

Pivotal Rabbit MQ

Three Day Workshop

Building Enterprise Applications using RabbitMQ

Version 3.6.1b

Pivotal™

Copyright Notice

Copyright © 2016 Pivotal Software, Inc. All rights reserved. This manual and its accompanying materials are protected by U.S. and international copyright and intellectual property laws.

Pivotal products are covered by one or more patents listed at <http://www.pivotal.io/patents>.

Pivotal is a registered trademark or trademark of Pivotal Software, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. The training material is provided “as is,” and all express or implied conditions, representations, and warranties, including any implied warranty of merchantability, fitness for a particular purpose or noninfringement, are disclaimed, even if Pivotal Software, Inc., has been advised of the possibility of such claims. This training material is designed to support an instructor-led training course and is intended to be used for reference purposes in conjunction with the instructor-led training course. The training material is not a standalone training tool. Use of the training material for self-study without class attendance is not recommended.

These materials and the computer programs to which it relates are the property of, and embody trade secrets and confidential information proprietary to, Pivotal Software, Inc., and may not be reproduced, copied, disclosed, transferred, adapted or modified without the express written approval of Pivotal Software, Inc.

Pivotal RabbitMQ Course Introduction

Objectives and Prerequisites

Logistics

- Participants list / registration
- Courseware / Internet access
- Working hours
- Lunch and breaks
- Toilets / Fire alarms / Emergency exits
- Other questions?
- Introductions



Course Objectives

- After this class, you should be able to do the following:
 - Describe messaging and asynchronous systems.
 - Describe AMQP (Advanced Message Queuing Protocol).
 - Install and administer RabbitMQ.
 - Access RabbitMQ from applications, using the Java binding to send and consume messages.
 - Describe reliable messaging development by using RabbitMQ APIs.
 - Describe RabbitMQ cluster and high availability.
 - Describe RabbitMQ's performance and security.

Course Introduction

- Course designed for Windows and Linux
 - 50% theory
 - 50% labs
- Memory keys:
 - Lab materials and documentation

Course Prerequisites

- Basic knowledge of core Java
- Working knowledge in Spring Tool Suite (STS) / Eclipse
- This is not an advanced course – no knowledge of RabbitMQ or AMQP is expected

Course Outline – Day 1

Module	Description
Introduction to Messaging and AMQP	Advantages of messaging in IT systems, overview of the AMQP model.
RabbitMQ Installation	The RabbitMQ project and its main features, how to install and setup an instance, main configuration settings, plugin mechanism. Lab.
Development and Integration	Accessing RabbitMQ from applications, using the Java client to send and consume messages. Lab.

Course Outline – Day 2

Module	Description
Reliable Messaging Development	What can go wrong in messaging applications, how to safely send and receive messages using persistence, transactions, and acknowledgments. Lab.
Clustering	Understanding RabbitMQ clustering, setting up a cluster, load balancing, network partition handling strategies. Lab.
High Availability	Node failures, mirrored queues, slaves synchronization, failover handling for the client. Lab.

Course Outline – Day 3

Module	Description
Plugins	Federation and shovel, using LDAP for authentication, the STOMP plugin.
Performance	Factors that impact performance, flow control, message paging, best practices. Lab.
Spring AMQP	Using Spring AMQP for application development. Lab.
Security	Virtual hosts, securing communications between client and broker, protecting exchanges and queues. Lab.
Operations and Monitoring	Production settings and tuning. Tips and techniques to monitor your RabbitMQ cluster.

Let's Get Started!

Introduction to Messaging and AMQP

Agenda

- **Messaging and Asynchronous Systems**
- Why JMS isn't enough
- Introduction to AMQP
- RabbitMQ case studies

What is Messaging?

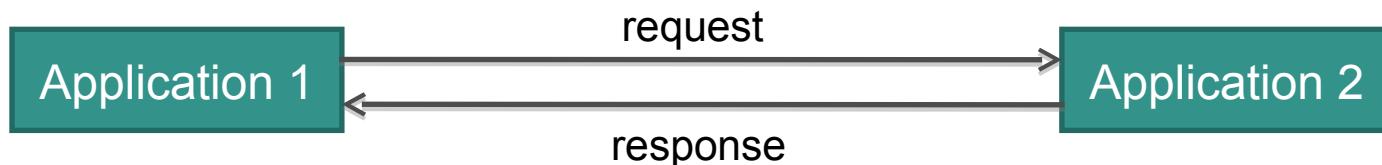
- Messaging is a way to make applications / systems communicate
- Messaging is sometimes called an "integration style"
- Messaging eases decoupling between applications
 - Applications can evolve independently.
- Messaging is often referred to as "Message Oriented Middleware" (MoM)
- Messaging server typically called a *broker*
 - Broker ensures reliable dispatching of messages

What is a Message?

- Messages consist of a payload and multiple headers
- Payload is the actual content to exchange
 - Can be a string, a byte array (binary serialized object)
 - Often serialized with data exchange format (JSON, XML)
- Headers are metadata
 - Key/value pairs
 - Can be technology-specific or custom
 - E.g., routing (where to go, whom to answer to, etc.) ...
- Messaging technologies usually come with their subtleties
 - An AMQP message can have several kinds of metadata (header, properties, delivery annotations), a body, and even a footer!

Synchronous vs. Asynchronous

- When an application wants to talk to another application, it can send a message to it ...
 - Synchronously



- Asynchronously



Synchronous vs. Asynchronous

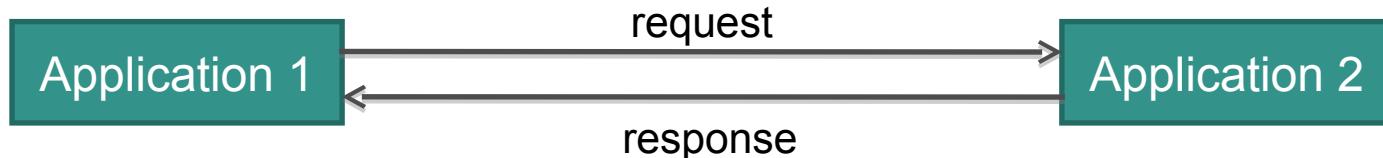
- Real-world comparison:
 - Synchronous = phone
 - Asynchronous = SMS

NOTE

Asynchronous messaging decouples the senders and receivers, more than synchronous remote method calls.

Synchronous Messaging

- Sending application must know about receiving application
 - Host, port, protocol, endpoint
- Sending application is blocked until receiving application answers
- What happens if the receiving application doesn't respond?
 - Wait?
 - Crash?
- HTTP is an example of synchronous messaging



Asynchronous Messaging



- Sending application knows only about the broker
- Sending application can "fire and forget" if it doesn't need a response
 - Request / reply also supported
- Receiving application consumes messages whenever it wants
 - Constant polling, notification, batch de-queuing
 - It consumes messages rather than receives them
- JMS and AMQP are examples of asynchronous messaging

Decoupling

- The broker decouples the sender and the receiver...
 - Spatially
 - They don't need to be co-located
 - Temporally
 - No need of immediate responses
 - Processing can happen in the background
 - Receiver doesn't have to be up when message is sent
 - Logically
 - Sender and receiver don't know about each other
 - Broker can use advanced routing

Use Cases

- Simple producer – consumer
 - Send message for further processing
 - E.g., a web app places an order for further processing
- Request / reply
 - Send message and wait for response
 - E.g., to throttle or scale processing on the consumer side
- Publish / subscribe
 - Send message for multiple consumption
 - E.g., order sent to inventory and billing systems

Simple Producer – Consumer



Simple Producer – Consumer

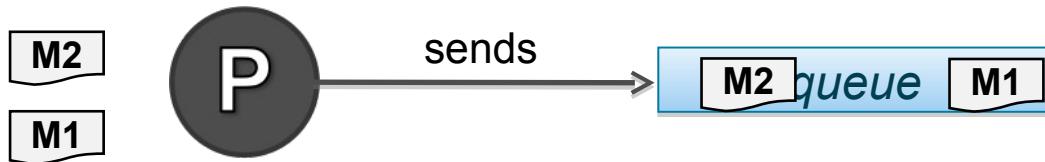
Temporal Decoupling – 1



*Consumer goes down
for upgrade*

Simple Producer – Consumer

Temporal Decoupling – 2



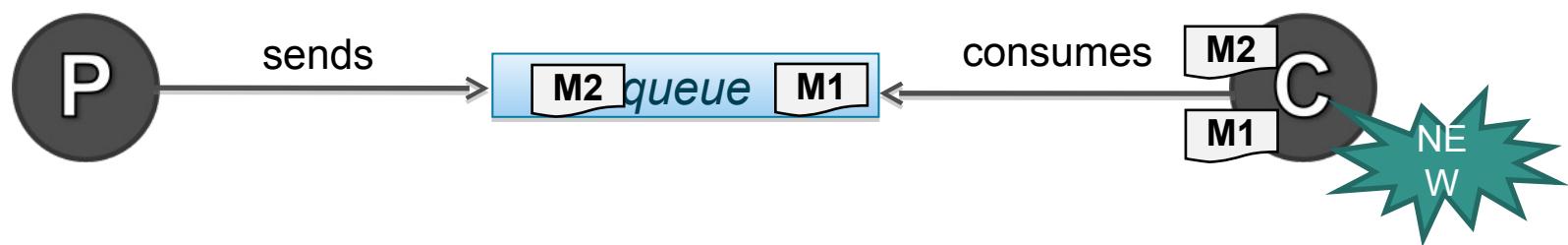
*Consumer goes down
for upgrade*

Simple Producer – Consumer

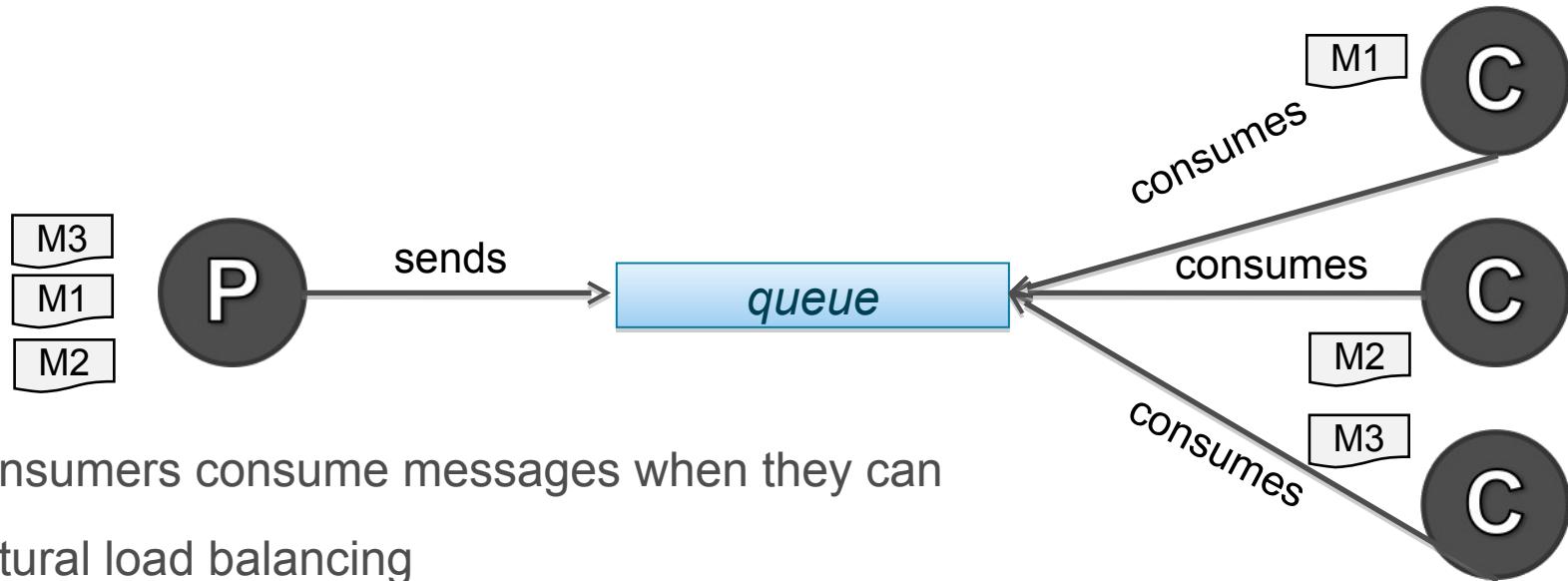
Temporal Decoupling – 3

New version

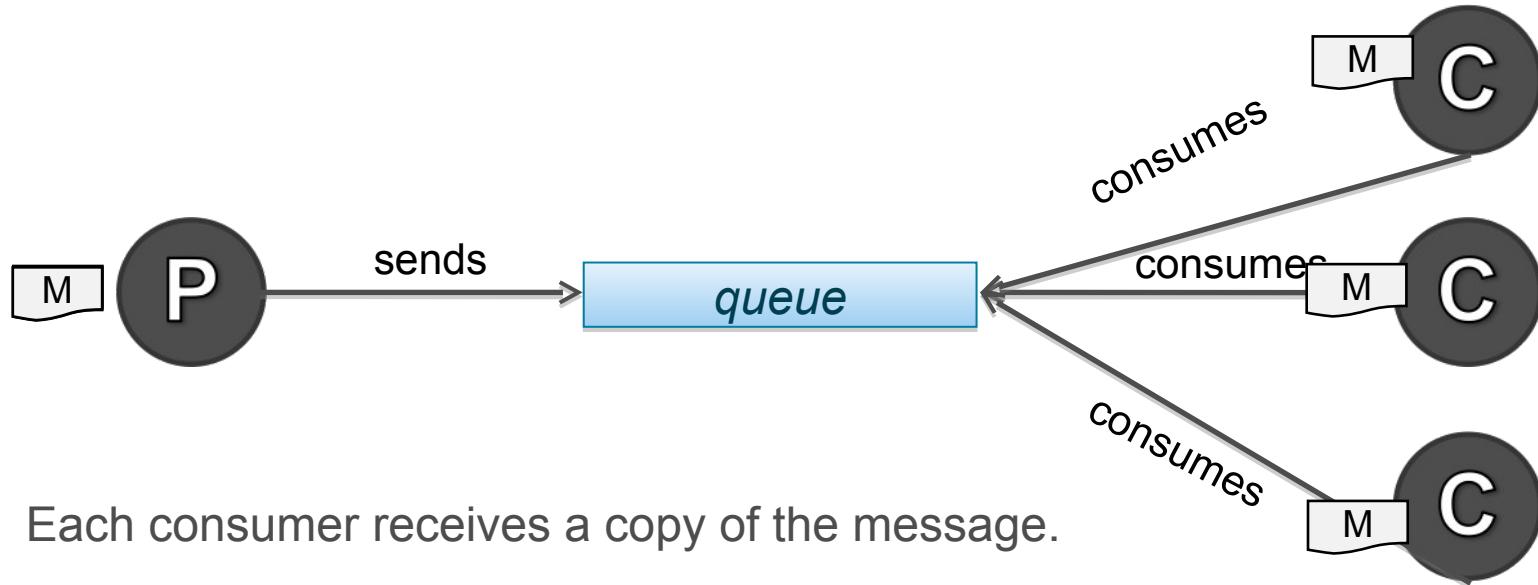
of consumer comes up



Simple Producer – Multiple Consumers

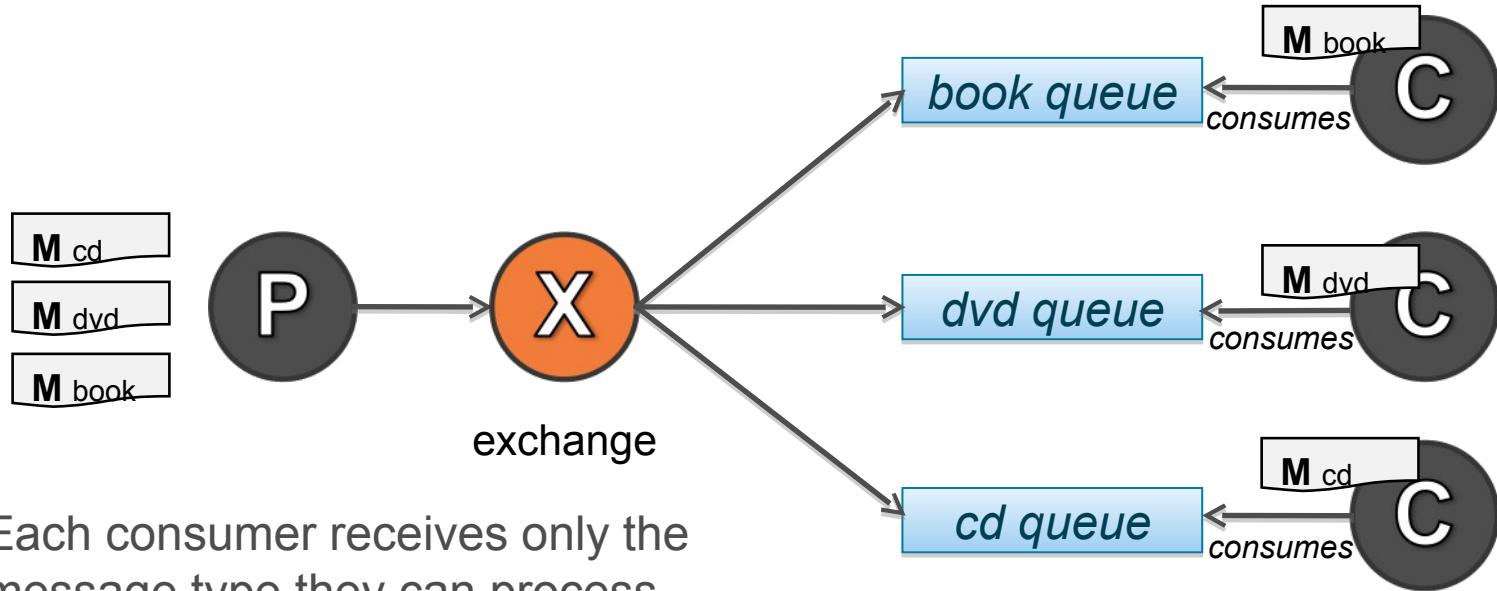


Publish / Subscribe



- Each consumer receives a copy of the message.

Routing



- Each consumer receives only the message type they can process.

Pros and Cons of Messaging

- Pros
 - Scalability
 - Loose coupling
- Cons
 - Complexity
 - Broker can be a single point of failure

Messaging in the Cloud

- Asynchronous messaging is an ideal integration tool for cloud deployments
 - Elastic
 - Scalable
 - Robust
 - Decoupled
- RabbitMQ is the preferred mechanism for integrating Pivotal Cloud Foundry applications



Cloud
AMQP

Pivotal™

Agenda

- Messaging and Asynchronous Systems
- Why JMS isn't enough
- Introduction to AMQP
- RabbitMQ case studies

JMS

- JMS stands for Java Message Service
- JMS is part of Java Enterprise Edition
- JMS is a Java API.
- JMS solves the vendor lock-in for Java apps
- JMS supports
 - persistent messages
 - reliable messaging
 - distributed transactions (through JTA and XA)
- JMS doesn't provide advanced routing rules
 - One can use selectors on a destination

NOTE

JMS doesn't standardize the network communication, brokers do whatever they want on the wire.

Why JMS Isn't Enough

- JMS = Java Message Service
- Everything is in the name: Java
- Sender and receiver become coupled through the language
- Limitation is manageable in one enterprise system
- Limits interoperability between different companies' systems

NOTE

Support for other languages/platforms is broker specific.

Giving Up On JMS?

- JMS is a mature, stable, and reliable standard
- JMS is Java specific
- AMQP is standardized at the protocol level ...
- ... and AMQP broker usually provides JMS bridges!
 - Can help upgrade to AMQP by keeping the same application code

Agenda

- Messaging and Asynchronous Systems
- Why JMS isn't enough
- **Introduction to AMQP**
- RabbitMQ case studies

AMQP

- AMQP stands for Advanced Message Queuing Protocol
- AMQP
 - Aims to provide an open standard for messaging
 - Enables complete interoperability for messaging middleware
 - Defines the network protocol and the semantics of broker services
- AMQP is open, interoperable, and platform agnostic

NOTE

AMQP is an application protocol, like HTTP and SMTP.

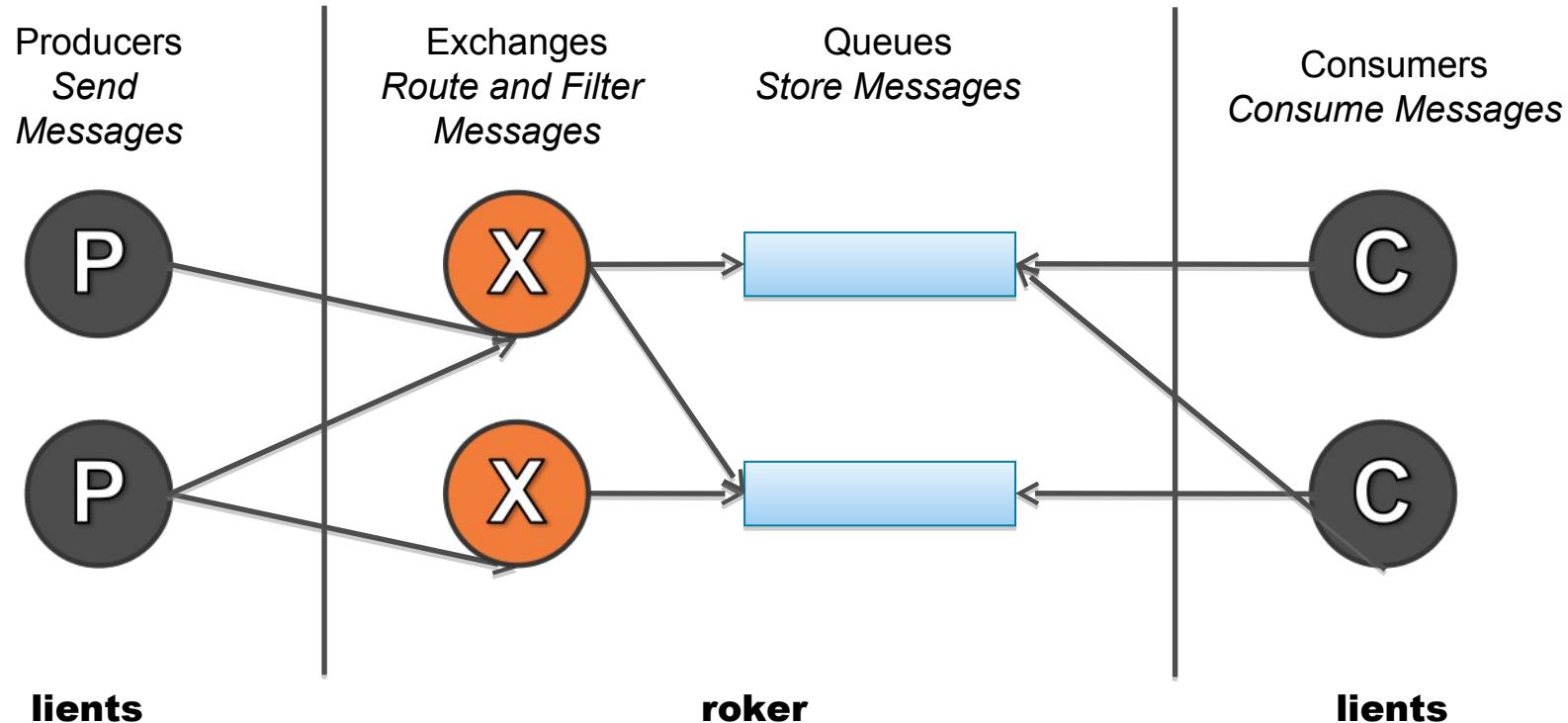
History of AMQP

- Development started in 2004 by JP Morgan and iMatix
- AMQP Working Group was born when other companies joined the effort
 - See members at <http://www.amqp.org/about/members>
- Specification version 1.0 final in October 2011
 - Downloadable at <http://www.amqp.org/resources/download>
- AMQP originated in the finance industry ...
- ... but it addresses a large range of middleware problems

NOTE

This training focuses on AMQP 0.9.1, the most popular and widespread version.

The AMQP Model (v0.9.1)



JMS / AMQP Comparison

	JMS	AMQP
Defined by	Java Community Process	AMQP Working Group
Scope	Java API	Application protocol
API	Yes	No
Interoperable	No (broker specific)	Yes
Distributed transactions	Yes	Yes*
Routing	No	Yes

NOTE

* RabbitMQ implements AMQP but doesn't support distributed transactions.

Agenda

- Messaging and Asynchronous Systems
- Why JMS isn't enough
- Introduction to AMQP
- **RabbitMQ case studies**

RabbitMQ case study: New York Times

- System provides subscription services for news, video feeds, etc.
- Dozens of RabbitMQ instances
- Deployment across 6 AWS zones
- Upon launch, the system autoscaled to 500 K users
- Connection times stayed stable around 200 ms

Source

<http://lists.rabbitmq.com/pipermail/rabbitmq-discuss/2014-January/032920.html>

RabbitMQ case study: Travis CI

- Hosted continuous integration service
- Build logs are forwarded to RabbitMQ for live display
- Messages contain an incrementing counter to identify ordering
- RabbitMQ clusters are hosted on CloudAMQP
- Travis CI handles 74 K builds per day

Source

<https://blog.pivotal.io/pivotal/case-studies/continuous-integration-scaling-to-74000-builds-per-day-with-travis-ci-rabbitmq>

Summary

- Asynchronous messaging facilitates decoupling between systems
- Common messaging patterns:
 - Simple producer/consumer
 - Request/reply
 - Publish-subscribe
- AMQP is an open standard for messaging
 - A binary network protocol specification
 - Not just a Java interface specification like JMS!

RabbitMQ Installation

Agenda

- RabbitMQ Installation
- Startup at Boot
- Basic Administration

Capabilities

- The most popular AMQP implementation
 - Implements AMQP 0-8, AMQP 0-9-1, AMQP 1.0
 - Implements STOMP, MQTT
- Written in Erlang using OTP (Open Telecom Platform)
 - Designed for scale and fault tolerance
- Fully open-source under the MPL (Mozilla Public License)
- Consists of:
 - An AMQP message broker
 - Client libraries for multiple languages and platforms

Installation overview

- Use official RabbitMQ packages
 - Not OS default packages, they can be old
- Erlang VM is required
 - You may have to install it yourself, e.g. CentOS family
- For latest instructions, refer to <http://www.rabbitmq.com/download.html>

Installation on Fedora / RedHat

- Install a recent Erlang runtime system, e.g. from Erlang Solutions

```
$ wget http://packages.erlang-solutions.com/erlang-solutions-1.0-1.noarch.rpm  
$ sudo rpm -Uvh erlang-solutions-1.0-1.noarch.rpm  
$ sudo yum install erlang
```

- Install RabbitMQ

- Get the most recent RPM directly from the official site
<http://www.rabbitmq.com/download.html>

```
$ wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.1/rabbitmq-server-3.6.1-  
1.noarch.rpm  
$ sudo rpm --import https://www.rabbitmq.com/rabbitmq-signing-key-public.asc  
$ sudo yum install rabbitmq-server-3.6.1-1.noarch.rpm
```

Installation on Debian / Ubuntu

- Add the RabbitMQ official repository to your /etc/apt/sources.list:

```
$ echo 'deb http://www.rabbitmq.com/debian/ testing main' |  
  sudo tee /etc/apt/sources.list.d/rabbitmq.list
```

- Add the public key to your trusted key list using apt-key(8):

```
$ wget -O- https://www.rabbitmq.com/rabbitmq-signing-key-public.asc |  
  sudo apt-key add -
```

- Update the packages list and install RabbitMQ (use sudo if needed)

```
$ sudo apt-get update  
$ sudo apt-get install rabbitmq-server
```

Installation on Windows

- Download and install Erlang from <http://www.erlang.org/download.html>
- Download and install RabbitMQ from
<http://www.rabbitmq.com/install-windows.html>
- The installer will install the broker as a *service*, and create a “RabbitMQ Server” folder in the Start menu.

Installation on Mac OS X

- Installation package includes Erlang.
- Download and install RabbitMQ from
<http://www.rabbitmq.com/install-standalone-mac.html>
- MacPorts and Homebrew packages also available
 - You can use either the single download or *Homebrew* in the lab

Starting and Stopping RabbitMQ

- Running RabbitMQ is straightforward
 - Assuming the bin directory is in the PATH

```
$ rabbitmq-server
```

- Running the daemon in the background:

```
$ rabbitmq-server -detached
```

- Stopping RabbitMQ

- Assuming the bin directory is in the PATH

```
$ rabbitmqctl stop
```

NOTE

By default Rabbit is started on MS Windows as a “service”, so you *wouldn’t* manage it this way and *rabbitmqctl stop* is ignored. See later slide on Windows Service.

RabbitMQ Startup Log Sample (first lines)

```
=INFO REPORT==== 6-Apr-2016::12:07:36 ===
Starting RabbitMQ 3.6.1 on Erlang 18.3
Copyright (C) 2007-2016 Pivotal Software, Inc.
Licensed under the MPL. See http://www.rabbitmq.com/

=INFO REPORT==== 6-Apr-2016::12:07:36 ===
node          : rabbit@vm-master1
home dir      : /var/lib/rabbitmq
config file(s) : /etc/rabbitmq/rabbitmq.config (not found)
cookie hash   : 2k4H5hGmec00T+6wzMtLhQ==
log           : /var/log/rabbitmq/rabbit@vm-master1.log
sasl log      : /var/log/rabbitmq/rabbit@vm-master1-sasl.log
database dir  : /var/lib/rabbitmq/mnesia/rabbit@vm-master1

=INFO REPORT==== 6-Apr-2016::12:07:37 ===
Memory limit set to 397MB of 993MB total.

=INFO REPORT==== 6-Apr-2016::12:07:37 ===
Disk free limit set to 50MB

=INFO REPORT==== 6-Apr-2016::12:07:37 ===
Limiting to approx 924 file handles (829 sockets)
```

Checking RabbitMQ Status

- To check the status of the RabbitMQ server:

```
$ rabbitmqctl status
```

- Should produce output similar to the following:

```
Status of node 'rabbit@vm-master1' ...
[{pid,14400},
 {running_applications,[{rabbit,"RabbitMQ","3.6.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.13.3"},
                        {os_mon,"CPO  CXC 138 46","2.4"},
                        {xmerl,"XML parser","1.3.10"},
                        {rabbit_common,[],"3.6.1"},
                        {ranch,"Socket acceptor pool for TCP protocols.",
                           "1.2.1"},
                        {sasl,"SASL  CXC 138 11","2.7"},
                        {stdlib,"ERTS  CXC 138 10","2.8"},
                        {kernel,"ERTS  CXC 138 10","4.2"}]},
 {os,{unix,linux}},
 {erlang_version,"Erlang/OTP 18 [erts-7.3] [source-d2a6d81] [64-bit] [async-threads:64] [hipe]
 [kernel-poll:true]\n"},
 ...]
```

Agenda

- RabbitMQ Installation
- Startup at Boot
 - Red Hat Linux, Debian Linux, MS Windows, MacOS
- Basic Administration



Running RabbitMQ on Boot in RedHat

- Install the RabbitMQ init script in the different run-levels

```
$ chkconfig rabbitmq-server on
```

- Starting the daemon manually:

```
$ service rabbitmq-server start
```

- Stopping the daemon manually:

```
$ service rabbitmq-server stop
```



CentOS

Pivotal™

Running RabbitMQ on Boot in Debian



- Install the RabbitMQ init script in the different run-levels

```
$ update-rc.d rabbitmq-server defaults
```

- Starting the daemon manually:

```
$ /etc/init.d/rabbitmq-server start
```

- Stopping the daemon manually:

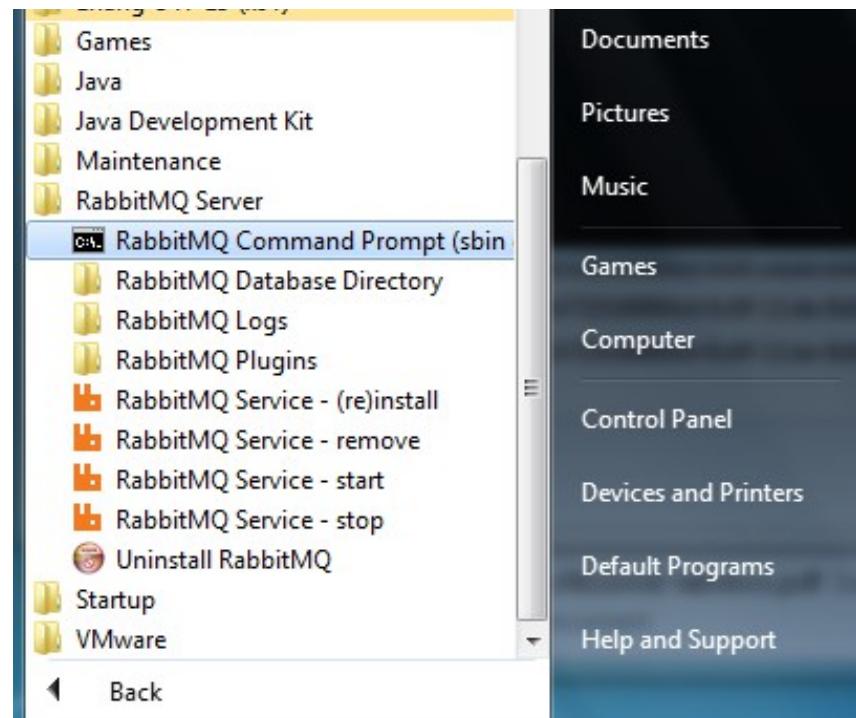
```
$ /etc/init.d/rabbitmq-server stop
```



Running Rabbit as a Windows Service – 1



- Installation *automatically* does this.
- Three options to control
 - Use Windows Services panel (next slide)
 - Use `rabbitmq-service start` and `rabbitmq-service stop`
 - Use shortcuts in the Start menu





Running Rabbit as a Windows Service – 2

Computer Management

Name Description Status Startup Type Log On As

Name	Description	Status	Startup Type	Log On As
PNRP Machine Name Publication S...	This service ...	Manual	Local Service	
Portable Device Enumerator Service	Enforces gr...	Manual	Local Syste...	
Power	Manages p...	Started	Automatic	Local Syste...
Print Spooler	Loads files t...	Started	Automatic	Local Syste...
Problem Reports and Solutions Co...	This service ...	Manual	Local Syste...	
Program Compatibility Assistant Se...	This service ...	Started	Automatic	Local Syste...
Protected Storage	Provides pr...	Manual	Local Syste...	
Quality Windows Audio Video Experi...	Quality Win...	Manual	Local Service	
RabbitMQ	Multi-proto...	Started	Automatic	Local Syste...
Remote Access Auto Connection ...	Creates a co...	Manual	Local Syste...	
Remote Access Connection Manager	Manages di...	Manual	Local Syste...	
Remote Desktop Configuration	Remote Des...	Manual	Local Syste...	
Remote Desktop Services	Allows user...	Manual	Network S...	
Remote Desktop Services UserMod...	Allows the r...	Manual	Local Syste...	
Remote Procedure Call (RPC)	The RPCSS	Started	Automatic	Network S...

