# Department of Computing

**Course Code: CS332**

**Class: BSCS9ABC**

**Lab 10: Introduction to RabbitMQ**

**CLO4: Develop Distributed Applications/Systems**

**Date: April 18, 2022**

**Time: 9:00 am -11:50am &2:00 am – 5:00 pm**

# Instructor: Dr. Farzana Jabeen

**Fatima Seemab**

**291310**

**BSCS 9B**

# Lab 10: Introduction to RabbitMQ

**Introduction**

RabbitMQ is a message broker. The principal idea is pretty simple: it accepts and forwards messages. This lab introduces the concepts of RabbitMQ.

**Objectives**

Understand and implement RabbitMQ

**Tools/Software Requirement**

* Eclipse
* RabbitMQ

**Description**

RabbitMQ is a message broker. The principal idea is pretty simple: it accepts and forwards messages. You can think about it as a post office: when you send mail to the post box you're pretty sure that Mr. Postman will eventually deliver the mail to your recipient. Using this metaphor RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is the fact that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data ‒ messages.

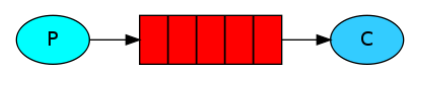
RabbitMQ, and messaging in general, uses some jargon like

Producing means nothing more than sending. A program that sends messages is a producer. We'll draw it like that, 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 like that, 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

****

**What can RabbitMQ do for you?**

Messaging enables software applications to connect and scale. Applications can connect to each other, as components of a larger application, or to user devices and data. Messaging is asynchronous, decoupling applications by separating sending and receiving data.

You may be thinking of data delivery, non-blocking operations or push notifications. Or you want to use publish / subscribe, asynchronous processing, or work queues. All these are patterns, and they form part of messaging.

RabbitMQ is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

## Feature Highlights

### Reliability

RabbitMQ offers a variety of features to let you trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability.

### Flexible Routing

Messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic. For more complex routing you can [bind exchanges together](https://www.rabbitmq.com/extensions.html#routing) or even write your own exchange type as a plugin.

### Clustering

Several RabbitMQ servers on a local network can be [clustered](https://www.rabbitmq.com/clustering.html) together, forming a single logical broker.

### Federation

For servers that need to be more loosely and unreliably connected than clustering allows, RabbitMQ offers a federation model.

### Highly Available Queues

Queues can be [mirrored](https://www.rabbitmq.com/ha.html) across several machines in a cluster, ensuring that even in the event of hardware failure your messages are safe.

### Multi-protocol

RabbitMQ supports messaging over [a variety of messaging protocols](https://www.rabbitmq.com/protocols.html).

In the below mentioned code, we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker.

Before coding download the RABBITMQ server from [\\10.3.20.65\downloads\RABBITMQ](file://10.3.20.65/downloads/RABBITMQ)



Figure 1: SENDER CODE



Figure 2: RECEIVER CODE

The receiver will print the message it gets from the sender via RabbitMQ. The receiver will keep running, waiting for messages (Use Ctrl-C to stop it), so try running the sender from another terminal.

Open rabbitMq command line and type following command

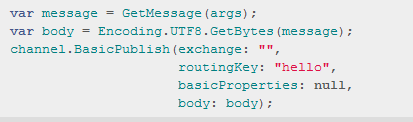
rabbitmqctl list\_queues.

Your task is after implementing the above mentioned code. Please go to below mentioned link and implement the publish subscribe mechanism using RabbitMQ by adding a exchange (<https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html>)

#### 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:

****

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

**Durability basics**

There are two types of queues, exchanges and messages from a persistence point of view:

* Durable: messages and other resources are saved to disk so they are available even after a server restart. There’s some overhead incurred while reading and saving messages. If durability is set to true then the resource will be persisted to disk and not only exist in memory. You’ll probably want to have the durable parameter set to true in a real life messaging scenario
* Non-durable: the resources are persisted in memory only. They disappear after a server restart but offer a faster service

|  |  |
| --- | --- |
|  | channel.QueueDeclare("my.first.queue", true, false, false, null); |

We have the following parameters to the function:

* The queue name
* Whether it is durable
* Whether it is exclusive, which means whether the queue is exclusively used for the connection
* Whether it should be auto-deleted
* The same dictionary object with custom options as in ExchangeDeclare

**Lab Tasks**

Complete the above mentioned steps.

