**Institute of Technology, Tralee**

**Computing Department**

Distributed Computing – Message Brokers

**Learning Outcomes**

* MOM described
* JMS basics.
* RabbitMQ example
* MQTT example
* Other Protocols: AMQP, STOMP

**Message-oriented Middleware**

**Message-oriented middleware (MOM)** is infrastructure focused on sending and receiving messages that allows application modules to be distributed over heterogeneous platforms.

It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces.

API's that extend across diverse platforms and networks are typically provided by MOM.

MOM is software that resides in both portions of client/server architecture and typically supports asynchronous calls between the client and server applications.

Message queues provide temporary storage when the destination program is busy or not connected.

MOM reduces the involvement of application developers with the complexity of the client/server mechanism.

MOM comprises a category of inter-application communication software that generally relies on asynchronous message-passing, as opposed to a request-response metaphor.

We have already come across this model with Redis (pubsub commands) and you have also possible seen the observer pattern in software patterns.

**JMS Basics**

JMS is a proprietary MOM API developed by Sun/Oracle for Java

It provides a good introduction to Message Queue Systems as it introduced a lot of the terminology we use in this and other products

**JMS components**

Java Message Service (JMS) is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. A JMS Provider is the software that implements the Java Message Service (JMS) specification for a messaging product's brokers and clients.

A JMS messaging product is comprised of the following components:

* Brokers
* Messages
* Destinations
* Clients
* Connections
* Sessions

**JMS broker:** A JMS broker provides clients with connectivity, and message storage/delivery functions

**Messages:** A messages is an object that contains the required heading fields, optional properties, and data payload being transferred between JMS clients.

**Destinations:** Destinations are maintained by the message broker. They can be either queues or topics.

**Queues:** A queue is a destination that contains messages sent from a producer that await delivery to one consumer. Messages are delivered in the order sent. A message is removed from the queue once it has been acknowledged as received by the consumer. In this one-to-one messaging model, producers are *senders* and consumers are *receivers*.



Point-to-point Messaging

**Topics:** Topics are used to send messages to one or more consumers. Producers publish messages to a topic and one or more consumers subscribe to the topic. In this one-to-many messaging scenario, producers are also referred to as *publishers* and consumers as *subscribers*.



**Publish-Subscribe Messaging**

**JMS Clients**

A JMS client is an application that uses the services of the message broker. There are two client types in a JMS system:

* Producer
  + Producers create messages and send or publish them to the broker for delivery to a specified destination.
* Consumer
  + Consumers retrieve messages from a destination.

**Connections**

Connections are the technique used by clients to specify a protocol and credentials for a sustained client interaction with a broker.

**Sessions**

Sessions are defined by a client on a connection established with a broker. Each session defines whether the messages will form transactions, and – if not -- the acknowledgement mode for messages.

RabbitMQ

[**http://www.rabbitmq.com/**](http://www.rabbitmq.com/)

RabbitMQ is an open source 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.

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.

### 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](http://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](http://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](http://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](http://www.rabbitmq.com/protocols.html) (See below; MQTT, AMPQ, STOMP): <https://www.rabbitmq.com/protocols.html>

### Many Clients

There are RabbitMQ clients for almost any [language](http://www.rabbitmq.com/devtools.html) you can think of. <https://www.rabbitmq.com/devtools.html>

### Management UI

RabbitMQ ships with an easy-to use [management UI](http://www.rabbitmq.com/management.html) that allows you to monitor and control every aspect of your message broker.

### Tracing

If your messaging system is misbehaving, RabbitMQ offers [tracing](http://www.rabbitmq.com/firehose.html) support to let you find out what's going on.

### Plugin System

RabbitMQ ships with a variety of [plugins](http://www.rabbitmq.com/plugins.html) extending it in different ways, and you can also [write your own](http://www.rabbitmq.com/plugin-development.html).

## And Also...

### Commercial Support

Commercial [support, training and consulting](http://www.rabbitmq.com/services.html) are available from Pivotal.

