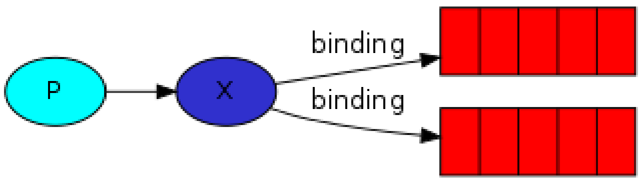
Secondly, once we disconnect the consumer the queue should be automatically deleted.

In the Java client, when we supply no parameters to `queueDeclare()` we create a non-durable, exclusive, autodelete queue with a generated name:

```
String queueName = channel.queueDeclare().getQueue();
```

At that point `queueName` contains a random queue name. For example it may look like `amq.gen-JzTY20BRgKO-HjmUJj0wLg`.

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*.

```
channel.queueBind(queueName, "logs", "");
```

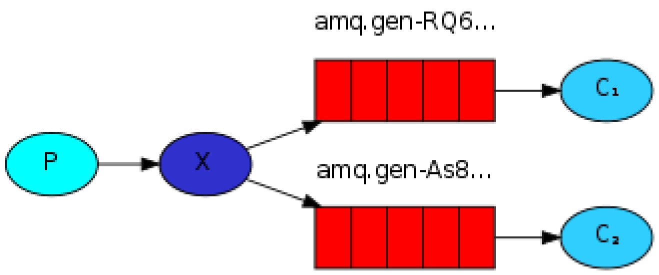
From now on the `logs` exchange will append messages to our queue.

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 `EmitLog.java` program:

```
import java.io.IOException;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

public class EmitLog {

    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv)
                  throws java.io.IOException {

        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");

        String message = getMessage(argv);

        channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());
        System.out.println(" [x] Sent '" + message + "'");

        channel.close();
        connection.close();
    }
    //...
}
```

(EmitLog.java source)

As you see, after establishing the connection we declared the exchange. This step is necessary 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 `ReceiveLogs.java`:

```java
import com.rabbitmq.client.*;

import java.io.IOException;

public class ReceiveLogs {
  private static final String EXCHANGE_NAME = "logs";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
    String queueName = channel.queueDeclare().getQueue();
    channel.queueBind(queueName, EXCHANGE_NAME, "");

    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
      @Override
      public void handleDelivery(String consumerTag, Envelope envelope,
                                 AMQP.BasicProperties properties, byte[] body) throws
IOException {
        String message = new String(body, "UTF-8");
        System.out.println(" [x] Received '" + message + "'");
      }
    };
    channel.basicConsume(queueName, true, consumer);
  }
}
```

(ReceiveLogs.java source)

Compile as before and we're done.

```
javac -cp $CP EmitLog.java ReceiveLogs.java
```

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

```
java -cp $CP ReceiveLogs > logs_from_rabbit.log
```

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

```
java -cp $CP ReceiveLogs
```

And of course, to emit logs type:

```
java -cp $CP EmitLog
```

Using `rabbitmqctl list_bindings` you can verify that the code actually creates bindings and queues as we want. With two `ReceiveLogs.java` 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