Actions

Services More Actions

RabbitMQ More Actions

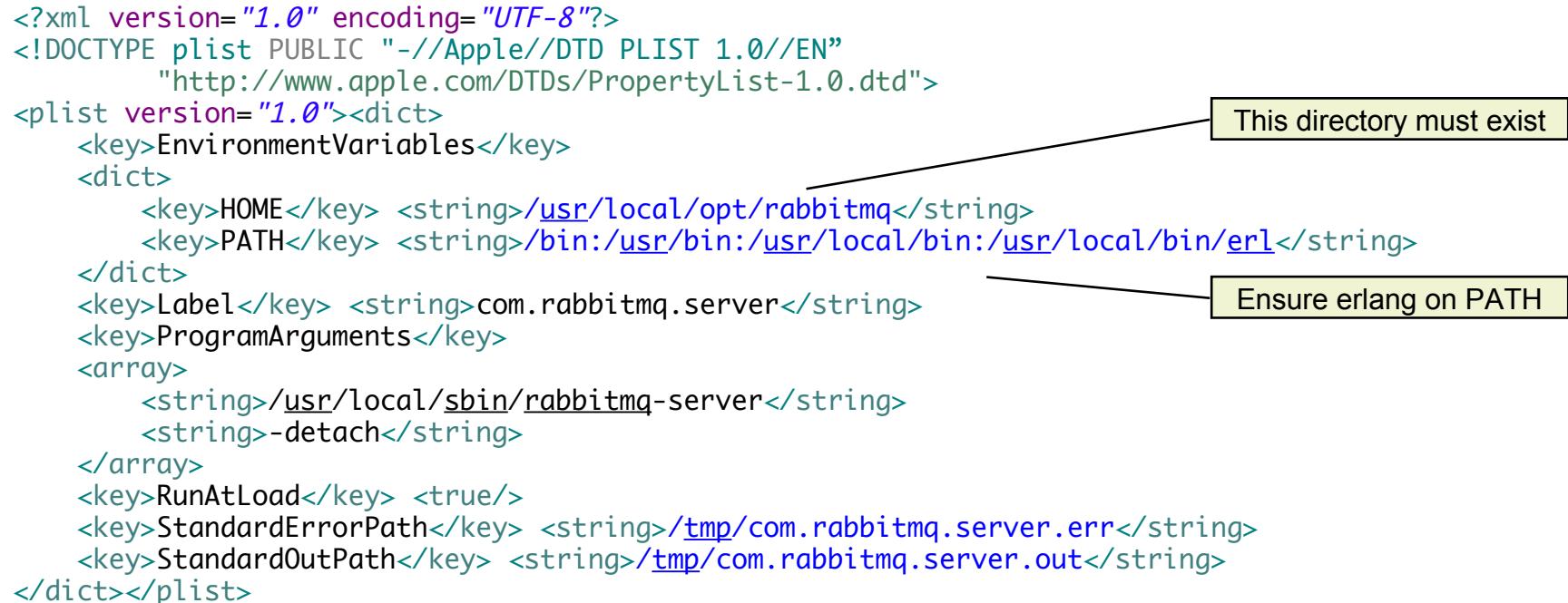
The screenshot shows the Windows Computer Management console. The left navigation pane is expanded to show 'Computer Management (Local)' with several categories like System Tools, Storage, and Services and Applications. The 'Services' node under Services and Applications is selected. The main pane displays a list of services, with 'RabbitMQ' highlighted. A red oval highlights the toolbar above the table, specifically the first four icons: Stop, Start, Pause, and Continue.

Running RabbitMQ on Boot on MacOS – 1

- Create a daemon definition `com.rabbitmq.server.plist`
 - An XML file in either:
 - `/Library/LaunchDaemons` – runs as root on startup, all users
 - `/Users/your-name/Library/LaunchAgents` – runs when *you* log in
- Automatically loaded any run by *launchd*
 - You can test it manually
 - `launchd load com.rabbitmq.server`
 - `launchd start com.rabbitmq.server`

Running RabbitMQ on Boot on MacOS – 2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0"><dict>
  <key>EnvironmentVariables</key>
  <dict>
    <key>HOME</key> <string>/usr/local/opt/rabbitmq</string>
    <key>PATH</key> <string>/bin:/usr/bin:/usr/local/bin:/usr/local/bin/erl</string>
  </dict>
  <key>Label</key> <string>com.rabbitmq.server</string>
  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/sbin/rabbitmq-server</string>
    <string>-detach</string>
  </array>
  <key>RunAtLoad</key> <true/>
  <key>StandardErrorPath</key> <string>/tmp/com.rabbitmq.server.err</string>
  <key>StandardOutPath</key> <string>/tmp/com.rabbitmq.server.out</string>
</dict></plist>
```



This diagram shows annotations on the launchd plist code:

- A callout points to the `HOME` key with the text "This directory must exist".
- A callout points to the `erl` component in the `PATH` string with the text "Ensure erlang on PATH".

com.rabbitmq.server.plist – launchd definition file for RabbitMQ Server

Lab

Installation and Management
"Installation"
(Up to and including section 2.2)

Agenda

- RabbitMQ Installation
- Startup at Boot
- **Basic Administration**

Customizing the RabbitMQ Installation

- Customized by 2 configuration files
 - `rabbitmq-env.conf`
 - `rabbitmq.config`
- Location of configuration files is distribution specific
 - Generic UNIX - `$RABBITMQ_HOME/etc/rabbitmq/`
 - Debian - `/etc/rabbitmq/`
 - RPM - `/etc/rabbitmq/`
 - Mac OS X (Macports) - `$(install_prefix)/etc/rabbitmq/`, the Macports prefix is usually `/opt/local`
 - Mac OS X (Homebrew) - `/usr/local/etc/rabbitmq`
 - Windows - `%APPDATA%\RabbitMQ\`

Environment

- Environment variables are sourced
 - In the `rabbitmq-env` shell script
 - From the file `/etc/rabbitmq/rabbitmq-env.conf` (recommended – does not exist after install, you must create it)
- `rabbitmq-env.conf` is not used on Windows
- Variables are listed at <http://www.rabbitmq.com/configure.html>
 - See also the `rabbitmq-env.conf` manpage for your version
- Examples
 - `RABBITMQ_NODE_IP_ADDRESS`: network interfaces binding
 - `RABBITMQ_MNESIA_BASE`: database files location

```
$ man rabbitmq-env.conf
```

Configuration File

- Formatted as an Erlang configuration file
 - Options are listed at <http://www.rabbitmq.com/configure.html>
 - Example is shown below.

```
[{rabbit, [  
    {heartbeat, 30}, —————— TCP keepalive heartbeat interval  
    {tcp_listeners, [5673]}, —————— List of AMQP listener ports  
    {default_user,<<"guest2">>}, —————— Default username/password  
    {default_pass,<<"guest2">>} ——————  
]}]. —————— Don't forget the closing "period"!
```

NOTE

These are sample settings to show the various options formats. Note the syntax for strings.

Classic Configuration

- RabbitMQ works great out-of-the-box!
- Use configuration for:
 - Changing ports
 - Setting TCP keepalive heartbeat interval
 - Changing logging levels
 - SSL
 - LDAP integration
 - etc.

Logging

- Logs are written to
 - `/var/log/rabbitmq` (UNIX platforms)
 - `$USER_HOME\AppData\Roaming\RabbitMQ\log` (Windows)
- Logs are rotated using standard logrotate mechanism in Unix
 - By default, logs are rotated weekly
 - Configuration in `/etc/logrotate.d/rabbitmq-server`
- Up to 6 log files are created:
 - `startup_log`
 - `startup_err`
 - `<nodename>@<hostname>.log` (eg. `"rabbit@centos4.log"`)
 - `<nodename>@<hostname>-sasl.log` (eg. `"rabbit@centos4-sasl.log"`)
 - `shutdown_log`
 - `shutdown_err`

Logging

- **`startup_log`**: Messages related to server startup.
- **`startup_err`**: Errors on server startup.
- **`<nodename>@<hostname>.log`** (eg. “`rabbit@centos4.log`”): Main server log file.
- **`<nodename>@<hostname>-sasl.log`** (eg. `rabbit@centos4-sasl.log`): Contains Erlang process crash reports.
- **`shutdown_log`**: Server shutdown log file.
- **`shutdown_err`**: Errors on server shutdown

Data Files

- RabbitMQ uses the Erlang Mnesia distributed database to persist state.
 - A distributed key-value Erlang database
 - Designed for high availability
- Mnesia stores system state (exchange, queue metadata)
 - Does *not* store message content (is stored separately in a persistency log file)
- Data is by default kept in
 - `/var/lib/rabbitmq/mnesia` (UNIX platforms)
 - `$USER_HOME\AppData\Roaming\RabbitMQ\db` (Windows)
- Structure
 - `$DB_ROOT/<nodename>/` (Mnesia DB)
 - `$DB_ROOT/<nodename>.pid` (pidfile)
 - `$DB_ROOT/<nodename>-plugins-expand/` (generated launcher scripts)

RabbitMQ Plugins

- Plugins extend the functionality of the RabbitMQ servers. Examples include:
 - Management
 - LDAP
 - Federation
- A list of official plugins can be found at <http://www.rabbitmq.com/plugins.html>
- Most (including Management plugin) are disabled by default, must be enabled.

```
$ rabbitmq-plugins enable rabbitmq_management
```
- Server must be restarted for changes to take effect.
- Available plugins are displayed using the “rabbitmq-plugins” list command.
 - Dependent plugins automatically enabled when a parent is enabled are indicated by a lower-case “e”.

RabbitMQ Plugins

- Broker will log number of enabled plugins on startup:

```
$ sbin/rabbitmq-server

RabbitMQ 3.6.1. Copyright (C) 2007-2016 Pivotal Software, Inc.
## ##
## ## Licensed under the MPL. See http://www.rabbitmq.com/
## ##
##### Logs: /.../var/log/rabbitmq/rabbit@MacBook-Pro.log
##### ##      /.../var/log/rabbitmq/rabbit@MacBook-Pro-sasl.log
#####
Starting broker... completed with 6 plugins.
```

Management Plugin

- A web application
 - With administration capabilities
 - Graphical monitoring (queue depths, throughput ,etc.)
- Default port is 15672
- Default login credentials are user “guest”, password “guest”

Management Plugin Screenshot

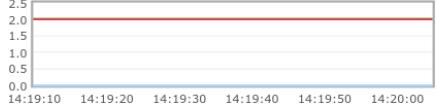
User: guest Cluster: rabbit@myserver ([change](#)) Log out
RabbitMQ 3.6.1, Erlang 18.2

[Overview](#) [Connections](#) [Channels](#) [Exchanges](#) [Queues](#) [Admin](#)

Overview

[Totals](#)

Queued messages (chart: last minute) (?)



Status	Count
Ready	2
Unacked	0
Total	2

Message rates (chart: last minute) (?)

Currently Idle

Global counts (?)

Connections: 0 Channels: 0 Exchanges: 10 Queues: 4 Consumers: 0

[Node](#)

[Ports and contexts](#)

[Import / export definitions](#)

HTTP API | Command Line Update [every 5 seconds](#)

Last update: 2016-04-06 14:20:15

Management Using the CLI

- Statistics for exchanges and queues accessible from the CLI

```
$ rabbitmqctl list_queues [-p <vhostpath>] [<infos>...]
$ rabbitmqctl list_exchanges [-p <vhostpath>] [<infos>...]
$ rabbitmqctl list_bindings [-p <vhostpath>] [<infos>...]
$ rabbitmqctl list_connections [<infos>...]
$ rabbitmqctl list_channels [<infos>...]
$ rabbitmqctl list_consumers [-p <vhostpath>]
$ rabbitmqctl _____
```

Running without args displays options

Management Using the CLI

```
$ rabbitmqctl list_queues name durable messages_ready
Listing queues ...
quot      true      0
quotes    true      0
test.queue      true     76
...done.
$
$ rabbitmqctl list_exchanges name type durable
Listing exchanges ...
amq.direct      direct  true
amq.fanout      fanout  true
amq.headers     headers true
amq.match       headers true
amq.rabbitmq.log      topic  true
amq.rabbitmq.trace      topic  true
amq.topic       topic   true
quotations     fanout  true
quotes        fanout  true
test.exchange   direct  true
...done.
```

Virtual Hosts

- Use virtual hosts (“vhosts”) to isolate sets of producers-consumers i.e. provides different namespaces
- Prevents exchange and queue name clashes across multiple applications.
- Default vhost is “/”
- Adding, deleting, and listing vhosts

```
$ rabbitmqctl add_vhost <vhostpath>
$ rabbitmqctl delete_vhost <vhostpath>
$ rabbitmqctl list_vhosts
```

User Management

- Create a user:

```
$ rabbitmqctl add_user <username> <password>
```

- Delete a user:

```
$ rabbitmqctl delete_user <username>
```

- List users:

```
$ rabbitmqctl list_users
```

Tracing

- RabbitMQ provides the “firehose tracer”
 - Fine-grained logs
 - Details on what happens in the server
- The firehose tracer
 - Has a performance penalty
 - Must be explicitly activated
- Consumable using AMQP from the amq.rabbitmq.trace exchange
 - ‘publish.<exchangename>’ as routing key for inbound messages
 - ‘deliver.<queuename>’ as routing key for outbound messages
- Headers and body are copies of the original

```
$ rabbitmqctl trace_on [-p <vhost>]  
$ rabbitmqctl trace_off [-p <vhost>]
```

Troubleshooting

- Check the status of the broker with rabbitmqctl status
- If the broker doesn't start
 - Check the log files: `startup_log`, `startup_err`, and server log files
 - Check `erl_crash.dump` file in the directory where the broker was started
 - Often caused by the wrong formatting of the config file
- If plugin won't activate, run rabbitmq-plugins list
- To verify AMQP traffic, use the firehose
- Check the mailing list
 - <https://groups.google.com/forum/#!forum/rabbitmq-users>

Summary

- RabbitMQ can be installed on various platforms including Windows, Mac OSX, and Linux
- Installation customized by 2 configuration files
 - `rabbitmq-env.conf`
 - `rabbitmq.config`
- Server management can be performed using the `rabbitmqctl` CLI or the web management plugin GUI

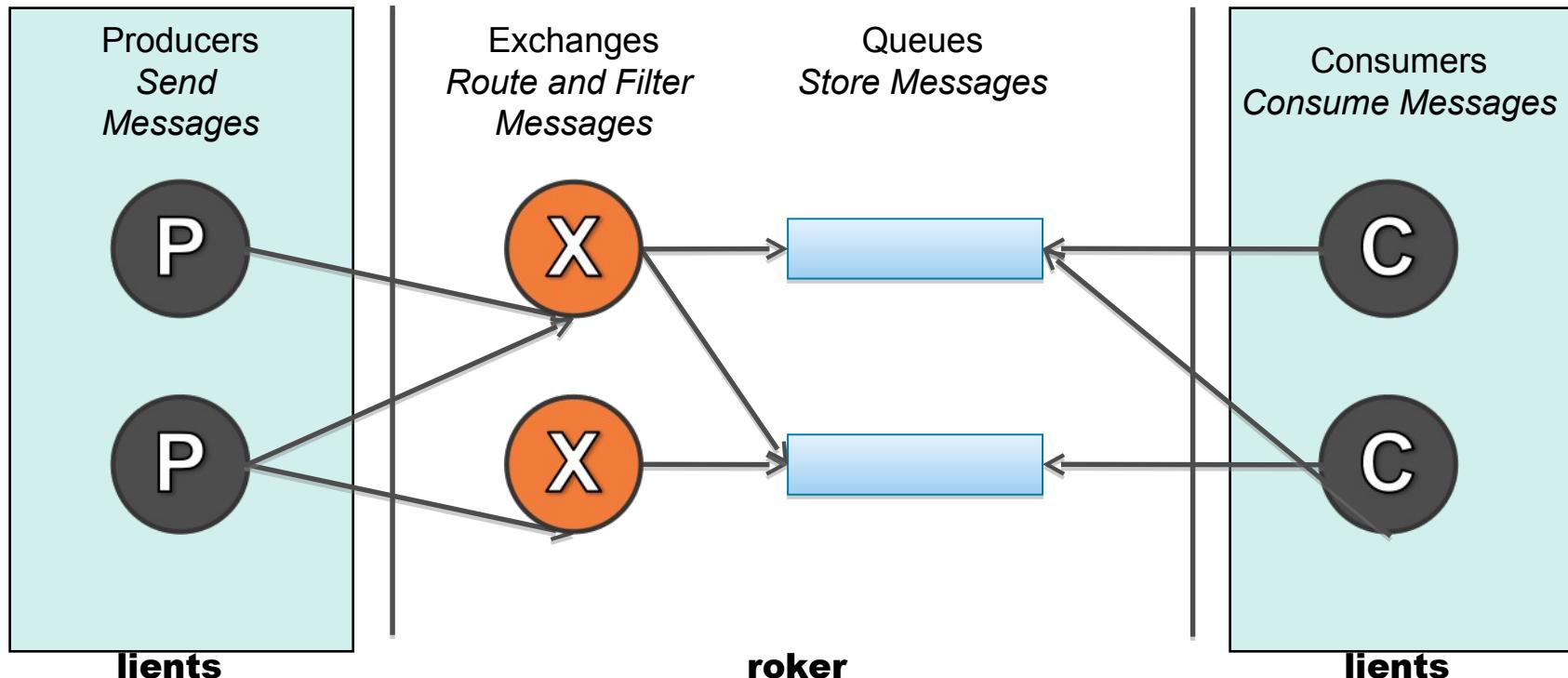
Installation and Management "Using the Management Plugin" (Section 2.3)

RabbitMQ Development and Integration

Agenda

- **Development Basics**
 - Clients
 - Java Client Basics
- Client AMQP Resource Management
 - Routing Messages
 - Exchanges
 - Message Ordering
- Use cases and patterns

The AMQP Model

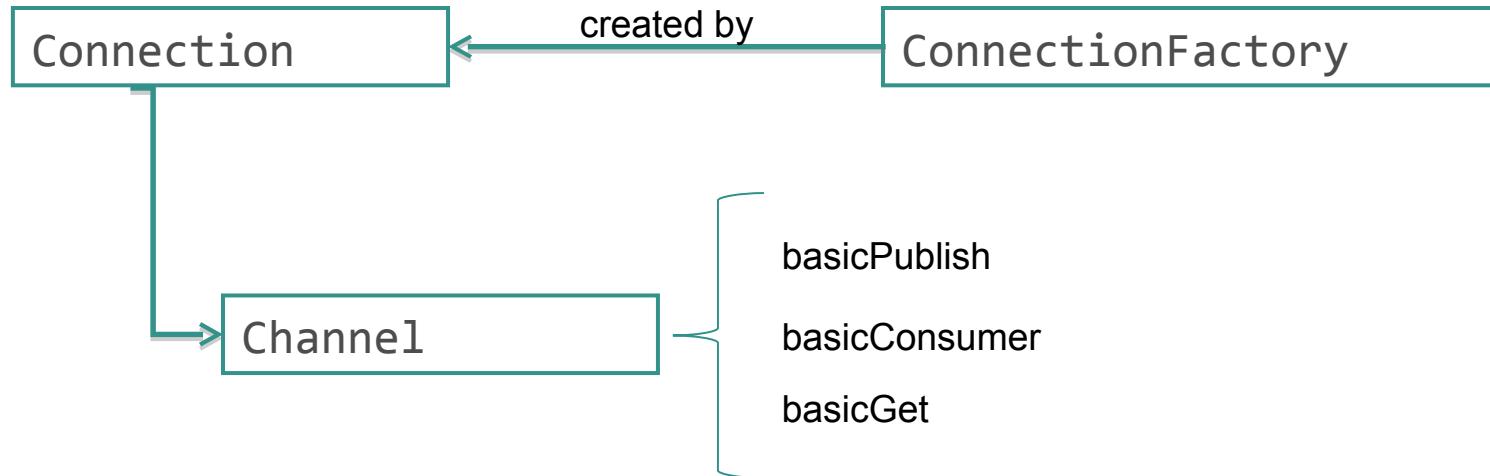


Clients

- Official clients are:
 - Java
 - C# (.NET)
 - Erlang
 - JMS (only available in Pivotal commercial release)
- Community-maintained clients include:
 - Python, Perl, Ruby
 - PHP
 - C/C++
 - JavaScript
 - Lisp, OCaml, and many more...
- For a full list go to <http://www.rabbitmq.com/devtools.html>

Java Client Hierarchy

- Java client provides a simple interface.
- Channel class is the main component.



Creating an AMQP Connection (Java Client)

- The Connection interface represents an AMQP connection.
 - All connection parameters have a default value
 - Override as necessary using setters

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

{...}

ConnectionFactory factory = new ConnectionFactory();
factory.setUsername("guest");
factory.setPassword("guest");
factory.setVirtualHost("/");
factory.setHost("localhost");
factory.setPort(5672);

Connection connection = factory.newConnection();
```

Creating an AMQP Connection (Java Client)

- AMQP URI's may also be used

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

{...}

ConnectionFactory factory = new ConnectionFactory();
factory.setUri("amqp://username:password@hostname:port/vhost");

Connection connection = factory.newConnection();
```

- URL with default vhost (/) needs escaping

amqp://guest:guest@localhost:5672/%2f

Creating an AMQP Channel

- The connection will be used to create a Channel
- The channel object is used to produce and consume messages.
- Channels are thread-safe, but the recommendation is to use one channel per thread in a multi-threaded application.

```
import com.rabbitmq.client.Channel;  
{...}  
Channel channel = connection.createChannel();
```

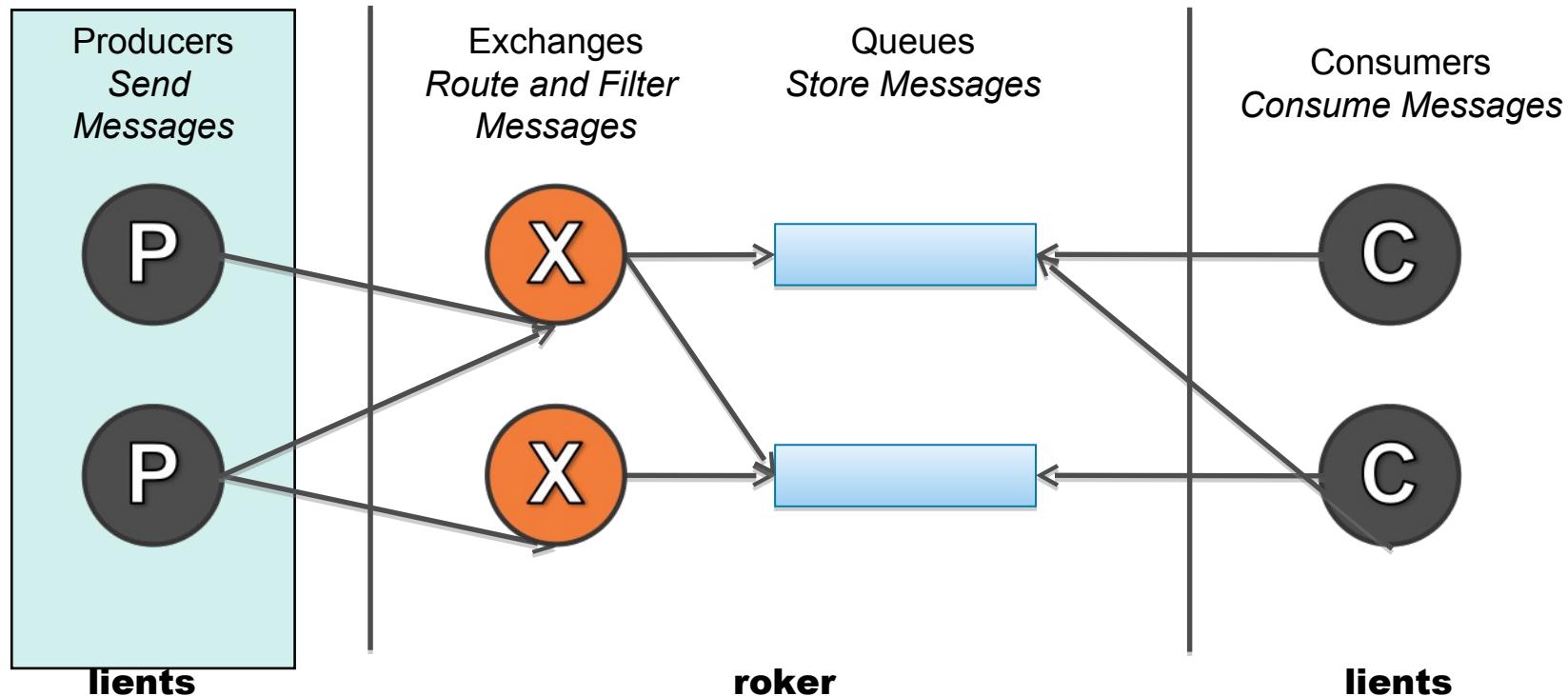
Closing Resources

- Connection must be closed, channel can be closed also.
- Closing the connection also closes the open channels.

```
{...}
```

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

The AMQP Model



Publishing Messages

- AMQP messages are published to an exchange
- The Channel.basicPublish() method is used
- A simple publisher:

AMQP.BasicProperties object

```
byte[] message = "My first AMQP message!".getBytes();
channel.basicPublish("exchangeName", "routingKey", null, message);
```

- AMQP.BasicProperties defines all the properties of your message
 - contentType, encoding, headers, correlationId, deliveryMode, expiration, messageId, replyTo, etc.

Publishing Messages

- A more advanced publisher:

```
byte[] message = "My second AMQP message!".getBytes();  
  
AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()  
    .contentType("text/plain").correlationId("id#3740")  
    .priority(1).userId("rabbit").build();  
  
channel.basicPublish("exchangeName", "routingKey", true, true, props, message);
```

AMQP.BasicProperties builder

Mandatory & Immediate flags

- Mandatory flag

- the message will be sent back to the sender if it cannot be routed to at least one queue (use Channel.setReturnListener to be notified)

- Immediate flag

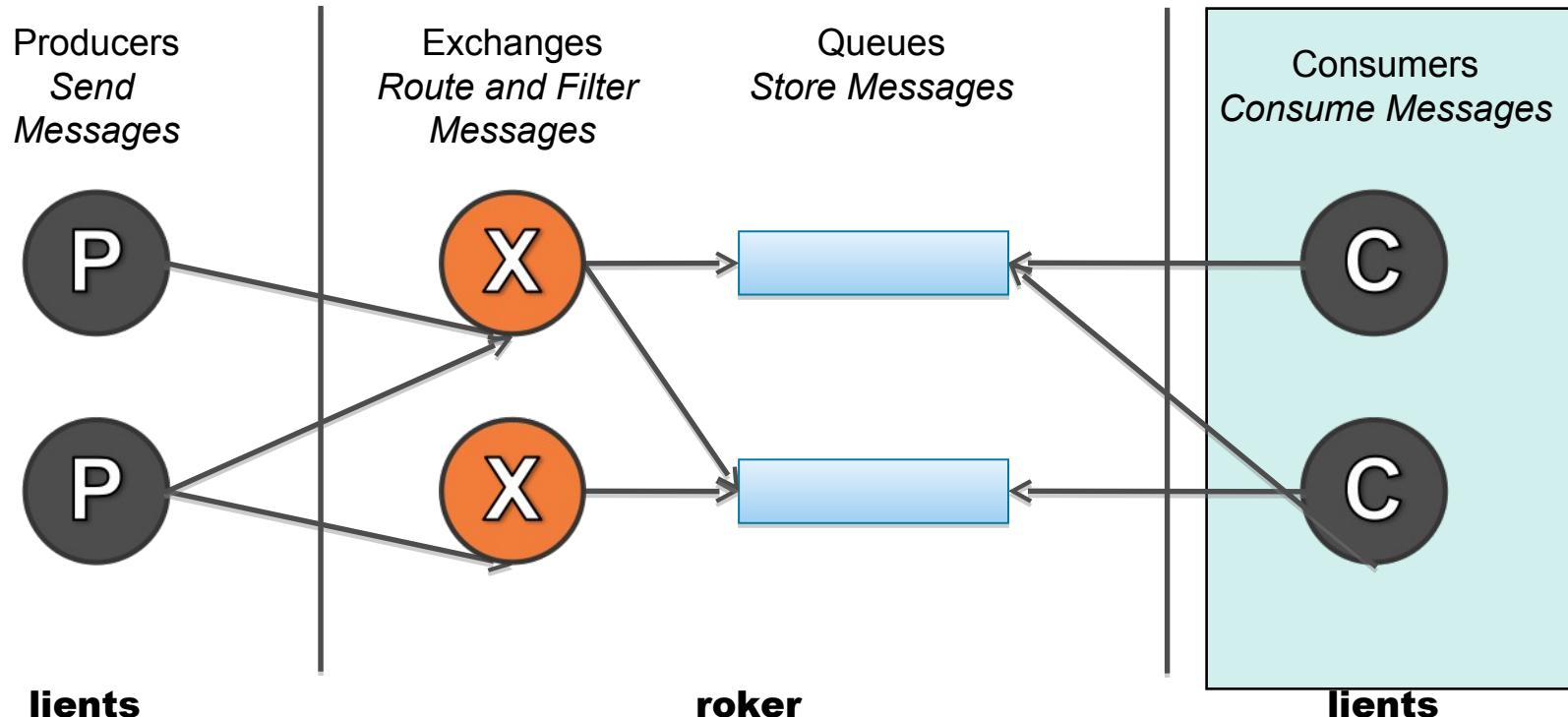
- the message will be handled as unroutable if at least one of the queues that would receive the message has no subscription on it. RabbitMQ does not currently support this flag.

Pre-Defined Properties

- The MessageProperties class contains pre-defined properties.

```
byte[] message = "My second AMQP message!".getBytes();
AMQP.BasicProperties props = MessageProperties.TEXT_PLAIN;
channel.basicPublish("exchangeName", "routingKey", true, true, props, message);
```

The AMQP Model



Receiving Messages

- Two different ways to receive AMQP messages
 - Asynchronously (subscription)
 - `Channel.basicConsume()`
 - Consumer callback mechanism
 - Non-blocking
 - Synchronously (polling)
 - `Channel.basicGet()`
 - Message is explicitly requested by caller

Receiving Messages Asynchronously

- The callback is achieved with the Consumer interface
- Important methods are:
 - handleCancel()
 - handleCancelOk()
 - handleConsumeOk()
 - handleDelivery()
 - handleRecoverOk()
 - handleShutdownSignal
- Implement them when you want your consumers to be notified of delivery, cancel, shutdown, or consume events

DefaultConsumer Class

- Java client provides a DefaultConsumer class
- Mostly empty, developers override required methods
 - Avoids implementing all the methods from Consumer
- Consumer implementations typically extend this class

Receiving Messages Asynchronously

- Channel.basicConsumer() example:

```
boolean autoAck = true;

channel.basicConsume("queue", autoAck,
    new DefaultConsumer(channel) {
        @Override
        public void handleDelivery( String consumerTag,
                                    Envelope envelope,
                                    AMQP.BasicProperties properties,
                                    byte[] body) throws IOException {
            String contentType = properties.getContentType();
            String routingKey = envelope.getRoutingKey();

            // process message here ...
        }
    });
});
```

Receiving Messages Synchronously (Polling)

- Channel.basicGet() example:

```
boolean autoAck = true;

GetResponse response = channel.basicGet("queueName", autoAck);

if (response != null) {
    AMQP.BasicProperties props = response.getProps();
    byte[] body = response.getBody();
    Envelope envelope = response.getEnvelope();

    // process message here ...
}
```

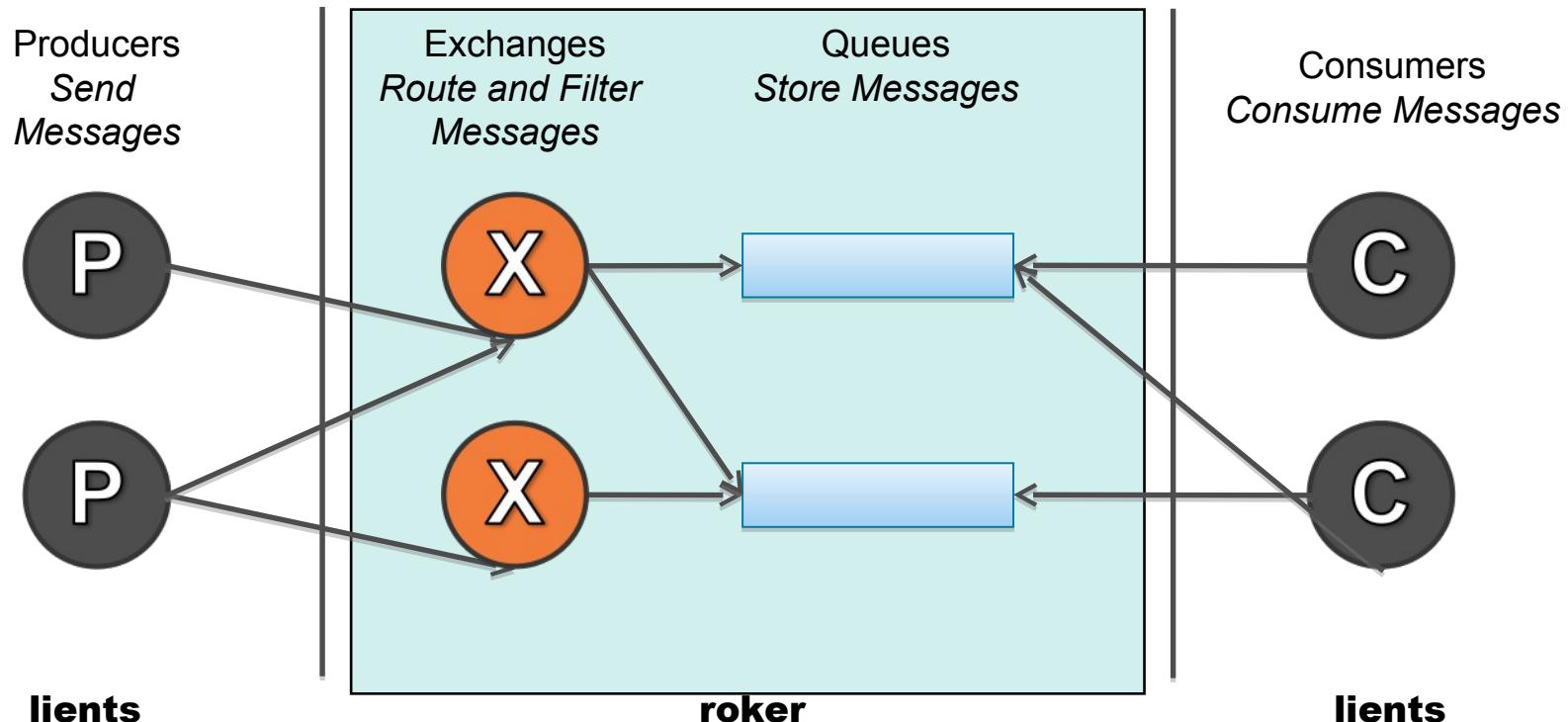
Lab

"Java Message Sender and Receiver"

Agenda

- Development Basics
 - Clients
 - Java Client Basics
- **Client AMQP Resource Management**
 - Routing Messages
 - Exchanges
 - Message Ordering
- Use cases and patterns

The AMQP Model



Managing and Creating AMQP Resources

- Queues, exchanges and bindings
 - They always exist on the broker, when they exist at all
 - They can be created on the fly by clients
 - They can be durable (i.e. can survive broker restarts)
- Clients can create and manage AMQP resources on the broker side
 - Exchange creation / deletion
 - Queue creation / purge / deletion
 - Binding creation / deletion

NOTE

Resource creation is idempotent, as long as the already-existing resource has the exact same attributes (durability, auto-deletion, etc.)

Using Queues and Exchanges

- Clients can declare queues and exchanges
- Declaring a resource ensures it exists, creating it if necessary
 - Exchange declaration is realized on a channel
 - Example is shown below.