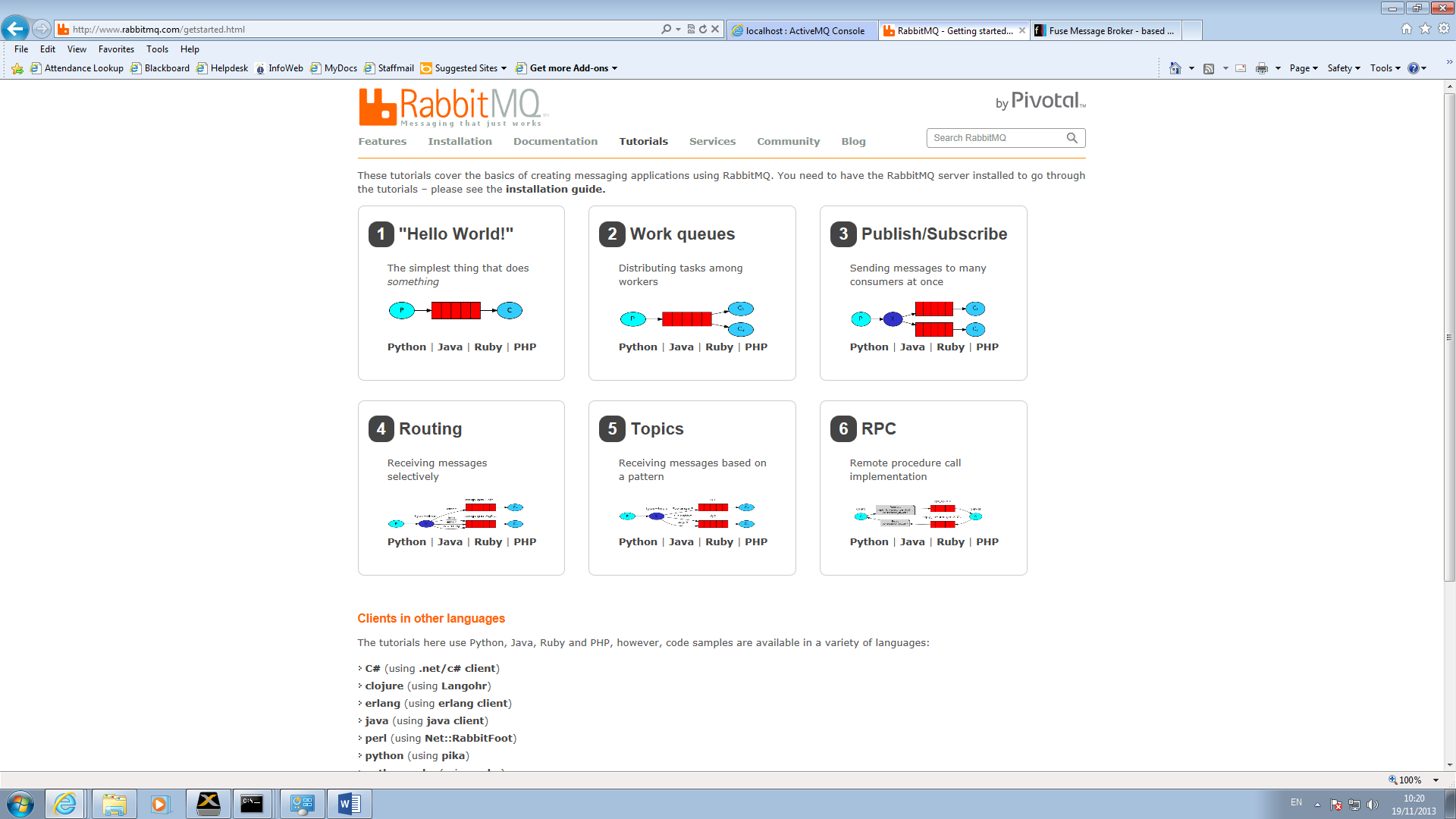
### Large Community

There's a large [community](http://www.rabbitmq.com/devtools.html) around RabbitMQ, producing all sorts of clients, plugins, guides, etc. Join our [mailing list](https://lists.rabbitmq.com/cgi-bin/mailman/listinfo/rabbitmq-discuss) to get involved!

**rabbitMQ Tutorials**

There are a number of tutorials you can do on the rabbitMQ site: <https://www.rabbitmq.com/getstarted.html>

Latest code sources and latest instructions at: <https://github.com/rabbitmq/rabbitmq-tutorials/tree/master/java>



Download and install RabbitMQ for Windows <http://www.rabbitmq.com/install-windows.html> Note it also needs Erlang installed (as Administrator)

Start RabbitMQ server

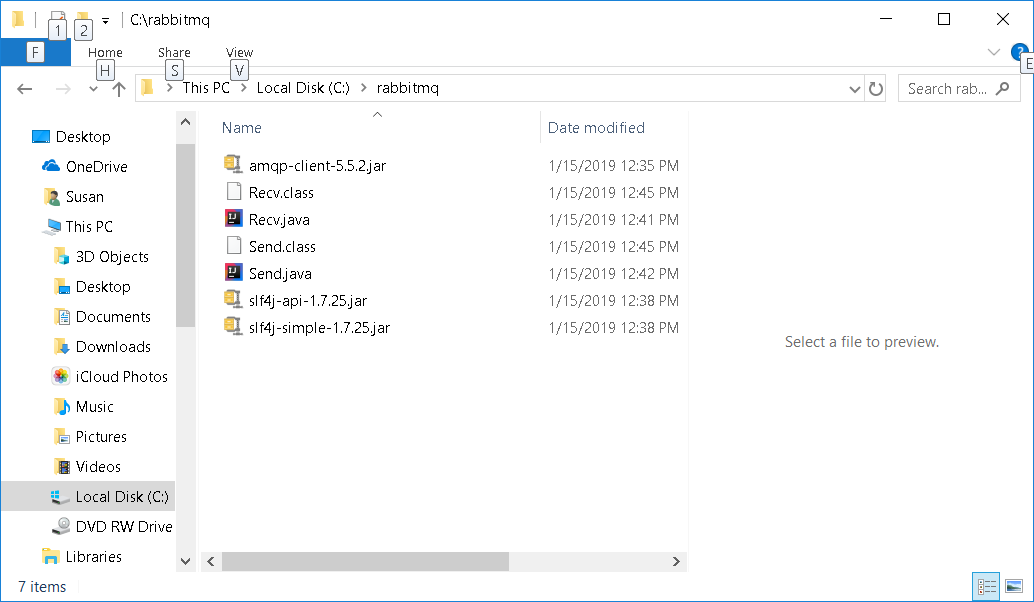
Download the Java client <http://www.rabbitmq.com/java-client.html>

(Note there is an official Docker image for RabbitMQ - https://hub.docker.com/\_/rabbitmq/)

Do the following tutorials from <http://www.rabbitmq.com/getstarted.html>

**Exercise 1 Hello World**

Download the Recv.java and Send.java for the folder with the Java client and compile. Note you also need the Simple Logging for Java library as the Java client uses this. I have put it in the folder for today.



Hello World

**Receiver**

public class Recv {

private final static String QUEUE\_NAME = "hello";

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, false, false, false, null);

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

DeliverCallback deliverCallback = (consumerTag, delivery) -> {

String message = new String(delivery.getBody(), "UTF-8");

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

};

channel.basicConsume(QUEUE\_NAME, true, deliverCallback, consumerTag -> { });

}

}

**Sender**

public class Send {

private final static String QUEUE\_NAME = "hello";

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(QUEUE\_NAME, false, false, false, null);

String message = "Hello World!";

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

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

}

}

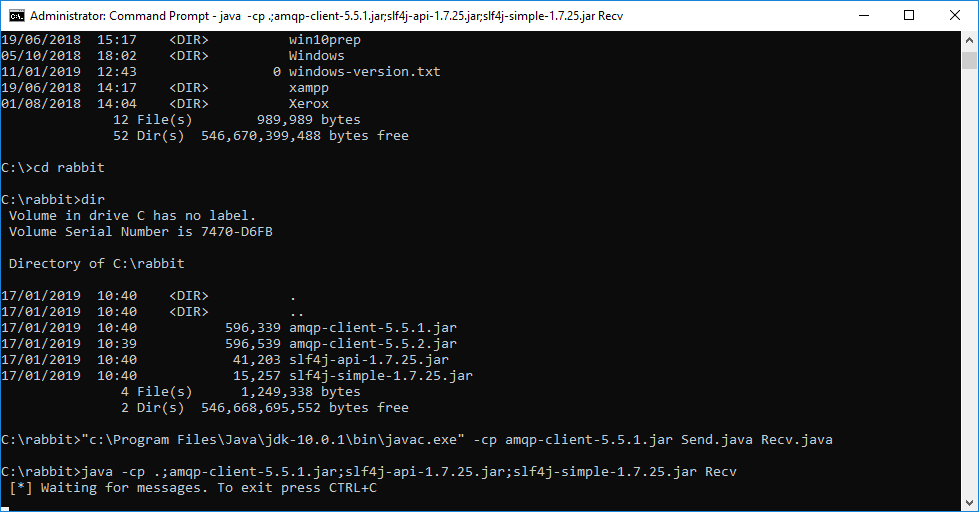
}

To compile

"c:\Program Files\Java\jdk-10.0.1\bin\javac.exe" -cp amqp-client-5.5.2.jar Send.java Recv.java

