RabbitMQ
Open Source Enterprise Messaging

Messaging with AMQP and RabbitMQ

Systems Integration

PBA Softwareudvikling/BSc Software Development

Tine Marbjerg

Fall 2018

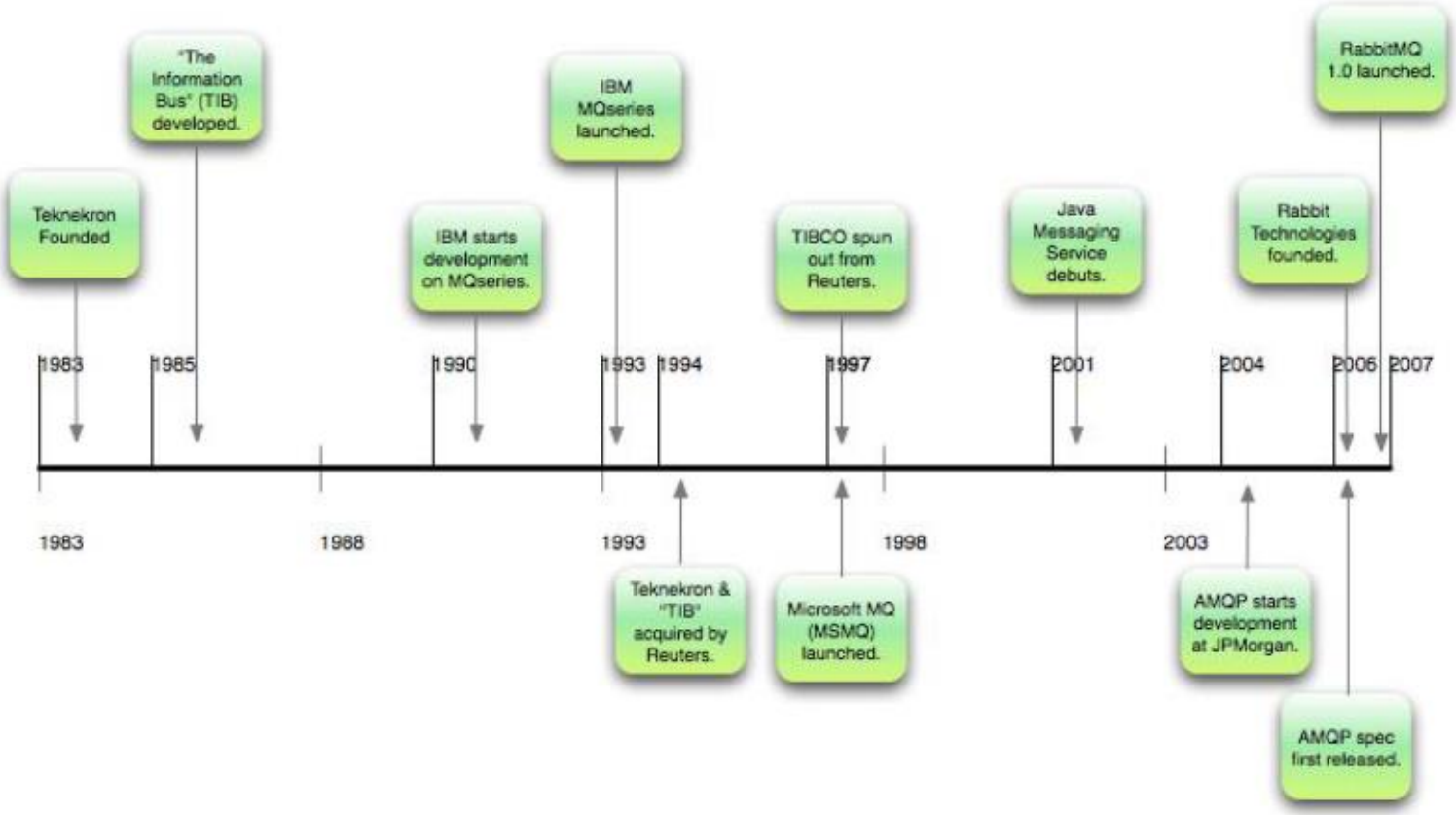
Agenda for Today

- RabbitMQ and AMQP (Advanced Message Queuing Protocol)
- AMQP terminology
- Implementations of selected messaging patterns

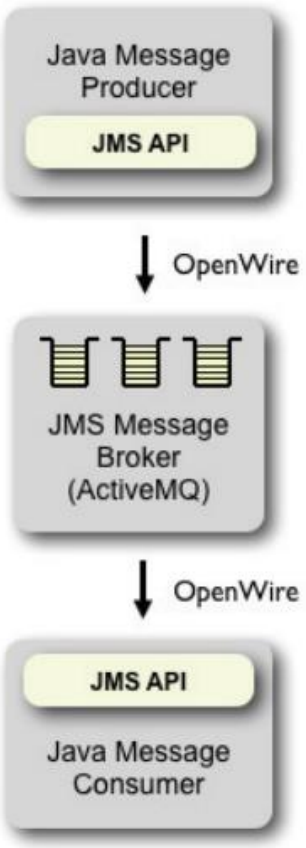
Messaging Implementations

- **AMQP**
  - Open standard over-the-wire protocol = (theoretical) full interoperability between vendor implementations
  - Language agnostic (Libs for 10+ languages)
- **JMS**
  - Only an API standard = no interoperability between vendors of JMS implementations
  - Pretty much bound to Java
- **MSMQ**
  - Microsoft’s own messaging system non-open, non-free standard
  - One vendor only
  - API's to few languages at codeplex.com etc.
  - Requires MS Windows to run