**Simple Hello World Program**

CODE:

Sender:

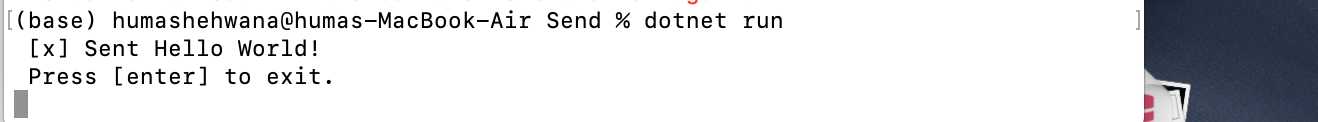
|  |
| --- |
| using System;  using RabbitMQ.Client;  using System.Text;  class Send  {  public static void Main()  {  var factory = new ConnectionFactory() { HostName = "localhost" };  using(var connection = factory.CreateConnection())  using(var channel = connection.CreateModel())  {  channel.QueueDeclare(queue: "hello",  durable: false,  exclusive: false,  autoDelete: false,  arguments: null);  string message = "Hello World!";  var body = Encoding.UTF8.GetBytes(message);  channel.BasicPublish(exchange: "",  routingKey: "hello",  basicProperties: null,  body: body);  Console.WriteLine(" [x] Sent {0}", message);  }  Console.WriteLine(" Press [enter] to exit.");  Console.ReadLine();  }  } |

Receiver:

|  |
| --- |
| using RabbitMQ.Client;  using RabbitMQ.Client.Events;  using System;  using System.Text;  class Receive  {  public static void Main()  {  var factory = new ConnectionFactory() { HostName = "localhost" };  using(var connection = factory.CreateConnection())  using(var channel = connection.CreateModel())  {  channel.QueueDeclare(queue: "hello",  durable: false,  exclusive: false,  autoDelete: false,  arguments: null);  var consumer = new EventingBasicConsumer(channel);  consumer.Received += (model, ea) =>  {  var body = ea.Body.ToArray();  var message = Encoding.UTF8.GetString(body);  Console.WriteLine(" [x] Received {0}", message);  };  channel.BasicConsume(queue: "hello",  autoAck: true,  consumer: consumer);  Console.WriteLine(" Press [enter] to exit.");  Console.ReadLine();  }  }  } |

**Chart

Description automatically generated**

OUTPUT:****

**Publish Subscribe Model:**

CODE:

Sender:

|  |
| --- |
| #!/usr/bin/env python import pika import sys  connection = pika.BlockingConnection(  pika.ConnectionParameters(host='localhost')) channel = connection.channel()  channel.exchange\_declare(exchange='logs', exchange\_type='fanout')  message = ' '.join(sys.argv[1:]) or "info: Hello World!" channel.basic\_publish(exchange='logs', routing\_key='', body=message) print(" [x] Sent %r" % message) connection.close() |

Receiver:

|  |
| --- |
| import pika  connection = pika.BlockingConnection(  pika.ConnectionParameters(host='localhost')) channel = connection.channel()  channel.exchange\_declare(exchange='logs', exchange\_type='fanout')  result = channel.queue\_declare(queue='', exclusive=True) queue\_name = result.method.queue  channel.queue\_bind(exchange='logs', queue=queue\_name)  print(' [\*] Waiting for logs. To exit press CTRL+C')  def callback(ch, method, properties, body):  print(" [x] %r" % body)  channel.basic\_consume(  queue=queue\_name, on\_message\_callback=callback, auto\_ack=True)  channel.start\_consuming() |

**OUTPUT:**

**Graphical user interface, application

Description automatically generated**

Graphical user interface, application

Description automatically generated

**Text

Description automatically generated**

**Deliverables**

Submit the code files and screenshots on LMS.