RabbitMQ by Pivotal

6 RPC

Request/reply pattern example

Python Java Ruby PHP C# JavaScript Go Elixir Spring AMQP

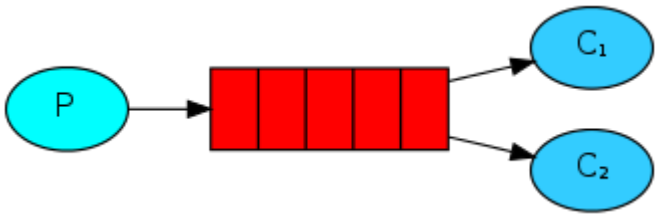
7 Publisher Confirms

Reliable publishing with publisher confirms

Java

Work Queues

(using the Java Client)



In the first tutorial we wrote programs to send and receive messages from a named queue. In this one we'll create a *Work Queue* that will be used to distribute time-consuming tasks among multiple workers.

The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a *task* as a message and send it to a queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

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.

This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

Preparation

In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the `Thread.sleep()` function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by `Hello...` will take three seconds.

We will slightly modify the *Send.java* code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it `NewTask.java`:

```
String message = String.join(" ", argv);

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

Our old *Recv.java* program also requires some changes: it needs to fake a second of work for every dot in the message body. It will handle delivered messages and perform the task, so let's call it `Worker.java`:

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
  String message = new String(delivery.getBody(), "UTF-8");

  System.out.println(" [x] Received '" + message + "'");
  try {
    doWork(message);
  } finally {
    System.out.println(" [x] Done");
  }
};
boolean autoAck = true; // acknowledgment is covered below
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag -> { });
```

Our fake task to simulate execution time:

```
private static void doWork(String task) throws InterruptedException {
```

```
        for (char ch: task.toCharArray()) {
            if (ch == '.') Thread.sleep(1000);
        }
    }
```

Compile them as in tutorial one (with the jar files in the working directory and the environment variable `CP`):

```
javac -cp $CP NewTask.java Worker.java
```

Round-robin dispatching

One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

First, let's try to run two worker instances at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the worker program. These consoles will be our two consumers - C1 and C2.

```
# shell 1
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C

# shell 2
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
```

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
# shell 3
java -cp $CP NewTask First message.
# => [x] Sent 'First message.'
java -cp $CP NewTask Second message..
# => [x] Sent 'Second message..'
java -cp $CP NewTask Third message...
# => [x] Sent 'Third message...'
java -cp $CP NewTask Fourth message....
# => [x] Sent 'Fourth message....'
```

```
# ... [x] Sent 'Fourth message....'
java -cp $CP NewTask Fifth message.....
# => [x] Sent 'Fifth message.....'
```

Let's see what is delivered to our workers:

```
java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'First message.'
# => [x] Received 'Third message...'
# => [x] Received 'Fifth message.....'

java -cp $CP Worker
# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'Second message..'
# => [x] Received 'Fourth message....'
```

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

Message acknowledgment

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code, once RabbitMQ delivers a message to the consumer it immediately marks it for deletion. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message *acknowledgments*. An ack(nowledgement) is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it.

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

Manual message acknowledgments are turned on by default. In previous examples we explicitly turned them off via the `autoAck=true` flag. It's time to set this flag to `false` and send a proper acknowledgment from the worker, once we're done with a task.

```
channel.basicQos(1); // accept only one unack-ed message at a time (see below)

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
  String message = new String(delivery.getBody(), "UTF-8");

  System.out.println(" [x] Received '" + message + "'");
  try {
    doWork(message);
  } finally {
    System.out.println(" [x] Done");
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
  }
};
boolean autoAck = false;
channel.basicConsume(TASK_QUEUE_NAME, autoAck, deliverCallback, consumerTag -> { });
```

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered.

Acknowledgement must be sent on the same channel that received the delivery. Attempts to acknowledge using a different channel will result in a channel-level protocol exception. See the doc guide on confirmations to learn more.

Forgotten acknowledgment

It's a common mistake to miss the `basicAck`. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use `rabbitmqctl` to print the `messages_unacknowledged` field:

```
sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

On Windows, drop the sudo:

```
rabbitmqctl.bat list_queues name messages_ready messages_unacknowledged
```

Message durability

We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as *durable*:

```
boolean durable = true;
channel.queueDeclare("hello", durable, false, false, null);
```

Although this command is correct by itself, it won't work in our present setup. That's because we've already defined a queue called `hello` which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example `task_queue`:

```
boolean durable = true;
channel.queueDeclare("task_queue", durable, false, false, null);
```

This `queueDeclare` change needs to be applied to both the producer and consumer code.