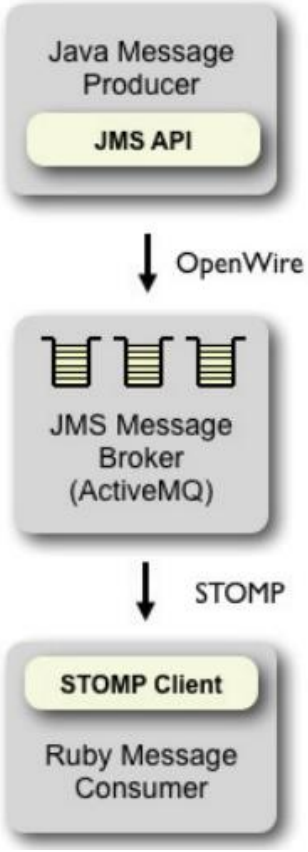
Short Timeline of Message Queueing



Connecting Messaging Systems

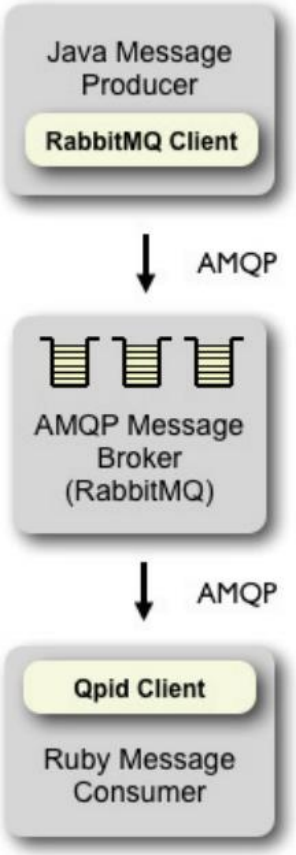


Same language
all OK



Different language
Different protocol
~Vendor lock-in

Different language
Different protocol
No vendor support



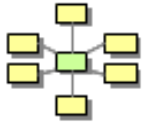
Same protocol
all OK

AMQP

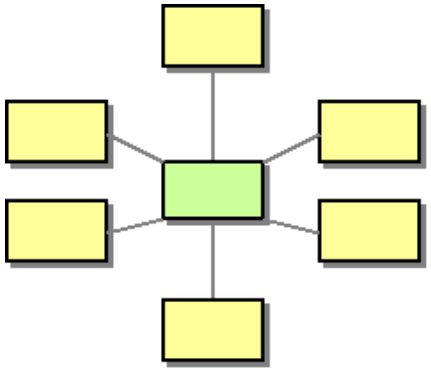
- Standard messaging protocol across platforms
- No standard API to program up against
- Rather, it is specification for industry standard wire-level binary protocol that describes how messages are structured and sent across the network

- So what client API and message broker should you use?
  - It doesn't matter ☺
  - Use AMQP compliant client library
  - Use AMQP compliant message broker

Message Broker (322)



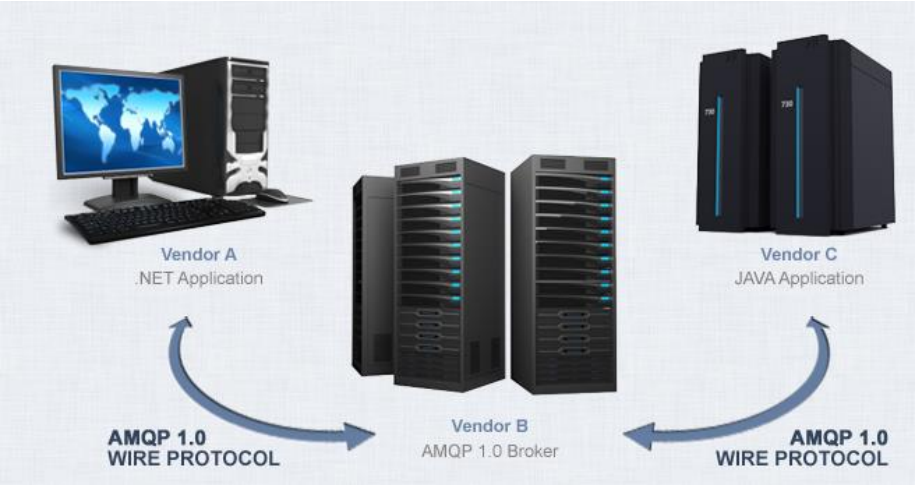
- How can you decouple the destination of a message from the sender and maintain central control over the flow of messages?
  - Use a central Message Broker that can receive messages from multiple destinations, determine the correct destination, and route the message to the correct channel.



- It's a hub-and-spoke architectural style
  - *Message Broker* isn't monolithic component. Internally, it uses the design patterns presented in Routing chapter
  - RabbitMQ is message broker

AMQP Message Broker Implementations

- Several broker implementations:
  - RabbitMQ by VMware
  - Qpid by Apache
  - MRG by Redhat (variant of Qpid)



- Several big business users:
  - OpenStack platform
  - JPMorgan
  - Deutche Börse (German stock exchange)
  - AT&T, Google and many more.

