RabbitMQ is a message broker: it accepts and forwards messages.

You can think about it as a post office:

In this analogy, RabbitMQ is a post box, a post office and a postman.

It accepts, stores and forwards binary blobs of data - messages.

* Producer: A program that sends messages is a producer.
* Consumer: A program that mostly waits to receive messages.
* Queue: A queue is the name for a post box which lives inside RabbitMQ.
  + Although messages flow through RabbitMQ and your applications, they can only be stored inside a queue.
  + A queue is only bound by the host's memory & disk limits, it's essentially a large message buffer.
  + Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue.

The producer, consumer, and broker may reside on the different host.

Prerequisites

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (5672).

Download the client library amqp-client-4.1.0.jar and its dependencies (SLF4J API and SLF4J Simple).

Copy those files in your working directory.

There are a number of clients for RabbitMQ in many different languages. We'll use the Java client provided by RabbitMQ.

Send.java : The publisher connect to RabbitMQ(Server), send a single message, then exit.

* Name the queue:
  + **private** **final** **static** String ***QUEUE\_NAME*** = "hello";
* Create a connection to the server.

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

* + The connection abstracts the socket connection, and takes care of protocol version negotiation and authentication.
  + Here we connect to a broker on the local machine.
* Create a channel,

Channel channel = connection.createChannel();

* + Channel provides most of the API for getting things done
* To send, we must declare a queue
  + Then we can publish a message to the queue:
  + Declaring a queue is idempotent - it will only be created if it doesn't exist already.
  + The message content is a byte array,

channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);

String message = "Hello World!";

channel.basicPublish("", ***QUEUE\_NAME***, **null**, message.getBytes("UTF-8"));

Receiving

Our consumer is pushed messages from RabbitMQ, so we'll keep it running to listen for messages and print them out.

DefaultConsumer

DefaultConsumer is a class implementing the Consumer interface we'll use to buffer the messages pushed to us by the server.

Since server pushes the messages asynchronously, we provide a callback in the form of an object that will buffer the messages until we're ready to use them. That is what a DefaultConsumer subclass does.

Just like sender

* Open a connection and a channel, and
* Declare the queue from which we're going to consume.
  + Note this matches up with the queue that send publishes to.
  + We declare the queue here, as well. Because we might start the consumer before the publisher, we want to make sure the queue exists before we try to consume messages from it.

**Work Queue**

Work Queue is used to distribute time-consuming tasks among multiple workers.

Idea: 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.

Concepts:

**Message acknowledgment:**

What happens if one of the consumers starts a long task and dies (its channel is closed, connection is closed, or TCP connection is lost) with it only partly done.

Once RabbitMQ delivers a message to the customer it immediately removes it from memory.

So, if you kill a worker you will lose the message it was just processing.

Also, You will also lose all the messages that were dispatched to this particular worker but were not yet handled.

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back from 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 without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it.

Message acknowledgments are turned on by default.

You can turned them off via the autoAck=true flag.

Set this flag to false to send a proper acknowledgment from the worker.

**boolean** autoAck = **false**;

channel.basicConsume(***TASK\_QUEUE\_NAME***, autoAck, consumer);

**Message durability**

If RabbitMQ server stops or crashes it will forget the queues and messages.

Two things are required to make sure that messages aren't lost:

* Mark the queue durable
* Mark 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**);

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

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());

Fair dispatch

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

Not dispatching 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 busy.

use the basicQos method with the prefetchCount = 1

**int** prefetchCount = 1;

channel.basicQos(prefetchCount);

Running:

First run two worker instances at the same time on different console for runnning two consumers (workers).

They will both get messages from the queue, but how exactly? Let's see.

In the third console publish new tasks (execute 5/6 times) by running new tasks 5/6 times.

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.

Exchanges

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Instead, the producer can only send messages to an exchange.

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.

Here are a few exchange types available:

* direct,
* topic,
* headers
* fanout - it broadcasts all the messages it receives to all the queues it knows

Creating an exchange of type fanout with the name logs:

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

previously we were using queues which had a specified name

Giving a queue a name is important when you want to share the queue between producers and consumers.

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.

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();

That relationship between exchange and a queue is called a binding.

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

Bindings can take an extra routingKey parameter.

The meaning of a binding key depends on the exchange type. The fanout exchanges, which we used previously, simply ignored its value.

Routing

we're going to make it possible to subscribe only to a subset of the messages. For example, we will be able to direct only critical error messages to the log file (to save disk space), while still being able to print all of the log messages on the console.

Direct exchange allow filtering messages based on their severity i.e. key.

A message goes to the queues whose binding key exactly matches the routing key of the message.

Remember binding key associated with queue and routing key is associated with the message.