LAB 8: THE PUBLISH-SUBSCRIBE ARCHITECTURE

In this lab, you will develop three Message-Queue-based (MQ) applications using RabitMQ (https://www.rabbitmq.com). This document has been prepared reusing the contents of the official RabbitMQ tutorials. Let's get started by installing all of the required software for this lab:

Software Installation

Please follow the instruction in the following URL to download and install Erlang (required by RabbitMQ server) and the RabbitMQ server itself:

Windows: https://www.rabbitmq.com/install-windows.html

Ubuntu: https://www.rabbitmq.com/install-debian.html

MacOS: https://www.rabbitmq.com/install-homebrew.html

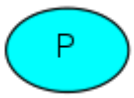
Please click on the **Allow** button when you see the Firewall pop-ups (allowing connection from **Private Network** would be an optimal option).

What is RabbitMQ?

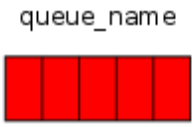
RabbitMQ is a message broker. In essence, it accepts messages from *producers*, and delivers them to *consumers*. In-between, it can route, buffer, and persist the messages according to rules you give it.

RabbitMQ, and messaging in general, uses some jargon.

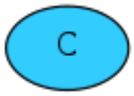
Producing means nothing more than sending. A program that sends messages is a *producer*. We'll represent a producer as follows, with "P":



A queue is the name for a mailbox. It lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can be stored only inside a *queue*. A *queue* is not bound by any limits, it can store as many messages as you like - it's essentially an infinite buffer. Many *producers* can send messages that go to one queue - many *consumers* can try to receive data from one *queue*. A queue will be drawn as follows, with its name above it:



Consuming has a similar meaning to receiving. A *consumer* is a program that mostly waits to receive messages. On our drawings, it's shown with "C":



Note that the producer, consumer, and broker do not have to reside on the same machine; indeed in most applications they don't.

Hello World - Application

In this part of the lab, we'll write two programs in Java; a producer that sends a single message, and a consumer that receives messages and prints them out. We'll gloss over some of the detail in the Java API, concentrating on this very simple thing just to get started. It's a "Hello World" of messaging.

In the diagram below, "P" is our producer and "C" is our consumer. The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.

NOTE: Please do not type in any code until the **Putting it All Together** section**. Please read the code and explanation thoroughly.**

Sending

We'll call our message sender Send and our message receiver Recv. The sender will connect to RabbitMQ, send a single message, then exit.

In **Send.java**, we need some classes imported:

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

Set up the class and name the queue:

```
public class Send {

 private final static String QUEUE_NAME = "hello";

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

then we can create a connection to the server:

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

The connection abstracts the socket connection, and takes care of protocol version negotiation and authentication and so on for us. Here we connect to a broker on the local machine - hence the *localhost*. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.

Next we create a channel, which is where most of the API for getting things done resides.

To send, we must declare a queue for us to send to; then we can publish a message to the queue:

```
  channel.queueDeclare(QUEUE_NAME, false, false, false, null);
  String message = "Hello World!";
  channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
  System.out.println(" [x] Sent '" + message + "'");
```

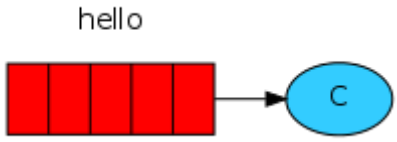
Declaring a queue is idempotent - it will only be created if it doesn't exist already. The message content is a byte array, so you can encode whatever you like there.

Lastly, we close the channel and the connection;

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

Receiving

That's it for our sender. Our receiver is pushed messages from RabbitMQ, so unlike the sender which publishes a single message, we'll keep it running to listen for messages and print them out.



The code (in **Recv.java**) has almost the same imports as Send:

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
```

The extra QueueingConsumer is a class we'll use to buffer the messages pushed to us by the server.