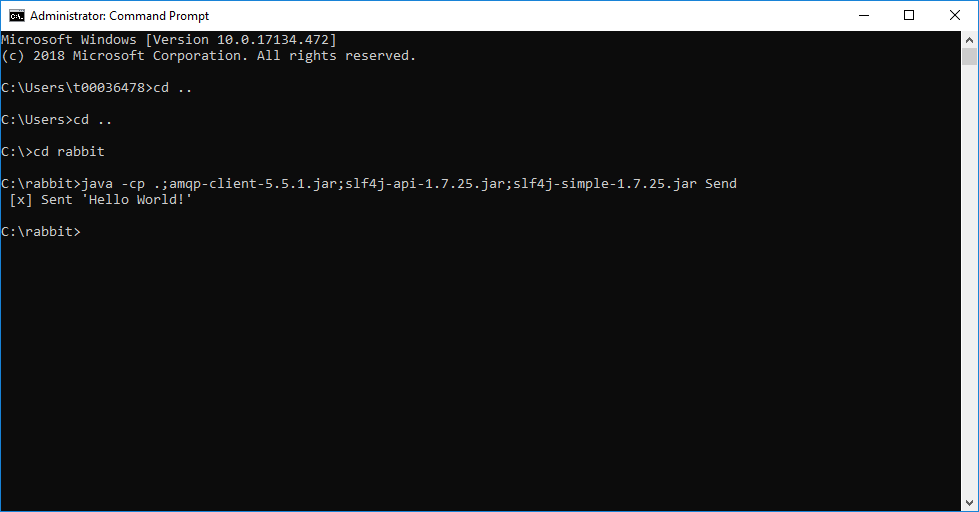
To run Receiver

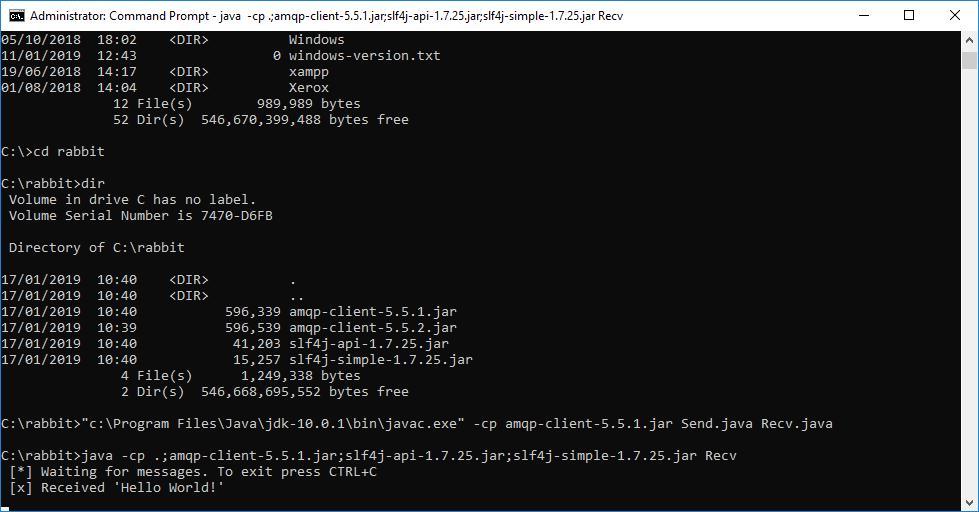
java -cp .;amqp-client-5.5.2.jar;slf4j-api-1.7.25.jar;slf4j-simple-1.7.25.jar Recv



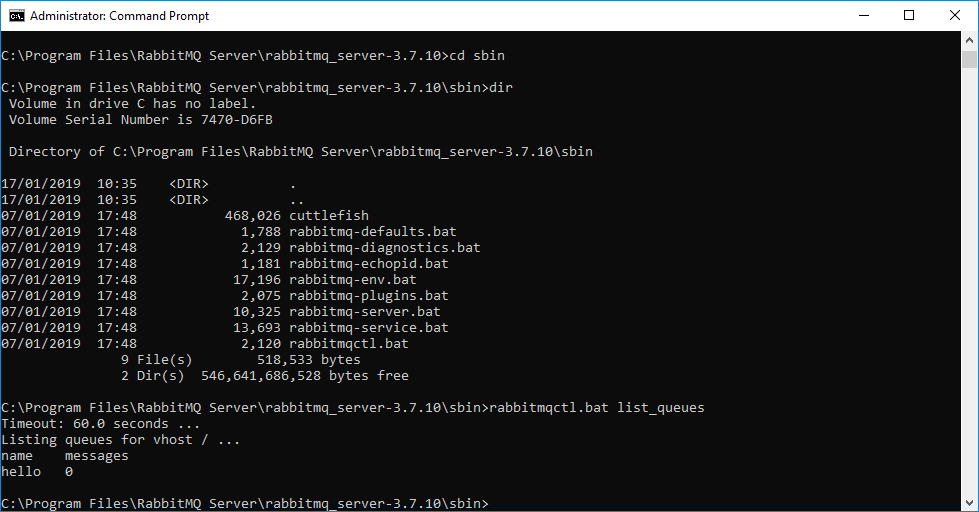
To run Sender

java -cp .;amqp-client-5.5.2.jar;slf4j-api-1.7.25.jar;slf4j-simple-1.7.25.jar Send





To see queues – from sbin folder of rabbitMQ installed folder



(Another exercise - Now you could build the project example in Netbeans/IntelliJ/Eclipse)

Exercise 3. Publish Subscribe – (In your own time)

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 first exercise 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 example:

* 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 that type, and call it logs:

channel.exchange\_declare(exchange='logs',

exchange\_type='fanout')

The fanout exchange is very simple. It just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

In exercise 1 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 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.

## Temporary queues

As you may remember previously we were using queues that had specific names (remember the hello queue in exercise 1). Giving a queue a name is important when you want to share the queue between producers and consumers.

For this application we want to hear about all log messages. 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

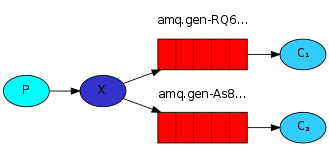


We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a binding.