At this point we're sure that the `task_queue` queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by setting `MessageProperties` (which implements `BasicProperties`) to the value `PERSISTENT_TEXT_PLAIN`.

```
import com.rabbitmq.client.MessageProperties;

channel.basicPublish("", "task_queue",
            MessageProperties.PERSISTENT_TEXT_PLAIN,
            message.getBytes());
```

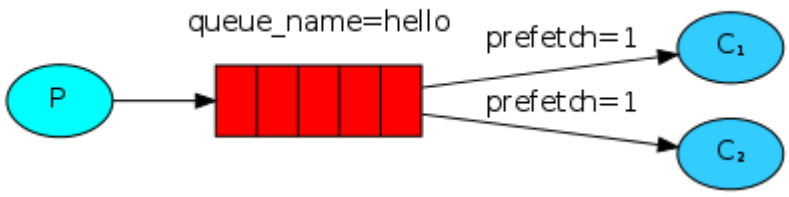
Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do `fsync(2)` for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use publisher confirms.

Fair dispatch

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



In order to defeat that we can use the `basicQos` method with the `prefetchCount = 1` setting. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still

acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```
int prefetchCount = 1;
channel.basicQos(prefetchCount);
```

> **Note about queue size**
>
> If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

Putting it all together

Final code of our `NewTask.java` class:

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.MessageProperties;

public class NewTask {

  private static final String TASK_QUEUE_NAME = "task_queue";

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

        String message = String.join(" ", argv);

        channel.basicPublish("", TASK_QUEUE_NAME,
                MessageProperties.PERSISTENT_TEXT_PLAIN,
                message.getBytes("UTF-8"));
        System.out.println(" [x] Sent '" + message + "'");
```

```
    }
  }

}
```

(NewTask.java source)

And our `Worker.java`:

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

public class Worker {

  private static final String TASK_QUEUE_NAME = "task_queue";

  public static void main(String[] argv) throws Exception {

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

    channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    channel.basicQos(1);

    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");

        System.out.println(" [x] Received '" + message + "'");
        try {
            doWork(message);
        } finally {
            System.out.println(" [x] Done");
            channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
        }
```

```
        };
        channel.basicConsume(TASK_QUEUE_NAME, false, deliverCallback, consumerTag -> { });
    }

    private static void doWork(String task) {
        for (char ch : task.toCharArray()) {
            if (ch == '.') {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException _ignored) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
}
```

(Worker.java source)

Using message acknowledgments and `prefetchCount` you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.

For more information on `Channel` methods and `MessageProperties`, you can browse the JavaDocs online.

Now we can move on to tutorial 3 and learn how to deliver the same message to many consumers.

Production [Non-]Suitability Disclaimer

Please keep in mind that this and other tutorials are, well, tutorials. They demonstrate one new concept at a time and may intentionally oversimplify some things and leave out others. For example topics such as connection management, error handling, connection recovery, concurrency and metric collection are largely omitted for the sake of brevity. Such simplified code should not be considered production ready.

Please take a look at the rest of the documentation before going live with your app. We particularly recommend the following guides: Publisher Confirms and Consumer Acknowledgements, Production Checklist and Monitoring.

Getting Help and Providing Feedback

If you have questions about the contents of this tutorial or any other topic related to RabbitMQ, don't hesitate to ask them on the RabbitMQ mailing list.

Help Us Improve the Docs <3

If you'd like to contribute an improvement to the site, its source is available on GitHub. Simply fork the repository and submit a pull request. Thank you!