Setting up is the same as the sender; we 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.

```
public class Recv {

 private final static String QUEUE_NAME = "hello";

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

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

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

Note that we declare the queue here, as well. Because we might start the receiver before the sender, we want to make sure the queue exists before we try to consume messages from it.

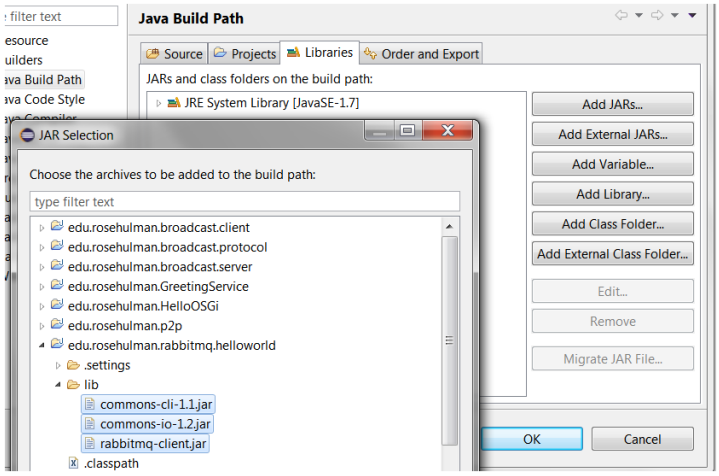
We're about to tell the server to deliver us the messages from the queue. Since it will push us messages asynchronously, we provide a callback in the form of an object that will buffer the messages until we're ready to use them. That is what a DefaultConsumer subclass does.

```
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(QUEUE_NAME, true, consumer);
```

Putting It All Together

Please follow the following steps to build and run your Hello World message queue application:

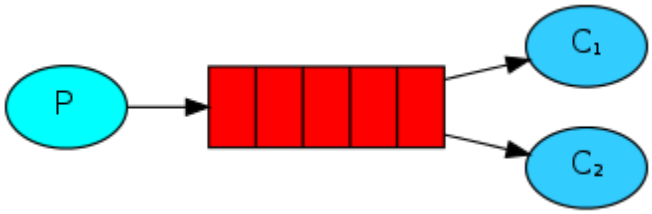
1. Create a Java project in Eclipse. Name it **edu.rosehulman.rabbitmq**.
2. Create a **lib** folder in the project.
3. **Download and extract** the Java Client for RabbitMQ in your download folder. Here is the download URL: http://www.rose-hulman.edu/class/csse/binaries/RabbitMQ/rabbitmq-java-client-bin-3.5.6.zip.
4. Copy **commons-cli-1.1.jar**, **commons-io-1.2.jar**, and **rabbitmq-client.jar** from the extracted **rabbitmq-java-client-bin-3.5.6** directory to the newly created **lib** folder of the project.
5. Right-click on the project in Eclipse -> Properties -> Java Build Path -> Libraries Tab -> Add JARs -> Select the three jars -> OK.



6. This will put the three jar files in the **CLASSPATH** of the project.
7. Create a new package in the project, name it: **edu.rosehulman.rabbitmq.helloworld**.
8. Create **Send.java** and **Recv.java** (classes) in the package.
9. Copy the code for **Send.java** from GitHub (https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/java/Send.java) to the **Send.java** file in your project.
10. Copy the code for **Recv.java** from GitHub (https://github.com/rabbitmq/rabbitmq-tutorials/blob/master/java/Recv.java) to the **Recv.java** file in your project.

11. Run **Send.java** as a regular Java application. You should see an appropriate message in Console.
12. Run **Recv.java** as a regular Java application. The receiver should receive the “Hello World!” message published by the sender.
13. **Terminate the Receiver** by pressing on the **stop** button in the Eclipse console.
14. Create a **pdf file** (name it **Lab8.pdf**). Add snapshots of both sender and receiver console to the pdf.
15. Add “C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.5.6\sbin” or the appropriate installation **sbin** directory to the PATH environment variable.
16. Open command prompt and type: **rabbitmqctl list_queues**.
17. You should see the **hello** queue with **0 messages** in it.
18. Run **Send.java** one more time and retype the **list_queues** command. You should see 1 unread message in the queue.
19. Add the snapshot of the command output to the pdf.
20. Run **Recv.java** to consume the message, which should make the queue empty again.
21. Redo the step #13.

Work Queue Application



In the **Hello World Application**, 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.

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.

Create a new package in the existing project, name it: **edu.rosehulman.rabbitmq.workqueue**.

We will slightly modify the *Send.java* code in our previous package, to allow arbitrary messages to be sent from the command line. Copy *Send.java* from previous package to the new package and name it NewTask.java. Modify the code as follows. This program will schedule tasks to our work queue:

```
...
channel.queueDeclare(QUEUE_NAME, false, false, false, null);

System.out.println("Type messages below. Type quit to exit the program.");
Scanner scanner = new Scanner(System.in);
while(scanner.hasNextLine()) {
            String message = scanner.nextLine().trim();

            if(message.equalsIgnoreCase("quit"))
                        break;

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

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 pop messages from the queue and perform the task. Let's copy the *Recv.java* from previous package to the new package and call it *Worker.java*:

```
...
final 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 + "'");
    try {
      doWork(message);
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    finally {
      System.out.println(" [x] Done");
```

```
    }
  }
};
channel.basicConsume(QUEUE_NAME, true, consumer);
...
```

Our fake task to simulate execution time:

```
private static void doWork(String task) throws InterruptedException {
    System.out.println(" [x] Doing Work: " + task);
    for (char ch: task.toCharArray()) {
        if (ch == '.') Thread.sleep(1000);
    }
}
```

Round-robin dispatching

One of the advantages of using a Task Queue is the ability to easily parallelize work. If we are building up a backlog of work, we can just add more workers and that way, our application can 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. In the third one, we'll publish new tasks. Once you've started the **consumers** you can **publish** a few messages as follows:

```
Type messages below. Type quit to exit the program.
T.
 [x] Sent 'T.'
T..
 [x] Sent 'T..'
T...
 [x] Sent 'T...'
T....
 [x] Sent 'T....'
T.....
 [x] Sent 'T.....'
quit
```

Let's see what is delivered to our workers:

C1:

```
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'T.'
[x] Done
[x] Received 'T...'
[x] Done
[x] Received 'T.....'
[x] Done
```

C2:

```
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'T..'
[x] Done
[x] Received 'T....'
[x] Done
```

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 workers.

Add the snapshots of producer's console and all of the three worker's console to the pdf.

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 customer it immediately removes it from memory. 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 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 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 only when the worker connection dies. It's fine even if processing a message takes a very, very long time.

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

```
channel.basicQos(1);

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

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

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:

$rabbitmqctl list_queues name messages_ready messages_unacknowledged

Listing queues ...

hello 0 0

...done.

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());
```

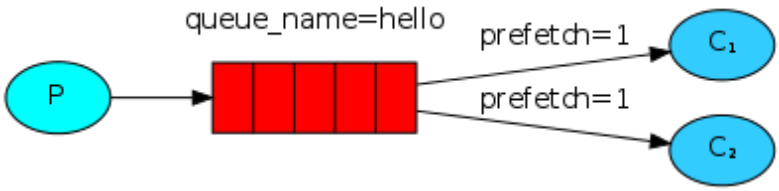
Note on message persistence

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 (https://www.rabbitmq.com/confirms.html).

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 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:

```
public class NewTask {
  private final static String QUEUE_NAME = "task_queue";

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

    channel.queueDeclare(QUEUE_NAME, true, false, false, null);

    System.out.println("Type messages below. Type quit to exit the program.");
    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNextLine()) {
       String message = scanner.nextLine().trim();

       if(message.equalsIgnoreCase("quit"))
             break;

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

       System.out.println(" [x] Sent '" + message + "'");
```