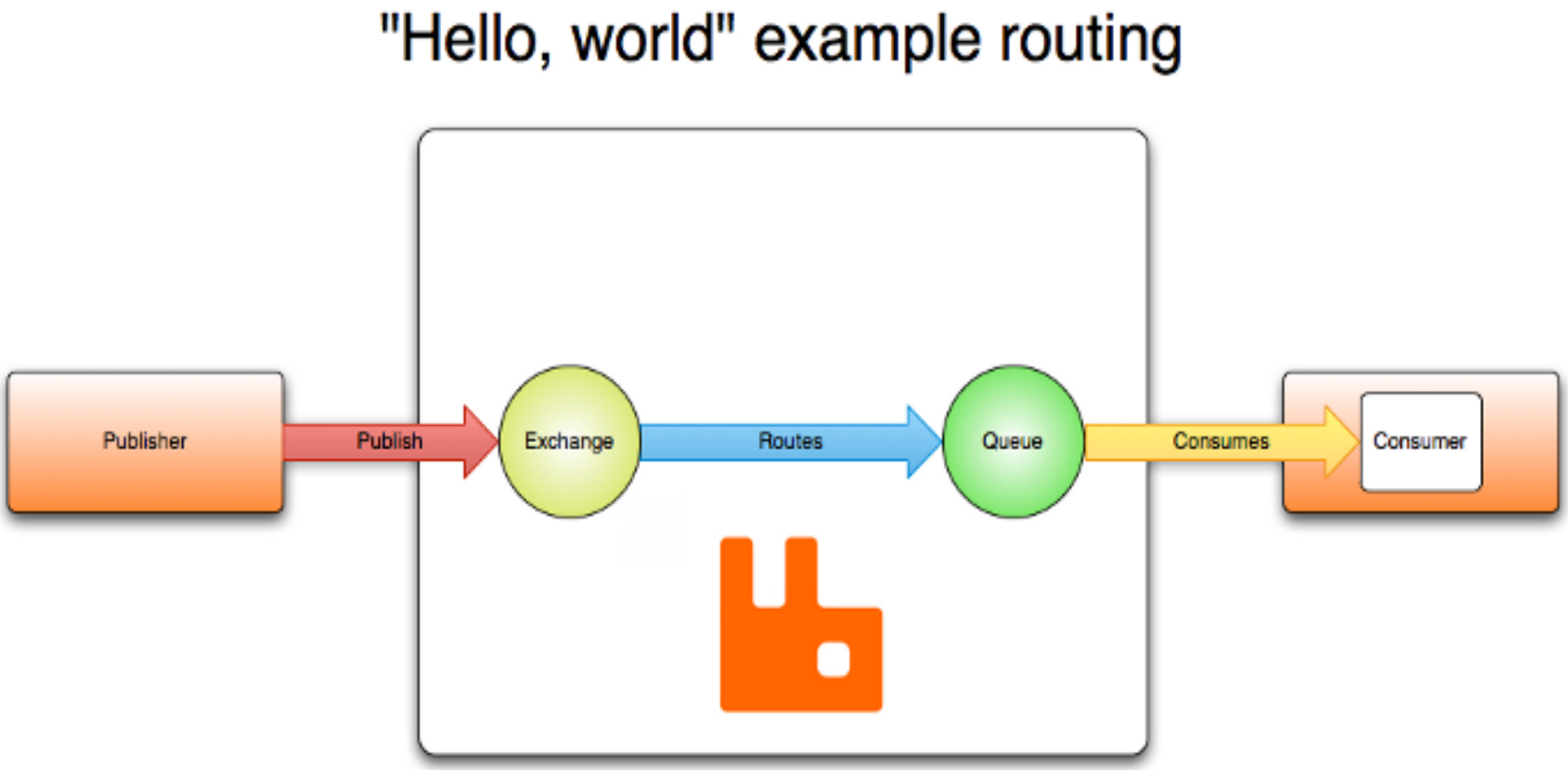
RabbitMQ

- Message Broker written in Erlang
- Can scale to over 15.000 messages pr. node pr. sec.
- Can have over 100 million concurrent queues

Fast like:

Chews messages like:

Terminology 1



Terminology 2

- In AMQP, you never send a message to a **queue**
- Producers publish messages to **exchanges**
- Consumers subscribe to **queues** to have messages delivered

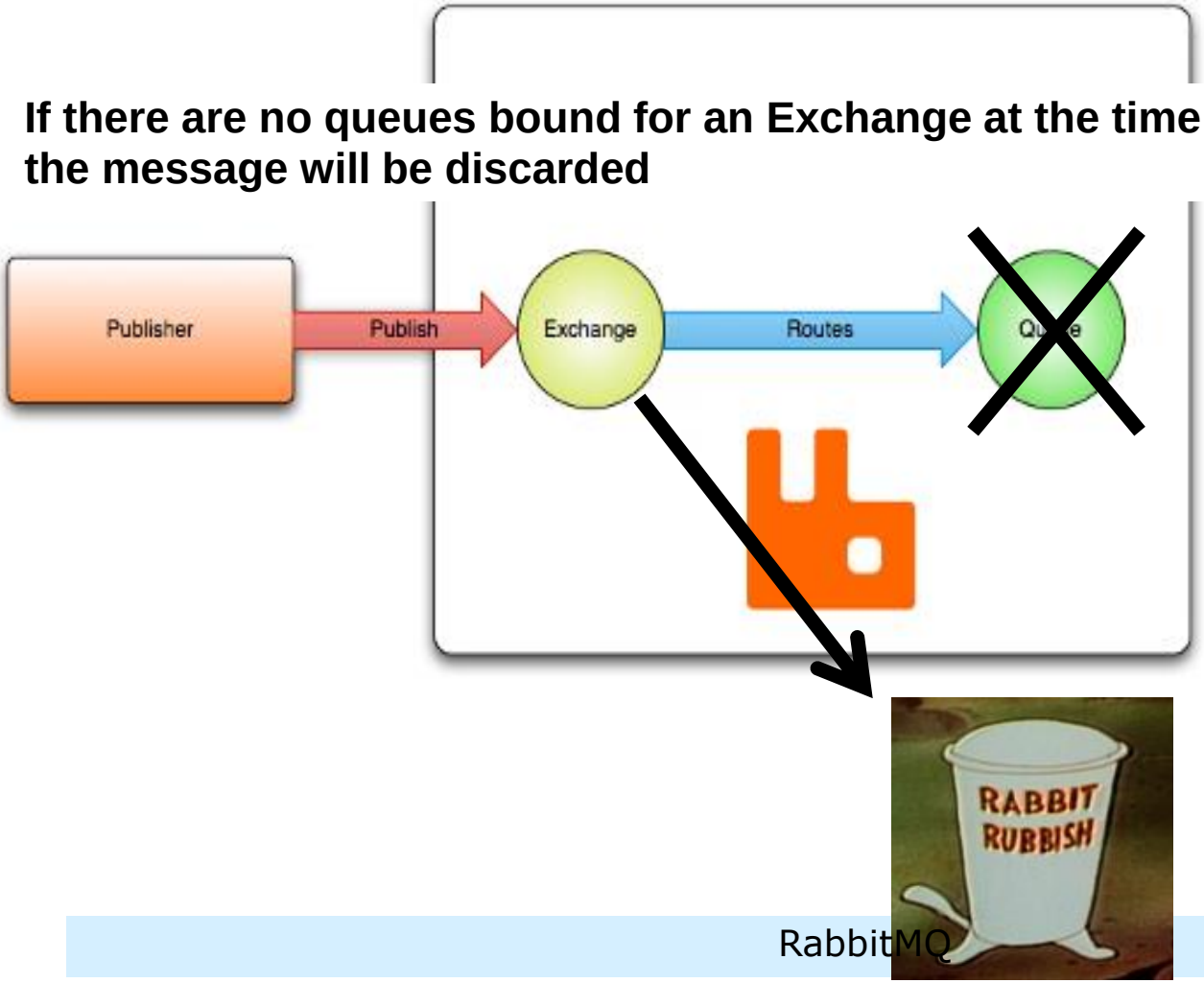
- But then how do messages get to a Queue?
  - Queues are connected to Exchanges via Bindings

- (Almost) Everything can be done in the programming language = no prior configuration with the tool.