```
channel.exchangeDeclare("exchangeName", "direct", true);
```

Exchange type

Durability flag

NOTE

Exchange types are described in the next slides.

Using Queues and Exchanges

- Queue declaration is realized on a channel

```
channel.queueDeclare("queueName", true, false, false, null);
```

Durability flag

Exclusive flag

Auto-delete flag

Arguments

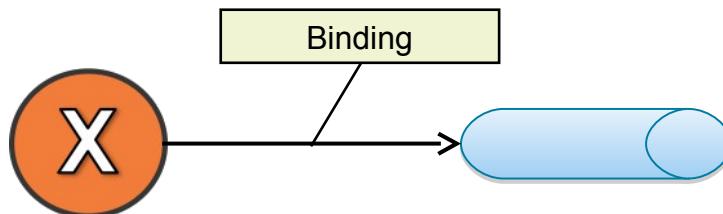
- Durability
 - Message can survive broker restarts (more on this in next module)
- Exclusive
 - Only the current client can connect to this queue
 - Exclusive queues will be deleted when the client closes the connection
- Auto-delete
 - The queue will be deleted after the last subscription is removed from it

Binding Queues and Exchanges

- Clients need to bind queues and exchanges on channel
- Routing key needs to be provided in order to bind the exchange and the queue together

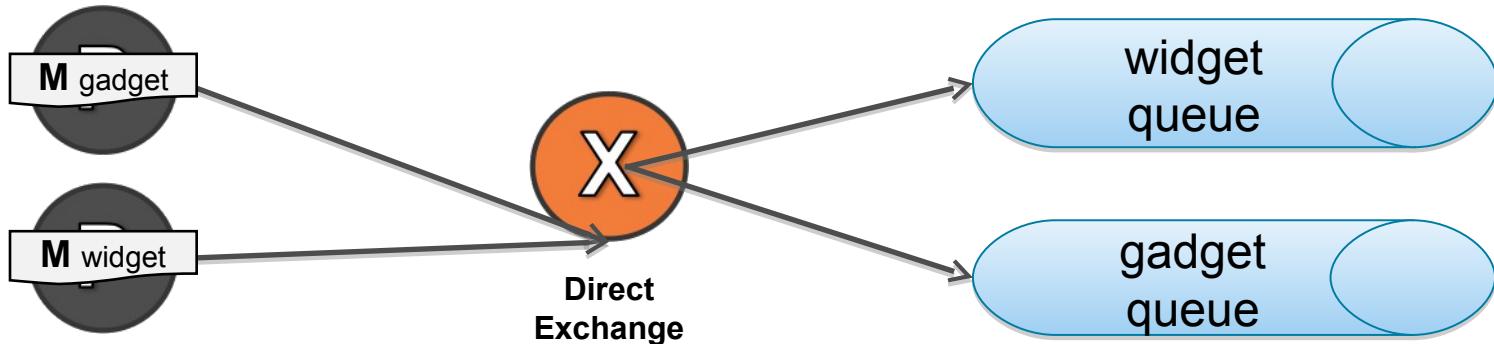
```
channel.queueBind("queueName", "exchangeName", "widget");
```

Routing key



Routing Logic to Queues

- AMQP embeds routing logic using exchange types and bindings.
- Messages are published with a special header: “routingkey”
 - Pattern matching is performed between the message routingkey and the exchange bindings
 - Based on the match results, in combination with the type of exchange, the message is routed to the appropriate queue(s).



Removing Bindings and Purging Queues

- To remove a binding:

```
channel.queueUnbind("queueName", "exchangeName", "widget");
```

- To purge all messages from a queue (note this does not delete the queue):

```
channel.queuePurge("queueName");
```

Deleting Resources

- To delete a queue:

```
channel.queueDelete("queueName");
```

- To delete an exchange:

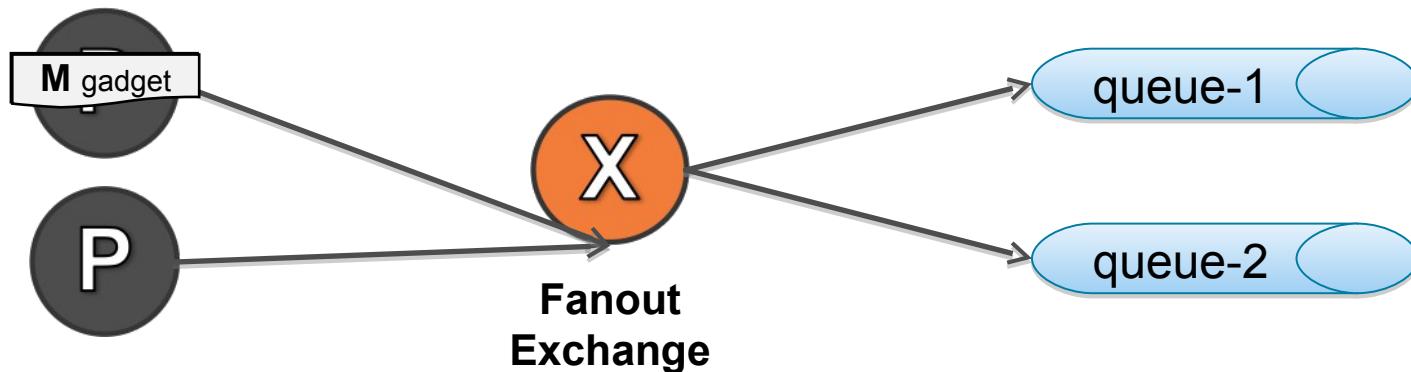
```
channel.exchangeDelete("exchangeName");
```

Exchange Types

- 4 AMQP default exchange types:
 - Fanout
 - Direct
 - Headers
 - Topic
- Each type of exchange has different routing capabilities.
- It is also possible to write your own custom exchange.
- RabbitMQ includes a custom consistent hash exchange as part of the installation.

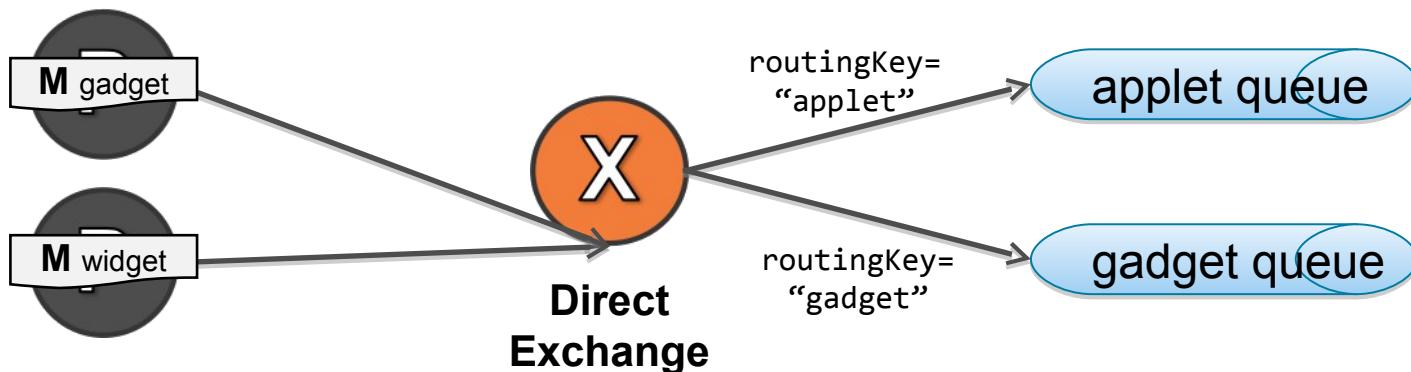
Fanout Exchanges

- Think “broadcast” or “publish-subscribe”
- No routing key evaluation
- Bindings don’t need routing key criteria
- Always matches
- Sends messages to all the bound queues



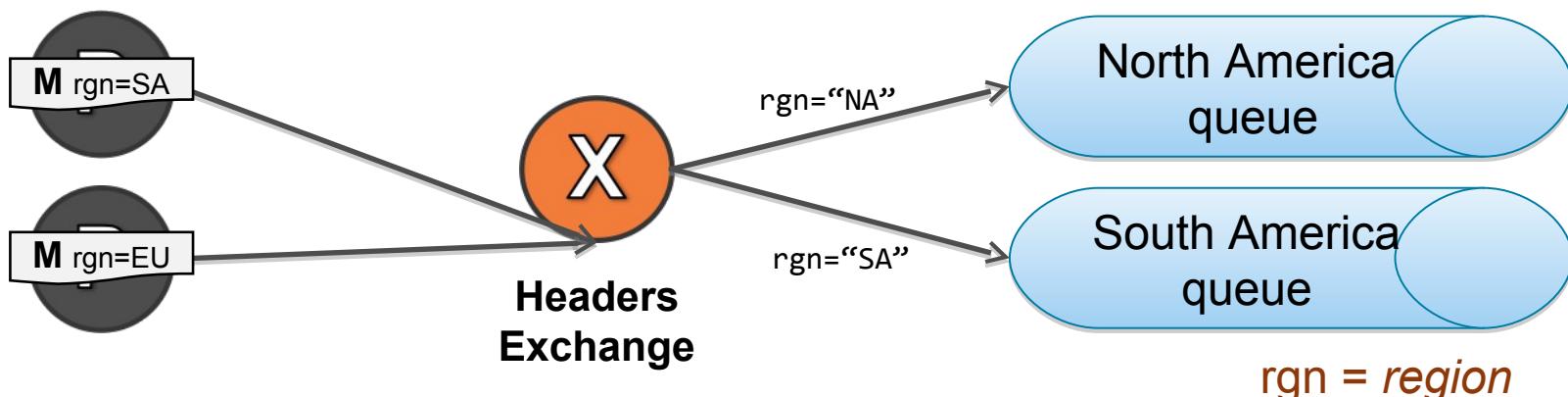
Direct Exchanges

- Routing key is mandatory
- Otherwise, message is not routed and is discarded
- Sends messages to all the bound queues where the routing key matches the binding.



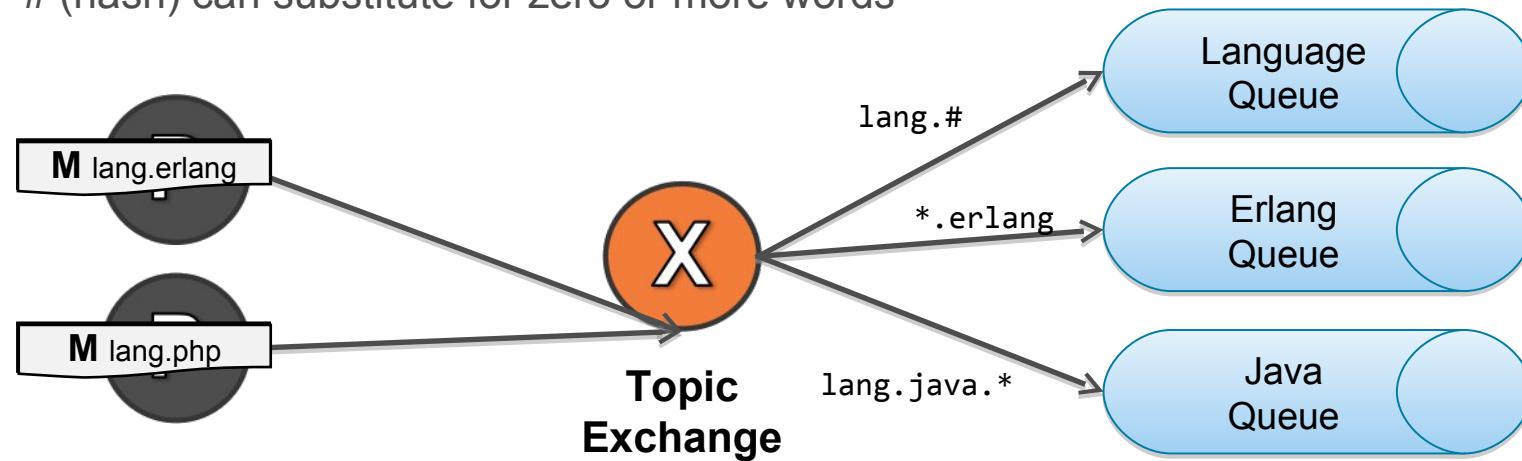
Headers Exchanges

- No routing key evaluation
- Message headers are key/value pairs
- Like the direct exchange, but matches on headers and not on message routing key
- Can match on any or all key-value pairs specified in the binding
- Again, message is discarded if no matching binding is found



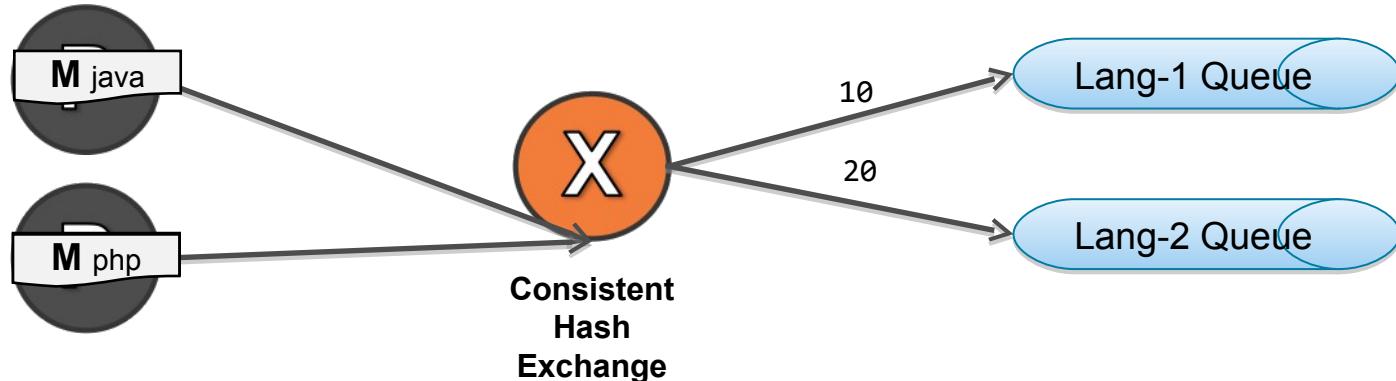
Topic Exchanges

- Topic exchange matches the message routing key on binding-defined pattern
- The binding pattern supports wildcards
 - * (star) can substitute for exactly one word
 - # (hash) can substitute for zero or more words



Consistent Hash Exchange

- Consistent hash exchange routes messages to bound queues based on hashed value of routing key
- Each queue essentially represents a "bucket" of the hash
- The routingKey binding value is an integer that specifies the weighting for each queue
 - In example below, the "Lang-2 Queue" will receive twice as many messages as the "Lang-1 Queue"



Message ordering guarantees

- RabbitMQ provides strong message ordering guarantees
- Messages are always held in publication order in *one* queue
- Nevertheless, relying too much on message ordering in consumers is fragile
 - Limits the processing to a single consumer, it's not scalable
 - With multiple consumers, processing messages in strict order doesn't make much sense
- It's still possible to observe out of order messages
 - If the queue has multiple consumers
 - If some consumers *requeue* messages

NOTE

A consumer can requeue a message for various reasons (bad format, rejection). The reliability module covers the requeuing mechanisms.

Priority queues

- RabbitMQ can prioritize the delivery of messages to consumers
 - Messages with high priority are delivered *before* other messages
 - Priority takes over publication order
- Use priority when some messages are more urgent than others
 - E.g. messages coming from “premium” customers vs “standard” customers
- How to use priority in RabbitMQ?
 - Declare a queue with a max priority level (e.g. 10)
 - Send messages with the priority property set to the priority level (e.g. 8)
- Priority semantics:
 - Messages with a high priority level are delivered first
 - Messages with no priority property set are treated as if their priority level were 0

Setting up priority

```
Map<String, Object> arguments = new HashMap<>();  
arguments.put("x-max-priority", 10);  
channel.queueDeclare("queueWithPriority", true, false, false, arguments);
```

Use the x-max-priority argument when creating the queue

```
byte[] message = "This message should be processed quickly!".getBytes();
```

```
AMQP.BasicProperties props = new AMQP.BasicProperties.Builder()  
.priority(8)  
.build();
```

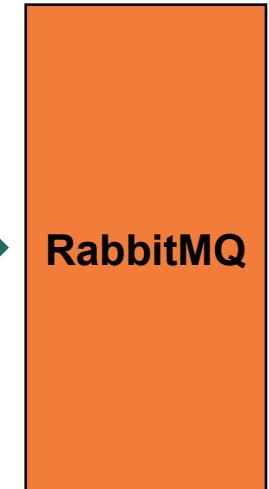
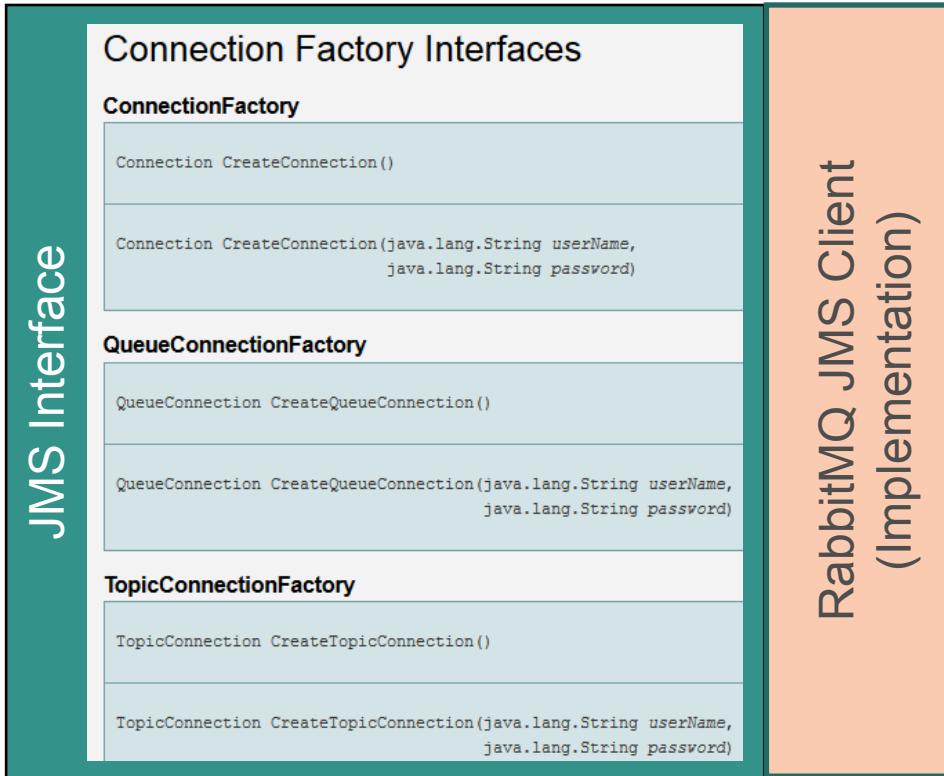
Use the priority property when sending the message to set the priority level

```
channel.basicPublish("exchangeName", "routingKey", true, true, props, message);
```

Priority queues, more details

- Priority queues cannot change the number of priorities they support
- The priority level can be between 0 and 255
 - Pick a reasonable max level, like 10
- Each level of priority has some in-memory and on-disk cost
 - Another good reason to set a reasonable max level!

JMS Client



Higher Level Abstractions

- Client API's are modeled on the AMQP specification
 - Low-level
- Higher-level abstractions are often required
 - Automatic reconnection
 - Failover
 - Built-in mechanism for marshalling/unmarshalling payloads
 - Uniform error handling

Higher Level Abstractions

- Java
 - Spring AMQP
 - Spring Integration
- Spring AMQP
 - Minimize boilerplate code with an AmqpTemplate
 - Encode and decode payloads with MessageConverter
 - Provide resilience mechanisms (retry...)
 - Similar to the JMS support in Spring Framework
- Spring Integration
 - Provide unidirectional calls (one-way) over AMQP
 - Provide bidirectional calls (request-reply) over AMQP

Even Higher Level Abstractions

- Need to do it yourself
- De-duplication if needed for consumers
- Buffering while disconnected for producers
 - Cyclic buffers for holding unsent messages
 - Sender thread(s) with failover on a set of broker nodes

Lab

Message Routing (Part 1: "Exchange Types and Bindings")

Agenda

- Development Basics
 - Clients
 - Java Client Basics
- Client AMQP Resource Management
 - Routing Messages
 - Exchanges
 - Message Ordering
- **Use cases and patterns**

Use Cases and Patterns

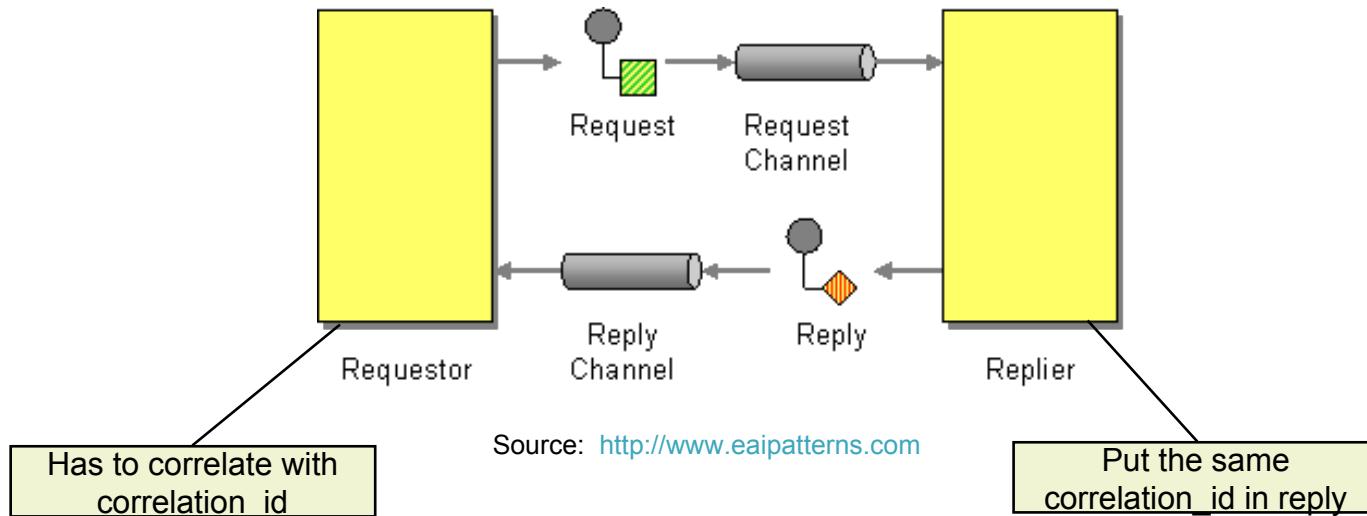
- In messaging, there are lots of common patterns and use cases
 - They are often called Enterprise Integration Patterns (EIP)
 - <http://www.eaipatterns.com/>
 - Enterprise Integration Patterns book by Gregor Hohpe and Bobby Woolf
- AMQP and RabbitMQ provide a foundation to use these patterns in messaging applications
 - RPC: request / reply communications
 - Work queue: distribute tasks among consumers
 - Publish / subscribe: send message to many consumers at once

RPC – Remote Procedure Call

- This pattern is also known as request-reply
 - An initial message is sent to a consumer (request)
 - Timeout in case the response takes too long to come
 - A message is sent back to the original producer (reply)
- Hidden mechanisms play a role in this request-reply pattern
 - The request needs to be sent with extra headers
 - `reply_to` in order to know on which queue reply should be sent back
 - `correlation_id` in order to correlate the response and the reply
 - The reply also has extra headers
 - `correlation_id` has to be the same as the request to be able to correlate both messages

RPC – Remote Procedure Call

- The requestor can implement the timer process
 - This is optional, but risky if not implemented



RPC - RpcClient

- Java client comes with two convenient classes to manage client and server side

```
try {  
    RpcClient rpcClient = new RpcClient(channel, "exchangeName", "key", 500);  
  
    byte[] request = "my request".getBytes();  
    byte[] reply = rpcClient.primitiveCall(request);  
  
    // do business with reply message  
  
} catch (TimeoutException e) {  
  
    // manage timeout here
```

Specify channel, exchange, routing key, and timeout (ms)

This method call returns either
-reply
-TimeoutException

RPC - RpcServer

- Need to override one method to specify behaviour

```
RpcServer rpcServer = new RpcServer(channel, "queueName") {  
  
    @Override  
    public byte[] handleCall(byte[] request, BasicProperties replyProperties) {  
  
        // do business to create reply message  
        return "reply!!".getBytes();  
    }  
};  
rpcServer.mainloop();
```

Specify channel and queue name

Override the handleCall() method

Create the reply message

Start the looping process

RPC – RabbitMQ optimization

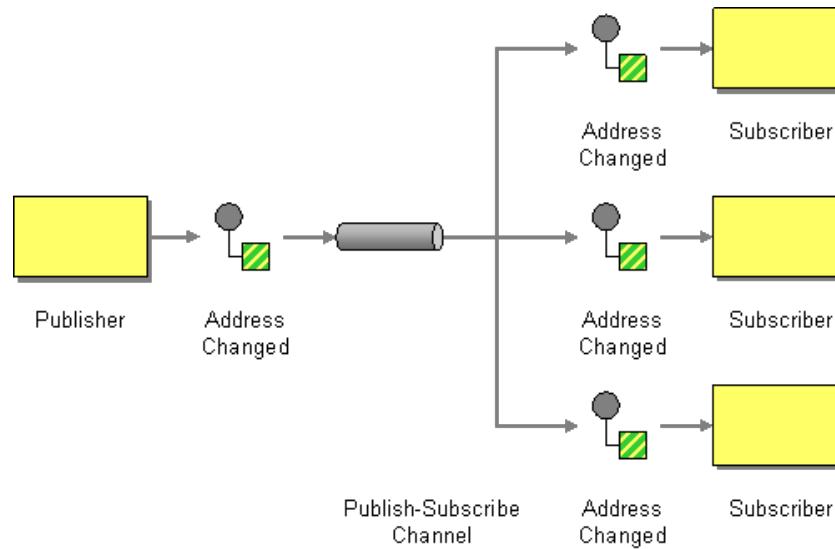
- The RPC client usually waits for the response on a dedicated queue
 - Do this if there's nothing better in your messaging infrastructure...
 - But this is inefficient
- RabbitMQ provides the *direct reply-to* feature to make RPC faster
 - By providing the `amq.rabbitmq.reply-to` special queue
- The Java `RpcClient` and `RpcServer` classes use this special queue
 - No need to worry about optimization if you're using them
- Any custom RPC mechanism should use the special queue
 - In case the used client doesn't already implement RPC this way

Work Queue – Predictable Dispatching

- With AMQP and RabbitMQ, you can of course run several consumers on the same queue
- Contrary to some other messaging brokers, AMQP messages are round-robin dispatched
 - For example, if you have two consumers connected to a queue
 - The queue has 10 messages
 - Both consumers will receive exactly five messages
 - Consumer 1 : the first, third, fifth, seventh, ninth
 - Consumer 2 : the second, fourth, sixth, eighth, tenth
- In JMS, you cannot predict which consumer will get the next message (random, based on the consumer's load and speed)

Publish / Subscribe

- This pattern allows us to send a message (or copy of message) to many consumers at the same time
 - All active consumers at a given time will receive the same copy of the message



Source: <http://www.eapatterns.com>

Publish / Subscribe

- Implemented using topic destinations in JMS.
- In AMQP, fanout exchanges are used to implement the publish-subscribe pattern.
- Alternatively, exchange types with multiple identical bindings can be used to route copies to multiple queues.

Summary

- Numerous clients supported in various languages
- RabbitMQ Java client has robust support for managing server resources, and publishing and consuming messages
- 4 different exchange types dictate message routing
- It is possible to implement common messaging patterns in RabbitMQ

Message Routing (Part 2: "Patterns")

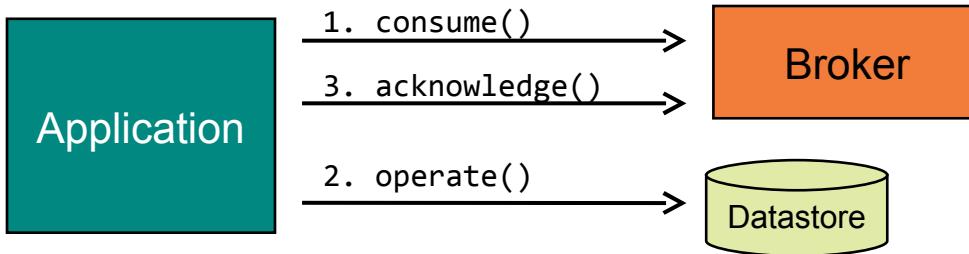
RabbitMQ and Reliable Messaging Development

Agenda

- Durability and persistence
- AMQP Transactions
- AMQP Acknowledgements
- Dead letter exchanges
- Multiple transactional resources

Messaging and Reliability

- Messaging is rarely isolated
- Messaging is usually interleaved with business processing
- Typical reception workflow:



- “Acknowledge” means a message has been properly processed
- Once acknowledged, the message is removed from the queue

NOTE

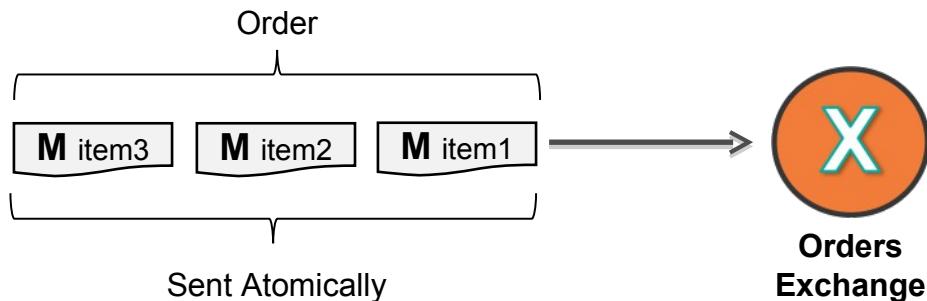
The acknowledgement is sent to the broker, not to the producer of the message!

RabbitMQ and reliability

- Many things can go wrong during message publication or reception
- AMQP specifies how to exchange messages in a reliable manner
- RabbitMQ implements these reliable mechanisms

What Can Go Wrong?

- The broker crashes; sent but not-yet-consumed messages are lost!
 - The broker can provide *persistence*
- Business processing after reception times out, or is temporarily down
 - The broker can *redeliver* the message
- Messages must be sent as a group, but the application fails in the middle
 - Thanks to transactions, the broker can provide *atomic operations*



Durability and Persistence

- Durability is the 'D' in ACID, eg.
 - surviving a server restart
 - surviving a broker crash
- Durability happens at three different levels:
 - Exchange
 - Queue
 - Message (referred to as “persistence”)
- Durability only works if queues are durable and messages are persistent.

Durable Exchanges

- Setting up a durable exchange:

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

// declare the durable exchange
channel.exchangeDeclare("my-exchange", "direct", true, false, null);
```

Durable flag

Auto-delete flag

- Consequences:

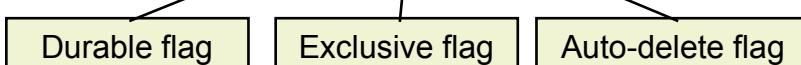
- The exchange definition (i.e. metadata) is written to disk
- If the broker crashes or shuts down, the exchange is still there when the broker is restarted
- There's no need to re-declare the exchange between restarts

Durable Queues

- Setting up a durable queue:

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

// declare the durable queue
channel.queueDeclare("my-queue", true, false, false, null);
```



- Consequences:

- The queue definition (i.e. metadata) is written to disk
- If the broker crashes or shuts down, the queue is still there when the broker is restarted
- There's no need to re-declare the queue between restarts

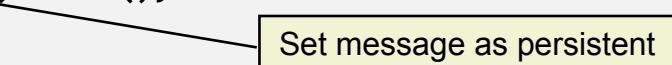
Persistent Messages

- Creating a persistent message:

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

// set the message as persistent (deliveryMode = 2)
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
AMQP.BasicProperties props = builder.deliveryMode(2).build();

String message = "Hello World!";
channel.basicPublish("my-exchange", "key", props, message.getBytes());
System.out.println("Sent '" + message + "'");
```



Set message as persistent

- Consequences:
 - Persists the message to the file system (in persistency log file)
 - If stored in durable queue, the message will still be there between restarts

Agenda

- Durability and persistence
- **AMQP Transactions**
- AMQP Acknowledgements
- Dead letter exchanges
- Multiple transactional resources

Transactions in AMQP

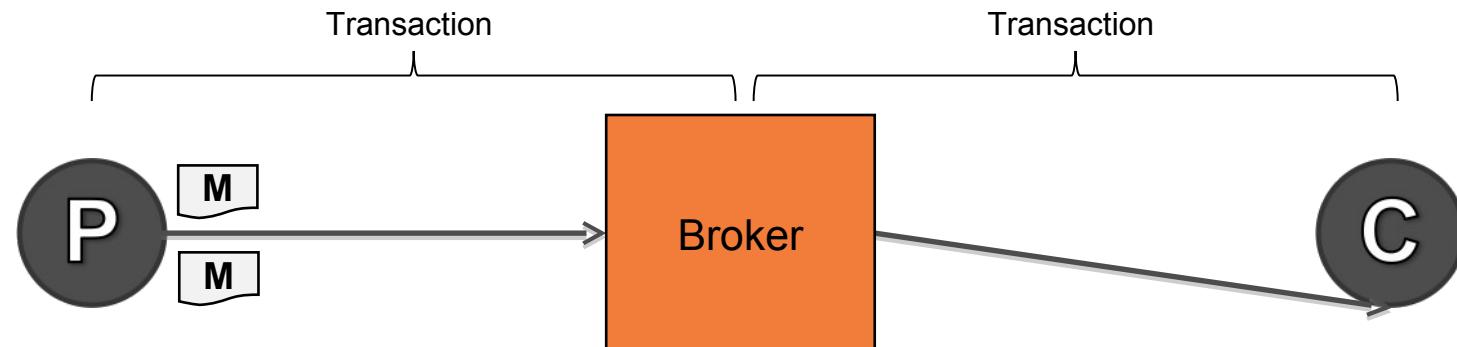
- AMQP specifies transaction semantics
- RabbitMQ implements part of the semantics
 - More on the limitations later
- AMQP transactions are pretty similar to JMS transactions but they do differ in the following ways:
 - Acknowledgment (ack) of received messages
 - Can be on a group of messages
 - Transaction to publish and acknowledge messages
 - Commit and Rollback
 - Can be on a group of messages

NOTE

There's no acknowledgment for publishing in AMQP. RabbitMQ has "Publisher Confirms" extension to ensure message(s) make it to the broker.