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

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

## Putting it all together



The producer program, which emits log messages. The most important bit 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:

**EmitLog**

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");

try (Connection connection = factory.newConnection();

Channel channel = connection.createChannel()) {

channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");

String message = argv.length < 1 ? "info: Hello World!" :

String.join(" ", argv);

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

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

}

}

}

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.

**ReceiveLogs**

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");

DeliverCallback deliverCallback = (consumerTag, delivery) -> {

String message = new String(delivery.getBody(), "UTF-8");

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

};

channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });

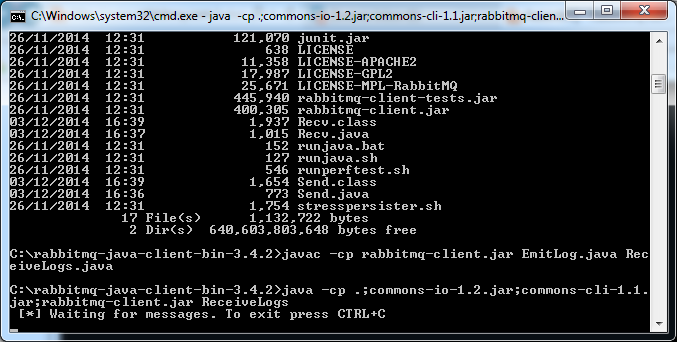
}

}

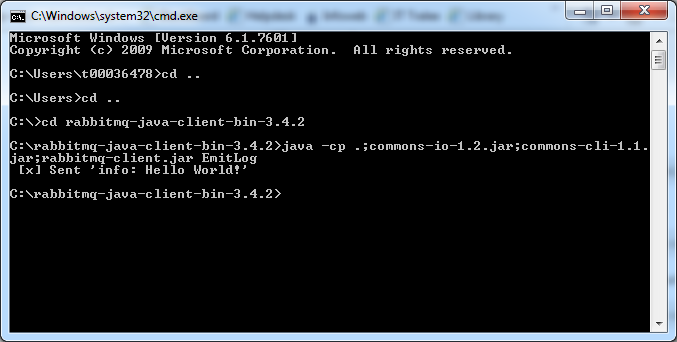
Compile as before and we're done

Note use the correct name for the Java client (amqp-client-5.5.2.jar) in the code below.

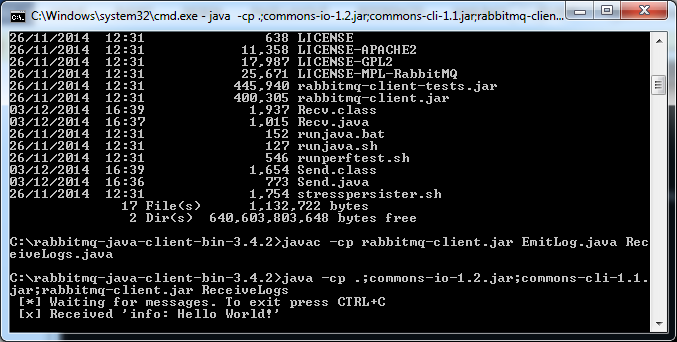
Here I run the Receiver



And an Emitter



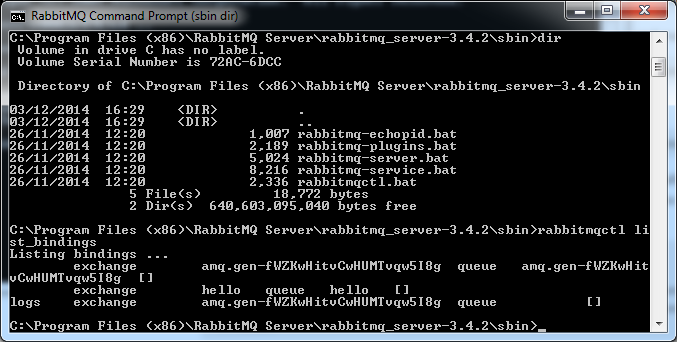
The Receiver gets the message



If you want you can run another Receiver and store the messages in a log file

java -cp $CP ReceiveLogs > logs\_from\_rabbit.log

You can view all the bindings from the sbin folder by calling *rabbitmqctl list\_bindings*



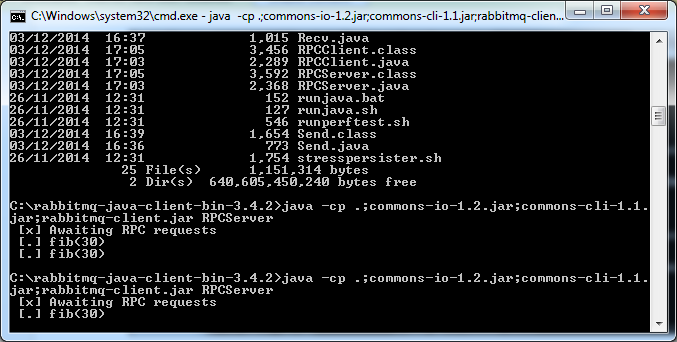
(Now build the project example in Netbeans/Intellij/Eclipse and review the code)

**Exercise 6. RPC – (In your own time)**

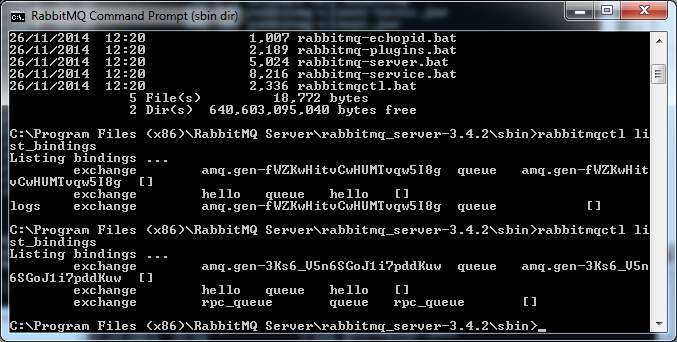
Download the code for the RPC exercise. Use the correct names for the files in the examples shown below.

Work through the tutorial.