Exchange without Queue

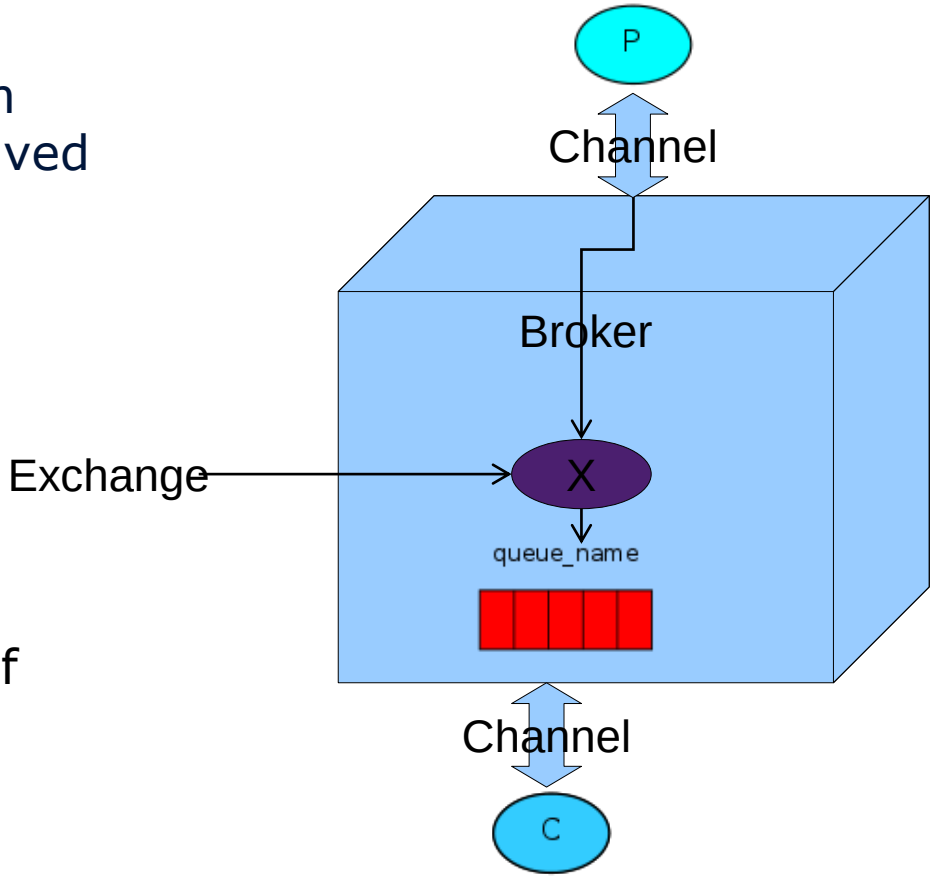
"Hello, world" example routing

If there are no queues bound for an Exchange at the time of publishing a message, the message will be discarded



Producer Consumer

- Messages are sent from **P**roducer app and received by **C**onsumer app



- The queue is a buffer of messages

Producer Consumer
Basic send


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

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

1. Make the factory
2. Point at the broker
3. Make conn. to broker
4. Make a channel
5. Make a queue through the channel
6. Send the message

Producer Consumer Basic send 2

- But wait a minute
- Aren't we actually sending through the queue here?
  - No, RabbitMQ uses the default exchange to bind the queue.

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

Producer Consumer Basic receive


```
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");

  QueueingConsumer consumer = new QueueingConsumer(channel);
  channel.basicConsume(QUEUE_NAME, true, consumer);

  while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());
    System.out.println(" [x] Received '" + message + "'");
  }
 }
}
```

Same as in sender

Make the consumer

Attach the consumer to the queue

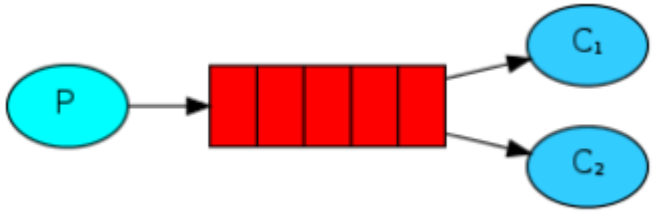
Polling consumer

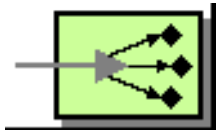
Demo

- Simple java program that sends, and receives
- Asynchronous version

- The Http interface to Rabbit
- Amount of messages Rabbit can transfer pr. second.
- Max sent: 8.000/s
- Max received: 25.000/s
  - Memory ~ the data that are being sent

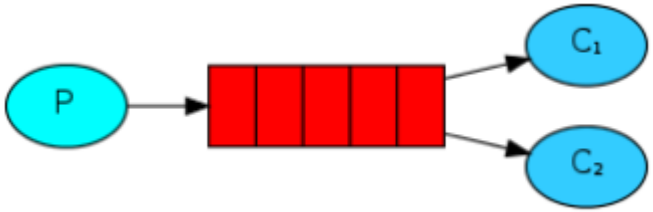
Work Queues (Competing Consumers)





- Distribution of work through several consumers
- Normally in round robin style
  - All consumers get equal amount of messages
  - Does not take into account that processing time could differ from message to message, leaving a large queue on one node, while idling others.

Work Queues – fair dispatch



- Prefetchcount = 1 as parameter in method channel.basicQos
  - Ensures that every consumer only gets stacked one message at a time

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

- Otherwise same codebase as with basic produce/consume
  - See example in “Work queues” tutorial: https://www.rabbitmq.com/tutorials/tutorial-two-python.html

- Notice auto acknowledgement in tutorial – also see next slide

Auto Acknowledgment

Auto acknowledgment set to false

- A way to be sure that every task that is sent, is also completed

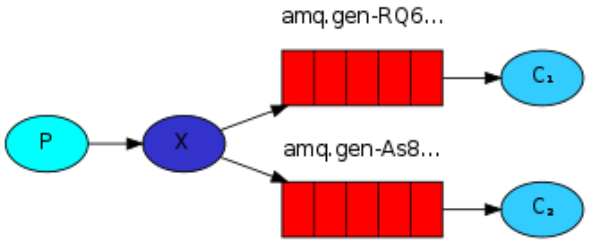
```
QueueingConsumer consumer = new QueueingConsumer(channel);
boolean autoAck = false;
channel.basicConsume("hello", autoAck, consumer);

while (true) {
  QueueingConsumer.Delivery delivery = consumer.nextDelivery();
  //...
  channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
}
```