Transactions Across Senders and Consumers

- Transactions don't span the sender and the consumer
- A common transaction would couple the sender to the consumer
- Messaging is all about decoupling!
- Transaction semantics are only between the sender-broker and broker-consumer

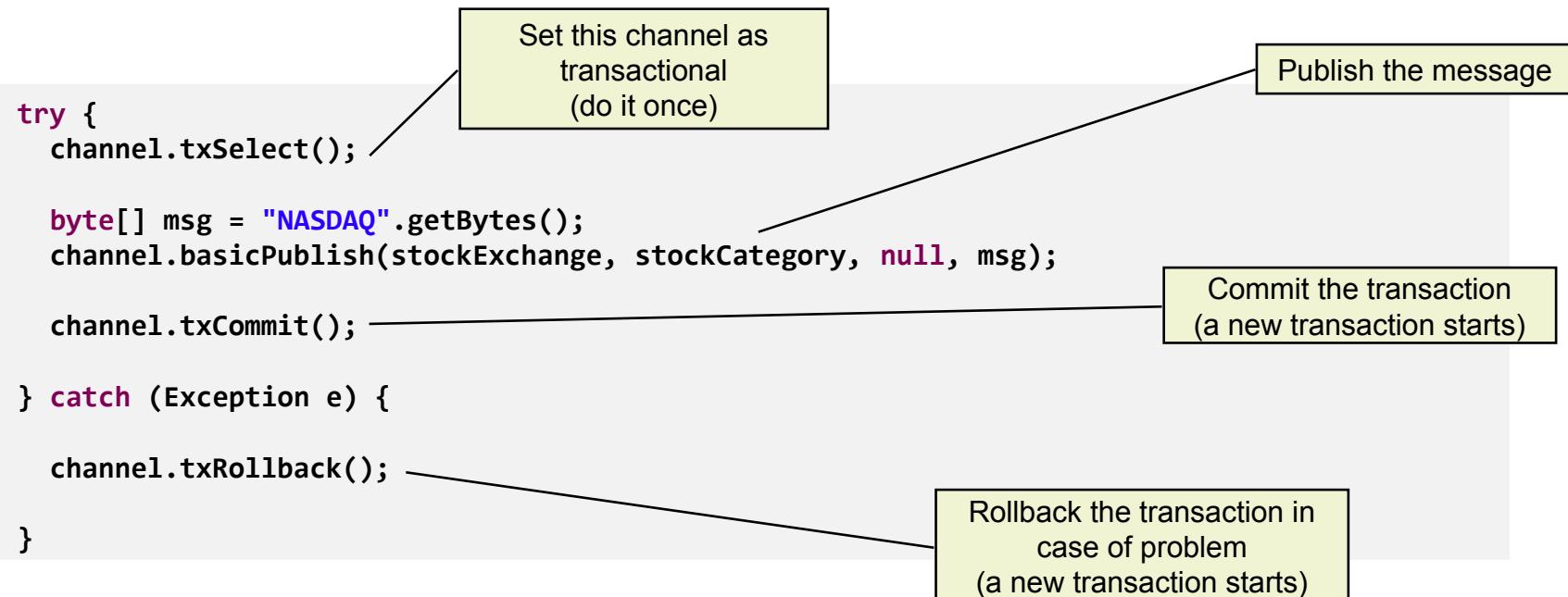


AMQP Transactions

- Acknowledgment doesn't exist on the publisher side in AMQP
 - RabbitMQ has a publisher confirms extension
- Transaction is managed at the channel level
 - Channel.txSelect()
 - Set this channel as transacted
 - Channel.txCommit()
 - Commit all the published messages since the last txSelect
 - A new transaction starts immediately after committing
 - Channel.txRollback()
 - Rollback all the published messages since the last txSelect
 - Cancel all the acknowledgments since the last txSelect
 - A new transaction starts immediately after rollback

AMQP Transactions

- Transaction usage for publishers:



AMQP Transactions

- In the RabbitMQ Management Console:

Channels

Channel	User name	Mode (?)	Details			
			Prefetch	Unacked	Unco	Settled
127.0.0.1:58905:1	guest	T	0	1		

- Messages stay in uncommitted state
 - Until the transaction is committed or rolled back

AMQP Transactions

- Publishing messages in batch mode, all messages will be committed together

```
channel.txSelect();

byte[] msgNasdaq = "NASDAQ".getBytes();
byte[] msgDJ = "DOWJONES".getBytes();
byte[] msgSP500 = "SP500".getBytes();

channel.basicPublish(stockExchange, stockCategory, null, msgNasdaq);
channel.basicPublish(stockExchange, stockCategory, null, msgDJ);
channel.basicPublish(stockExchange, stockCategory, null, msgSP500);

channel.txCommit();
```

Channel	User name	Mode (?)	Prefetch
127.0.0.1:59001:1	guest	T 3 uncommitted messages 0 uncommitted acks	0

AMQP Transactions

- Rolling back published messages
 - All the messages will be rolled back and will not reach the exchange

```
channel.txSelect();

byte[] msgNasdaq = "NASDAQ".getBytes();
byte[] msgDJ = "DOWJONES".getBytes();

channel.basicPublish(stockExchange, stockCategory, null, msgNasdaq);
channel.basicPublish(stockExchange, stockCategory, null, msgDJ);

// business exception is thrown here!
// must cancel all publications

channel.txRollback();
```

Agenda

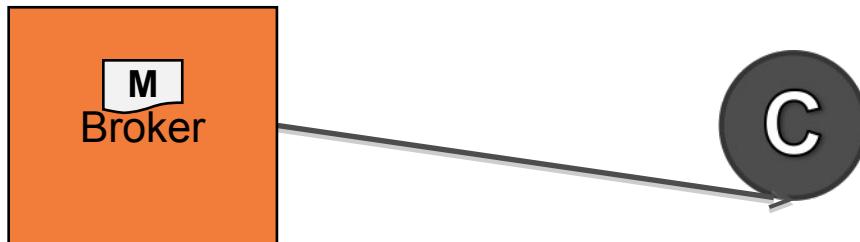
- Durability and persistence
- AMQP Transactions
- **AMQP Acknowledgements**
- Dead letter exchanges
- Multiple transactional resources

Did Message Reach the Broker?

- Without transactions:
 - Client uses fire-and-forget
 - They can't know whether the message(s) reached the broker or not
- With transactions:
 - When txCommit() returns, it means the broker got the message(s)
- With publisher confirms:
 - When the sequence number of the message is returned in the handleAck() callback method

Receiving Messages

- How does the broker know that the consumer processed the message?



- The consumer can:
 - Have the broker acknowledge messages automatically on delivery (“auto-ack”)
 - Receive, process, and explicitly send the acknowledgement
 - Start a transaction, receive messages, acknowledge messages, and commit acknowledged messages

AMQP Acknowledgements

- Acknowledgments are for receivers only
- Receiver
 - Can acknowledge one or a group of messages
 - Gets the delivery tag from the message and uses it to acknowledge on the channel
- Messages that are not acknowledged are put back in the queue

AMQP Acknowledgements

- If fetching messages synchronously:
 - Envelope is an argument of the handleDelivery() method

```
boolean autoAck = false;  
GetResponse response = channel.basicGet("queueName", autoAck);  
  
if (response != null) {  
  
    AMQP.BasicProperties props = response.getProps();  
    byte[] body = response.getBody();  
    long deliveryTag = response.getEnvelope().getDeliveryTag();  
  
    // do business logic here  
  
    boolean multiple = false;  
    channel.basicAck(deliveryTag, multiple);  
  
}
```

Set auto-ack to false

Get the delivery tag from the envelope

Single acknowledgement

AMQP Acknowledgements

- If using asynchronous callbacks
 - Specify acknowledgement mode in the `basicConsume()` method

```
boolean autoAck = false;  
channel.basicConsume("queueName", autoAck, listener);
```

Explicit acknowledgements

- Envelope is passed in as an argument of the `handleDelivery()` method

```
public void handleDelivery(String consumerTag,  
                           Envelope envelope,  
                           AMQP.BasicProperties properties,  
                           byte[] body) throws java.io.IOException {  
  
    // Process message here  
  
    // Ack the message  
    this.getChannel().basicAck(envelope.getDeliveryTag(), false);  
}
```

Get the delivery tag
from the envelope

Do not acknowledge
multiple messages

AMQP Acknowledgements

- Until the message is acknowledged by the client
 - There will be some unacknowledged messages



Overview Connections Channels Exchanges **Queues** Users Virtual Hosts

Queues

All queues

Overview				Messages			Message rates		
Name	Exclusive	Parameters	Status	Ready	Unacked	Total	incoming	deliver / get	ack
queueName		D	Idle	23	1	24			

No Acknowledgement?

- There's no timeout in AMQP
 - If the processing takes a long time, and the ack doesn't come, then the broker won't resend the messages
- When a client quits, the unacked messages are redelivered
 - Client quits = client closes connection
- Broker can't release unacked messages
 - Not good for memory!
- Use rabbitmqctl or the management plugin to track unacked messages

Acknowledging Multiple Messages

- Only one call to Channel.basicAck(), with parameters:
 - the tag of a delivered message (this message and all previous unacknowledged messages are acknowledged)
 - multiple flag set to true

```
boolean autoAck = false;
GetResponse response1 = channel.basicGet("queueName", autoAck);
// do business with response1

GetResponse response2 = channel.basicGet("queueName", autoAck);
// do business with response2

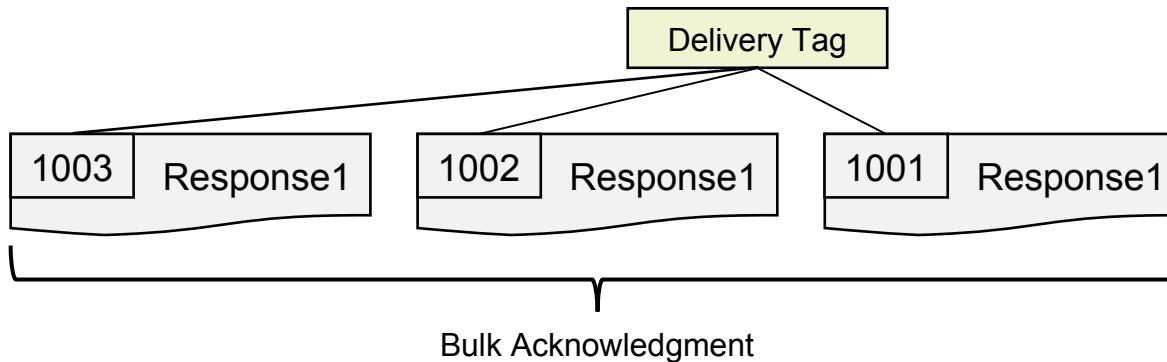
GetResponse response3 = channel.basicGet("queueName", autoAck);
// do business with response3

long latestDeliveryTag = response3.getEnvelope().getDeliveryTag();
boolean multiple = true;
channel.basicAck(deliveryTag, multiple);
```

Multiple acknowledgement

Acknowledging Multiple Messages

- All the messages that match the criteria below will be acknowledged in the same call to `basicAck()`:
 - Messages received on the same channel
 - Messages not yet acknowledged
 - Messages with a delivery tag less than or equal to the delivery tag given to the `basicAck()` method



```
channel.basicAck(1003, true);
```

AMQP Rejections

- Instead of acknowledging, consumers can also reject messages
 - Message can be requeued if needed (requeue flag)
 - The `basicReject()` method doesn't discard multiple messages
- Best practices:
 - Try to avoid requeuing messages that are likely to be repeatedly rejected by the consumer (poison messages)
 - Messages that cause the consumer to crash are likely to do so again.

```
public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body) throws java.io.IOException {

    long deliveryTag = envelope.getDeliveryTag();

    // something went wrong...
    boolean requeue = true;
    this.getChannel().basicReject(deliveryTag, requeue);
}
```

Requeue flag

AMQP Rejections – RabbitMQ Extension

- RabbitMQ offers an extra method to reject messages
- It is done by calling the `Channel.basicNack()` method
 - This is not part of the AMQP specification
 - Can reject all previous unacknowledged messages

```
public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body) throws java.io.IOException {

    long deliveryTag = envelope.getDeliveryTag();

    // something went wrong...
    boolean requeue = true;
    boolean multiple = true;
    this.getChannel().basicNack(deliveryTag, multiple, requeue);
}
```

Multiple flag

Requeue flag

AMQP Transactions Over Acknowledgements

- Messages can be acknowledged inside a transaction
- Typical workflow:
 - `txSelect()` before receiving
 - Business processing (can be message sending), ack
 - `txCommit()`
- Useful to make sending an acknowledgment atomic
 - E.g., consume, send, ack, commit everything

AMQP Transactions Over Acknowledgements

- The message is acknowledged after the `basicAck()`
- But the acknowledgment isn't committed before the `txCommit()`

```
Start the transaction  
channel.txSelect();  
  
boolean autoAck = false;  
GetResponse response = channel.basicGet("queueName", autoAck);  
  
// do business logic here  
  
long deliveryTag = response.getEnvelope().getDeliveryTag();  
channel.basicAck(deliveryTag, false);  
  
channel.txCommit();
```

Use explicit acknowledgements

Acknowledge the message

Commit the transaction

AMQP Transactions Over Acknowledgements

- Before acknowledgement:

Channels

		Details					
Channel	User name	Mode (?)	Prefetch	Unacked	Unconfirmed	Status	
127.0.0.1:59832:1	guest	T	0	1	0	Idle	

Queues

Overview							Messages	
Name	Exclusive	Parameters	Status	Ready	Unacked	Total		
queueName		D	Idle	22	1	23		

AMQP Transactions Over Acknowledgements

- After acknowledgment but before commit:

Channels

		Details					
Channel	User name	Mode (?)	Prefetch	Unacked	Unconfirmed	Status	
127.0.0.1:59874:1	guest	T	0	0	0	Idle	

Queues

Overview						Messages		
Name	Exclusive	Parameters	Status	Ready	Unacked	Total		
queueName		D	Idle	22	1	23		

AMQP Transactions Over Acknowledgements

- After commit:

Queues

▼ All queues

Overview					Messages		
Name	Exclusive	Parameters	Status	Ready	Unacked	Total	
queueName		D	Idle	22	0	22	

Reception: Acknowledgement or Transaction?

- For simple cases, prefer acknowledgment
 - It's simpler and fits most of the cases
 - Performs better than transactions
- Fine-grained control with transactions
 - Can make reception/acknowledgement AND sending atomic!

Agenda

- Durability and persistence
- AMQP Transactions
- AMQP Acknowledgements
- **Dead letter exchanges**
- Multiple transactional resources

Dead Letter Exchanges

- What should I do with rejected, not requeued messages?
 - They're usually “poisonous”, but worth keeping track of
- A queue can specify a dead letter exchange
- Rejected, not requeued messages are sent to this dead letter exchange
 - The exchange must exist when the first message is dead-lettered
 - It's a “normal” exchange (any type, any binding)
- Dead letter queuing can lead to a human intervention
 - Last chance processing
- RabbitMQ adds information to dead-lettered messages
 - Queue, reason, time, routing keys

NOTE

A message is also sent to the queue dead letter exchange when its TTL expires or when the queue length limit is exceeded.

Dead letter exchanges

- Declare dead letter exchange at queue creation
 - x-dead-letter-exchange and x-dead-letter-routing-key arguments

▼ Add a new queue

Name:	<input type="text" value="quotations"/> *
Durability:	<input type="text" value="Durable"/>
Auto delete:	(?) <input type="text" value="No"/>
Message TTL:	(?) <input type="text"/> ms
Auto expire:	(?) <input type="text"/> ms
Max length:	<input type="text"/>
Dead letter exchange:	(?) <input type="text" value="rejected.records"/>
Dead letter routing key:	(?) <input type="text" value="rejected.quotations"/>
Arguments:	<input type="text"/> = <input type="text"/> String ▾

The exchange the rejected/expired messages are sent to

Optional routing key that replaces the original one

Add queue

Lab

Implementing a Reliable Message Flow (Part 1: "Durability and Persistence, Acknowledgements, and Transactions")

Agenda

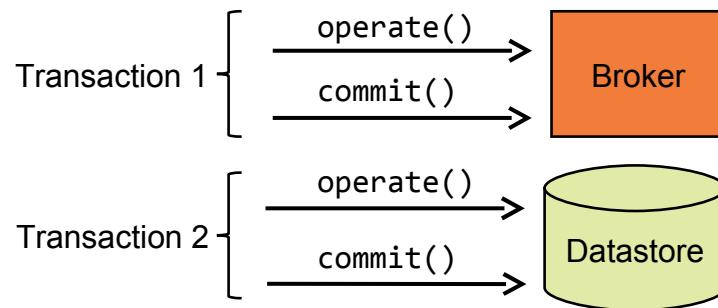
- Durability and persistence
- AMQP Transactions
- AMQP Acknowledgements
- Dead letter exchanges
- **Multiple transactional resources**

Multiple Transactional Resources

- Message processing is usually interleaved with business processing
 - E.g., on message reception, update the database
- Two transactional resources!
 - Broker (with message acknowledgment)
 - Database (commit the changes)
- All of this needs to be atomic
 - No partial failures!

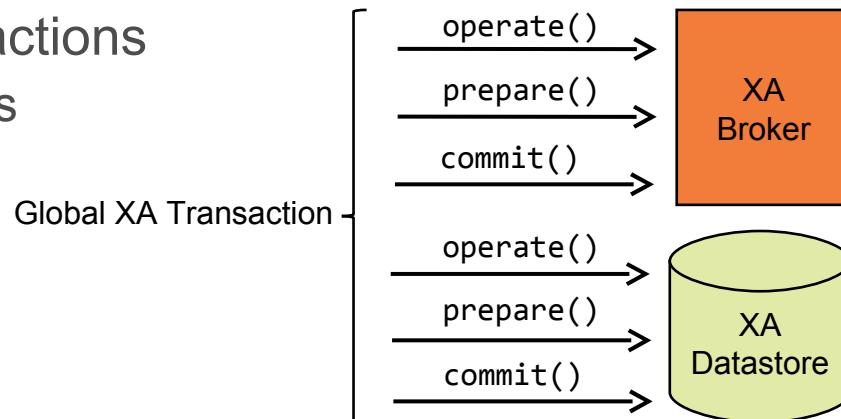
Local Transactions on Multiple Resources

- Two transactional resources (database, broker)
 - Each resource transaction has to be committed individually
 - A rollback on a single transaction doesn't affect any other transaction
 - Cannot enlist transactions
- Referred to as *local* transactions



Global Transactions

- Two transactional resources (database, broker)
 - A transaction can be committed/rolled back atomically
 - Resources has to support XA protocol
 - Resources are coordinated by a dedicated transaction manager
- Referred to as global transactions
 - Or *distributed* transactions



Local Transactions in Java

- Local Java transactions are managed:
- Directly by the developer
 - Calling the `beginTX()` and `commitTX()` or `rollbackTX()` method
 - Either by directly using the client resource API, or JTA
 - Have to write the technical code inside the business code
- By the container / lightweight container
 - Spring automatically manages TX, hiding them for the developer
 - Through some helper classes, or by configuration / annotations
 - By creating a Java EE Resource managed by the container

Global Transactions in Java

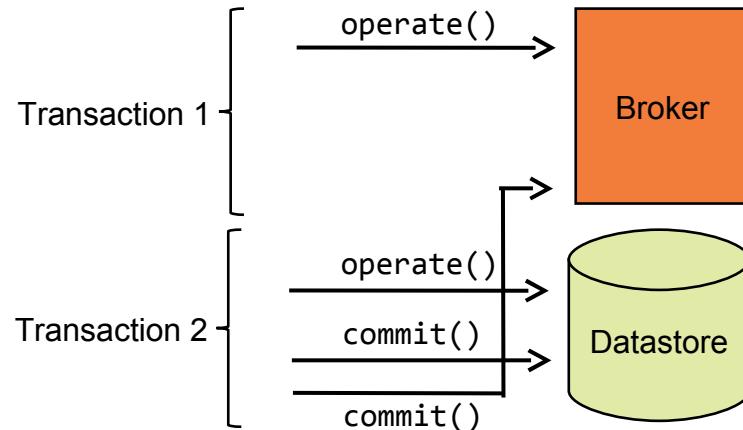
- JTA has support for global transactions
- A dedicated JTA transaction manager handles global transactions
- The JTA transaction manager is usually part of a full blown Java EE application server
 - Standalone JTA transaction managers also exist

XA Drawbacks

- XA Transactions have limitations and drawbacks
 - It is generally costly and impacts performance
 - It can be difficult to set-up
 - It is not 100% reliable (there is still a window where things can go wrong)

Best Effort Pattern (1-Phase Commit)

- The workaround to XA is to use the best effort pattern
 - It is very close to the XA model without drawbacks
 - The critical window is very short (two operations for two commits)
 - When compared to local TX, the best effort pattern has no performance overhead
 - But it's not real XA



Transactions in AMQP

- AMQP supports both local and global transactions
- RabbitMQ supports only local transactions
 - 1-Phase Commit of the best effort pattern is well suited for AMQP / RabbitMQ
- Spring AMQP supports best effort pattern out of the box

Transactions in AMQP

- Best effort pattern can lead to duplicate messages
 - If something goes wrong between the DB and the broker commits
 - Message goes back to the queue and is redelivered
 - Business processing can happen twice!
- Processing must be idempotent
 - Certain types of processing are idempotent (e.g., update a flag to true)
 - Otherwise, must make processing idempotent (detect duplicate, ignore it, and send ack)

Summary

- Durability and persistence protect against message loss
- AMQP transactions improve reliability
- AMQP acknowledgements should be used by consumers to inform the broker when processing has completed
- Dead letter exchanges can be useful for handling rejected messages
- Best-effort pattern is recommended when there are multiple transactional resources

Lab

Implementing a Reliable Message Flow (Part 2: "Multiple Transactional Resources and Best Effort")

RabbitMQ Clustering

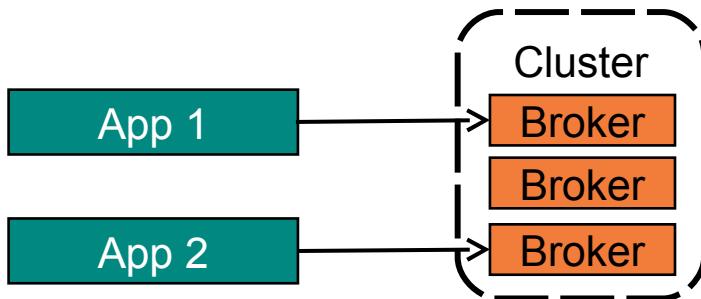
Agenda

- Clustering overview
- Setting up clustering
- Network partitions

Clustering

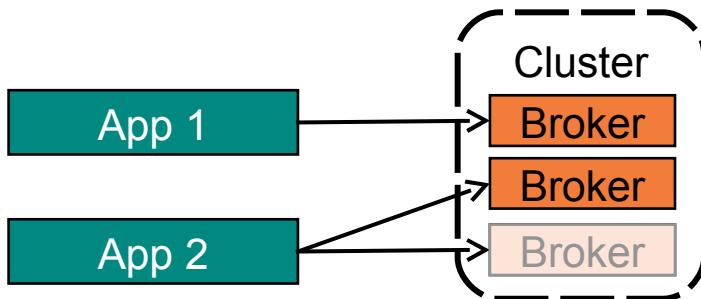
- Clustering means using multiple nodes to construct a single *logical* broker
 - All nodes behave identically to clients
 - Clients see same exchanges, queues, users, vhosts
- Clustering enables scalability and high availability
- Scalability: scale messaging throughput by adding nodes
 - Several nodes to publish to and consume from
- High availability: no interruption of service if a node fails
 - No single point of failure

Clustering



- Scalability:

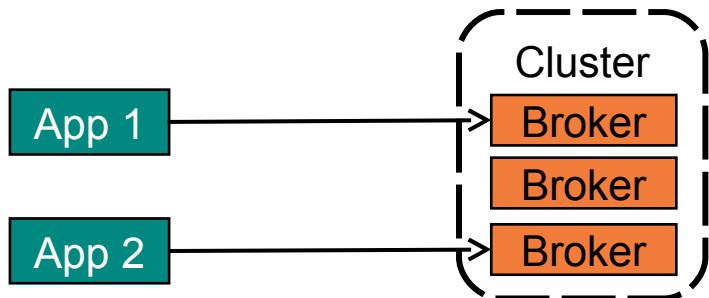
- Load is balanced across nodes
- Is not automatic, must be designed accordingly



- High Availability (HA):

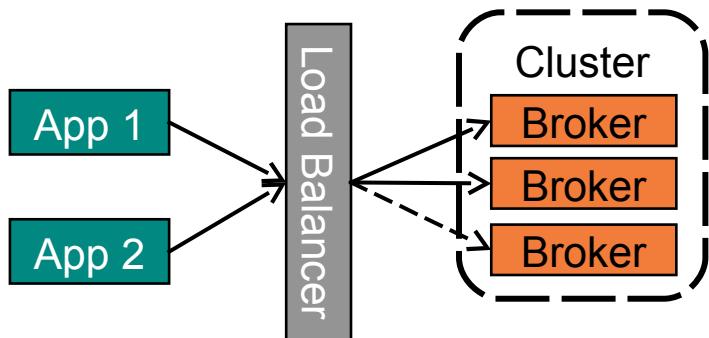
- If node fails, client can connect to another one
- Is only one part of an HA solution, clustering by itself is not inherently HA

Clustering Client Connectivity



- Option 1:

- Clients have list of nodes to connect to, iterate over list until they connect successfully
- Advantages: no additional components required
- Disadvantages: clients must be aware of the cluster membership, and update configurations when the cluster changes



- Option 2:

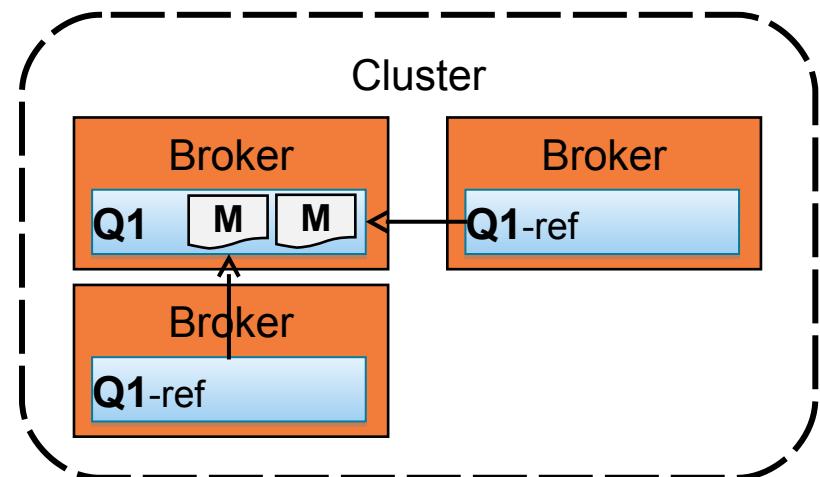
- Clients connect through a software or hardware load balancer
- Advantages: clients always connect to a single virtual IP, do not need to be aware of cluster membership
- Disadvantages: single point of failure, additional component required

Load Balancer

- The load balancer connects client to a node as selected by the specified algorithm eg. least connections, round robin, etc.
- Clients don't have to know all the nodes and deal with failures
- Load balancing can be at the TCP-level, no knowledge of AMQP necessary
- All classical solutions work
 - HAProxy – <http://haproxy.1wt.eu/>
 - IPVS – <http://www.linuxvirtualserver.org/software/ipvs.html>
 - Other software solutions
 - Hardware solutions

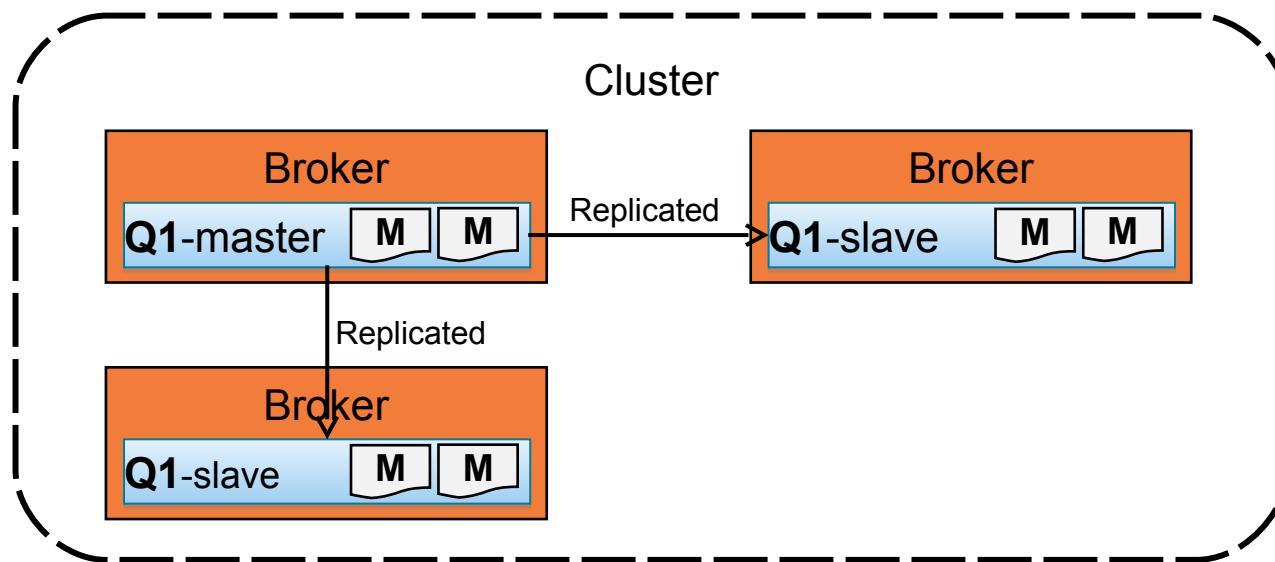
Clustering does *not* equal full HA

- Standard default queues are not replicated across nodes in the cluster
- A single node in the cluster contains the queue messages
- Other nodes simply hold pointers to the node with the actual messages



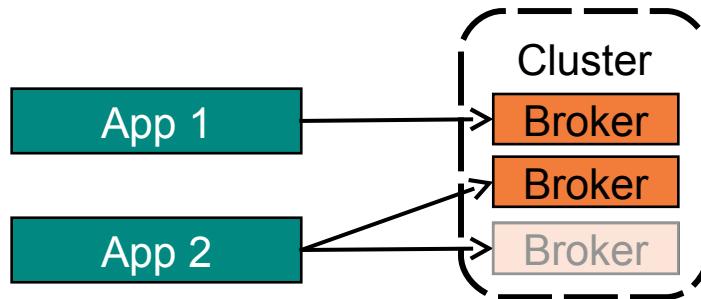
Clustering + Mirrored Queues = Full HA

- Mirrored queues replicate the messages to other node(s) in the cluster.
- In the event of the node hosting the master copy going down, another node can take over



Client Side Failover

- Clients must be aware of node failures
- Clients must handle reconnections



Clustering

- It is important to understand the concepts behind RabbitMQ clustering, since:
 - A failure isn't fully transparent for a client (especially a consumer)
 - Messages in a queue can be lost after a failure

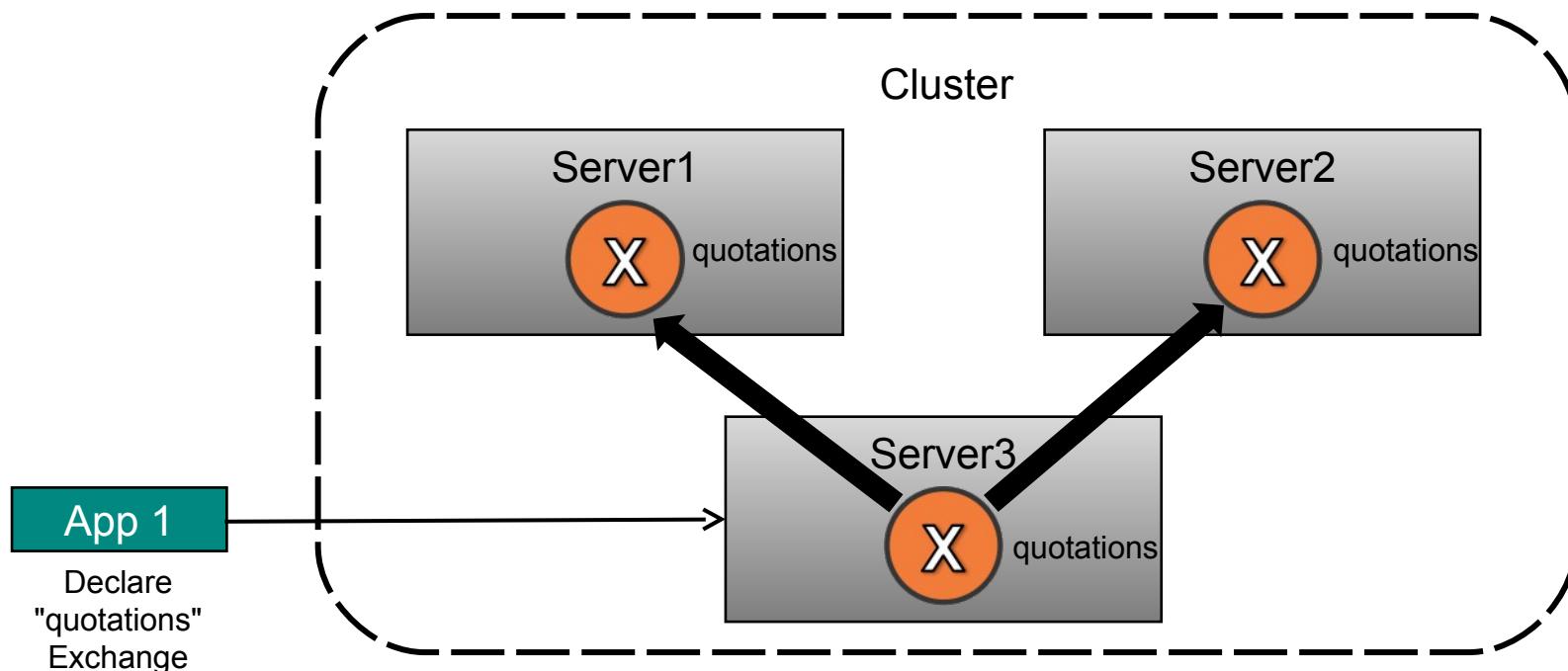
NOTE

Some components can help to cope with these concerns (e.g., a load balancer to make reconnection simpler for clients).

What's Inside a Cluster?

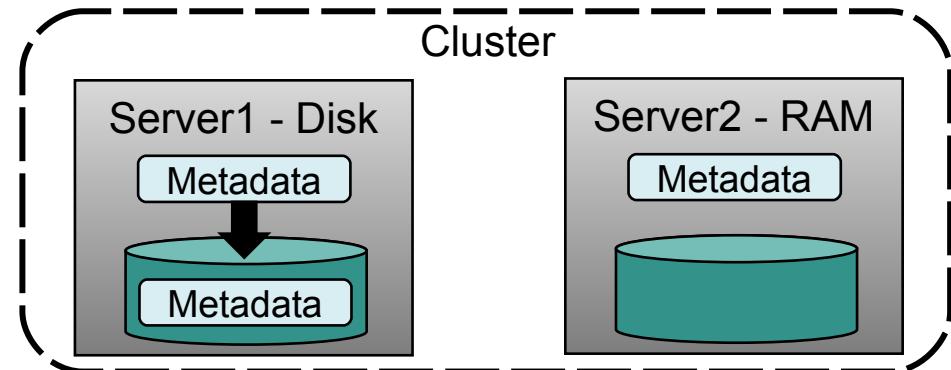
- A single or cluster RabbitMQ node stores metadata on
 - Users
 - Queues
 - Bindings
 - Exchanges
 - Virtual hosts
- A cluster node also stores metadata about the other nodes
- Any declaration on a node is replicated to the other nodes i.e. the metadata is replicated to all nodes in the cluster

Cluster Behavior for an Exchange Declaration



RAM vs Disk Nodes

- A node can store metadata in RAM or on disk
 - RAM storage allows fast access (but metadata is lost in case of a restart)
 - Disk storage is slower but metadata will survive a restart
- A cluster node can be either a RAM or a disk node
- A single RabbitMQ node can *only* be a disk node
 - You don't want to lose durable definitions between restarts!