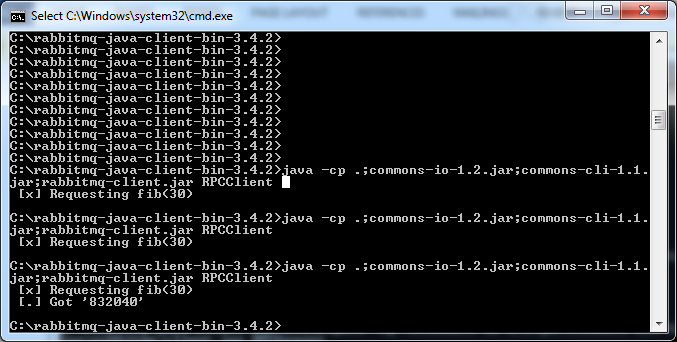
Calling the RPCServer



Showing the bindings



Calling the RPCClient



(Now build the project example in Netbeans etc. and review the code)

Exercise 4: Topics – (In your own time)

Do this exercise in your own time. We will also cover a topic example below using MQTT

**Message Queue Protocols**

There are a number of Message Queue Protocols; by default RabbitMQ uses AMQP. Other protocols include MQTT and STOMP. RabbitMQ will allow you to use these other protocols if you wish.

In the next exercise we will use another MOM product called Mosquitto which implements the MQTT protocol.

**MQTT**

<https://www.networkworld.com/article/3142778/internet-of-things/internet-of-things-messaging-part-1-introducing-mqtt.html>



When IoT devices generate messages, they send stuff like status reports and environmental measurements; when they receive, they’re being told what to do (“open that valve” or “send your status”) etc

Now there are many ways to handle IoT messaging (RPC, REST etc) but one of the most powerful and, for that matter most interesting, technologies are message brokers.

MQTT is…

… a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks.

The design principles are to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery.

These principles also turn out to make the protocol ideal of the emerging “machine-to-machine” (M2M) or “Internet of Things” world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

In other words, MQTT implementations are excellent choices for endpoint devices such as Raspberry Pi’s, Arduinos, smart phones OSes, and pretty much any platform that can benefit from simple and low overhead messaging.

The MQTT protocol is, as of writing, at [version](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html) 3.1.1 (<http://docs.oasis-open.org/mqtt/mqtt/>) with version 5 a candidate standard since 2018 and has become [an OASIS standard](https://www.oasis-open.org/org). “brokers” are now be called MQTT “servers” while publishers and subscribers are called “clients”

IANA has reserved TCP/IP port 1883 for use by MQTT servers, along with port 8883 for MQTT over SSL (note that using SSL introduces an additional communications performance overhead).

The MQTT protocol has five key features:

* Topic-Based Routing. Messages in MQTT implementations are labelled by topic hierarchies such as state/building/room/device/sensor and subscriptions can use wildcards so, for example, all of the sensor data for all laboratories in all buildings in California could be subscribed to as california/+/laboratory/# (+ is a single level wildcard while # matches all following lower levels.
* Support for Clean Sessions. When an endpoint connects to an MQTT server for the very first time and new MQTT session is created and both the server and the client note the session by storing data about the state. This saved session state allows the broker to resume message delivery after reconnection so messages that haven’t been sent or received due to disconnection can be delivered. Clean sessions are an option.
* Quality of Service. MQTT’s QoS support allows subscribers and publishers (collectively, “clients”) to select one of three levels of service:
* QoS 0, also referred to as “fire and forget”, doesn’t require acknowledgement from a client that have been received. This level potentially sacrifices delivery reliability (messages can be lost) for speed so is more suited for reliable connections.
* QoS 1, “at least once” delivery, requires the sender (either the server or client as appropriate) to retry if no acknowledgement is received within the timeout window. On poor quality connections QoS 1 retries can impact performance and duplicate messages are possible.
* QoS 2, “exactly once” delivery, is like QoS 1 but with a subsequent status exchange to ensure duplicate messages are not delivered. The extra communications has an even higher potential overhead than QoS 1.
* Retained Messages. When a publisher marks a message in a topic “to be retained” that message is delivered first when an endpoint first subscribes to that topic. Only one retained message is allowed per topic. Consider this to be a “welcome” or “last status” message from a publisher.
* Last Will and Testament Messages. When connecting to a server, a publisher or subscriber can  specify a LWT message for a topic so that if that endpoint disconnects ungracefully, all subscribers to the topic will receive the message. On a graceful disconnection, LWT messages are disposed of.

MQTT and Security

V5.0 - MQTT Client and Server implementations SHOULD offer Authentication, Authorization and secure communication options. Applications concerned with critical infrastructure, personally identifiable information, or other personal or sensitive information are strongly advised to use these security capabilities.

**Implementation**

We will implement an MQTT server and clients on a windows machine using Mosquitto

The Mosquitto (<https://mosquitto.org/>) broker

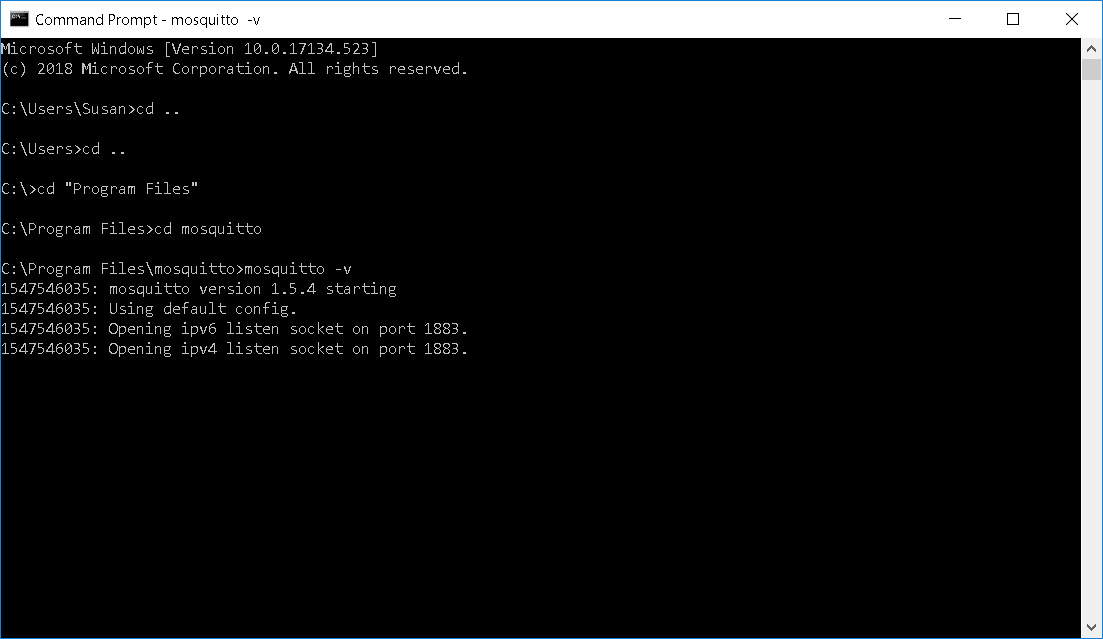
Mosquito currently supports MQTT version 3.1 and 3.1.1 and support for the [proposed MQTT v5](https://issues.oasis-open.org/browse/MQTT/fixforversion/14060/?selectedTab=com.atlassian.jira.jira-projects-plugin:version-summary-panel), which introduces scalability and protocol improvements [is under way](https://mosquitto.org/2016/08/mqtt-v5-draft-features/).