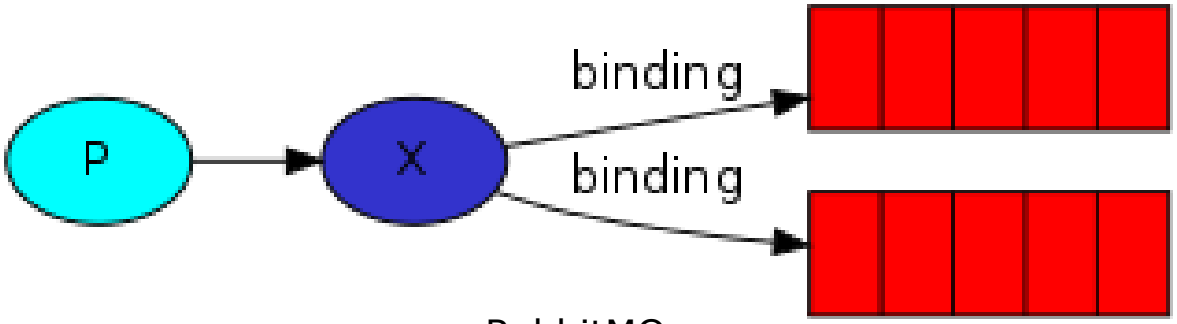
To reject a message call one of the following:

```
channel.basicNack(delivery.getEnvelope().getDeliveryTag(), false, true);
//or
channel.basicReject(delivery.getEnvelope().getDeliveryTag(), true);
```

Publish Subscribe

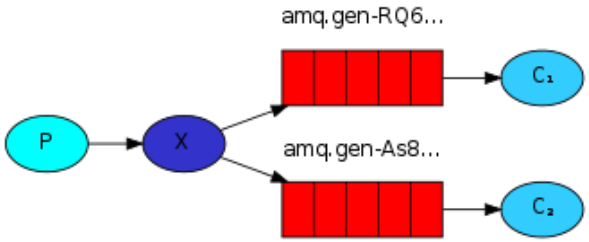


- **Fanout exchange** is used when we have one message to several consumers.
- Producer makes a name given exchange of the type Fanout
- Each Consumer makes a queue and binds the queue to the exchange.
- When the broker receives a message, it looks at the bindings for the exchange. If no bindings are made, it deletes the message. Otherwise it sends the message to the bounded queues.



RabbitMQ

Publish Subscribe - sender




```
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 message = getMessage(argv);

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

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

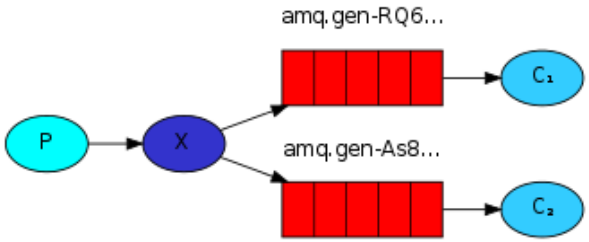
Same

Declare the exchange

Send the message

Not a word about the queues ☺

Publish Subscribe - receiver




```
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");

QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(queueName, true, consumer);

while (true) {
  QueueingConsumer.Delivery delivery = consumer.nextDelivery();
  String message = new String(delivery.getBody());

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

Make the connection to the exchange

Make a temp queue with auto generated name

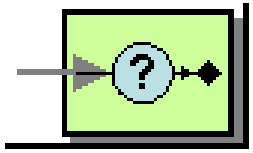
Bind the queue and exchange together

Demo

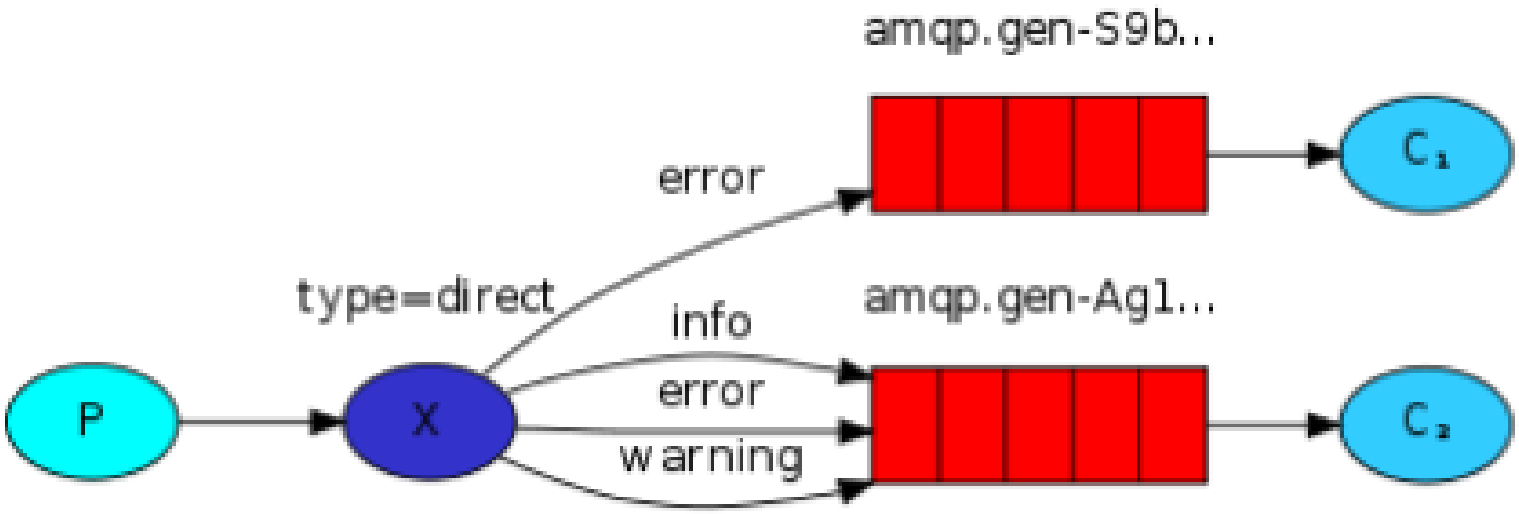
- **Pub/sub app**
- **Let’s look at the exchanges and queues in Rabbit http**

Break!

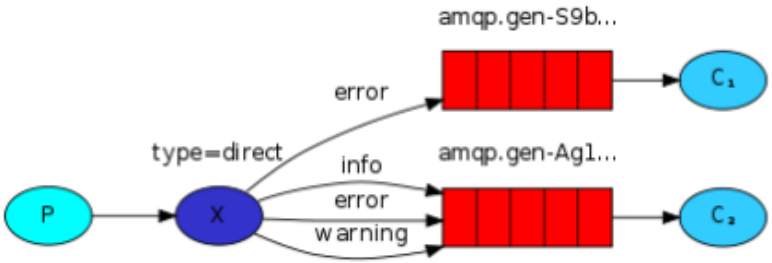
Routing (Selective Consumer) Key binding



- A way to selectively consume messages based on the routing key
- Distributes the messages based on the binding key. If queue has the same key K as message.routing_key R (K=R) then send the message to that queue.