RAM vs Disk Nodes

- General rule: always use disk nodes
 - This is safer
 - A cluster of only RAM nodes would result in data loss if the cluster crashed
- RAM node are faster, but only for resource management
 - E.g., creating a queue
- Disk nodes are rarely a source of bad performance
 - Instead, investigate other potential root causes such as persistent messages, acknowledgments, etc.

Behind the Scenes

- Clustering exchanges and bindings
 - Just a routing table
 - Clustering them means distributing them on the nodes
 - Each node can make routing decisions without network calls
- Clustering users and virtual hosts
 - Clustering them means distributing them on the nodes
 - In classic cases, doesn't account for much replication traffic
- Clustering a queue
 - All nodes replicate the queue metadata
 - But only one node stores the queue's content (messages)
 - RabbitMQ sets up pointers on all other nodes, to the node with the queue contents

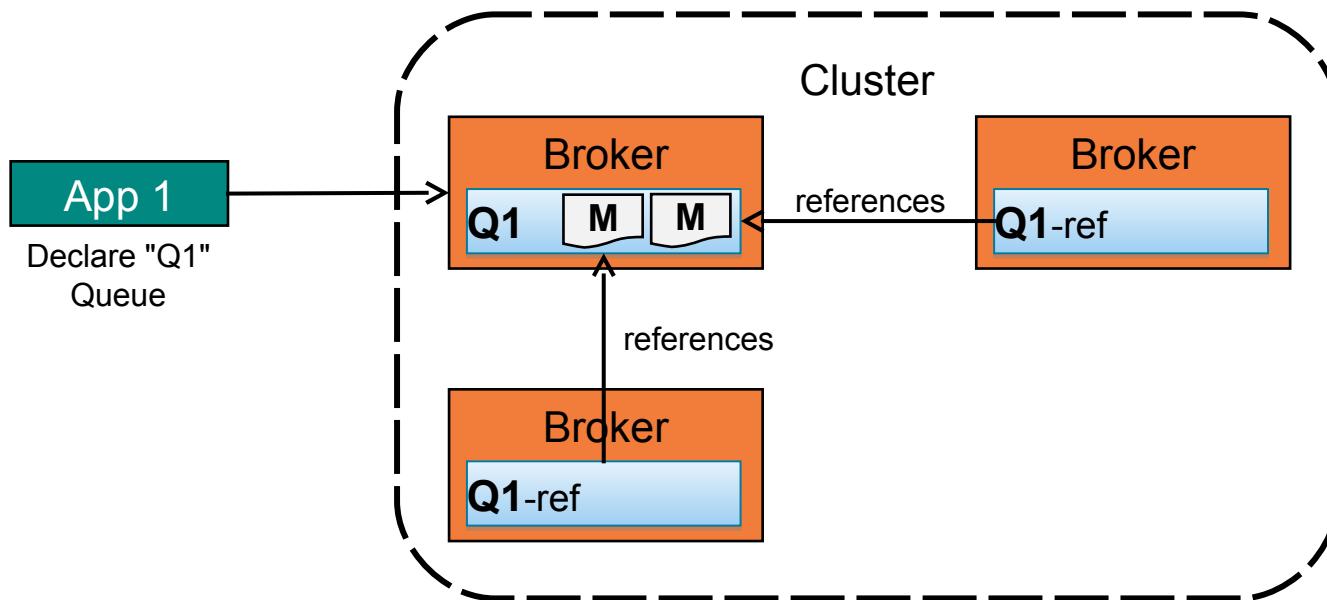
NOTE

RabbitMQ provides mirrored queues to replicate queue messages on multiple nodes. The high availability module covers mirrored queues.

Queue Creation

- Creating a standard (non-mirrored) queue on a cluster
 - All nodes will see it (cluster)
 - $(n - 1)$ nodes will just have a pointer to the real queue on a node
 - 1 node will have the physical storage
 - RabbitMQ puts the physical queue on the node the client declaring the queue was connected to at the time the queue was declared
 - No rebalancing is done

Standard Non-Mirrored Queue in a Cluster



Agenda

- Clustering overview
- **Setting up clustering**
- Network partitions

Creating a Cluster (overview)

- Prerequisites
 - Must be same RabbitMQ and Erlang versions on all nodes
 - A special file must be copied to all cluster nodes for RabbitMQ to communicate between nodes (next slide)
- Creating a cluster on three machines
 - server1, server2, server3 in the next slides
- Start RabbitMQ on all nodes
 - For all nodes except the first:
 - Stop the RabbitMQ application, leaving Erlang running
 - Run the RabbitMQ clustering, telling it where to find the other cluster nodes
 - Restart the RabbitMQ application

Erlang Cookie

- RabbitMQ's clustering builds on top of Erlang OTP
 - Open Telecom Platform
- OTP requires nodes to share the same cookie to communicate
- It's just a text file to copy between nodes, eg:

```
CGUKYMYPRRNFOSUYVXPE
```

- User running RabbitMQ needs read permissions on the file

Step 1: Copy the Erlang Cookie

- Start the server on one of the nodes

```
server1$ service rabbitmq-server start
```

- It creates the Erlang cookie if it doesn't already exist:
 - `/var/lib/rabbitmq/.erlang.cookie` (UNIX)
 - `C:\Users\Current User\.erlang.cookie` or `C:\Documents and Settings\Current User\.erlang.cookie` (Windows)
- Copy the Erlang cookie to the correct location on all the other nodes you want to cluster
 - User running RabbitMQ needs read permissions on the file!

Step 2: Start The Other Servers

- Login to all the other servers
- Start RabbitMQ on all the other servers

```
server2$ service rabbitmq-server start
```

```
server3$ service rabbitmq-server start
```

Step 2: Start The Other Servers

- Confirm the RabbitMQ servers are running by checking their status
- Confirm the RabbitMQ servers are all standalone
 - i.e. not in a cluster already

```
server1$ rabbitmqctl cluster_status
Cluster status of node rabbit@server1 ...
[{nodes,[{disc,[rabbit@server1]}]}, {running_nodes,[rabbit@server1]}]
...done.
```

```
server2$ rabbitmqctl cluster_status
Cluster status of node rabbit@server2 ...
[{nodes,[{disc,[rabbit@server2]}]}, {running_nodes,[rabbit@server2]}]
...done.
```

```
server3$ rabbitmqctl cluster_status
Cluster status of node rabbit@server3 ...
[{nodes,[{disc,[rabbit@server3]}]}, {running_nodes,[rabbit@server3]}]
...done.
```

Step 3: Stop the joining nodes

- Choose a server that will be the first cluster node
 - Don't touch it; leave it running
 - You can also stop it (`stop_app`) and reset it, to be sure to start from a clean state
- On all other servers
 - Stop the RabbitMQ application

NOTE

A reset removes the node from any cluster it belongs to, removes all metadata, and deletes all persistent messages. Do it only if you want a fresh cluster!

Step 3: Stop the joining nodes

```
server2$ rabbitmqctl stop_app  
Stopping node rabbit@server2 ...  
...done.
```

Leaves the Erlang
VM running.

```
server3$ rabbitmqctl stop_app  
Stopping node rabbit@server3 ...  
...done.
```

Step 4: Join the Node to the Cluster

- On all servers that have undergone step 3
 - Join the cluster by specifying the other nodes
- Starting with server2

Join as a RAM node

Specify the node to join

```
server2$ rabbitmqctl join_cluster --ram rabbit@server1
Clustering node rabbit@server2 with [rabbit@server1] ...
...done.
server2$ rabbitmqctl start_app
Starting node rabbit@server2 ...
...done.
```

NOTE

Joining a cluster resets the joining node.

Step 5: Check the Cluster Status

- Confirm the status on the newly clustered nodes

```
server1$ rabbitmqctl cluster_status
Cluster status of node rabbit@server1 ...
[{nodes,[{disc,[rabbit@server1]},{ram,[rabbit@server2]}]},  

 {running_nodes,[rabbit@server2,rabbit@server1]}]
...done.
```

A 2-node cluster

```
server2$ rabbitmqctl cluster_status
Cluster status of node rabbit@server2 ...
[{nodes,[{disc,[rabbit@server1]},{ram,[rabbit@server2]}]},  

 {running_nodes,[rabbit@server1,rabbit@server2]}]
...done.
```

Both nodes are up

Step 5: RAM vs. Disk Nodes

- Going back to the cluster status:

```
server1$ rabbitmqctl cluster_status
Cluster status of node rabbit@server1 ...
[{nodes,[{disc,[rabbit@server1]},{ram,[rabbit@server2]}]}, 
 {running_nodes,[rabbit@server2,rabbit@server1]}]
...done.
```

rabbit@server1 is a disk node

rabbit@server2 is a RAM node

Step 5: RAM vs. Disk Nodes

- How can we specify if a node is a RAM or disk?
- Going back to what we executed:

```
server2$ rabbitmqctl join_cluster --ram rabbit@server1
Clustering node rabbit@server2 with [rabbit@server1] ...
...done.
server2$ rabbitmqctl start_app
Starting node rabbit@server2 ...
...done.
```

- For a disk node, exclude the “--ram” parameter

```
server2$ rabbitmqctl join_cluster rabbit@server1
Clustering node rabbit@server2 with [rabbit@server1] ...
...done.
server2$ rabbitmqctl start_app
Starting node rabbit@server2 ...
...done.
```

Step 6: Join the Node to the Cluster

- On all servers that have undergone step 3
- Join the cluster by excluding the “--ram” parameter
- Let’s do that again with server3, as a disk node

server3 will join as a disk node

```
server3$ rabbitmqctl join_cluster rabbit@server1
Clustering node rabbit@server3 with [rabbit@server1, rabbit@server3] ...
...done.
server3$ rabbitmqctl start_app
Starting node rabbit@server3 ...
...done.
```

Step 7: Check the Cluster Status

- Confirm the status on the newly clustered nodes

```
server1$ rabbitmqctl cluster_status
Cluster status of node rabbit@server1 ...
[{"nodes": [{"disc": ["rabbit@server3", "rabbit@server1"], "ram": ["rabbit@server2"]}], "running_nodes": ["rabbit@server3", "rabbit@server2", "rabbit@server1"]}]
...done.
```

A 3-node cluster

```
server2$ rabbitmqctl cluster_status
Cluster status of node rabbit@server2 ...
[{"nodes": [{"disc": ["rabbit@server3", "rabbit@server1"], "ram": ["rabbit@server2"]}], "running_nodes": ["rabbit@server3", "rabbit@server1", "rabbit@server2"]}]
...done.
```

```
server3$ rabbitmqctl cluster_status
Cluster status of node rabbit@server3 ...
[{"nodes": [{"disc": ["rabbit@server3", "rabbit@server1"], "ram": ["rabbit@server2"]}], "running_nodes": ["rabbit@server1", "rabbit@server2", "rabbit@server3"]}]
...done.
```

All nodes are up

Changing a Node Type (RAM – Disk)

- To change the type of the node
 - Stop the application
 - Change the type with `change_cluster_node_type <new_type>`
 - Start the application

```
server2$ rabbitmqctl stop_app
server2$ rabbitmqctl change_cluster_node_type disk
server2$ rabbitmqctl start_app
```

Removing a Cluster Node

- Stop the application and reset the node

```
server1$ rabbitmqctl stop_app
Stopping node rabbit@server1 ...done.
server1$ rabbitmqctl reset
Resetting node rabbit@server1 ...done.
server1$ rabbitmqctl start_app
Starting node rabbit@server1 ...done
```

- server1 is now an independent broker
- Don't stop the only disk node in a cluster!
 - Change the type of another node from RAM to disk first

Upgrading a cluster

- 2 scenarios
 - The cluster can run during the upgrade
 - The cluster must be taken down
- When must the cluster be taken down?
 - When upgrading Erlang
 - When from major/minor RabbitMQ version to another
 - E.g. from 3.0.x to 3.1.x, from 2.x.x to 3.x.x

Upgrading a cluster (scenario 1)

- If the cluster can run
 - Just stop and upgrade nodes one after the other

Upgrading a cluster (scenario 2)

- All nodes must be stopped
 - Stop a disk node last (a.k.a. the “upgrader”)
- Change the RabbitMQ version (and Erlang if appropriate)
- Start the upgrader first
- The upgrader automatically updates the metadata structure if needed
- Start the other nodes

Clustering on a Single Machine

- A cluster can run on a single machine
- Very useful for experimenting failure scenarios
- Just need to avoid port collisions!

```
$ RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=server1 rabbitmq-server start  
$ RABBITMQ_NODE_PORT=5673 RABBITMQ_NODENAME=server2 rabbitmq-server start
```

- To stop:

```
$ rabbitmqctl -n server1 stop  
$ rabbitmqctl -n server2 stop
```

Clustering on a Single Machine

- Specify port for the management plugin if it's enabled:

```
RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=server1 \
RABBITMQ_SERVER_START_ARGS="-rabbitmq_management listener [{port,15672}]" \
rabbitmq-server start
```

NOTE

Remove any port settings the rabbitmq.config file contains.

Lab

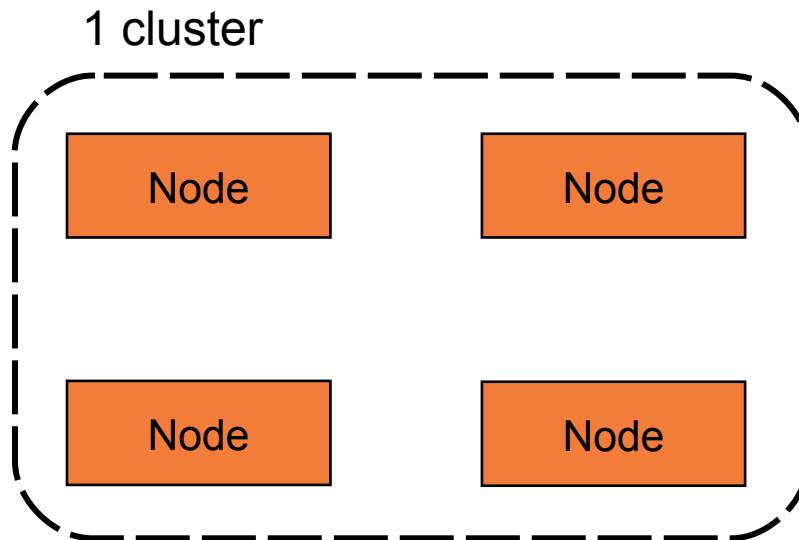
Setting Up A RabbitMQ Cluster

Agenda

- Clustering overview
- Setting up clustering
- **Network partitions**

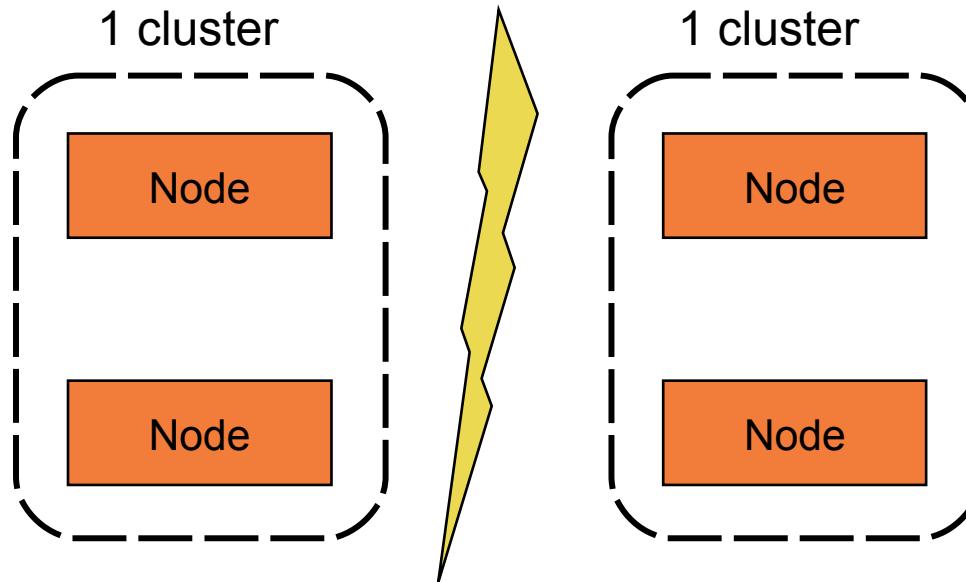
Network Partitions

- What happens when cluster nodes can't reach each other?



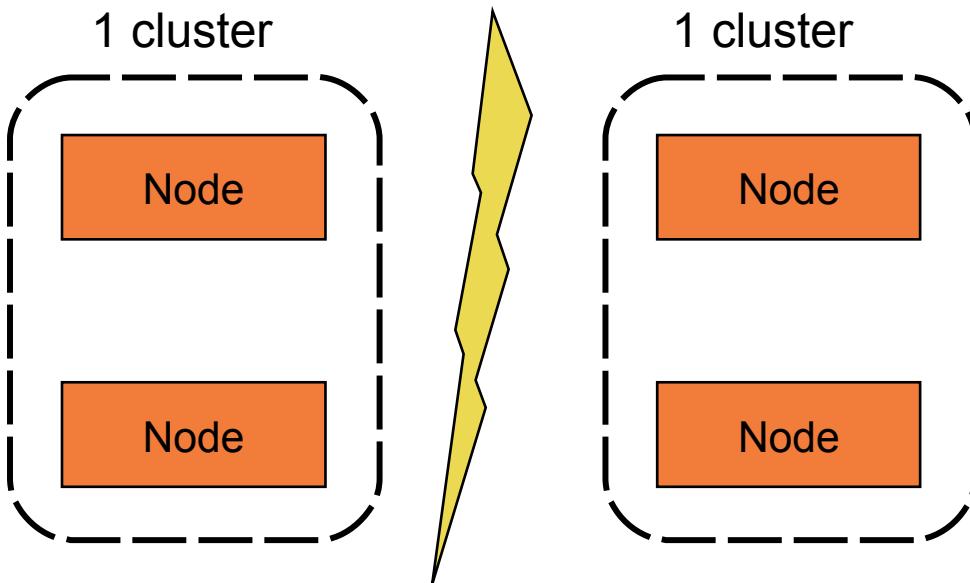
Network Partitions

- Default behavior in case of network partition: 2 independent clusters



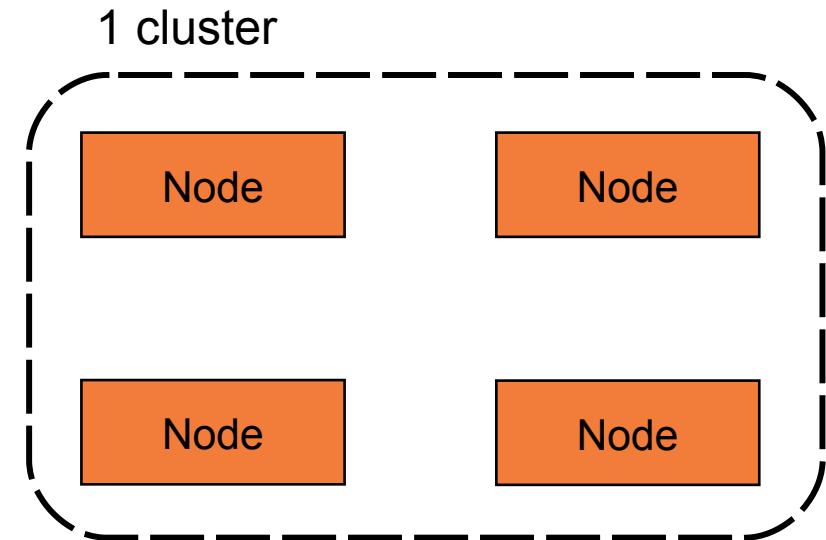
Network Partitions

- When network connectivity is restored, still 2 independent clusters
- This is the default behavior (a.k.a. “ignore”) : no explicit action after the partition



Network Partitions Recovery

- Choose one “winning” partition/cluster
- Stop and start all nodes in the “losing” cluster, they’ll rejoin the cluster
- Everything in the losing cluster is lost



Network Partitions Detection

- Partition happens after `net_ticktime` seconds of non-connectivity
 - Default is 60 seconds
- Nodes report the partition in the log (when connectivity is restored)

```
=ERROR REPORT==== 24-Mar-2016::18:02:30 ===  
Mnesia(rabbit@server1): ** ERROR ** mnesia_event got  
{inconsistent_database, running_partitioned_network, rabbit@server2}
```

- Information also available from the command line

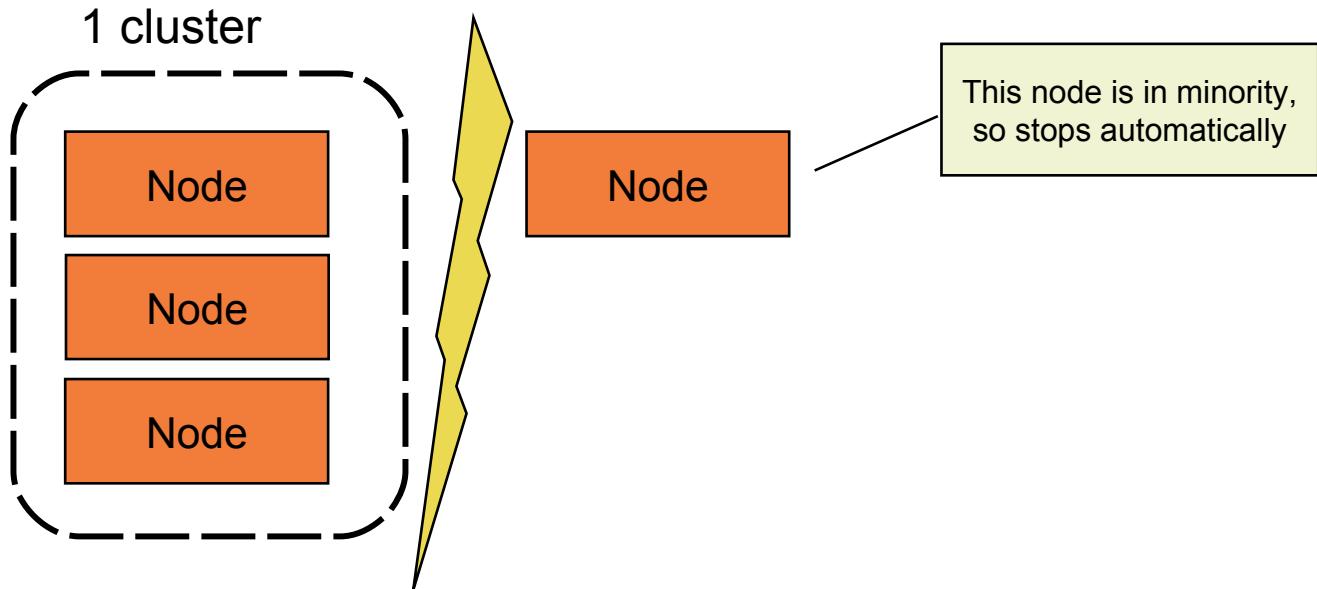
```
$ rabbitmqctl cluster_status  
Cluster status of node rabbit@server1 ...  
[{nodes,[{disc,[rabbit@server2,rabbit@server1]}]},  
 {running_nodes,[rabbit@server1,rabbit@server2]},  
 {partitions,[{rabbit@server1,[rabbit@server2]},  
 {rabbit@server2,[rabbit@server1]}]}]  
...done.
```

Network Partitions Handling

- Default behavior: RabbitMQ doesn't do anything
 - “ignore” mode
- Other modes are available, they can intervene:
 - When the partition is detected
 - When connectivity is restored
- Modes:
 - pause_minority: automatically stop nodes in a minority cluster
 - pause_if_all_down: automatically stop nodes that cannot reach a given list of other nodes
 - autoheal: on connectivity recovery, choose automatically a winning partition and restart the node in the losing partition

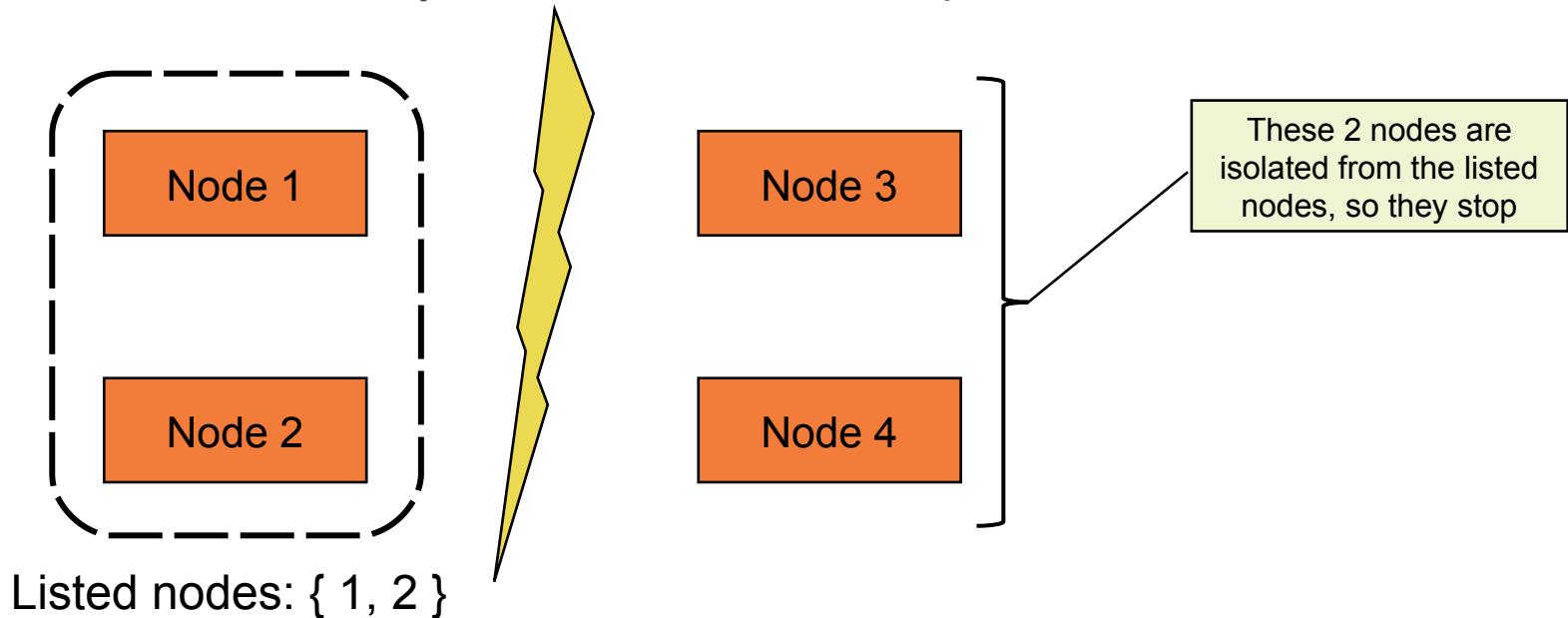
Network Partitions – pause_minority

- Nodes in a minority partition stop
 - Minority = fewer or equal than half the total number of nodes



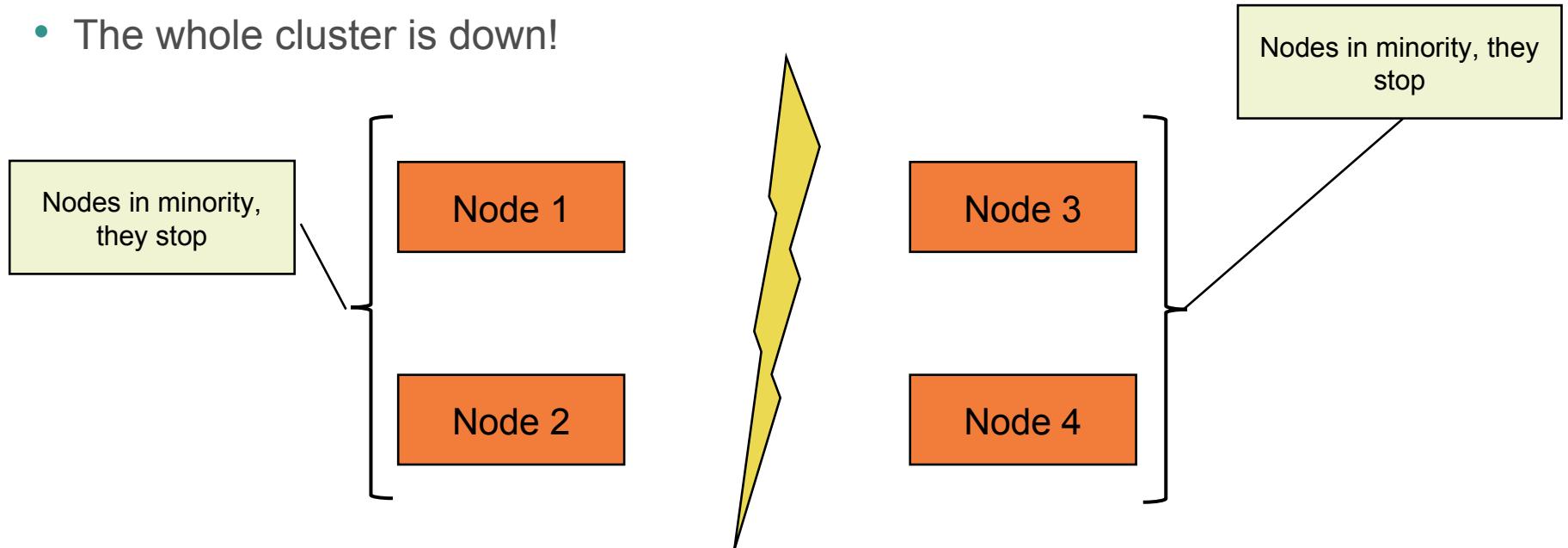
Network Partitions – pause_if_all_down

- List of “important” nodes at configuration time
- If one node loses connectivity with those nodes, it stops



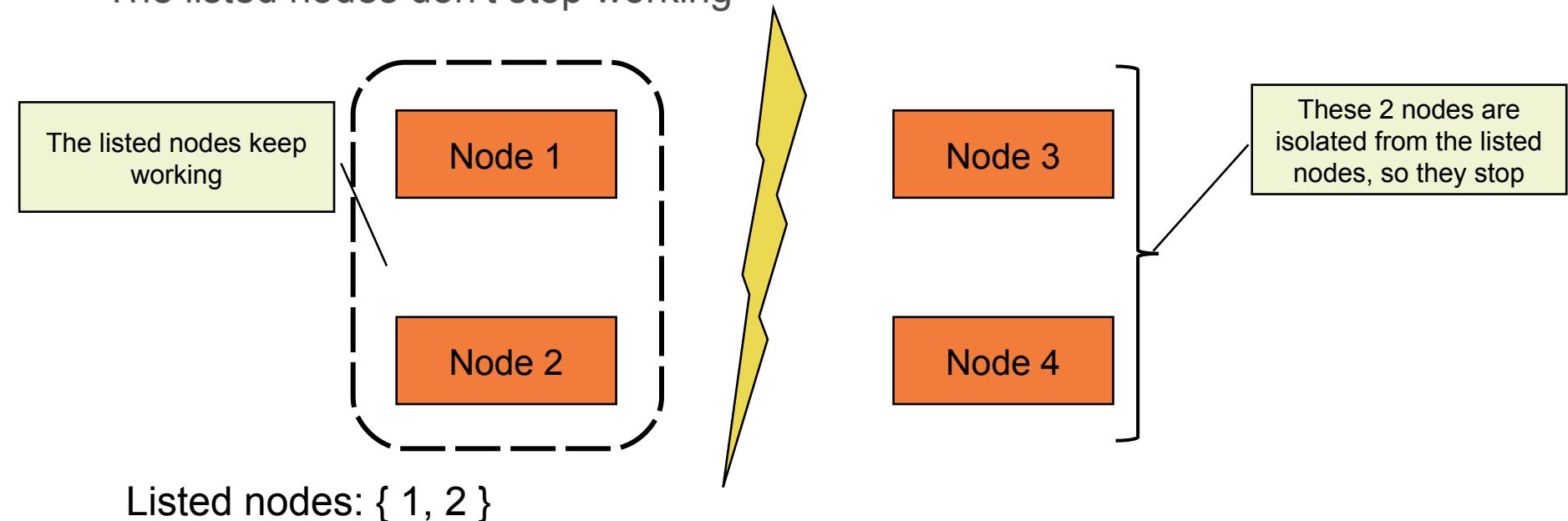
pause_minority vs. pause_if_all_down

- *With pause_minority*
- The whole cluster is down!