1. Download and install Mosquitto for windows x64 from <https://mosquitto.org/download/>
2. Download and install OpenSSL (the Win 64 v.1.1.1a Light) EXE from <http://slproweb.com/products/Win32OpenSSL.html>
3. See the readme.txt with Mosquitto if you are having problems
4. Run the Mosquitto Server (Broker) (cd to the moqquitto installation and run)

Open command prompt

Cd c:\program files\mosquitto

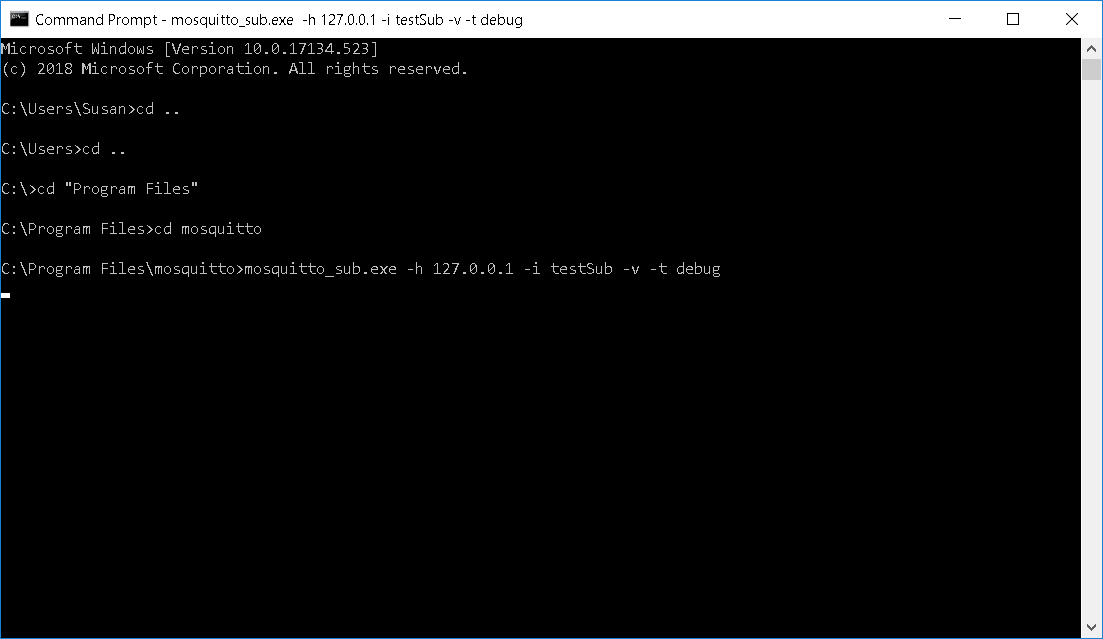
mosquito –v



The Mosquitto broker is now listening on the standard MQTT port, 1883, for both IPv4 and IPv6 MQTT requests.

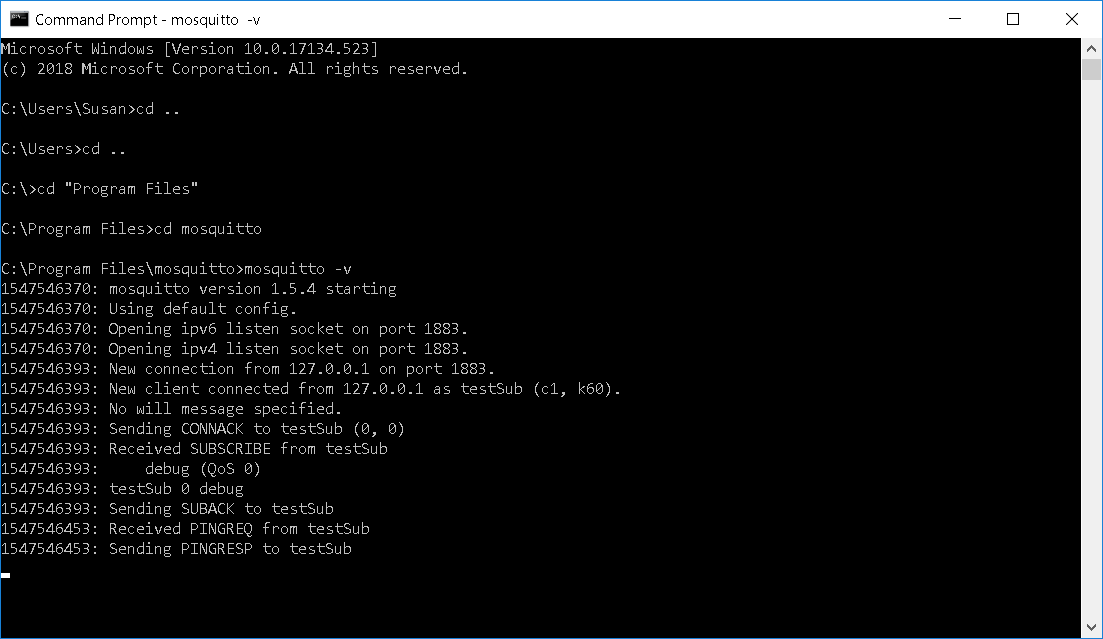
1. Now the server is running open another command prompt, cd to the mosquito folder and run a subscriber client

mosquitto\_sub.exe -h 127.0.0.1 -i testSub -v -t debug



The [Mosquitto subscriber](https://mosquitto.org/man/mosquitto_sub-1.html) (called **–i testSub**) is now waiting for messages (from the server at host **–h 127.0.0.1**) with the topic ‘**debug’**. Without the **-v** option the subscriber will only print the payload while with it, the subscriber prints both the topic and payload