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

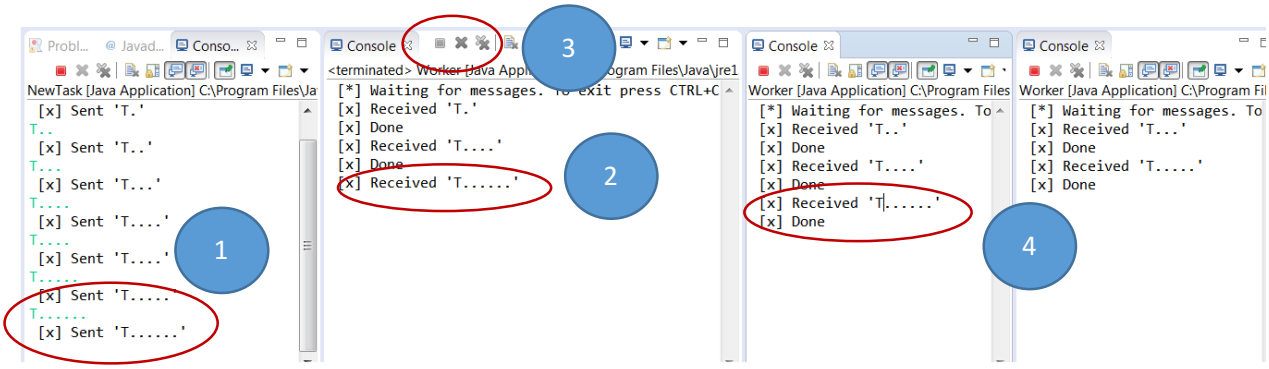
And our **Worker.java**:

```
public class Worker {

	private final static String QUEUE_NAME = "task_queue";

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

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

		channel.basicQos(1);

		final 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 + "'");
				try {
					doWork(message);
				}
				catch(Exception e) {
					e.printStackTrace();
				}
				finally {
					System.out.println(" [x] Done");
					channel.basicAck(envelope.getDeliveryTag(), false);

				}
			}
		};
		channel.basicConsume(QUEUE_NAME, false, consumer);
	}
	private static void doWork(String task) throws InterruptedException {
		System.out.println(" [x] Doing Work: " + task);
		for (char ch : task.toCharArray()) {
			if (ch == '.')
				Thread.sleep(1000);
		}
	}
}
```

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 (http://www.rabbitmq.com/releases/rabbitmq-java-client/current-javadoc/).

Here is a sample run of the producer and three consumer:



The figure above shows one producer (the left most Console) and three consumers (the three right Consoles). The circles highlight that the task **T……** was received by **C1**. **C1** was killed before completion of the **T……** task. Since the messages require acknowledgement, which did not happen through **C1**, the **T……** task was resent to C2, which eventually completes it.

Add a snapshot of similar worked out example to the pdf. Label the figure to show the sequence of events as shown above.

Publish-Subscribe Application

In the previous application 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.

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

Exchanges

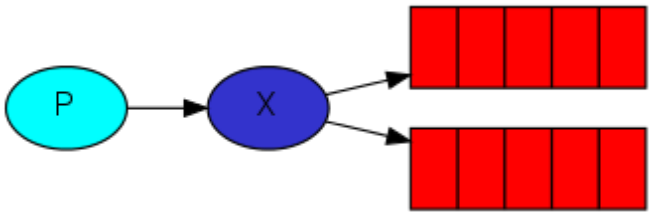
In the previous application, we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in RabbitMQ.

Let's quickly go over what we covered in the previous applications:

- 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:

```
$rabbitmqctl list_exchanges
Listing exchanges ...
      direct
amq.direct     direct
amq.fanout     fanout
amq.headers    headers
amq.match      headers
amq.rabbitmq.log       topic
```

```
amq.rabbitmq.trace      topic
amq.topic       topic
...done.
```

In this list there are 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.

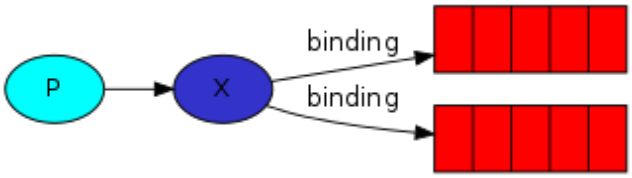
Secondly, once we disconnect the consumer the queue should be automatically deleted.

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*.