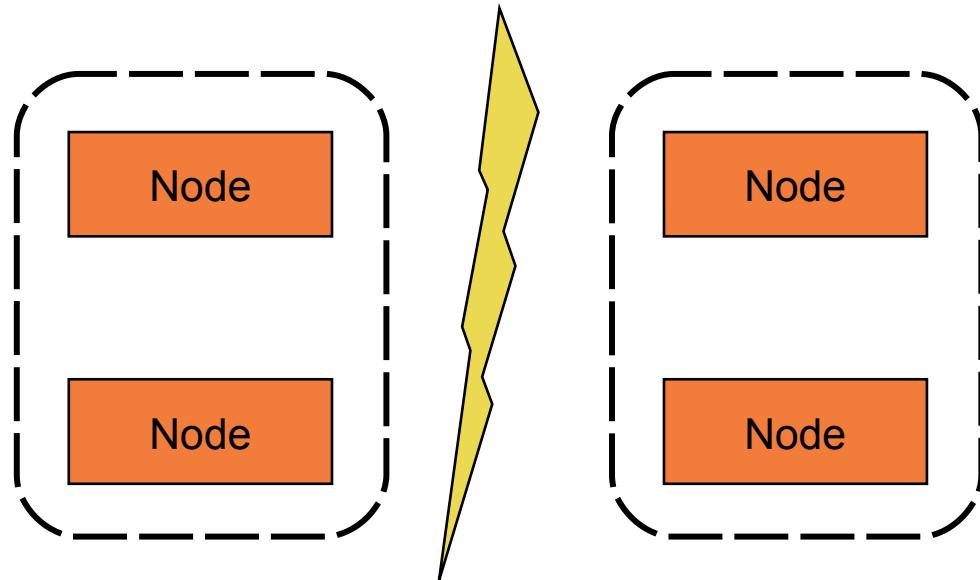
pause_minority vs. pause_if_all_down

- *With pause_if_all_down*
- The listed nodes don't stop working



Network Partitions – autoheal

- RabbitMQ chooses a winning partition when connectivity is restored
- How?
 - Partition with the most clients connected
 - In case of draw, partition with the most nodes
 - In case of draw, random choice



Network Partitions Handling Configuration

- In RabbitMQ configuration file

```
[  
{rabbit, [  
    {cluster_partition_handling, ignore }  
]}  
.]
```

ignore | pause_minority | autoheal

- With pause_if_all_down

```
[  
{rabbit, [  
    {cluster_partition_handling,  
     {pause_if_all_down, ['rabbit@server1', 'rabbit@server2'], ignore }  
]}  
.]
```

ignore | autoheal
(because listed nodes can also lose connectivity)

Network Partitions Handling

- Use ignore
 - If you have a very reliable network and plan to recover manually
- Use pause_minority / pause_if_all_down
 - If your network is less reliable
 - If you want minimal manual intervention once connectivity is restored
- Use autoheal
 - If you favor availability over consistency
 - If you can afford to lose some data

NOTE

The complete reference: <http://www.rabbitmq.com/partitions.html>

Summary

- RabbitMQ clusters form a single logical server
- Important to understand underlying clustering concepts
- Creating a cluster is easy
- RabbitMQ offers several options for handling network partitions

High Availability

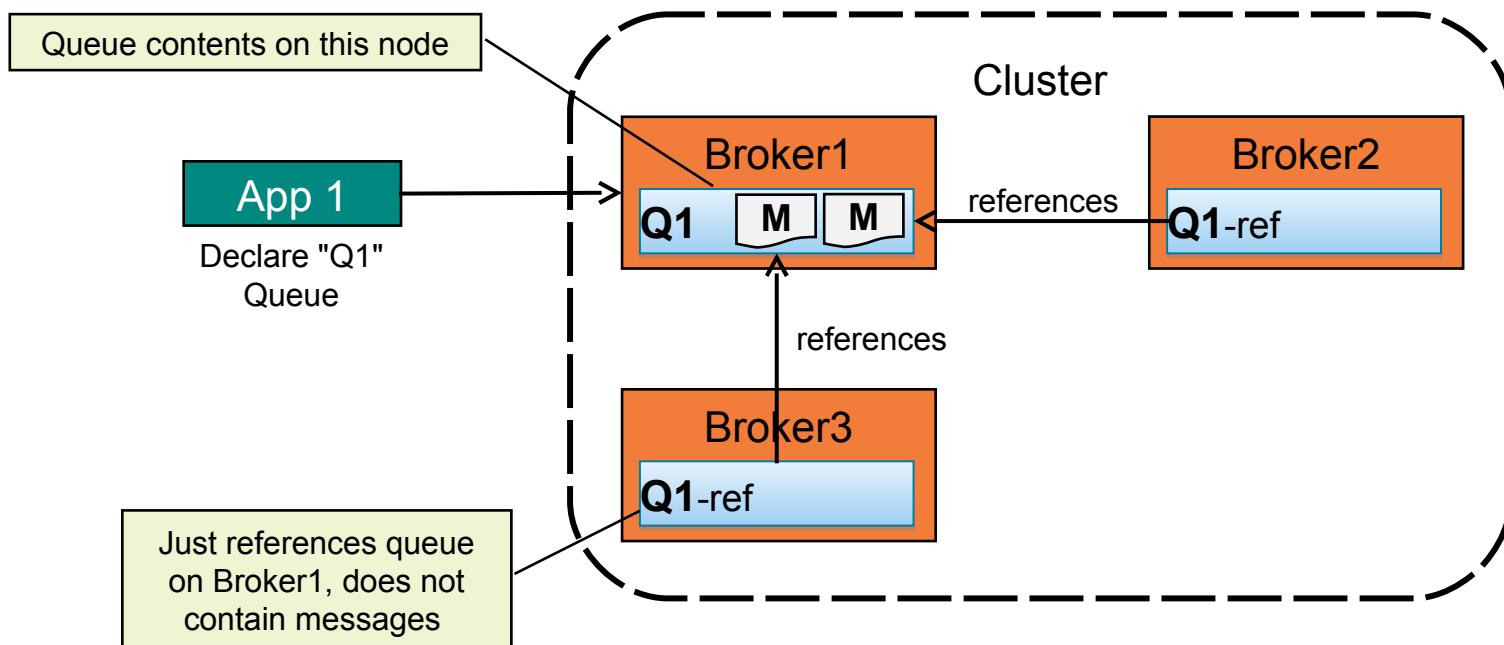
Agenda

- Cluster node failures and consequences
- Mirrored queues
- Slaves synchronization
- Failover handling for the client

Failure of a Cluster Node

- When a cluster node fails, the main concern is the queues it hosts
- Remember the defaults of a clustered queue:
 - All nodes share the metadata
 - One node contains the queue messages
 - Other nodes have a pointer to the owner of the queue
- Why don't we care about exchanges in case of failure?
 - The other nodes have the exchanges metadata
 - An exchange doesn't contain any messages, a queue does!

Reminder: Queue in a Cluster



Failure of a Cluster Node

- What happens if the “owner” node dies?
 - Consumers lose their subscriptions
 - New messages matching the bindings are silently discarded
- If queue was durable, it cannot be re-created
 - The owner must be restored
- If queue wasn’t durable, it can be re-created
 - Even if the former owner isn’t running

Node Failure: Consequences for an Exchange

- An exchange is just a lookup table
- The channel handles the routing from exchanges to queues
- If a node fails, all the other nodes know the lookup table
 - This is a cluster, each node has the metadata of all exchanges
- The exchange is still visible to clients connected to other nodes

Node Failure: Consequences for an Exchange

- What happens if a node fails while the message is being routed?
 - If the producer used "fire and forget", the message can be lost
 - To avoid lost messages on the producer side:
 - Use transactions
 - Or use RabbitMQ's publisher confirms extension
 - In both cases, the producer can know if the message didn't make it to the queue
- At the end, the producer needs to reconnect to another node
 - Or wait for the node to be up again

Agenda

- Cluster node failures and consequences
- **Mirrored queues**
- Slaves synchronization
- Failover handling for the client

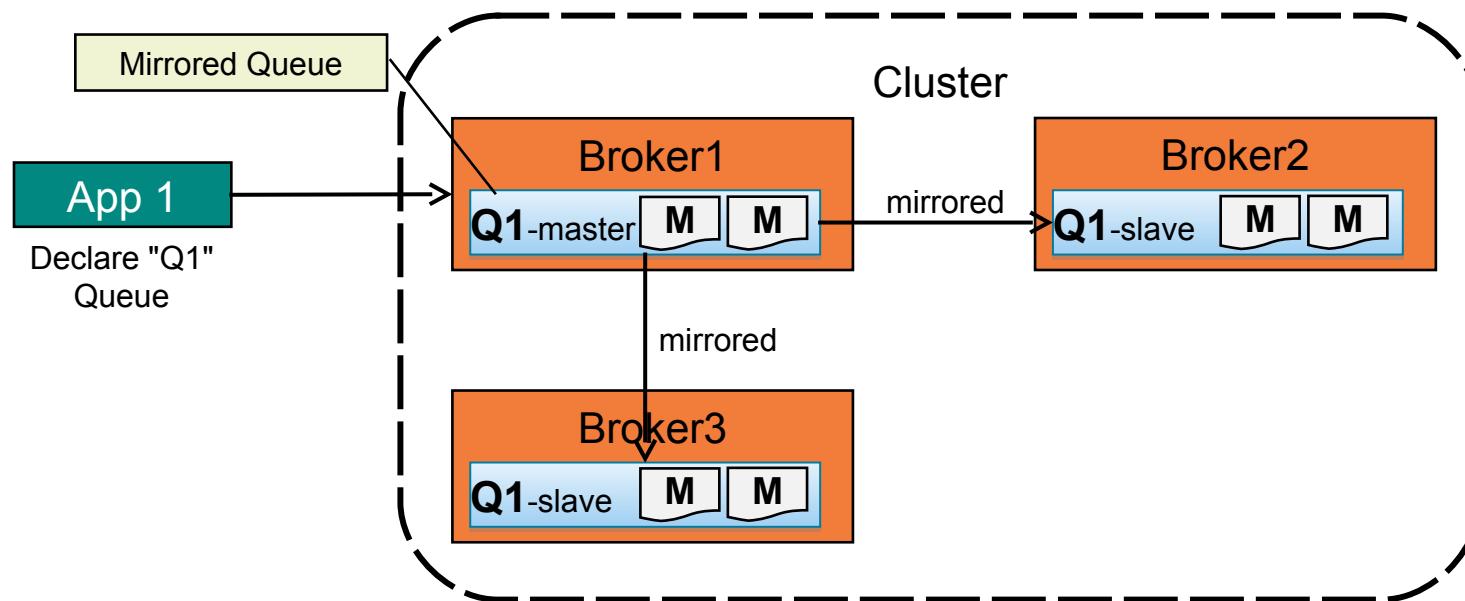
Mirrored Queues

- A mirrored queue replicates its content on multiple cluster nodes
- The content of a mirrored queue lives on one master node
- The content is replicated on other nodes of the cluster
- Mirrored queues provide high availability
- Mirrored queues also affect performance

NOTE

Mirrored queues were introduced in RabbitMQ 2.6.0. There was no built-in HA solution before mirrored queues.

Mirrored Queue



Mirrored Queue Creation

- Mirrored queues are designated through broker policies
- Regex's on queue names are used to determine whether policy applies
- Policies applied
 - At queue creation, if queue's name matches policy
 - At policy creation, on existing queues if they match
- Regular queues can be promoted to mirrored queues!

Mirrored Queue Policy Options

- Two HA parameters when creating the policy
 - ha-mode: mode of mirroring/replication
 - ha-params: parameters, depends on the mode

ha-mode	ha-params	Description
all	(none)	Mirrored across all nodes (simpler but slower)
exactly	Count of mirrors	Total number of mirrors, nodes are selected randomly
nodes	Names of nodes	Mirrored to the specified list of nodes. Fine-grained but implies knowledge of the nodes in the cluster

- A policy specifying exactly 2 mirrors will provide a good balance between redundancy and performance

Declaring the HA Policy

- The management plugin supports the creation of HA policies

▼ Add / update a policy

Name:	HA	*	
Pattern:	^ha\.	*	
Definition: (?)	ha-mode	= exactly	String ▾ *
	ha-params	= 2	Number ▾
		=	String ▾
Priority:			
<input type="button" value="Add policy"/>			

- The matching queue(s) become mirrored

Overview						Messages			Message rates		
Name	Node	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver / get	ack
ha.quotations	server1@acogoluegnes-zenbook +1		D	HA	Idle	0	0	0			

Queue Master Distribution Strategies

- How to distribute queue masters between nodes?
 - You don't want a node to be the master of too many queues
- 3 strategies:
 - Pick the node hosting the minimum numbers of masters
 - Pick the node the client that declares the queue is connected to
 - Pick a random node

NOTE

The default is to pick the node the client declaring the queue is connected to.

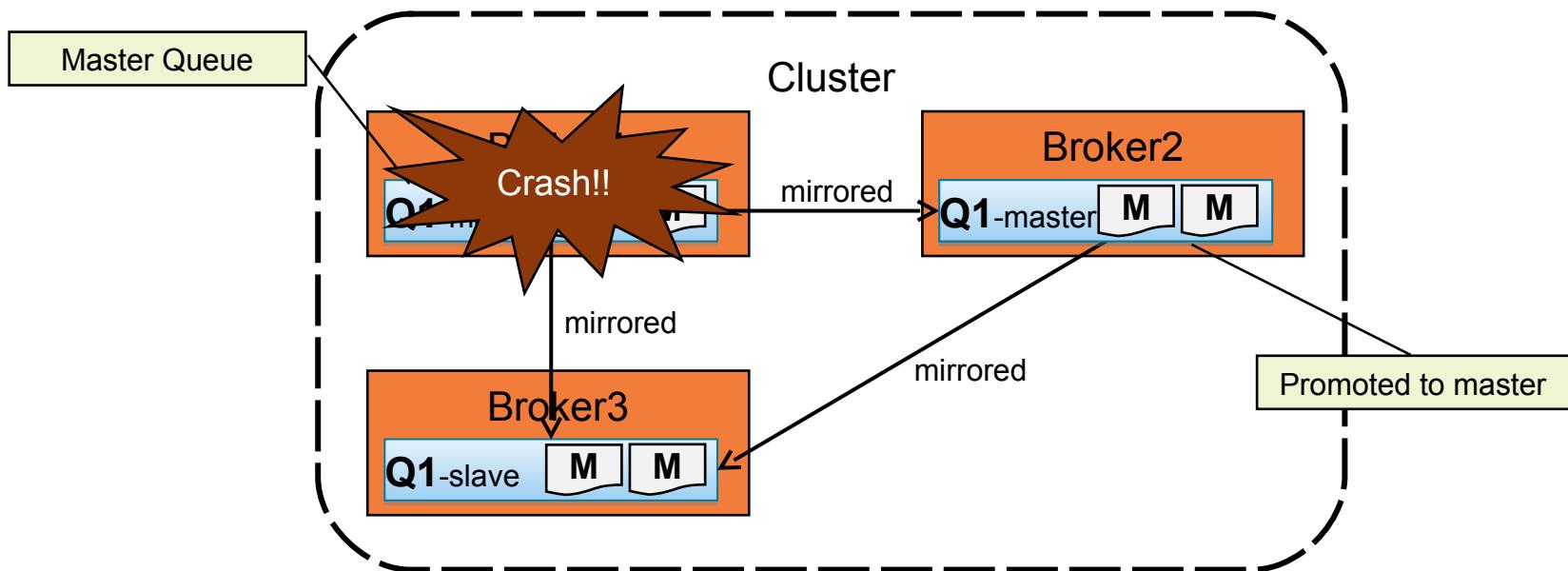
Queue Master Distribution Strategies

- 3 ways
 - Use the `x-queue-master-locator` queue declare argument
 - Use the `queue-master-locator` policy key
 - Use the `queue_master_locator` key in the configuration file
- 3 possible values:
 - `min-masters`
 - `client-local`
 - `random`

Failure of the Master of a Mirrored Queue

- The eldest slave becomes the new master
 - It has the best chance to be synchronized with the master
 - If it's not, messages held only on the master are lost!
- Messages that are pending acknowledgment are re-queued
- Consumers connected to the failing node must reconnect to another node
- Consumers of the remaining nodes don't have to do anything

Failure of the Master of a Mirrored Queue



Master Promotion Behavior

- Why does the newly promoted master re-queue unack'd messages?
- The new master can't know what truly happened to these messages
 - Perhaps the consumers sent the acknowledgment but...
 - ... The master could have failed
 - When the ACK was between the consumer and the master
 - When the ACK was broadcast from master to slaves
 - While the consumer is still processing the message
 - Consumers must
 - Pick up the new master
 - Be able to deal with duplicate messages

Nodes Joining a Cluster with Mirrored Queues

- If the HA Policy dictates there are not enough mirrors for a given queue when a newcomer joins the cluster:
 - Queue mirrors are created as slave queues
 - The new slave queue on the newcomer starts queuing incoming messages
 - Existing messages from the master queue are not replicated to the slave on the newcomer
 - There is no “catch-up” of existing messages
 - The newcomer is considered unsynchronized
 - As messages are consumed, slave queue on newcomer gets synchronized naturally
 - This is the default behavior: no synchronization when joining the cluster

NOTE

RabbitMQ provides a mechanism to automatically or manually synchronize new slave queues.

Nodes Joining a Cluster with Mirrored Queues

- If messages stay a long time in the queue, a newcomer doesn't bring much redundancy!
- Best practices
 - Start up all nodes and then create a mirrored queue
 - Do not use the broker to hold messages for long periods of times
 - Consume messages as fast as possible

Monitoring Mirrored Queues

- Slaves are synchronized:

Overview							Messages			Message rates		
Name	Node	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver / get	ack	
market.us	server1@acogoluegnes-zenbook +2		D	HA for market	Idle	0	0	0				

▼ Add a new queue

Synchronised mirrors: server2@acogoluegnes-zenbook,server3@acogoluegnes-zenbook

- One slave is not synchronized:

Overview							Messages			Message rates		
Name	Node	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver / get	ack	
market.us	server1@acogoluegnes-zenbook +1 +1		D	HA for market	Idle	16	0	16	0.00/s			

▼ Add a new queue

Unsynchronised mirrors: server2@acogoluegnes-zenbook

Monitoring Mirrored Queues

- Slaves are synchronized when the last 2 columns are identical

```
$ rabbitmqctl list_queues name pid slave_pids synchronised_slave_pids
Listing queues ...
quotations  <'s1@host'.1.12915.0> \
[<'s2@host'.2.10752.0>, <'s3@host'.3.8343.0>] \
[<'s2@host'.2.10752.0>, <'s3@host'.3.8343.0>]
...done.
```

- Slaves are unsynchronized when the last 2 columns are not identical

```
$ rabbitmqctl list_queues name pid slave_pids synchronised_slave_pids
Listing queues ...
quotations  <'s1@host'.1.12915.0> \
[<'s2@host'.2.10752.0>, <'s3@host'.3.8343.0>] \
[<'s2@host'.2.10752.0>]
...done.
```

Agenda

- Cluster node failures and consequences
- Mirrored queues
- **Slaves synchronization**
- Failover handling for the client

Synchronizing Unsynchronized Slaves

- Automatic synchronization when creating policy
 - Set ha-sync-mode key to automatic
 - Default is manual

▼ Add / update a policy

Name:	HA
Pattern:	^ha\.
Apply to:	Queues
Priority:	
Definition:	ha-mode = exactly ha-params = 2 ha-sync-mode = automatic =

Activate automatic synchronization

HA mode (?) | HA params (?) | HA sync mode (?)
Federation upstream set (?) | Federation upstream (?)
Queues Message TTL | Auto expire | Max length | Max length bytes
Dead letter exchange | Dead letter routing key
Exchanges Alternate exchange

Add policy

Synchronizing Unsynchronized Slaves

- How to synchronize manually?
- Command line

```
$ rabbitmqctl sync_queue myqueue
```

- Management plugin (in queue details)

Details					
Parameters	durable: true	Status	Idle since 2013-08-06 11:36:12	Node	server1@acogoluegnes-zenbook
Policy	ha	Consumers	0	Mirrors	server3@acogoluegnes-zenbook
Exclusive owner	None	Memory	86.9kB		server2@acogoluegnes-zenbook (unsynchronised) Synchronise

Mirrored Queues Synchronization

- Avoid explicit synchronization
 - Either automatic or manual
 - Queue is unresponsive during the sync
- Prefer the natural synchronization
 - Works well if consumers are working correctly

Synchronization Optimization

- RabbitMQ can synchronize messages in batches
 - Can make the sync much faster
 - E.g. from 60 seconds to a few seconds for 1M messages
- Use the `ha-sync-batch-size` key in the policy

Set a sync batch size

Definition:	Value	Type	*
ha-mode	= exactly	String	*
ha-params	= 2	Number	
ha-sync-mode	= automatic	String	
ha-sync-batch-size	= 20000	Number	
	=	String	

Add policy

Agenda

- Cluster node failures and consequences
- Mirrored queues
- Slaves synchronization
- **Failover handling for the client**

Failover on Client Side

- Node failures aren't transparent to client
- Clients must deal with node failures
- Clients bindings provide callbacks in case of errors
- These callbacks make the recovery code easier to write
- There are many failure scenarios: always test yours!

Shutdown Hook

- The connection has a shutdown hook
- The developer can add a ShutdownListener

```
connection.addShutdownListener(new ShutdownListener() {  
    @Override  
    public void shutdownCompleted(ShutdownSignalException cause) {  
    }  
});
```

Called when connection is closed

Shutdown Hook in Case Of Server Failure

- Shutdown listeners are called when the connection closes
- The close method can be called in normal condition
- Check the cause of the closing to execute recovery code

```
connection.addShutdownListener(new ShutdownListener() {  
    @Override  
    public void shutdownCompleted(ShutdownSignalException cause) {  
        if(cause.isHardError()) {  
            // recovery code  
        }  
    }  
});
```

True in case of connection failure

What To Do After Failure

- A client works against exchanges and queues
 - It can also define bindings
- Bindings and non-mirrored queues can disappear after a failure
- A client should re-declare everything it uses after it detected a failure
 - Especially non-mirrored queues and bindings
- Declarations are idempotent
 - They work even if the resource already exists...
 - ... as long as the to-be-declared resource has the same options as the existing one
- Don't take for granted your resources are still valid after a failure!

Reconnection After A Failure

- The cause of failure can be a crash or a network glitch
- Client can try to reconnect to the same node
 - No problem if cause was network glitch
 - Problem if node is down
- This is where a load balancer helps
 - The entry point for the client is the load balancer
 - Load balancer detects node failure and dispatches on available nodes
 - Client isn't coupled to cluster nodes
- If there's no load balancer, client needs to know nodes in advance
 - Doesn't have to be hardcoded, can be part of configuration

Client Reconnection (manual)

- Reconnect randomly or with any other algorithm

```
private static final String [] HOSTS =  
    {"server1", "server2", "server3"};  
...  
Connection connection = null;  
Random random = new Random();  
while(connection == null) {  
    try {  
        String host = HOSTS[random.nextInt(HOSTS.length)];  
        factory.setHost(host);  
        connection = factory.newConnection();  
    } catch (Exception e) {  
        // let's try again...  
    }  
}
```

Servers to connect to

Keep trying until
connection successful

Pick server randomly

Client Reconnection (built-in support)

- Built-in, but no control over the choice algorithm

```
Address[] addresses = new Address[]{  
    new Address("server1",5672),  
    new Address("server2",5672),  
    new Address("server3",5672),  
};  
Connection connection = factory.newConnection(addresses);
```

The servers to connect to

Pick the first that
succeeds in the array

Failover For Producers

- Failover for producers is easy as they're active
 - Same thing for active consumers
- This is the typical workflow of a producer:
 - Gets a channel from the connection
 - Sends messages
 - Closes the channel
- What should the producer consider?
 - Getting a workable channel: if the connection is dead, opening a channel throws an exception. The producer needs to open a new connection and to retry.
 - Ensure the sent messages made it to the broker: this isn't always a requirement, but sometimes it matters.

Producers Ensure Broker Got Sent Messages

- Remember, sending uses "fire and forget" by default
 - When the method returns, the message is on its way to the broker
- "Fire and forget" is fast, but doesn't provide guaranteed delivery
- Use transactions to be sure the messages reached the broker

```
channel.txSelect();  
  
byte[] msg = "NASDAQ".getBytes();  
channel.basicPublish(stockExchange, stockCategory, null, msg);  
  
channel.txCommit();
```

Create the transaction

Publish the message

**Commit the transaction. When this method returns,
it means the message reached the broker**

Publisher Confirms

- Transactions provide strong guarantees, but they're slow
 - They're also part of AMQP
- RabbitMQ provides the publisher confirmations extension
- Publisher can be sure messages made it to the broker
 - This is faster than transactions

```
channel.confirmSelect();
```

Enable publisher
confirmations on channel

```
byte[] msg = "NASDAQ".getBytes();
channel.basicPublish(stockExchange, stockCategory, null, msg);
```

```
channel.waitForConfirms();
```

Method blocks until broker
responds it received all messages

Failover for Asynchronous Consumers

- Asynchronous consumer can die with its connection
- Register a shutdown listener to handle node failure
- Just after connection creation:

```
public void connectAndListen() {  
    ...  
    connection.addShutdownListener(new ShutdownListener() {  
        @Override  
        public void shutdownCompleted(ShutdownSignalException cause) {  
            if(cause.isHardError()) {  
                connectAndListen();  
            }  
        }  
    });  
    ...  
}
```

Re-launch connection creation and listener registration after the connection dies.

Queues and Failure

- Remember, standard non-mirrored queues are only on one node at a time
 - Other nodes reference the owning node
- If the owning node dies, the queue disappears from the cluster!
- This can have profound impacts on consumers!

NOTE

Mirrored queues do survive a node failure, though all the messages may not.

Failure of a Node for an Async Consumer

- Use case:
 - Asynchronous consumer is consuming from a queue
 - Queue is on server 1
 - Consumer is connected to server 2
- Failure scenario
 - server2 dies
 - Consumer detects failure and manages to connect on server1
- Everything is fine!
 - server2 could have come back again, consumer could have reconnected to it
- It is when the owning node fails that things go wrong...

Failure of the Owner of a Queue

- Use case:
 - Async consumer is consuming from a queue
 - Queue is on server 1
 - Consumer is connected to server 2
- Failure scenario
 - server1 dies, the queue disappears from the cluster
 - Consumer doesn't detect anything, it keeps on listening and see nothing
 - server1 comes back to life, queue comes back in the cluster
 - Consumer still doesn't see anything!
- RabbitMQ provides an extension to notify the consumer

Consumer Cancellation Extension

- Consumer listens on a queue but isn't connected to owning node
- Consumer can be notified when owning node dies
 - It can recreate the queue on the node it is listening on

```
channel.basicConsume("quotations", true, new DefaultConsumer(channel) {  
    @Override  
    public void handleDelivery(String consumerTag, Envelope envelope,  
        AMQP.BasicProperties properties, byte[] body) throws IOException {}  
  
    @Override  
    public void handleCancel(String consumerTag) throws IOException {  
        recreateQueueAndRestartListening();  
    }  
});
```

Called when the owning node dies

Failure of the Master of a Mirrored Queue

- The queue still exists, messages aren't lost
- A consumer cancellation notification is not sent to consumers connected to other nodes
 - No action is required those consumers, they will continue to receive messages

Automatic recovery with the Java client

- The Java client supports automatic recovery of
 - Connections
 - Topology (queues, exchanges, bindings, consumers)
- In practice, in case of failure of the node the client is connected to:
 - The client reconnects and restores connection and channel listeners
 - The client can redeclare resources (queues, exchanges, etc)

NOTE

The Java client supports automatic recovery since version 3.3.0.

Automatic recovery activation

- Just activate the flag at the ConnectionFactory level

```
factory.setAutomaticRecoveryEnabled(true);  
Connection connection = factory.newConnection();
```

Set up the flag

Connection that will
recover automatically

- Even better to use an array of addresses

```
factory.setAutomaticRecoveryEnabled(true);  
Address[] addresses = new Address[]{  
    new Address("server1",5672),  
    new Address("server2",5672),  
    new Address("server3",5672),  
};  
Connection connection = factory.newConnection(addresses);
```

In case of failure,
automatically tries the
different addresses

Automatic recovery semantics

- Kicks in only when the client gets disconnected
- Restores connection and channel listeners
- Restores QoS setting on channels
- Doesn't handle re-publication (use public confirms or transaction instead)
- Restore resources
 - Resource restoration is activated by default, but can be disabled

NOTE

For full details about the automatic recovery:

<http://www.rabbitmq.com/api-guide.html#recovery>

Automatic recovery conclusion

- A feature of the Java client
 - Not available on all clients
- Just a commodity, doesn't prevent from knowing about important details
 - Failure of nodes, consumer cancellation, etc.
- Don't blindly rely on it
 - Test your failure scenarios

Recovery With Spring AMQP

- Higher-level modules can offer support for recovery
- This makes the client code less tedious and error-prone
- E.g. Spring AMQP takes care of recovery:
 - Automatic re-connection to the broker in case of failure
 - Automatic declaration callback to initialize resources after re-connection
 - Transparent retry

Summary

- Cluster node failures can result in message loss
- Mirrored queues provide high-availability and reduce the chance of message loss in the event of a node failure
- Slave synchronization policies can be configured
- Be certain to test client failover handling

Lab

Creating and Managing an HA Cluster

Plugins

Extending RabbitMQ functionality

LDAP, Shovel, Federation, STOMP

Agenda

- RabbitMQ plugins introduction
- LDAP authentication
- Shovel
- Federation
- Shovel vs Federation
- STOMP

RabbitMQ Plugins

- RabbitMQ capabilities can be extended with plugins
 - AMQP client
 - Management
 - STOMP
 - LDAP authentication
 - Shovel
 - Federation
 - etc.
- Can build custom plugins
 - <http://www.rabbitmq.com/plugin-development.html>

RabbitMQ Plugins

```
$ rabbitmq-plugins list
Configured: E = explicitly enabled; e = implicitly enabled
| Status:   * = running on rabbit@myhost
|/
[e*] amqp_client           3.6.1
[ ] cowboy                  1.0.3
[ ] cowlib                  1.0.1
[e*] mochiweb               2.13.0
[ ] rabbitmq_amqp1_0        3.6.1
[ ] rabbitmq_auth_backend_ldap 3.6.1
[ ] rabbitmq_auth_mechanism_ssl 3.6.1
[ ] rabbitmq_consistent_hash_exchange 3.6.1
[ ] rabbitmq_event_exchange 3.6.1
[ ] rabbitmq_federation     3.6.1
[ ] rabbitmq_federation_management 3.6.1
[E*] rabbitmq_management    3.6.1
[e*] rabbitmq_management_agent 3.6.1
[ ] rabbitmq_management_visualiser 3.6.1
[ ] rabbitmq_mqtt            3.6.1
[ ] rabbitmq_recent_history_exchange 1.2.1
[ ] rabbitmq_sharding         0.1.0
[ ] rabbitmq_shovel           3.6.1
[ ] rabbitmq_shovel_management 3.6.1
[ ] rabbitmq_stomp             3.6.1
[ ] rabbitmq_tracing          3.6.1
[e*] rabbitmq_web_dispatch   3.6.1
[ ] rabbitmq_web_stomp        3.6.1
[ ] rabbitmq_web_stomp_examples 3.6.1
[ ] sockjs                   0.3.4
[e*] webmachine              1.10.3
```

How to Install Plugins

- Plugins must be in the plugins directory
- Enabling or disabling plugins has no effect on a running RabbitMQ server ? restart is required to activate plugins
- Command to see the list of available / installed plugins

```
$ rabbitmq-plugins list
```

- Command to enable a plugin

```
$ rabbitmq-plugins enable <plugin-name>
```

- Command to disable a plugin

```
$ rabbitmq-plugins disable <plugin-name>
```

Agenda

- RabbitMQ plugins introduction
- **LDAP authentication**
- Shovel
- Federation
- Shovel vs Federation
- STOMP

LDAP Authentication

- This plugin lets the RabbitMQ server use an LDAP server to perform
 - Authentication (who can log in)
 - Authorization (with which permissions)
- Plugin must be enabled before using it (requires server restart)

```
$ rabbitmq-plugins enable rabbitmq_auth_backend_ldap
```

- The LDAP authentication provider needs to be added to the providers list in the configuration

```
[{rabbit,  
  [{auth_backends, [rabbit_auth_backend_ldap, rabbit_auth_backend_internal]}]  
 }].
```

LDAP Authentication Usage

- Minimal configuration file with some options might look like

```
[  
  {rabbit, [{auth_backends, [rabbit_auth_backend_ldap]}]},  
  {rabbitmq_auth_backend_ldap,  
   [ {servers, ["ldap.mycompany.com"]},  
     {user_dn_pattern, "cn=${username},ou=people,dc=example,dc=com"}  
     {port, "389"} ] }]  
].
```

The diagram illustrates the configuration options for LDAP authentication. It shows three components: 'LDAP server' (represented by a yellow box), 'Port' (also represented by a yellow box), and 'DN to bind' (another yellow box). Arrows connect the 'servers' and 'user_dn_pattern' fields in the configuration code to their respective annotations. The 'port' field is also annotated with a 'Port' box.

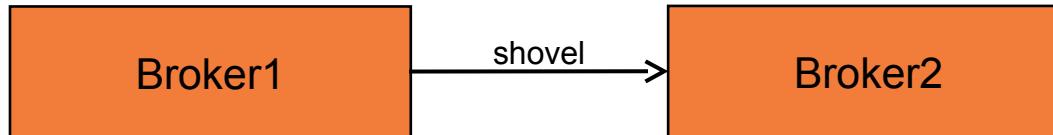
- Authorization can also be configured to control
 - vhost access
 - Resource (exchange, queue, binding)
 - Full access (i.e., administrator)

Agenda

- RabbitMQ plugins introduction
- LDAP authentication
- **Shovel**
- Federation
- Shovel vs Federation
- STOMP

Shovel Plugin

- Plugin transfers messages from queue on one broker to an exchange on another broker



- Shovel can be on
 - The source broker (typical)
 - A third broker
- Shovel is just a well-written client application
 - Reads messages from source
 - Forwards messages to destination
 - Deals with connection failures
 - Has plenty of options

Shovel Plugin

- Shovel has 2 modules
 - Shovel engine
 - Shovel management console
- Enable both plugins

```
rabbitmq-plugins enable rabbitmq_shovel
rabbitmq-plugins enable rabbitmq_shovel_management
```

Static and dynamic shovels

- There are 2 kinds of shovels
- Static shovels
 - Defined in the broker configuration file
 - Require a restart of the hosting broker to change
 - Many options
- Dynamic shovels
 - Defined in the broker's parameters
 - Can be created and deleted at runtime
 - Fewer options than static shovels

Static Shovel Configuration

- One of the simplest shovel configurations

```
[  
{rabbitmq_shovel,  
 [ {shovels, [ {my_first_shovel,  
             [ {sources,  
                   [ {brokers, [ "amqp://localhost:5672"  
                           ]}  
                     ]}  
                 , {destinations,  
                     [ {broker, "amqp://localhost:5673"}  
                       ]}  
                 , {queue, <<"market">>}  
                 , {publish_fields, [ {exchange, <<"from.market">>}  
                                      , {routing_key, <<"from_shovel">>}  
                                        ]}  
               ]}  
         ]}  
     ]}].
```

The shovel's name

Source and destination broker URI's

- Source queue
- Destination exchange
- Published routing-key

Shovel Plugin Management

- This configuration is displayed in the management console
 - Only when the shovel-management plugin is enabled

Overview Connections Channels Exchanges Queues Admin

Shovel Status

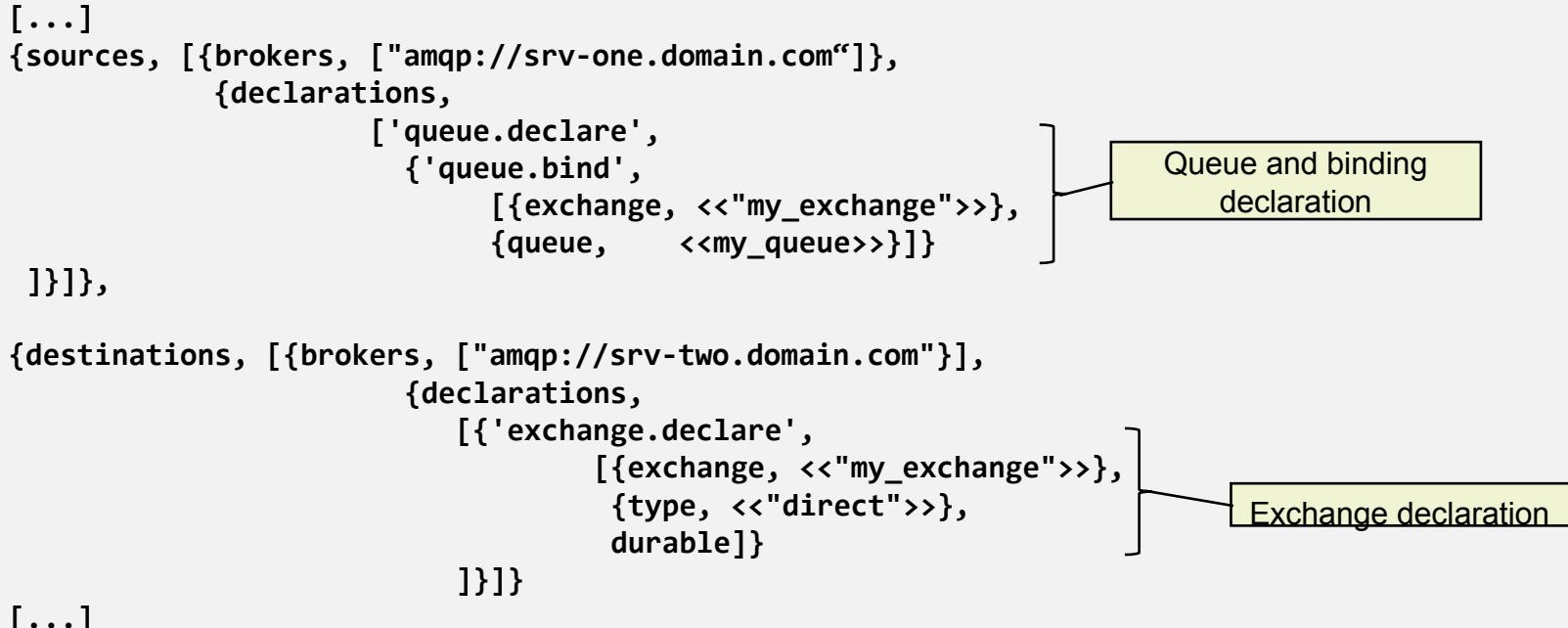
Name	State	Source	Destination	Last changed
my_first_shovel	running	type: network virtual_host: / host: localhost username: guest port: 5672 ssl: false	type: network virtual_host: / host: localhost username: guest port: 5673 ssl: false	2013-08-06 17:11:30

Users
Virtual Hosts
Policies
Shovel Status

Static Shovel Configuration

- Shovel plugin can also declare and bind its AMQP resources
 - Exchanges, bindings, queues

```
[...]  
{sources, [{brokers, ["amqp://srv-one.domain.com"]},  
           {declarations,  
             ['queue.declare',  
              {'queue.bind',  
               [{exchange, <<"my_exchange">>},  
                {queue,     <<my_queue>>}]}  
             ]}]],  
  
{destinations, [{brokers, ["amqp://srv-two.domain.com"]}],  
              {declarations,  
                [ {'exchange.declare',  
                  [{exchange, <<"my_exchange">>},  
                   {type, <<"direct">>},  
                   durable]}  
                ]}]}  
[...]
```



The code snippet shows static configuration for a Shovel. It defines sources and destinations. Sources include a broker at 'srv-one' and declarations for a queue and its binding to an exchange named 'my_exchange'. Destinations include a broker at 'srv-two' and declarations for an exchange named 'my_exchange' with type 'direct' and durability. Annotations with brackets and callouts highlight specific parts of the configuration:

- A bracket groups the 'queue.bind' section under a callout labeled "Queue and binding declaration".
- A bracket groups the 'exchange.declare' section under a callout labeled "Exchange declaration".