Routing
Key binding




```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");

String severity = getSeverity(argv);
String message = getMessage(argv);

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

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

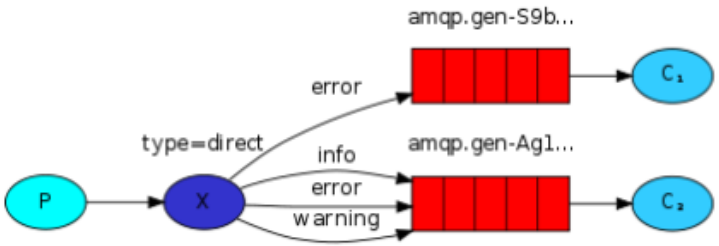
Make direct exchange

Make the routing key

Make the message

Send the message through the exchange with the given key

Routing
Key binding




```
private static final String EXCHANGE_NAME = "direct_logs";

 channel.exchangeDeclare(EXCHANGE_NAME, "direct");
 String queueName = channel.queueDeclare().getQueue();

 for(String severity : argv){
   channel.queueBind(queueName, EXCHANGE_NAME, severity);
 }

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

 QueueingConsumer consumer = new QueueingConsumer(channel);
 channel.basicConsume(queueName, true, consumer);

 while (true) {
   QueueingConsumer.Delivery delivery = consumer.nextDelivery();
   String message = new String(delivery.getBody());
   String routingKey = delivery.getEnvelope().getRoutingKey();