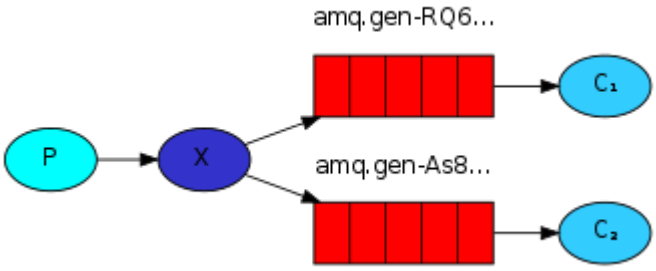
```
channel.queueBind(queueName, "logs", "");
```

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



Create a new package in the project and name it: **edu.rosehulman.rabbitmq.pubsub**.

The producer program, which emits log messages, doesn't look much different from the previous applications. 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:

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

import java.util.Scanner;


public class EmitLog {

  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");

    System.out.println("Type messages below. Type quit to exit the program.");
    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNextLine()) {
       String message = scanner.nextLine().trim();

       if(message.equalsIgnoreCase("quit"))
              break;

        channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
        System.out.println(" [x] Sent '" + message + "'");
    }
    scanner.close();

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

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**:

```
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);
  }
}
```

Run **two** RecvLogs followed by an EmitLog in three different Eclipse Consoles. Type in some messages on the EmitLog Console and verify that both subscribers can receive the broadcast/fanout messages.

Add the snapshots of the output to the pdf.

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:

```
$ 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.

Take a snapshot of the Terminal and add it to the pdf.

Exploring Further

RabbitMQ has a few more tutorials on selective- and topic-based routing that you can try out (https://www.rabbitmq.com/tutorials/tutorial-four-java.html). There are several implementation of MQ such as Apache's Active MQ, IBM's Websphere MQ, Oracle's AQ and Open Message Queue, JBoss Messaging, and so on. Several large-scale architectures use queuing service as their core-component for inter-process communication.

Do It Yourself

Enough following instructions! Let's be creative and solve a new problem using our new found understanding of RabbitMQ.

Assume you recently got hired as an Architect at **overstock.com**, an online retail company. It has been brought to your attention that the online service has been running very slow due to the large volume of transactions. Let's assume that the overstock.com runs on a single beefy server. When a client makes a purchase through the website, the sales application service sends the following sales order to your core logic where the order processing happens. The sales order has the following format:

```
<transaction>
        <id>10</id>
        <credit-card >
                <id>100</id>
                <amount>12</amount>
        </credit-card>
        <item>
                <id>200</id>
                <quantity>2</quantity>
        </item>
        <shipment>
                <id>500</id>
                <deliver-by>10/15/2015</deliver-by>
                <method-id>2</method-id>
        </shipment>
</transaction>
```

Rather than scaling vertically to improve performance, you decided to scale horizontally by distributing the processing logic over multiple servers. Here is what you envisioned. When the web service get the request from the client, it forwards that request to a distribution server (simulate this in a simple java program through console io and converting the input to xml format), the distribution service analyzes the order and sends parts of the order for further processing to the following internal servers:

1. Credit Card Processing (A simple java program that just prints out the received info followed by the " credit-card processed" message on the console)
2. Inventory Processing (A simple java program that just prints out the received info followed by the " inventory processed" message on the console)

3. Shipment Processing (A simple java program that just prints out the received info followed by the “shipment processed” message on the console)
4. Make other simplifying assumption, however, you must connect the distribution service with the three listed services using RabitMQ in your solution.

Append your new architecture diagram of the overstock.com to the pdf followed by snapshots of the four java consoles (distribution, credit-card, inventory, and shipment) in action.

Deliverables

- You will bundle the project into a **zip** file (**not rar** or other formats) and turn it in on Moodle.
- You will turn in **pdf** separately, which should contain all of the requested snapshots and the architecture diagram.

You may use http://x-stream.github.io/ for XML to Java object serialization/deserialization if you like. Look at the following 2-minutes tutorial: http://x-stream.github.io/tutorial.html.