Dynamic Shovel Declaration

- A dynamic shovel is defined as a named parameter
- Several ways to declare a dynamic shovel
 - rabbitmqctl set_parameter command
 - HTTP API
 - Management plugin (web UI)

Example of a JSON definition
for rabbitmqctl

```
{  
  "src-uri": "amqp://localhost:5672",  
  "src-queue": "market",  
  "dest-uri": "amqp://localhost:5673",  
  "dest-exchange": "from.market",  
  "dest-exchange-key": "from_shovel"  
}
```

Dynamic Shovel with the web UI

Overview Connections Channels Exchanges Queues Admin

Dynamic Shovels

▼ Shovels

... no shovels ...

▼ Add a new shovel

Name: *

Source: URI (?) Queue:

Destination: URI (?) Exchange: (Routing key:)

Prefetch count:

Reconnect delay: s

Add forwarding headers:

Acknowledgement mode:

Auto-delete

Users
Virtual Hosts
Policies
Shovel Status
Shovel Management

Dynamic Shovel Status

Overview Connections Channels Exchanges Queues Admin

Shovel Status

Name	State	Source		Destination		Last changed
my_dynamic_shovel dynamic	running	amqp://localhost:5672	market queue	amqp://localhost:5673	from.market : from_shovel exchange	2016-03-03 15:03:27

[HTTP API](#) | [Command Line](#)

Users

Virtual Hosts

Policies

Shovel Status

Shovel Management

Update

Agenda

- RabbitMQ plugins introduction
- LDAP authentication
- Shovel
- **Federation**
- Shovel vs Federation
- STOMP

Federation Plugin

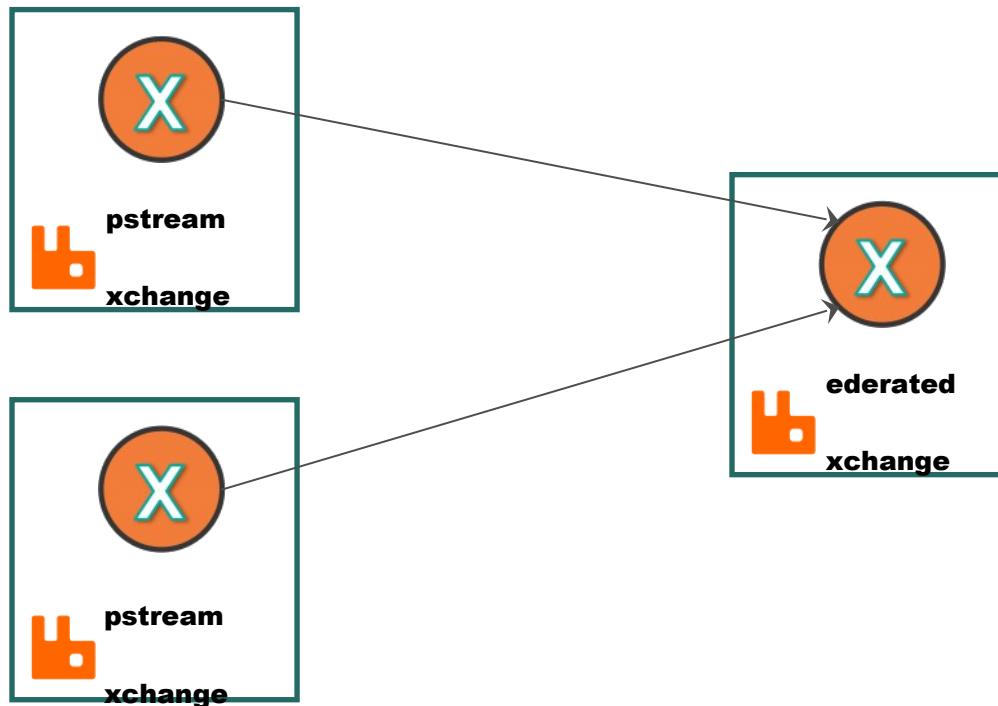
- Goal: forwarding messages between brokers without requiring clustering
- This is useful because
 - Federated brokers may have different users and virtual hosts
 - Federated brokers don't need to run in the same version of RabbitMQ and Erlang (whereas clustering requires it)
- Federation is designed to scale out publish / subscribe messaging across WAN
- Plugins need to be enabled

```
rabbitmq-plugins enable rabbitmq_federation  
rabbitmq-plugins enable rabbitmq_federation_management
```

Federation Plugin

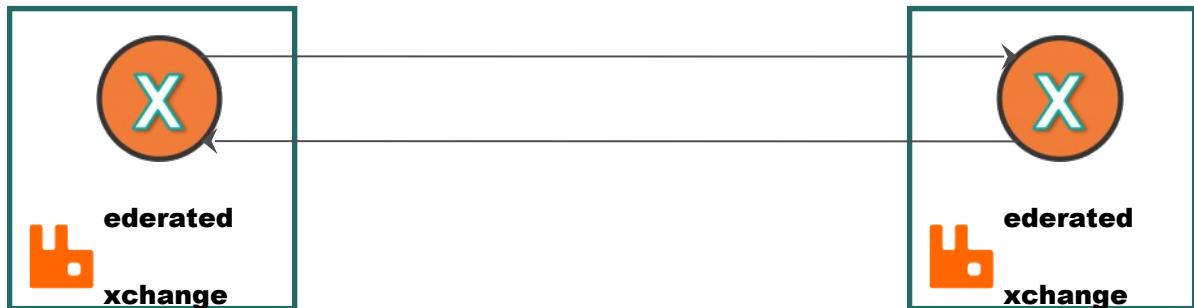
- Federation works at the exchange level or the queue level
 - But it's one or the other
- One broker receives messages published on another broker
- Federation works seamlessly within a cluster
 - Federated exchanges/queues are just plain resources
- Exchange level
 - Messages published to an exchange are also published to another exchange
- Queue level
 - Messages of a queue get forwarded to another queue

A federated exchange



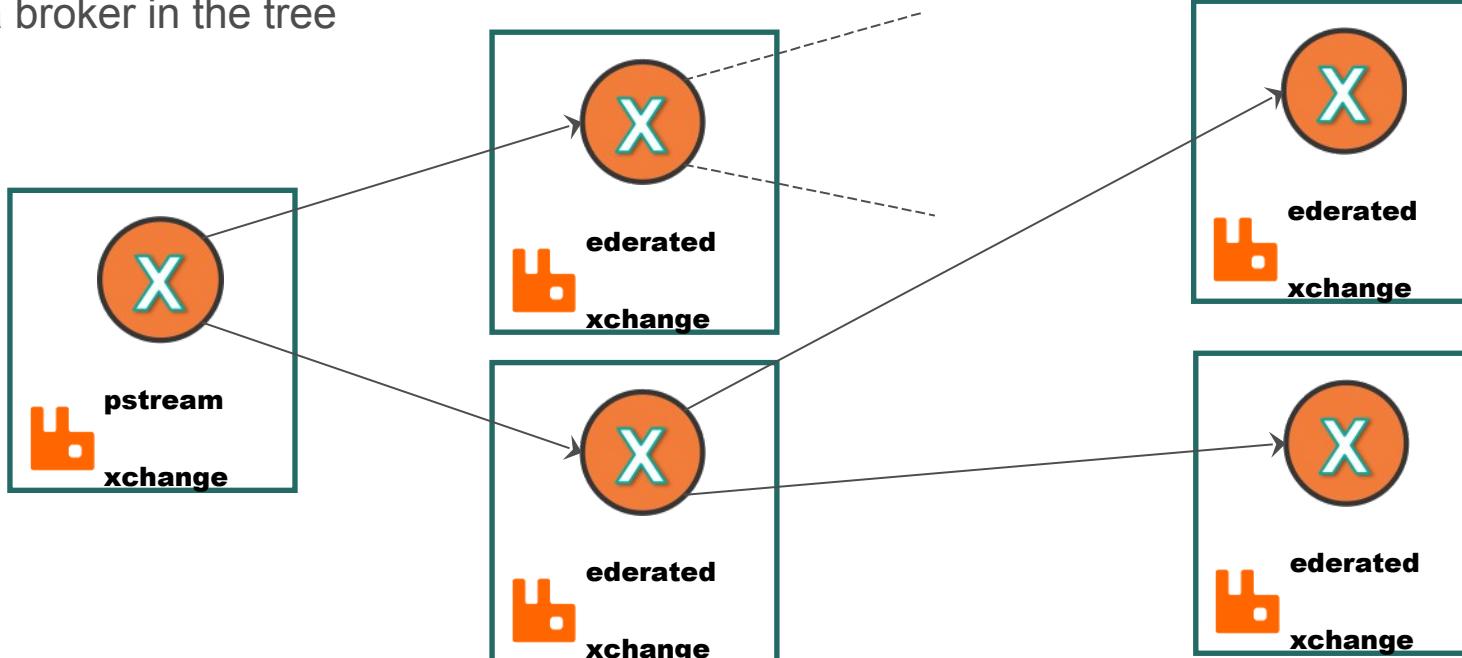
Federated exchange example: pair

- Set max-hops to 1
 - Messages are copied only once
- Consumer on one broker receives messages published on the other one
 - And vice-versa



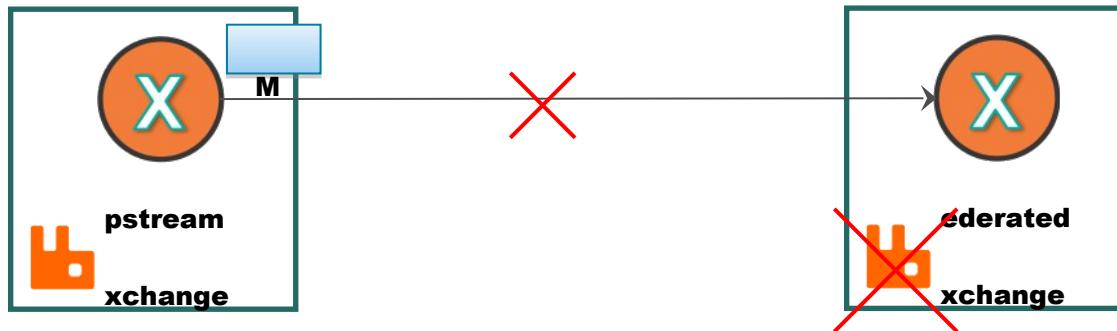
Federated exchange example: fan-out

- Messages published to upstream exchange can be received by any consumer connected to a broker in the tree



Federated exchange configuration

- Configuration takes place on the federated exchange's broker
 - No configuration on upstream exchanges' brokers
- Messages sent to upstream exchange forwarded to federated exchange
- Messages queued if federated broker is down



Federated exchange configuration steps

- Declare the upstream
 - The upstream server to connect to
- Declare the policy to apply the upstream to federated exchanges

NOTE

You can use rabbitmqctl, the HTTP API, or the web UI to configure the federation.

Declaring the upstream

- The upstream broker to connect to

The screenshot shows the RabbitMQ Management Console's Admin interface with the 'Federation Upstreams' tab selected. A callout box highlights the 'Name' field under 'General parameters'.

General parameters

- Name: *
- URI: *
- Prefetch count:
- Reconnect delay: s
- Acknowledgement Mode:
- Trust User-ID:

Federated exchanges parameters

- Exchange:
- Max hops:
- Expires: ms
- Message TTL: ms
- HA Policy:

Federated queues parameter

- Queue:

Add upstream

**Name of the upstream exchange
(default is to use the name of federated exchange)**

The policy

- Targeting exchange(s) of an upstream

Overview Connections Channels Exchanges Queues Admin

Policies

All policies

Filter: Regex (?)(?)

0 items, page size up to 100

... no policies ...

Add / update a policy

Name: federate-quotations *

Pattern: quotations *

Apply to: Exchanges

Priority:

Definition: federation-upstream = server2 String *

= String *

HA mode (?) | HA params (?) | HA sync mode (?)

Federation Federation upstream set (?) | Federation upstream (?)

Queues Message TTL | Auto expire | Max length | Max length bytes

Dead letter Exchange | Dead letter routing key

Exchanges Alternate exchange

Add policy

Select the to-be-federated exchange(s)

Select the upstream

NOTE

The policy can target an *upstream set*. This implies an extra configuration step with the `rabbitmqctl set parameter` command to define the upstream set.

Checking the federation

- The policy must apply to the exchange(s)

Overview Connections Channels **Exchanges** Queues Admin

Exchanges

▼ All exchanges (10)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Displaying 10 items , page size up to:

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D	0.00/s	0.00/s	
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.log	topic	D I			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
from.market	fanout	D	0.00/s	0.00/s	
quotations	fanout	D federate-quotations	0.00/s	0.00/s	

Checking the federation status

Overview Connections Channels Exchanges Queues Admin

Federation Status

Running Links

Upstream	URI	Exchange / Queue	State	Inbound message rate	Last changed
server2	amqp://localhost:5673	quotations exchange	running		2016-03-03 16:57:42

Users

Virtual Hosts

Policies

Shovel Status

Shovel Management

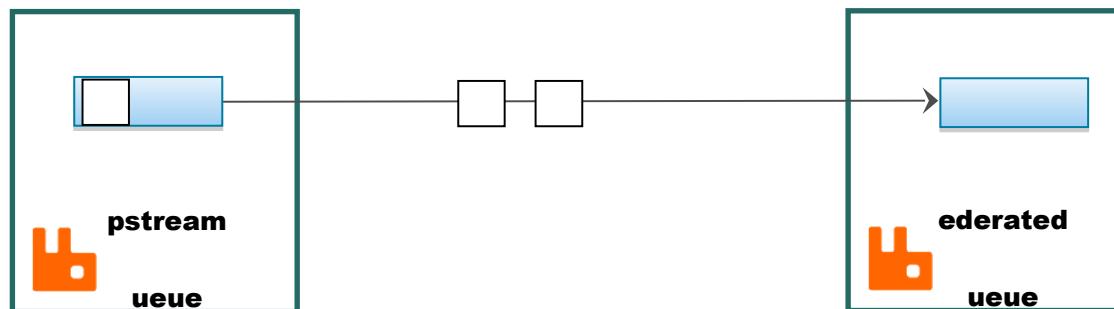
Federation Status

Federation Upstreams

Messages from server2 going through the “quotations” exchange are forwarded to the federated “quotations” exchange

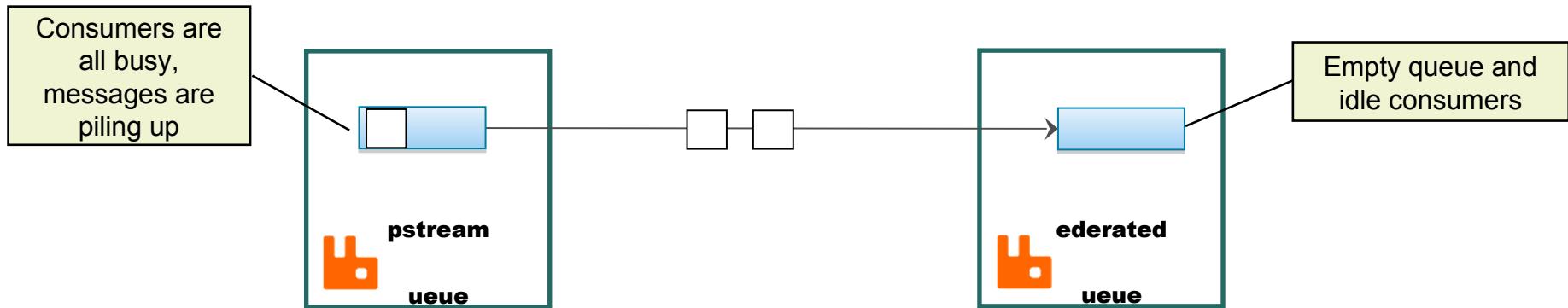
A Federated Queue

- The federated queue is downstream
- The federated queue can get messages from the upstream queue



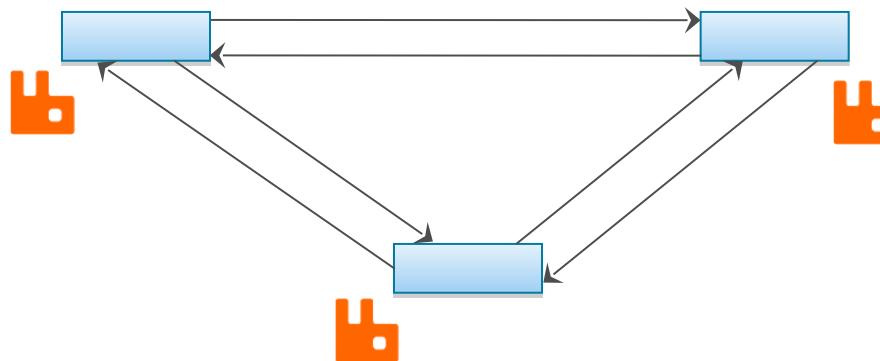
Federated queue, why?

- A way to load balance the load of a queue across nodes/clusters
- The federated queue shares its spare consuming capacity
 - When messages in the upstream queue piles up
 - When the federated queue has no messages and idle consumers



Federated queue example

- Queues are federated between each other
- Each of them can share its spare consuming capacity with the others



Federated queue configuration steps

- Configuration takes place on the federated queue's broker
 - No configuration on upstream queues' brokers
- Declare the upstream
 - The upstream server to connect to
- Declare the policy to apply the upstream to federated queues

NOTE

You can use rabbitmqctl, the HTTP API, or the web UI to configure the federation.

Declaring the upstream

- The upstream broker to connect to

The screenshot shows the RabbitMQ Management Console's Admin interface with the 'Federation Upstreams' tab selected. The 'Upstreams' section is expanded, showing a list of '... no upstreams ...'. Below it, the 'Add a new upstream' form is visible. The 'General parameters' group contains fields for Name (set to 'server2'), URI (set to 'amqp://localhost:5673'), Prefetch count, Reconnect delay, Acknowledgement Mode (set to 'On confirm'), and Trust User-ID (set to 'No'). The 'Federated exchanges parameters' group includes Exchange, Max hops, Expires, Message TTL, and HA Policy. The 'Federated queues parameter' group has a Queue field. At the bottom is an 'Add upstream' button. A callout box with a black border and light green background points to the 'Name' input field, containing the text: 'Name of the upstream queue (default is to use the name of federated queue)'.

Name: server2 *

URI: (?) amqp://localhost:5673 *

Prefetch count:

Reconnect delay: (s)

Acknowledgement Mode: (?) On confirm

Trust User-ID: (?) No

Exchange: (?)

Max hops: (?)

Expires: (ms)

Message TTL: (ms)

HA Policy: (?)

Queue: (?)

Add upstream

Name of the upstream queue
(default is to use the name of
federated queue)

The policy

- Targeting queue(s) of an upstream

Policies

All policies

Filter: Regex (?)(?)

0 items, page size up to

... no policies ...

Add / update a policy

Name: *

Pattern: *

Apply to:

Priority:

Definition: *

HA mode (?) | HA params (?) | HA sync mode (?)

Federation upstream set (?) | Federation upstream (?)

Queues Message TTL | Auto expire | Max length | Max length bytes
Dead letter exchange | Dead letter routing key

Exchanges Alternate exchange

Add policy

NOTE

The policy can target an *upstream set*. This implies an extra configuration step with the `rabbitmqctl set parameter` command to define the upstream set.

Checking the federation

- The policy must apply to the queue(s)

Queues								
Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
market	D federate-market	idle	0	0	0	0.00/s	0.00/s	

Checking the federation status

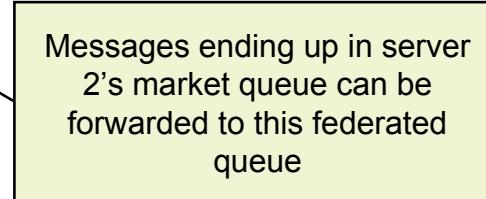
Overview Connections Channels Exchanges Queues Admin

Federation Status

▼ Running Links

Upstream	URI	Exchange / Queue	State	Inbound message rate	Last changed
server2	amqp://localhost:5673	market queue	running	0.00/s	2016-03-14 13:31:54

Users
Virtual Hosts
Policies
Federation Status
Federation Upstreams



Messages ending up in server 2's market queue can be forwarded to this federated queue

Agenda

- RabbitMQ plugins introduction
- LDAP authentication
- Shovel
- Federation
- **Shovel vs Federation**
- STOMP

Shovel or Federation?

- Conceptually, the Shovel and Federation plugin are similar
- They do not work at the same level
 - Shovel works at a lower level than federation
 - It simply consumes messages from a queue
 - And forwards it to an exchange on another broker
 - Federation mirrors messages from an exchange/queue to another
 - This is achieved with some optimizations
 - Main idea is to communicate only when it's required
- Shovel provides more control than Federation
 - Federation is more opinionated, Shovel is a customizable AMQP client

Shovel and Federation vs. Clustering

Shovel / Federation	Clustering
Brokers are logically separate and may have different owners.	A cluster forms a single logical broker.
Brokers can run different versions of RabbitMQ and Erlang.	Nodes must run the same version of RabbitMQ and frequently Erlang.
Brokers can be connected via unreliable WAN links. Communication is via AMQP (optionally secured by SSL), requiring appropriate users and permissions to be set up	Brokers must be connected via reliable LAN links. Communication is via Erlang internode messaging.
Brokers can be connected in whatever topology you arrange. Links can be one- or two-way.	All nodes connect to all other nodes in both directions.

Shovel and Federation vs. Clustering

Shovel / Federation	Clustering
Some exchanges in a broker may be federated while some may be local.	Clustering is all-or-nothing.
A client connecting to any broker can only see queues in that broker.	A client connecting to any node can see queues on all nodes in a virtual host

Agenda

- RabbitMQ plugins introduction
- LDAP authentication
- Shovel
- Federation
- Shovel vs Federation
- **STOMP**

STOMP Plugin

- This plugin provides a STOMP support in RabbitMQ
- It supports both STOMP 1.0, 1.1, and 1.2
- Plugin must be enabled

```
$ rabbitmq-plugins enable rabbitmq_stomp
```

- Adapter waits for connections on 61613

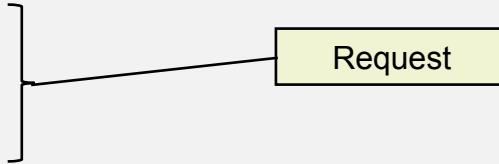
NOTE

STOMP (Streaming Text Oriented Message Protocol) is a text-based messaging protocol.

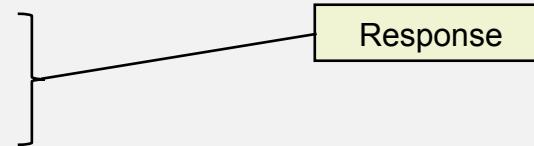
STOMP Connection

- STOMP uses TCP, Unix clients like telnet or nc (netcat) work (or *putty* on Windows)

```
$ nc localhost 61613  
CONNECT  
login:guest  
passcode:guest
```



```
^@  
CONNECTED  
session:session-gsKcEyvX3_9VC8mPu9UoxA  
heart-beat:0,0  
version:1.0
```



- ^@ inserts null byte, to send the request
 - Use Ctrl+@ to insert null byte
- Don't want to authenticate?

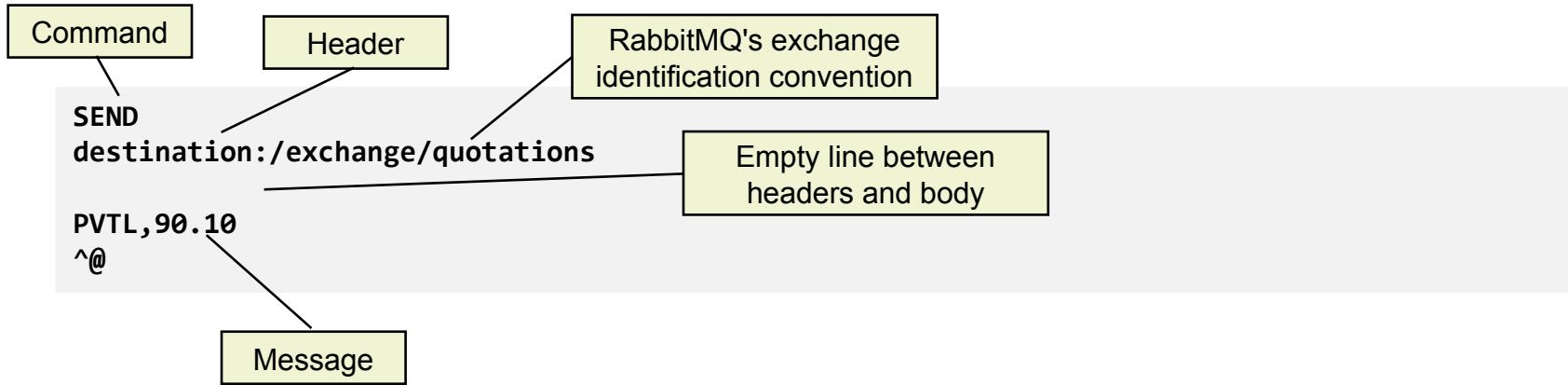
```
[  
    {rabbitmq_stomp, [{default_user,[{login, "guest"},  
                                {passcode, "guest"}]},  
                      {implicit_connect, true}]}]  
].
```

No authentication

STOMP Plugin Semantics

- STOMP specification is quite generic
- Values of destination header in SEND/MESSAGE frames broker-specific
 - MESSAGE symbolizes a message when the client receives it
 - SEND is used to send a message
 - SUBSCRIBE is used to register to a source of messages
- RabbitMQ supports theses destinations (SEND/SUBSCRIBE frames)
 - /exchange
 - /queue
 - /amq/queue (queues outside STOMP gateway, i.e., : existing queues)
 - /topic (in a JMS / STOMP manner)
 - /temp-queue (reply-to template)

Sending a STOMP Message to RabbitMQ



STOMP Plugin – Exchange Examples

- SUBSCRIBE

- /exchange/my-exchange/my-routing-key

Will create an exclusive and auto-delete queue on the direct exchange named "my-exchange", bind it using the routing key, and register a receiver for the current STOMP session

- SEND

- /exchange/my-exchange/my-routing-key

Will send to the exchange named "my-exchange" with the routing key "my-routing-key"

- /exchange destinations not suitable for consuming messages from *existing* queues

- A new queue is created for each subscriber.

STOMP Plugin – Queue Examples

- SEND and SUBSCRIBE

- /queue/my-queue

Will create a shared queue named "my-queue"

- /amq/queue/my-amqp-queue

- Send to the default exchange
 - Subscribe for the current STOMP session

Will use an existing AMQP queue

For the Latest Plugin Information

- For the latest available configuration and features
 - Please consult <http://www.rabbitmq.com/plugins.html>

Summary

- There are numerous plugins available that extend the functionality of RabbitMQ

Performance

Agenda

- Factors that impact performance
- Flow control
- Best practices

What Factors Impact Performance?

- Business factors:
 - Message size
 - Topology
- Client code:
 - Durability / message persistence
 - Acknowledgments
 - Transactions
 - Publisher confirms
 - Channel configuration
- Broker:
 - Queue durability
 - Clustering
 - High Availability (mirrored) queues

Business Factors

- Message size has huge impact on performance
 - Message creation or parsing
 - Network overhead
 - Storing the message on the broker
 - etc...
- Topology of your networks and where brokers are physically installed also impacts performance
 - Mainly because of network latency

Durability / Persistence

- Usage of the client API can have a big impact on performance
 - In good and bad
 - Really important to understand what you are doing
- With durability and persistence, all messages are written to disk
 - If durable flag is set on queue
 - If message is marked as persistent

```
// declare the durable queue
channel.queueDeclare("my-queue", true, false, false, null);

// set the message as persistent (deliveryMode = 2)
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
AMQP.BasicProperties props = builder.deliveryMode(2).build();

String message = "Hello World!";
channel.basicPublish("my-exchange", "key", props, message.getBytes());
```

Persisted to disk

Delivery Mode

- Non-persistent (1)
 - Messages are kept in memory (volatile)
 - *Even if the queue is marked as durable!!*
 - Default setting
- Persistent (2)
 - All messages are written to disk

```
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();

// set the message as non-persistent (default)
AMQP.BasicProperties props = builder.build();

// set the message as persistent (deliveryMode = 2)
AMQP.BasicProperties props = builder.deliveryMode(2).build();
```

Acknowledgements

- Using acknowledgments impact performance
- Client has to send explicit ACK to the broker
- Broker has to manage several cases:
 - Read but unacknowledged messages
 - Acknowledgments coming later
 - Rejections
- Acknowledgment
 - Can be set to automatic mode (auto-ack)
 - Explicitly acknowledged when clients ask for it
 - Behavior specified on
 - Channel.basicGet()
 - Channel.basicConsumer()

Acknowledgements

- Automatic acknowledgement

```
boolean autoAck = true;  
GetResponse response = channel.basicGet("queueName", autoAck);  
  
if (response != null) {  
    // do business logic here  
}
```

Auto-acknowledgement enabled

- Manual (client) acknowledgement

```
GetResponse response = channel.basicGet("queueName", false);  
  
if (response != null) {  
    long deliveryTag = response.getEnvelope().getDeliveryTag();  
    // do business logic here  
    boolean multiple = false;  
    channel.basicAck(deliveryTag, multiple);  
}
```

No Auto-acknowledgement

Manual acknowledgement

Transactions

- Transactions require extra communication to
 - Commit
 - Rollback
- When grouping multiple messages per transaction, the broker has to manage multiple uncommitted messages until the commit comes
- Transaction for consumers
 - Ability to group multiple consumed messages with a single commit
- Transaction for publishers
 - Message is not readable for others until it's committed
 - Add extra-delay for consumers to process the message

Transactions

- Using transactions to send messages atomically

```
channel.txSelect();

byte[] msgNasdaq = "NASDAQ".getBytes();
byte[] msgDJ = "DOWJONES".getBytes();
byte[] msgSP500 = "SP500".getBytes();

channel.basicPublish(stockExchange, stockCategory, null, msgNasdaq);
channel.basicPublish(stockExchange, stockCategory, null, msgDJ);
channel.basicPublish(stockExchange, stockCategory, null, msgSP500);

/*
 * 1) until this next instruction, broker has to manage uncommitted messages
 * in case a rollback is needed. This is costly.
 *
 * 2) until this next instruction, messages are also not available for consumers
 */
channel.txCommit();
```

Will commit the batch of messages

Publisher Confirms

- For the publisher, transactions are often unnecessarily weighted and decrease throughput up to a factor of 250
 - Especially when you commit messages one-by-one
- RabbitMQ has the concept of "publisher confirms"
 - Analogous to consumer acknowledgments, but on the publisher side
 - A confirmation is sent by the broker after it has accepted the message
- For both sending and receiving, use publisher confirms and consumer acknowledgments
 - Transactions are heavier
 - Confirms and acknowledgements are usually sufficient for most use cases

Publisher Confirms

- Client needs to put channel in confirm mode
 - Channel.confirmSelect()

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

channel.confirmSelect();
```

- Client can explicitly wait to have all published messages confirmed by the broker before doing more actions
 - Channel.waitForConfirms()
 - Channel.waitForConfirmsOrDie()

```
channel.basicPublish("exchangeName", "routingKey", null, "message".getBytes());

channel.waitForConfirms();
```

Delivery configuration on the client side

- The broker can limit what it delivers to consumers
- The consumer configures delivery at the channel level
- The broker continues deliveries, depending on consumer's ACK
- Channel.basicQos() to specify the "quality of service"
- Parameters:
 - prefetchCount: the number of unacknowledged messages the server will deliver to a channel (0 means there is no limit) in a single batch

“Quality of service” configuration

- `basicQos()` allows consumer-driven flow control
 - Consumer doesn't get overwhelmed
 - Method call is blocking when value is reached
- Useless if auto-ack is enabled: messages are acknowledged as soon as they are delivered

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

int prefetchCount = 100;
channel.basicQos(prefetchCount);

boolean autoAck = false;

while(true){
   .GetResponse response = channel.basicGet("queueName", autoAck);
}
```

100 messages will be delivered
before they are ack'd

“Quality of service” scope, the global flag

```
channel.basicQos(10);
```

Per consumer limit

“global” flag

```
channel.basicQos(10, false);  
channel.basicQos(15, true);
```

Per consumer limit

Per channel limit

With the settings above, 2 consumers can have 10 un-acked messages each, but the limit is 15 on the whole channel. This has some overhead. Prefer the (simple) per consumer limit setting.

NOTE

The meaning of the global flag in RabbitMQ is different than in the AMQP specification.