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

Make the exchange

Make an exchange binding
For every routing key

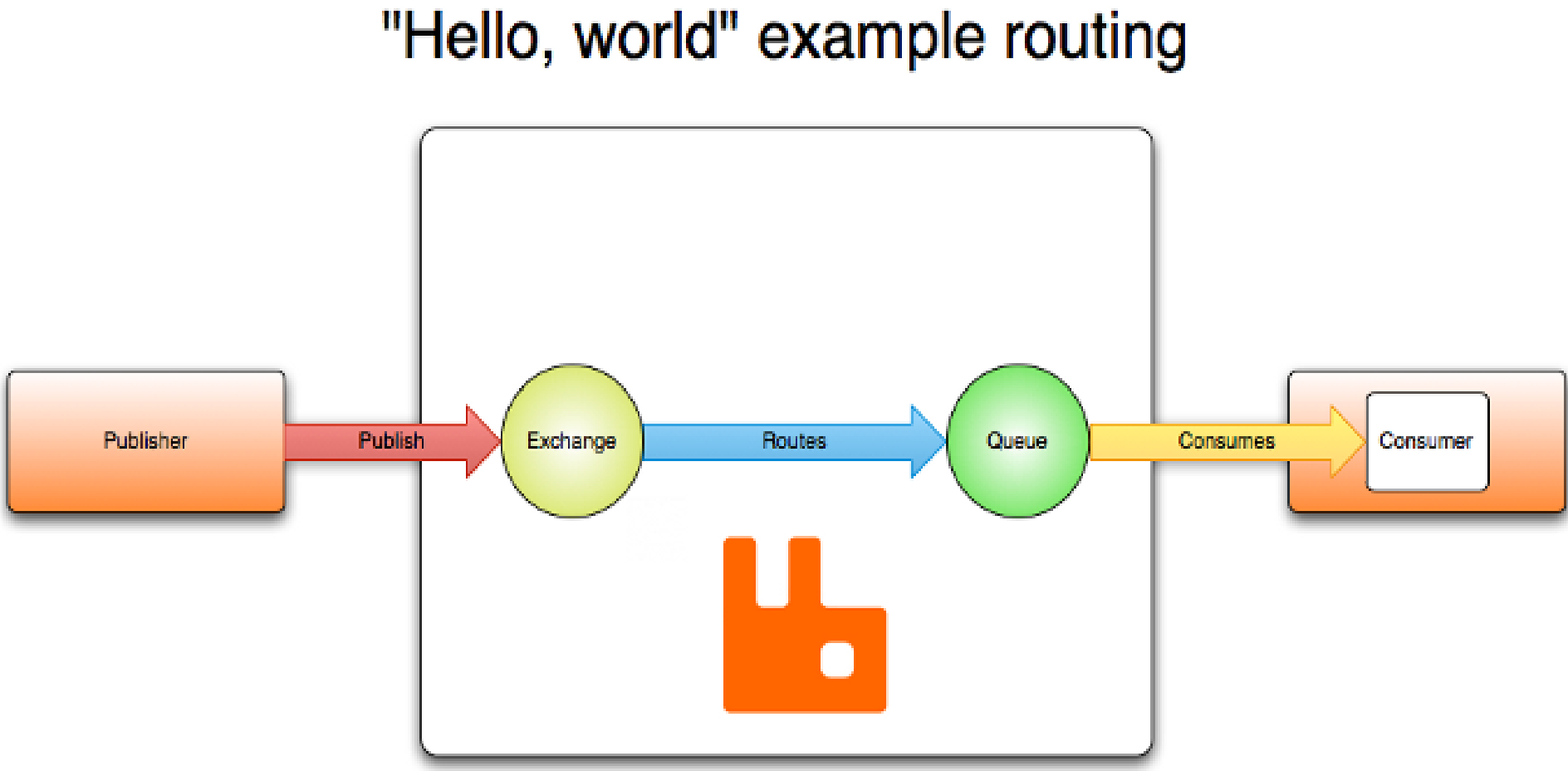
Routing key =
Severity level

Exchange Types

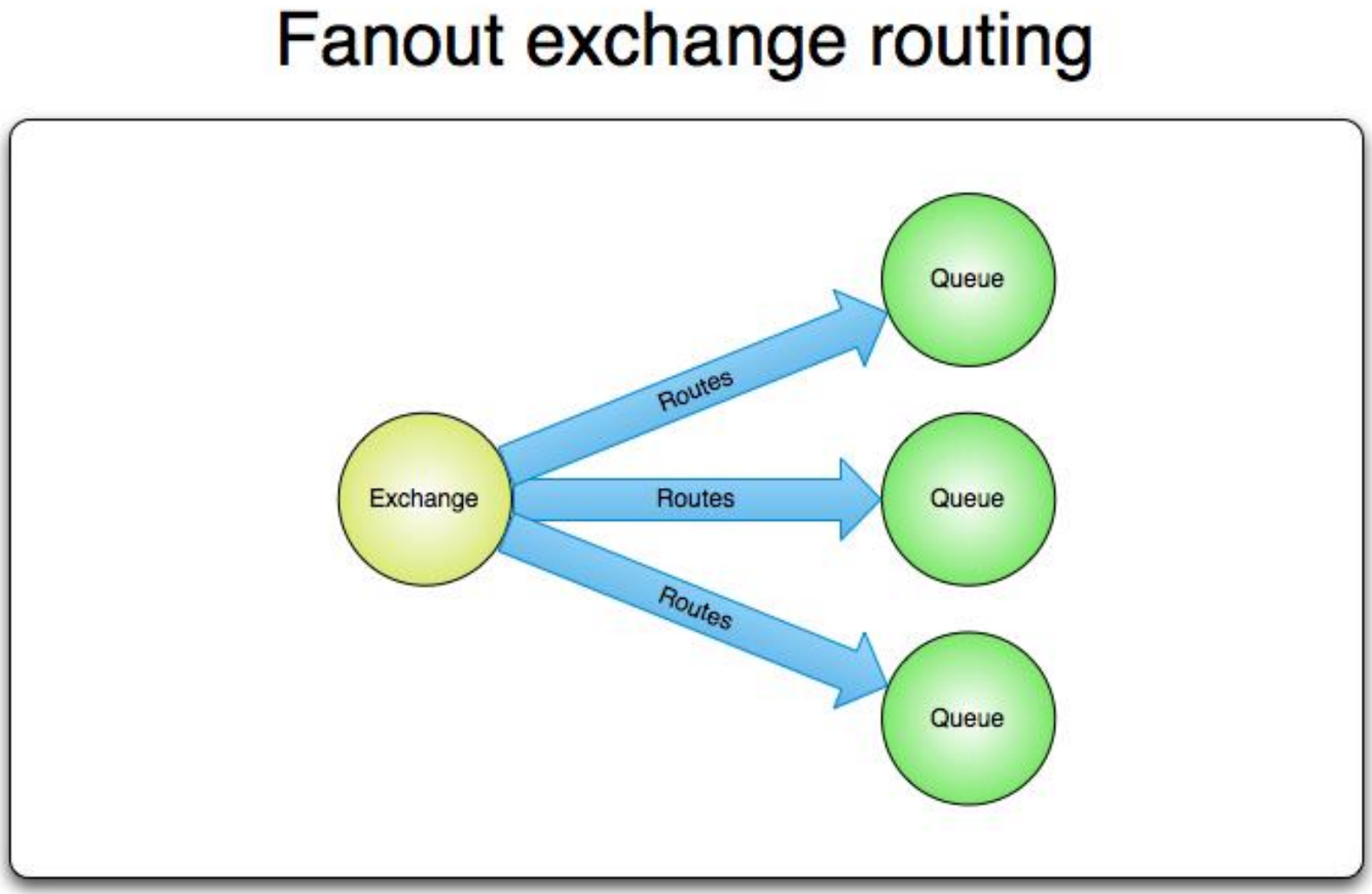
- **Fanout**
  - "Dumb" forward of messages to bound queues. Makes copies of the message
- **Default**
  - Takes the queue name and make it the binding key on the default exchange.
  - Is used in the producer/consumer and work queue examples.
- **Direct**
  - Distributes messages based on binding key.
  - If queue has the same key K as message.routing_key R (K=R), the message is sent to that queue.
- **Topic**
  - Covered very soon (today)

RECAP!