The Server shows the client’s connection

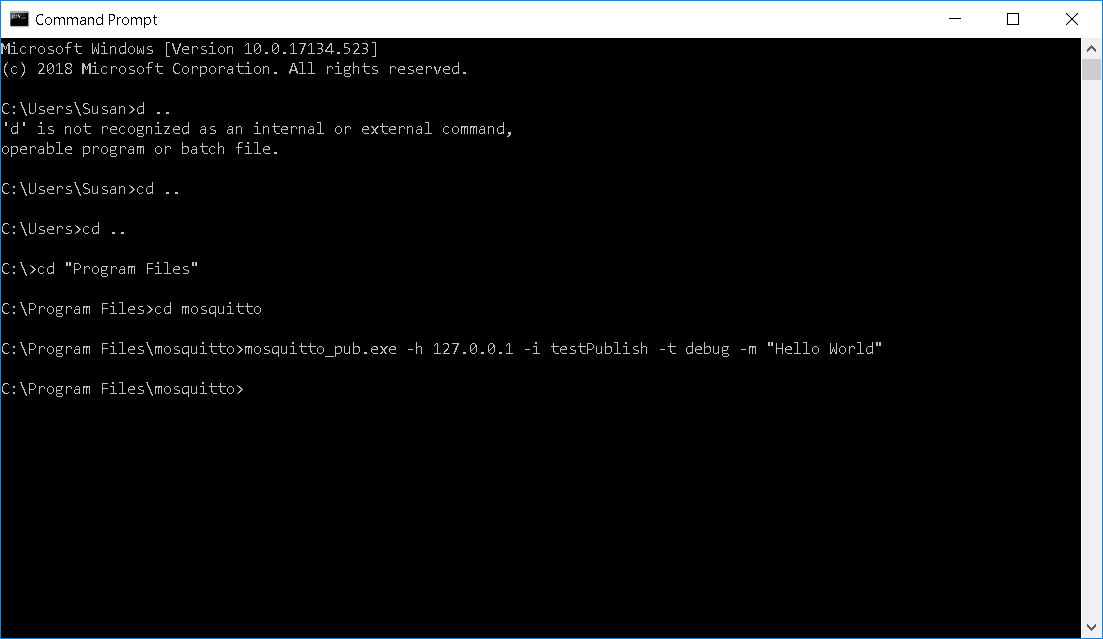


Observe the handshake taking place

* Client connection request on port 1883
* Connection accepted from client testSub
* No will message specified
* CONNACK sent back to the client testSub
* Client request a SUBSCRIBE to topic debug (with default QoS 0)
* testSub subscribed to debug
* SUBACK message sent back to testSub client.
* Perodically a PINGREQ and PINGRESP sent between client and server

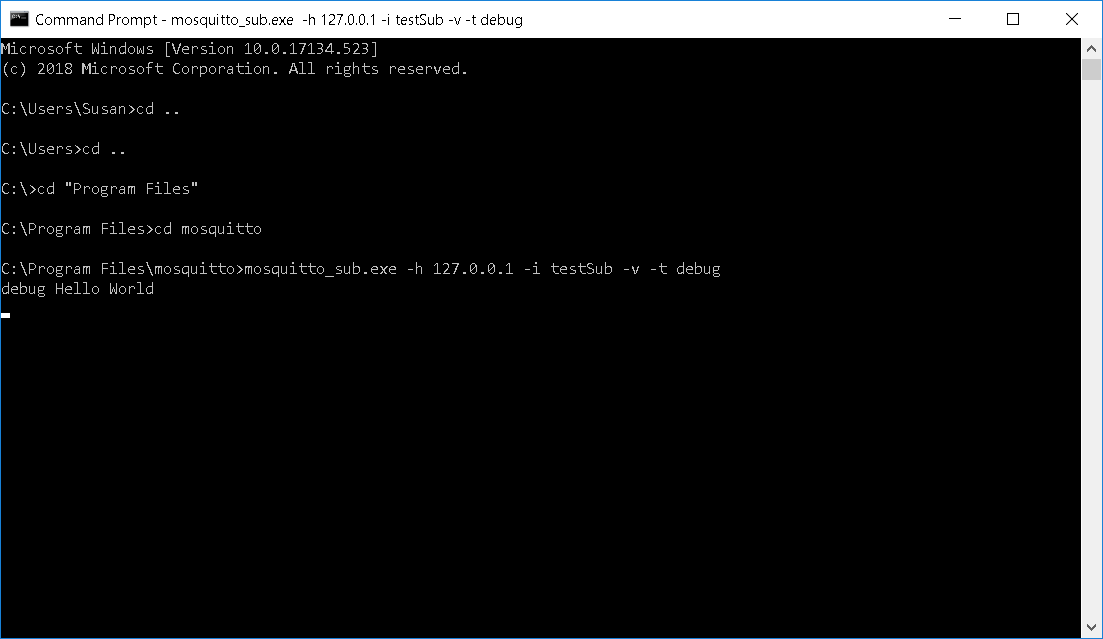
1. Open another command prompt and cd to the mosquito folder and run a publisher client

mosquitto\_pub.exe -h 127.0.0.1 -i testPublish -t debug -m "Hello World"

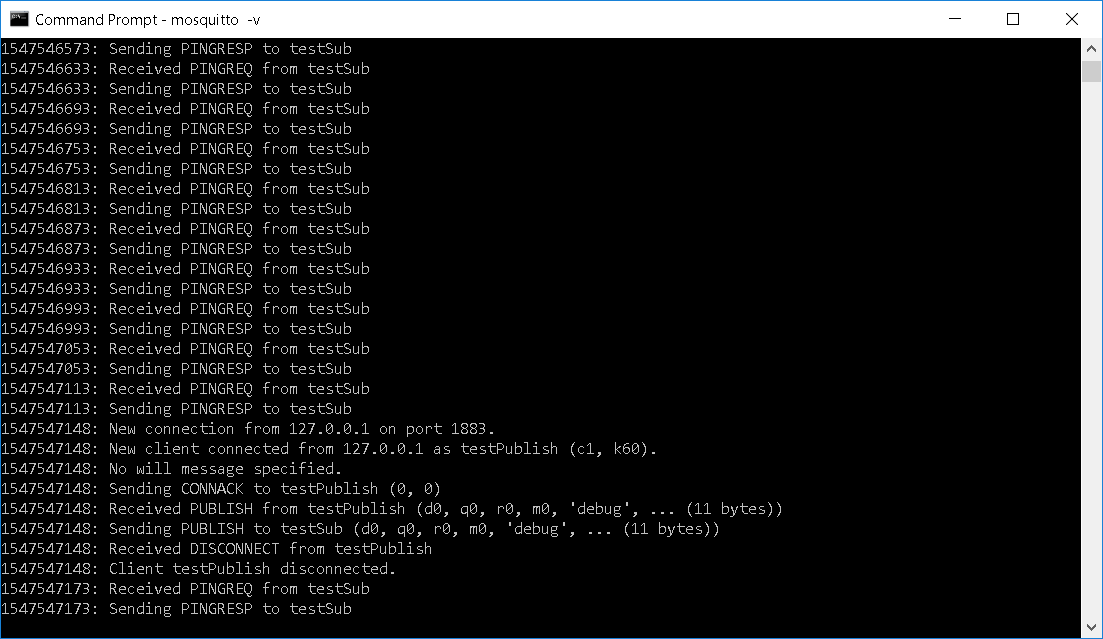


Note the client process sends the message to the topic and shuts down

Note the subscriber receives the message (and the topic)