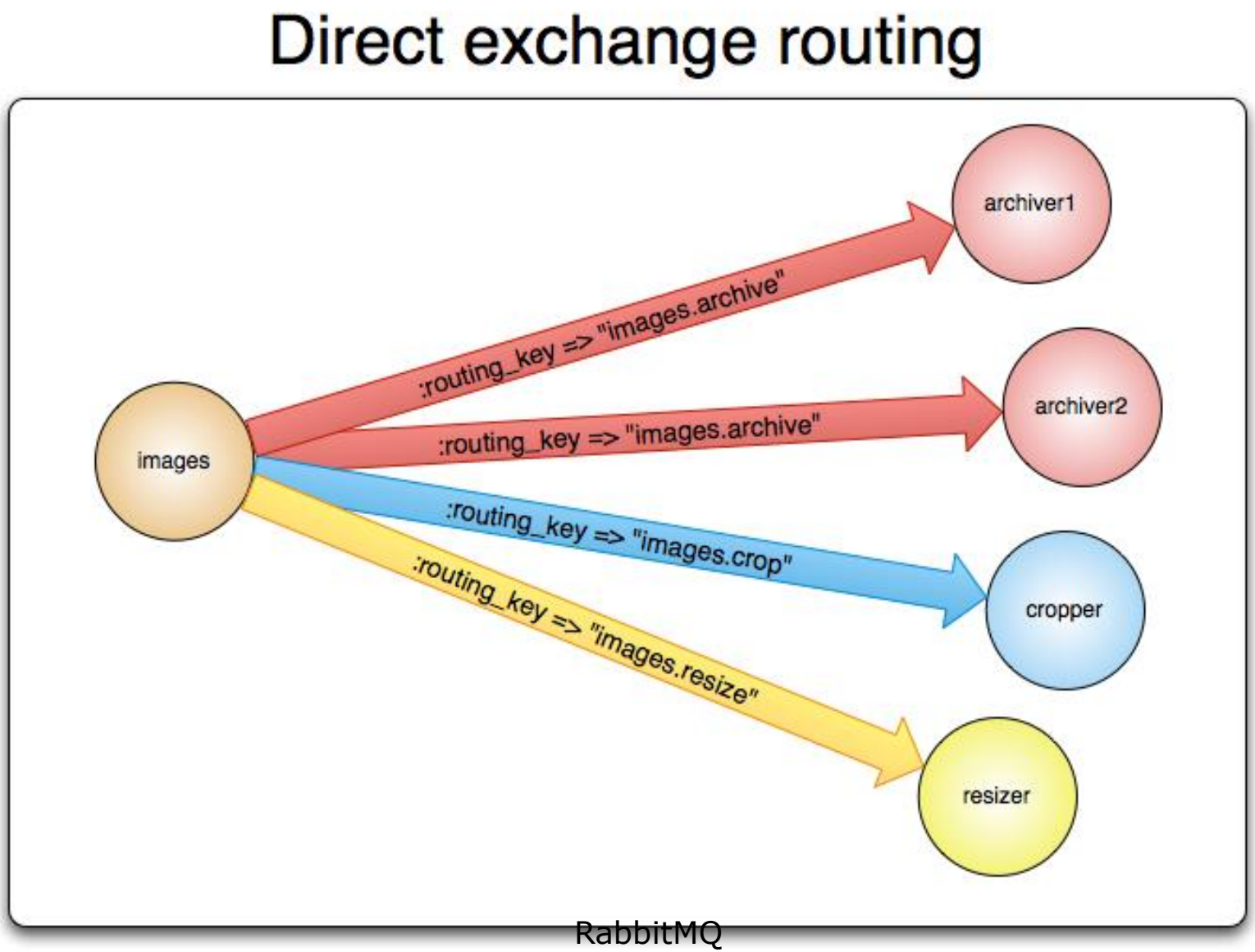
Terminology



Terminology

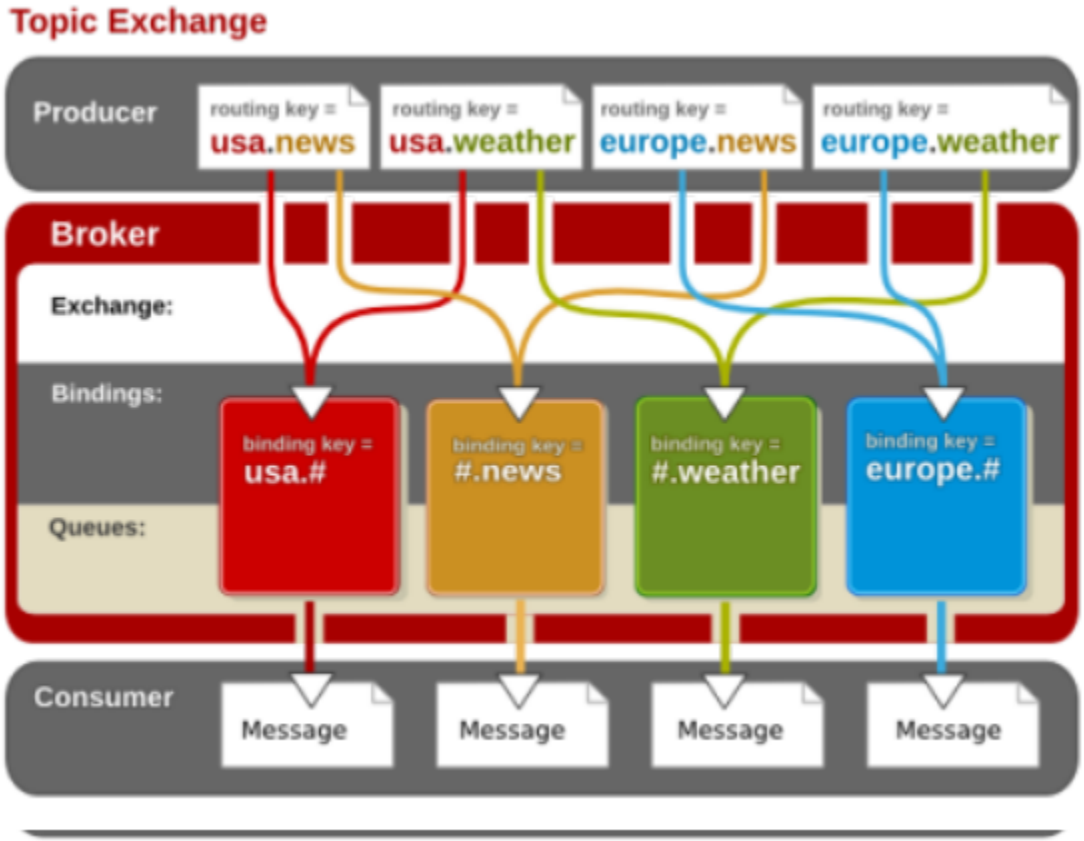


Terminology



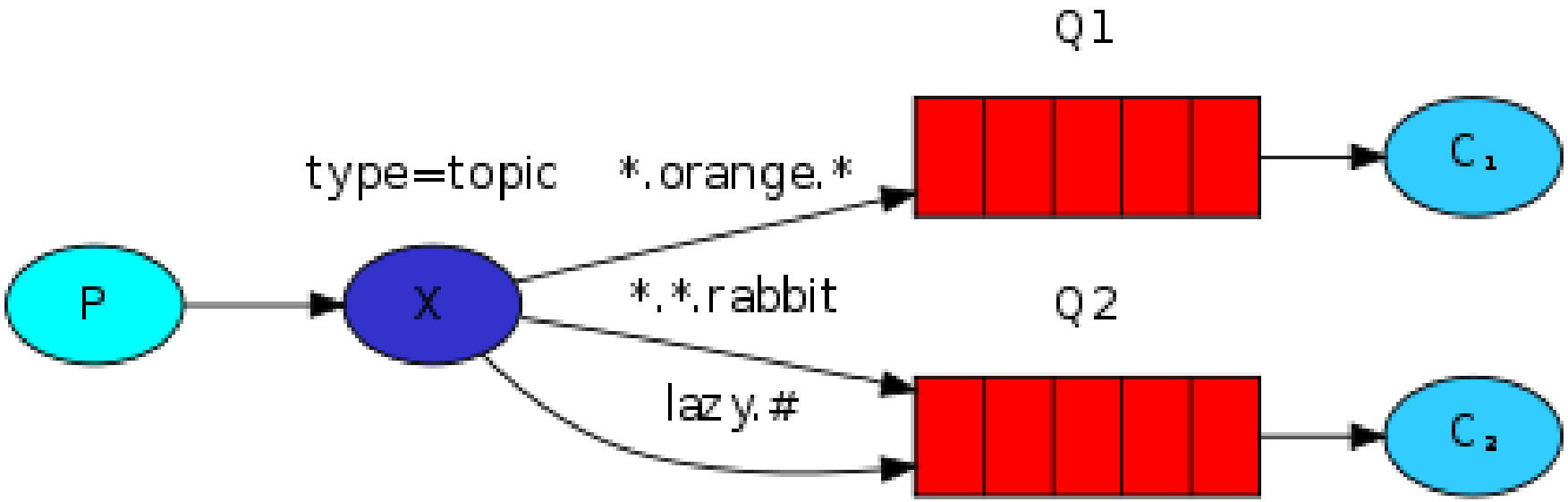
Topic Exchange

- Very fine grained way of distribute messages



Topic Exchange

- Very fine grained way of distributing messages, given a range of keywords, all separated with dots like “error.server2.tomcat”
- Has special routing keys:
  - * for one word
  - # for zero or more words



Topic Exchange Exercise

Who will get these messages?

quick.orange.rabbit
lazy.orange.elephant
quick.orange.fox
lazy.brown.fox
lazy.pink.rabbit
quick.brown.fox