Note the conversation of the server (beneath the ping requests and responses)



* Client connection request on port 1883
* Connection accepted from client testPublish
* No will message specified
* CONNACK sent back to the client testPublish
* PUBLISH message received from testPublish client to topic debug
* PUBLISH message sent to testSub client from debug topic
* DISCONNECT message from testPublish
* Client testPublish disconnected
* Pings continue to/from testSub

Wildcards

Much like REST's **GET**/**POST** commands that reference a URL, MQTT clients **publish** and **subscribe** to a topic. And, like URLs, topics use **/** delimeters and ascii-text categorization.

There is one clever trick that the broker can do with topics, that you **cannot** do with REST: wildcards! With wildcards, any client can subscribe to any part of that topic, rather than a fully qualified topic.

There are two wildcards available, **#** and **+**

**+** wildcard is used to get a **single level** of hierarchy**.**Here’s an example wildcard topic: sensors/fl3/temperature/+

This topic would let you monitor **all** of the temperature sensors on the 3rd floor.

You can get even more advanced with wildcard subscriptions with something like: sensors/+/temperature/+

Note that there are two **+** wildcards here! This topic would get *all*the temperature data on *all* floors, that are publishing to a temperature topic. And you’d get updates *any* time one of those sensors were published.

**#** wildcard can be used as a match for all remaining levels of hierarchy.

For example if you wanted all to subscribe to *everything* going on all *floors*, you’d use: sensors/#

Or maybe you want to listen in on all 10th floor sensors: sensors/fl10/#

As you can see, thinking through your topic hierarchy is required if you want to take advantage of the **+** and **#** wildcards!

1. Try as client subscribed to all topics: mosquitto\_sub -h test.mosquitto.org -t "#" –v

And publish to a topic with another client: mosquitto\_pub.exe -h 127.0.0.1 -i testPublish -t debug -m "Hello World"

**Other Mosquitto resources**

There are libraries available to use MQTT in java and python and other languages as well (including in Arduino’s C++)

Here is a snippet of Java code using a java mqtt library (http://tgrall.github.io/blog/2017/01/02/getting-started-with-mqtt/)

MqttClient client = new MqttClient("tcp://localhost:1883", MqttClient.generateClientId());

client.connect();

MqttMessage message = new MqttMessage();

message.setPayload("Hello world from Java".getBytes());

client.publish("iot\_data", message);

client.disconnect();

Here is what some Arduino code (C++) using the installed PubSubClient arduino library looks like

**#include <PubSubClient.h>**

**PubSubClient client();**

// Update these with values suitable for your network.

**IPAddress server(172, 16, 0, 2);**

**client.setServer(server, 1883);**

while (!client.connected()) {

Serial.print("Attempting MQTT connection...");

// Attempt to connect

**if (client.connect("arduinoClient")) {**

Serial.println("connected");

// Once connected, publish an announcement...

**client.publish("outTopic","hello world");**

// ... and resubscribe

**client.subscribe("inTopic");**

There is a test server at <https://test.mosquitto.org/>

* Try is with a client: mosquitto\_sub -h test.mosquitto.org –t debug –v
* And a client mosquitto\_pub -h test.mosquitto.org -t debug –m “Hello World”

There is an API description at <https://mosquitto.org/api/files/mosquitto-h.html>

## There is a monitor available to monitor messages on the queue at <https://github.com/eclipse/paho.mqtt-spy/wiki/GettingStarted> (May be missing dependency (pl.baczkowicz.mqttspy.Main)

Note that Amazon AWS IoT includes an MQTT server for connecting IoT devices: <https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html>

(also supported on Microsoft IoT Hub on Azure, Google Cloud IoT, UbiSoft etc)

<https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support>

We will use the MQTT API and library in Arduino when we look at IoT later.

**AMQP**

<https://www.amqp.org/>

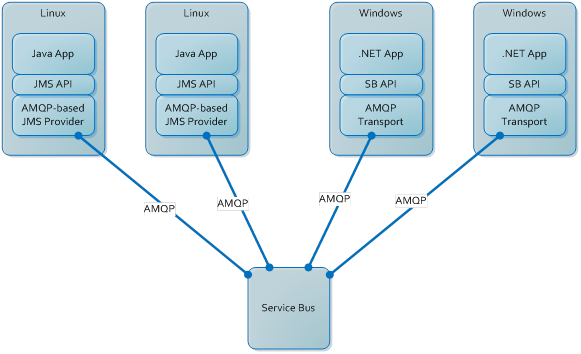
<https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-amqp-overview>

AMQP is a standard wire-level protocol and semantic framework for high performance enterprise messaging.

From the AMQP website:

AMQP is an Open Standard for Messaging Middleware.

By complying to the AMQP standard, middleware products written for different platforms and in different languages can send messages to one another. AMQP addresses the problem of transporting value-bearing messages across and between organisations in a timely manner.



AMQP enables complete interoperability for messaging middleware; both the networking protocol and the semantics of broker services are defined in AMQP.

**Other Protocols**

STOMP: Simple Text Oriented Message Protocol (<http://stomp.github.io/>)

**Which Protocol should I use for a project?**

<https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html>

**RabbitMQ Support them all**

RabbitMQ was initially built to support AMPQ but now RabbitMQ supports all these protocols; AMPQ, MQTT and STOMP so you could integrate different systems using different protocols using RabbitMQ (<http://www.rabbitmq.com/protocols.html>)

**Summary**

* **Proprietary Message Queue API: JMS**
* **MOM: RabbitMQ, Mosquitto**
* **Protocols: AMPQ, MQTT, STOMP**