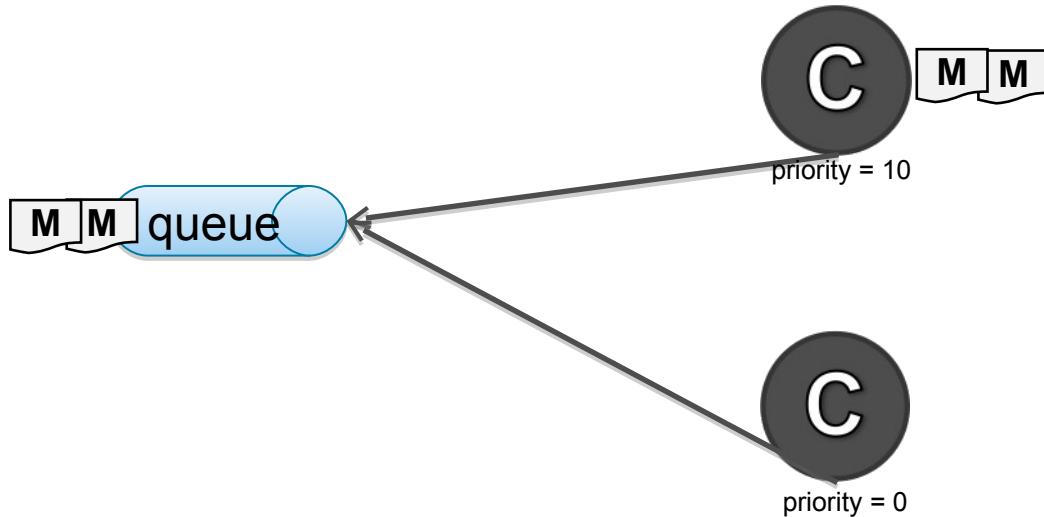
Consumer Priorities

- Consumers can declare a priority level when they start consuming
- RabbitMQ would then
 - first deliver messages to high-priority consumers
 - fall back to low-priority consumers when high-priority consumers are *blocked*
- When to use consumer priorities?
 - To prioritize consumers with some specific hardware, e.g. prefer consumers with SSD drives
 - To prioritize consumers connected to the master of a mirrored queue
- When does a consumer get blocked?
 - When it reaches its « quality of service » limit
 - When it experiments network congestion

NOTE

By default, RabbitMQ distributes messages to consumers in a round-robin fashion.

Consumer Priorities



Consumer cannot hold more than 3 un-acked messages

Broker falls back to low-priority consumer when high-priority consumer is blocked

Consumer priority declaration

```
int prefetchCount = 100;  
channel.basicQos(prefetchCount);
```

Use an appropriate quality of service, to control when the consumer gets blocked

```
Map<String, Object> arguments = new HashMap<>();  
arguments.put("x-priority", 10);  
channel.basicConsume("queue", false, arguments, consumer);
```

Use the x-priority argument when starting consuming

Set auto-ack flag to false, otherwise the consumer will consume all the messages the broker can send

Message Persistence

- Be aware of performance implications when specifying persistent messages
 - Every message will be written to disk
 - If publisher confirms are used and/or consumer acknowledgements, this can have significant impact on performance
 - Performance impact is due to additional disk I/O

Clustering and Mirrored Queues

- Certain types of clusters have mirrored-queues that replicate metadata and data on other queues in the cluster
 - Overhead to replicate all the data across the nodes
 - Commit and acknowledgment are also replicated across the nodes
 - If persistent messages, and/or publisher confirms, and/or consumer acknowledgements are used, performance impact will be higher with mirrored queues, as overhead is also "mirrored"

Agenda

- Factors that impact performance
- **Flow control**
- Best practices

Memory-based Flow Control

- RabbitMQ comes with a nice feature
 - Memory-based flow control
- Used to prevent the broker from accepting too much traffic from producers
 - So that memory is not filled in case of verbose producers
- Why is it needed?
 - By default on RabbitMQ, messages are accepted from clients even if they are not written to disk yet
 - Accepting messages is much faster than writing messages to disk
 - In a heavy load scenario, without any flow control available memory could be completely consumed

Memory-based Flow Control

- How does it work?
 - Producers are throttled to reduce throughput
 - Consumers aren't affected
 - Alarm is raised in the log
- The default memory threshold is 40% of the server RAM
 - This does not prevent RabbitMQ from using more than 40% RAM
 - It is just the point where flow control is turned on
- Usage
 - Memory threshold value can be modified in rabbitmq.config file
 - Value of 0 disables the flow control

```
[{rabbit, [{vm_memory_high_watermark, 0.4}]}]
```

Per-connection Flow Control

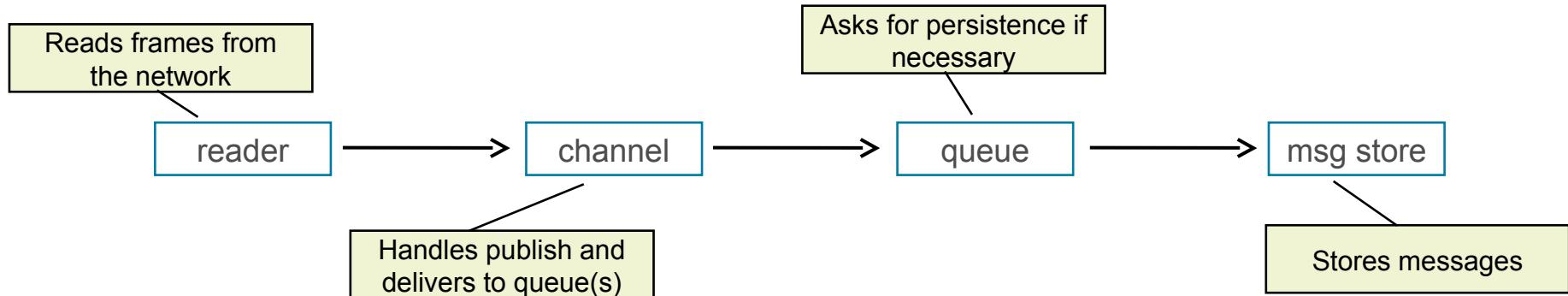
- What if a connection publishes too quickly?
- RabbitMQ blocks the connection automatically
- Happens when publishing is faster than routing
- Connection appears then as “blocked”
- Hard to monitor, as blocking can happen several times per second
 - Check `last_blocked_by` and `last_blocked_age` fields with `rabbitmqctl/web ui`

NOTE

RabbitMQ blocks only connections that publish messages.

Flow control internals

- The flow control is credit-based
- To understand it, let's look inside the rabbit
- Messages go through these RabbitMQ's internal components:

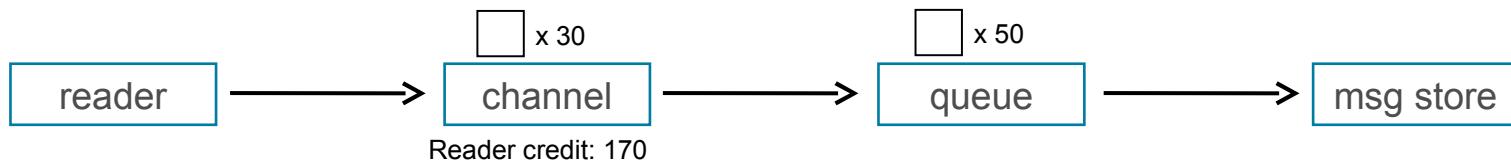
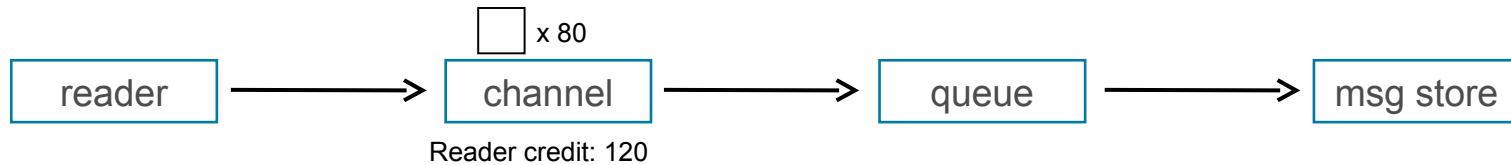
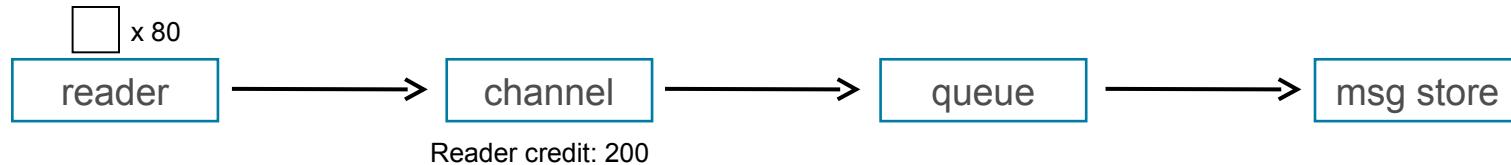


NOTE

At runtime, each component is an Erlang process.

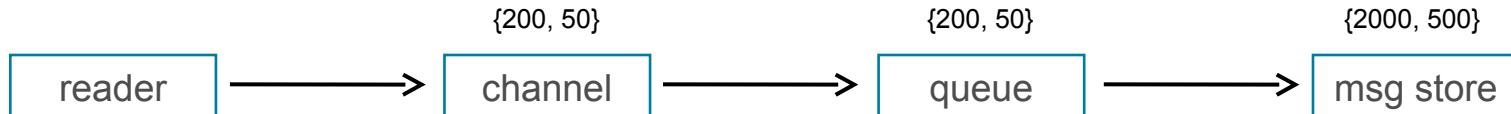
Credit flow

- Each process grants credits to upstream processes



Credit flow

- Process grants credits back when it has managed to handle messages
- If a process has no credit left, it's blocked by the downstream process
 - This is how the per-connection flow control is implemented, by blocking the reader
- The credit flow can be configured with 2 values
 - InitialCredit: the initial credit the process grants to upstream processes
 - MoreCreditAfter: the number of messages the process must handle before granting credit back
- Defaults:



Credit flow considerations

- The credit is *per process*
 - Many channels can send to the same queue
 - That's why a queue has an initial credit of 2000 from the message store
- Raising up the values
 - More messages going through the broker
 - More RAM usage
- Be careful with these settings, experiment with your workload



Credit flow configuration

- In RabbitMQ configuration file:

```
[  
{rabbit, [  
    {credit_flow_default_credit, {200, 50}},  
    {msg_store_credit_disc_bound, {2000, 500}}  
]}  
.]
```



NOTE

These settings have been introduced in RabbitMQ 3.5.5.

Blocked connection notifications

- RabbitMQ can notify clients when it blocks their connections
- Clients can then take some actions:
 - Throttle or stop their publishing
 - Add more consumers
- The point is to give some air to drain the queues

NOTE

Blocked connection notifications is an extension introduced in RabbitMQ 3.2. The official RabbitMQ clients support this feature.

Add a blocked connection listener

- Add the listener at the connection level:

```
ConnectionFactory factory = new ConnectionFactory();
Connection connection = factory.newConnection();
connection.addBlockedListener(new BlockedListener() {

    public void handleBlocked(String reason) { }

    public void handleUnblocked() { }

});
```

Blocked connection listener semantics

- Blocked event:
 - Called for RAM and disk flow control
 - Can be called several times before unblocked event!
- Unblocked event:
 - Called when all resource alarms have cleared and the connection is fully unblocked

Message paging

- Queues keep an in-memory cache of messages
 - To deliver messages to consumers as fast as possible
- If RabbitMQ needs memory, it pages out the messages to disk
- Paging « blocks » the queue
 - It can't receive new messages
- Strive to consume faster than you publish!

Message paging internals

- RabbitMQ's backing store is made of 2 parts
 - message store: stores content of messages
 - queue index: store per message per queue data
- Small messages are paged out in the queue index
 - This is an optimization
 - Default limit size is 4096
 - Set with `queue_index_embed_msgs_below` in config file

NOTE

A node has 2 backing stores, one for persistent messages and one for paging. They work the same way.

Message paging internals

- Broker pages first in the message store, then in the queue index
- Queue index paging doesn't always happen
 - Broker calculates how much memory to free
 - There a threshold to trigger the paging
 - Default is more than 2048 messages to page to trigger paging
- This is configurable, test with your own workload

NOTE

Queue index paging performance has been greatly improved in RabbitMQ 3.5.5.

Queue index paging configuration

- In RabbitMQ configuration file:

```
[  
{rabbit, [  
    {msg_store_io_batch_size, 2048}  
]}  
].
```

- Raising the value
 - Less paging (paging to disk is costly)
 - More RAM usage

Lazy queues

- Sometimes, queues fill up
 - Consumers can't keep up, or are shut down for maintenance
 - Consumers can be unstable
- If your queues fill up regularly, use *lazy queues*
- Lazy queues are good when you need very long queues
 - Millions of messages

Lazy queues semantics

- Lazy queues
 - Send messages to the disk if there's no consumer
 - Load messages in memory when requested by consumers
- Consequences
 - Lazy queues consume much less RAM than default queues
 - Lazy queues increase I/O, but not more than when using persistent messages

Lazy queues declaration

- 2 ways to declare a lazy queue
- Use an argument when declaring the queue
 - `x-queue-mode = lazy`
- Use a policy
 - `queue-mode = lazy`

NOTE

By using a policy, you can change the mode at runtime. Switching from default to lazy will page out the messages to disk, and thus block the queue.

Agenda

- Factors that impact performance
- Flow control
- **Best practices**

Best Practices on Client Side

- Prefer confirms and acknowledgments over transactions
 - These are lighter and generally satisfy most requirements
- Use durability and persistence only when required
 - This also has a big impact on performance
 - Efficient design avoids having to use persistence
 - Idempotency
 - Retry logic
- When dealing with multiple resources (e.g. queue + database)
 - Use best effort pattern

Best Practices on Server Side

- Do not deactivate the flow control mechanism
 - Server won't crash because it gets overwhelmed by publishers
- Monitor the health of your broker
 - By checking the log
 - By having an external health-check mechanism

Summary

- Numerous factors can impact performance
- RabbitMQ has built-in flow control to protect the server
- Use client and server best practices to maximize performance

Lab

Comparing and Benchmarking Classic Configurations

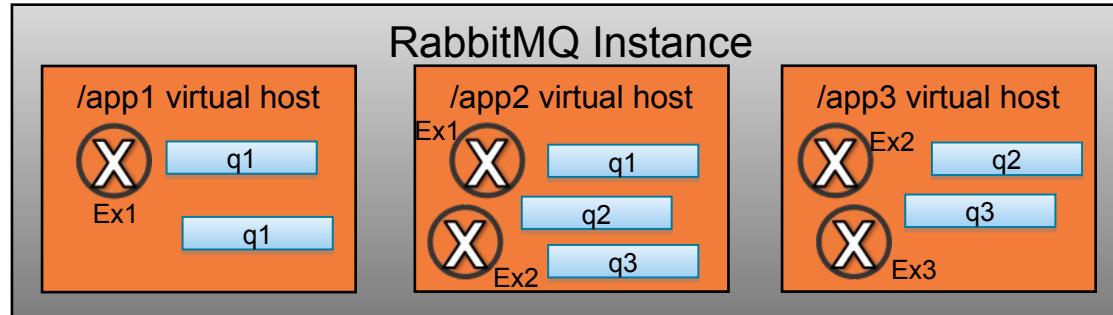
Security

Agenda

- **Virtual hosts, users, and access control**
- Authentication
- Secured communication

Virtual Hosts

- Virtual hosts are a way to logically partition a broker
 - Inside a virtual host, exchanges and queues are commonly referred to as “resources”
 - Resources are named entities in a given virtual host
- Two resources with the same name but in different virtual hosts are different
 - Analogous to different namespaces
- Cannot bind resources between different virtual hosts (/app1 Ex1 cannot bind to /app2 q3)



What are Virtual Hosts Good For?

- Logical organization
 - one virtual host for each application/system, to avoid name collision
- Security
 - a first level of access control is enforced when a client connects to the broker
- Access control
 - a user has permissions on resources inside a given virtual host

Virtual Host Management from the UI

Overview Connections Channels Exchanges Queues Admin

Virtual Hosts

All virtual hosts

Filter: 1 item (show at most

Overview		Messages			Data rates		Message rates	
Name	Users (?)	Ready	Unacked	Total	From clients	To clients	publish	deliver / get
/	guest							

Add a new virtual host

Name: *

Add virtual host

Command Line Virtual Host Management

- rabbitmqctl has commands to manage virtual hosts
 - add_vhost, delete_vhost, list_vhosts

```
$ rabbitmqctl add_vhost quotations
Creating vhost "quotations" ...
...done.
$ rabbitmqctl list_vhosts
Listing vhosts ...
/
quotations
...done.
```

Virtual Hosts, Users, and Access Control

- RabbitMQ allows for fine-grained access control
 - E.g., what a client can do with exchanges and queues in a virtual host
- Access control implies several notions:
 - Virtual hosts
 - Users
 - Permissions
- Let's cover users now before moving on to permissions!

User Management

- By default, RabbitMQ stores users in its internal database
- Users are managed through the management plugin and rabbitmqctl
- With plugins, users can be stored elsewhere
 - E.g., in an LDAP directory with the LDAP plugins
- Management plugin and rabbitmqctl can only see users in the internal database
- We're going to focus on internal database users

NOTE

Remember that users are also used for authentication!

User Management from the UI

Overview Connections Channels Exchanges Queues Admin Virtual host: /

Users

▼ All users

Filter: 1 item (show at most)

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/	•

(?)

▼ Add a user

Username: *

Password: *
 * (confirm)

Tags: (?)
[Admin] [Monitoring] [Policymaker] [Management] [None]

Add user

Command Line User Management

- rabbitmqctl has commands to manage users
 - add_user, delete_user, change_password, list_users

```
$ rabbitmqctl add_user quotation_app secret
Creating user "quotation_app" ...
...done.
$ rabbitmqctl change_password quotation_app newpassword
Changing password for user "quotation_app" ...
...done.
$ rabbitmqctl list_users
Listing users ...
guest      [administrator]
quotation_app      []
...done.
```

User Tags

- Users can have tags

```
$ rabbitmqctl list_users
Listing users ...
guest    [administrator]
quotation_app []
...done.
```

▼ All users				
Name	Tags	Can access virtual hosts	Has password	
guest	administrator	/	.	

- Extend existing permissions model
- Don't affect authentication

User Tags

- "management": Access restricted to user's virtual host(s)
- "policymaker": "management" *plus* ability to manage policies in user's virtual host(s)
- "monitoring": "policymaker", can also view other users and resources on all virtual hosts
- "administrator": "monitoring" plus ability to manage virtual hosts, users, and permissions
- See <https://www.rabbitmq.com/management.html#permissions>
- Use management console or *rabbitmqctl set_user_tags* to manage tags

```
$ rabbitmqctl set_user_tags quotation_app administrator
Setting tags for user "quotation_app" to [administrator] ...
...done.
```

Default Access

- On first start or after database deletion, broker creates a new database
- The new database has the following resources:
 - Virtual host named /
 - guest user with guest password
- The default user has full access to / virtual host
- Best practice: delete default user or change password
 - Especially on production or publicly-accessible broker
- To change default user credentials:

```
[  
  {rabbit, [  
    {default_user,<<"admin">>},  
    {default_pass,<<"changeit">>}  
  ]}  
].
```

Access Control

- RabbitMQ can enforce access rules
- Access rule = an operation on a resource
- Resource = exchange or queue in a virtual host
- Operation = configure, write, or read

The diagram illustrates the breakdown of a RabbitMQ access control command. A grey box contains the command: `rabbitmqctl set_permissions -p quotations quotation_app "^$" ".*" ".*"`. Above the command, two brackets indicate its components: one bracket labeled "Virtual Host" covers the prefix `rabbitmqctl set_permissions -p quotations`, and another bracket labeled "Regex patterns for authorized operations" covers the suffix `"^$" ".*" ".*"`. Below the command, four arrows point from the tokens to their corresponding meanings: `User` points to `quotations`, `Configure` points to `^$`, `Write` points to `.*`, and `Read` points to `.*`.

```
rabbitmqctl set_permissions -p quotations quotation_app "^$" ".*" ".*"
```

Virtual Host Regex patterns for authorized operations

User Configure Write Read

Operations

- Configure
 - Create or destroy resources, or alter their behavior
 - *Example:* declaring an exchange
 - *Example:* deleting a queue
- Write
 - Inject messages into a resource
 - *Example:* publishing to an exchange
 - *Example:* binding a queue to an exchange
- Read
 - Retrieve messages from a resource
 - *Example:* consuming messages from a queue
 - *Example:* purging a queue
- For a comprehensive reference:
<http://www.rabbitmq.com/access-control.html>

Permissions

- Permissions are expressed with regular expressions
- The regular expression is checked against a resource name
- One regular expression for each type of operation
- Common regular expressions
 - `^$` : matches nothing, prevents the operation on all resources
 - `.*` : matches everything, allows the operation on all resources
 - `^market\..*\|eu\..*$` : matches "market.us" and "eu.paris"

Access Control from the UI

Overview Connections Channels Exchanges Queues Admin Virtual host: All ▾

Virtual Host: quotation_app

▶ Overview **Virtual Hosts**

▼ Permissions

Current permissions

User	Configure regexp	Write regexp	Read regexp	
quotations	^\$.*	.*	Clear

Set permission

User: guest

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

Users **Virtual Hosts** Policies

Access Control from the Command Line

- rabbitmqctl has commands to manage access control
 - set_permissions, clear_permissions, list_permissions

```
$ rabbitmqctl set_permissions -p quotations quotation_app "[$]" ".*" ".*"  
Setting permissions for user "quotation_app" in vhost "quotations" ...  
...done.  
$ rabbitmqctl list_permissions -p quotations  
Listing permissions in vhost "quotations" ...  
quotation_app      [$]          .*          .*  
...done.
```

- Remember:
 - Permissions aren't global
 - Permissions are specific to a virtual host

Agenda

- Virtual hosts, users, and access control
- **Authentication**
- Secured communication

Authentication

- RabbitMQ uses a pluggable authentication mechanism
- Server and client negotiate how they exchange credentials
- RabbitMQ supports the most common mechanisms
 - Username / password (plain text or over SSL)
 - Client certificate
- Custom plugins can extend authentication mechanisms
 - *Example:* challenge / response

SASL

- The pluggable authentication mechanism is SASL
 - Simple Authentication and Security Layer
- SASL is embedded in AMQP and various protocols
- Thanks to SASL, AMQP doesn't need to worry about authentication
 - The onus is on the client and the broker

Supported Authentication Mechanisms

- PLAIN
 - Simple username / password
 - Password sent in plain text (can be protected with SSL)
- EXTERNAL
 - User's identity taken from outside the protocol
 - A plugin is usually in charge of determining the user's identity
 - *Example: rabbitmq-auth-mechanism-ssl* can take the client's certificate and check if the server trusts it

Where to Configure Authentication?

- Server side: rabbitmq.config

- PLAIN is the default

```
[  
  {rabbit, [  
    {auth_mechanisms, ['PLAIN']},  
    {auth_backends,[rabbit_auth_backend_internal]}  
  ]}  
].
```

- Client (Java binding)

- Again default is PLAIN

```
connectionFactory.setSaslConfig(DefaultSaslConfig.PLAIN);
```

Agenda

- Virtual hosts, users, and access control
- Authentication
- **Secured communication**

Secured Communication With SSL/TLS

- RabbitMQ has built-in support for SSL/TLS
- TLS can be used to:
 - Encrypt communication between the broker and its clients (when they exchange sensitive data on public networks)
 - Authenticate the clients AND the broker
- Don't use SSL any more
 - SSL was officially deprecated in 2015
- Try to use the latest versions of TLS
 - At least TLS 1.1, or even TLS 1.2
 - Be careful not to break clients that don't support those versions though

NOTE

SSL stands for Secure Socket Layer. TLS stands for Transport Layer Security. TLS is the new name/version of SSL.

RabbitMQ TLS Support and Erlang

- Prefer TLS over SSL
- RabbitMQ TLS builds on top of Erlang applications
 - new_ssl and crypto
- crypto needs OpenSSL
- There can be issues depending on the Erlang version and platform used
 - So: use Erlang 17.5 or later
- Check <http://www.rabbitmq.com/ssl.html>

TLS Configuration

- In rabbitmq.config:

```
[  
  {rabbit, [  
    {ssl_listeners, [5671]},  
    {ssl_options, [  
      {new_ssl_options,  
       {cacertfile,"/path/to/testca/cacert.pem"},  
       {certfile,"/path/to/server/cert.pem"},  
       {keyfile,"/path/to/server/key.pem"},  
       {verify,verify_peer},  
       {fail_if_no_peer_cert,true}  
     ]}  
   ]}.  
]  
  
Port for TLS connections  
Trusted certificate(s)  
Server's certificate  
Server's private key  
Client certificate needed  
Fails if client certificate isn't valid
```

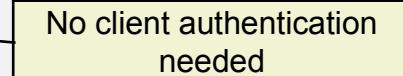
NOTE

Don't try to configure SSL/TLS without a bit of background on private/public keys, certificates, or without knowledge of OpenSSL!

TLS Configuration, no client authentication

- In rabbitmq.config:

```
[  
  {rabbit, [  
    {ssl_listeners, [5671]},  
    {ssl_options, [  
      {cacertfile, "/path/to/testca/cacert.pem"},  
      {certfile, "/path/to/server/cert.pem"},  
      {keyfile, "/path/to/server/key.pem"},  
      {verify, verify_none} ]}  
    ]}  
].
```



TLS Options

- new_ssl Erlang application provides options
 - Through the ssl_options key
- Example:
 - Don't ask for a client certificate (when you need only encryption). User can authenticate with username / password.
 - Ask client certificate and validate it against internal DB or LDAP.
- Check new_ssl configuration

TLS on the Client Side

- Depends on the client platform
- *Example:* Java
 - Certificates are usually manipulated through *keytool*
 - One can either:
 - Add server's certificate in JVM global keystore
 - add server's certificate on the connection factory – see
`ConnectionFactory.useSslProtocol(SSLContext)`

TLS with a Java client

- No server authentication

```
ConnectionFactory factory = new ConnectionFactory();
factory.setUsername("guest");
factory.setPassword("guest");
factory.setVirtualHost("/");
factory.setHost("localhost");
factory.setPort(5671);  
  
factory.setUseSslProtocol();  
  
Connection connection = factory.newConnection();
```

The diagram illustrates the configuration of a ConnectionFactory for a Java client. It shows three annotations pointing to specific lines of code:

- A callout box labeled "Configure as usual" points to the first five lines of code: `factory.setUsername("guest");`, `factory.setPassword("guest");`, `factory.setVirtualHost("/");`, `factory.setHost("localhost");`, and `factory.setPort(5671);`.
- A callout box labeled "Use appropriate port" points to the line `factory.setPort(5671);`.
- A callout box labeled "Trust any server" points to the line `factory.setUseSslProtocol();`.

TLS with a Java client

- With server authentication

```
char [] trustPassphrase = "rabbitstore".toCharArray();
KeyStore tks = KeyStore.getInstance("JKS");
tks.load(new FileInputStream("/path/to/truststore"), trustPassphrase);
TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(tks);

SSLContext c = SSLContext.getInstance("TLSv1.1");
c.init(null, tmf.getTrustManagers(), null);

ConnectionFactory factory = new ConnectionFactory();
(...)
factory.setUseSslProtocol(c);

Connection connection = factory.newConnection();
```

Creates trust store
(contains server certificate)

Adds trust store to SSL context

Uses SSL context in connection factory

Summary

- Virtual hosts allow for multi-tenancy and fine-grained access control
- Authentication mechanism is pluggable
- TLS is recommended for secured communication between clients and the broker

Lab

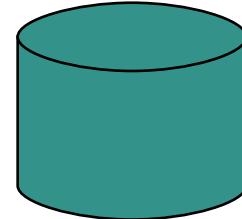
Securing RabbitMQ

Operations and Monitoring

Agenda

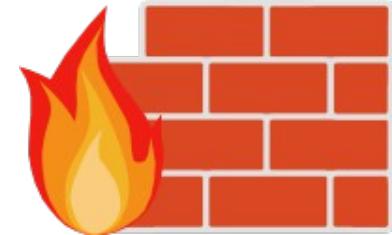
- Operations
 - Disk, firewall, heartbeat, tuning
- Monitoring
- How to monitor with the HTTP API
- Support for third-party monitoring tools

Disk



- RabbitMQ files
 - Mnesia database, message store
 - *Unix*: in /var/lib
 - *Windows*: in C:/Users/XXX/AppData/Roaming/RabbitMQ
 - Where XXX is the RabbitMQ user
- Transient messages can be paged to disk!
 - If memory alert kicks in
- Pay attention to this location!
 - Need at least 2 Gb free space

Firewall configuration

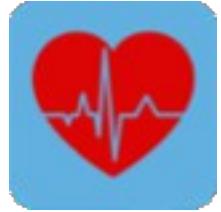


- The most common ports in RabbitMQ:
 - 4369, 25672 (Erlang)
 - 5672, 5671 (AMQP)
 - 15672 (management plugin)
 - 61613, 61614 (STOMP)
 - 1883, 8883 (MQTT)

NOTE

These ports can be changed by using RabbitMQ configuration.

Heartbeats



- Heartbeat timeout negotiated between server & client
 - To figure out when a connection is considered dead
 - Default is 60 seconds
- Heartbeat frames sent every (timeout / 2) seconds
 - After 2 missed heartbeats, the peer is considered unreachable
- Good value: between 6 and 12 seconds
 - Configurable at the client level

NOTE

Heartbeat mechanisms can be different between protocols (AMQP, STOMP.) These settings are for the AMQP protocol.

Recommendations

- Change default user credentials
- Don't use default vhost, use specific vhosts
 - Unless you're really in a single-tenant environment
- Don't use 32-bit Erlang
- Use odd number of nodes in your cluster
 - 3, 5, 7, ...
 - Ensures half a cluster stays up if network gets partitioned

Tuning

- Usually, 2 types of workloads
 - Throughput
 - Common goal, you want a *fast* rabbit!
 - “Internet of Things” (IoT)
 - Many connections (100s K or 1s M)
 - Deals with sensors or very large web applications
- Impossible to tune for both at the same time!

NOTE

Don't apply the following settings blindly, benchmark and test in your own environment, with your target workload.

Throughput: TCP buffers size

- Increase TCP connection buffers
- In RabbitMQ configuration:

```
[  
  {rabbit, [  
    {tcp_listen_options, [  
      {backlog, 128},  
      {nodelay, true}  
      {sndbuf, 196608},  
      {recbuf, 196608}  
    ]}  
  ]}  
].
```

Set buffers size to 192 KB
(default is usually 80-120 KB)
Always use the same value for
both buffers

NOTE

Increasing TCP buffer size also increases the amount of RAM each connection uses.

Throughput: Erlang VM I/O thread pool

- Pool of threads used for I/O operations
- Use environment variable for Erlang arguments:

```
RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS="+A 128"
```

Set pool size to 128

- Details:
 - Default size is 30
 - With 8+ cores, use pool size of 96+ (12 threads / core)
 - Don't increase too much, threads end up waiting on I/O anyway

Throughput: disable Nagle's algorithm

- Nagle's algorithm deals with the “small packet problem”
- Undesirable in highly interactive environments

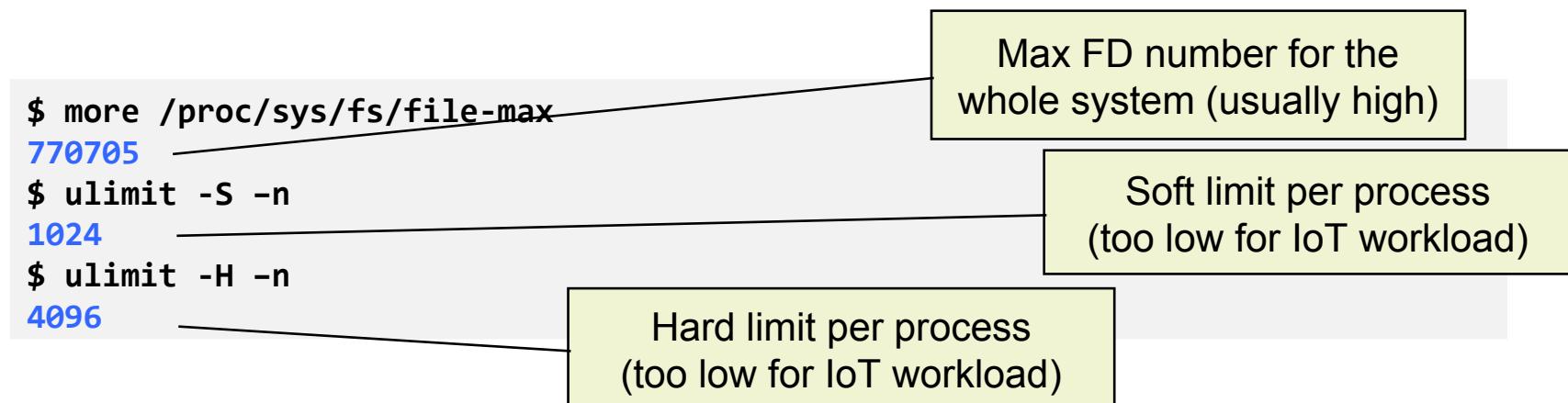
```
[  
  {kernel, [  
    {inet_default_connect_options, [{nodelay, true}]},  
    {inet_default_listen_options, [{nodelay, true}]}  
  ]},  
  {rabbit, [  
    {tcp_listen_options, [  
      {backlog, 4096},  
      {nodelay, true}  
    ]}  
  ]}  
].
```

Disable Nagle for inter-node connections

Disable Nagle to serve client connections

IoT: (Unix) Open File Descriptors (FD) Limit

- OS uses file descriptors to handle files and connections
- Defaults are usually way too low for IoT workloads:



IoT: (Unix) Open FD limit settings

- Per user and per process
- In /etc/security/limits.conf

rabbitmq	soft	nofile	4096
rabbitmq	hard	nofile	8192

NOTE

Configuration can change between distributions, check your own distribution documentation.

IoT: (Unix) Open FD limit recommendation

- Take both connections and files number into account
- Multiply the number of connections per node by 1.5
 - E.g: expected connections = 100 K => limit = 150 K

IoT: TCP buffers size

- Decrease TCP connection buffers to limit RAM usage
- Favor number of connections over throughput

```
[  
  {rabbit, [  
    {tcp_listen_options, [  
      {backlog, 128},  
      {nodelay, true},  
      {sndbuf, 32768},  
      {recbuf, 32768}  
    ]}  
  ]}  
].
```

Set buffers size to 32 KB
(default is usually 80-120 KB)
Always use the same value for
both buffers

NOTE

You'll need to find a trade-off between throughput and RAM usage, depending on your workload.

IoT: Connection backlog

- Increase the length of the queue of unaccepted connections
- Helps to handle a massive reconnection scenario

```
[  
  {rabbit, [  
    {tcp_listen_options, [  
      {backlog, 4096},  
      {nodelay, true}  
    ]}  
  ]}  
].
```

Increase the length of queue of
unaccepted connections
(default is 128)

NOTE

The length must also be changed at the OS level, with the `net.core.somaxconn` option (default is 128.)

IoT: Miscellaneous Recommendations

- Same recommendations as for Throughput
 - Tune the Erlang VM I/O thread pool
 - Disable Nagle's algorithm

Some additional resources...

- Very good resources in the official documentation
- Production checklist
 - <https://www.rabbitmq.com/production-checklist.html>
- Networking
 - <https://www.rabbitmq.com/networking.html>

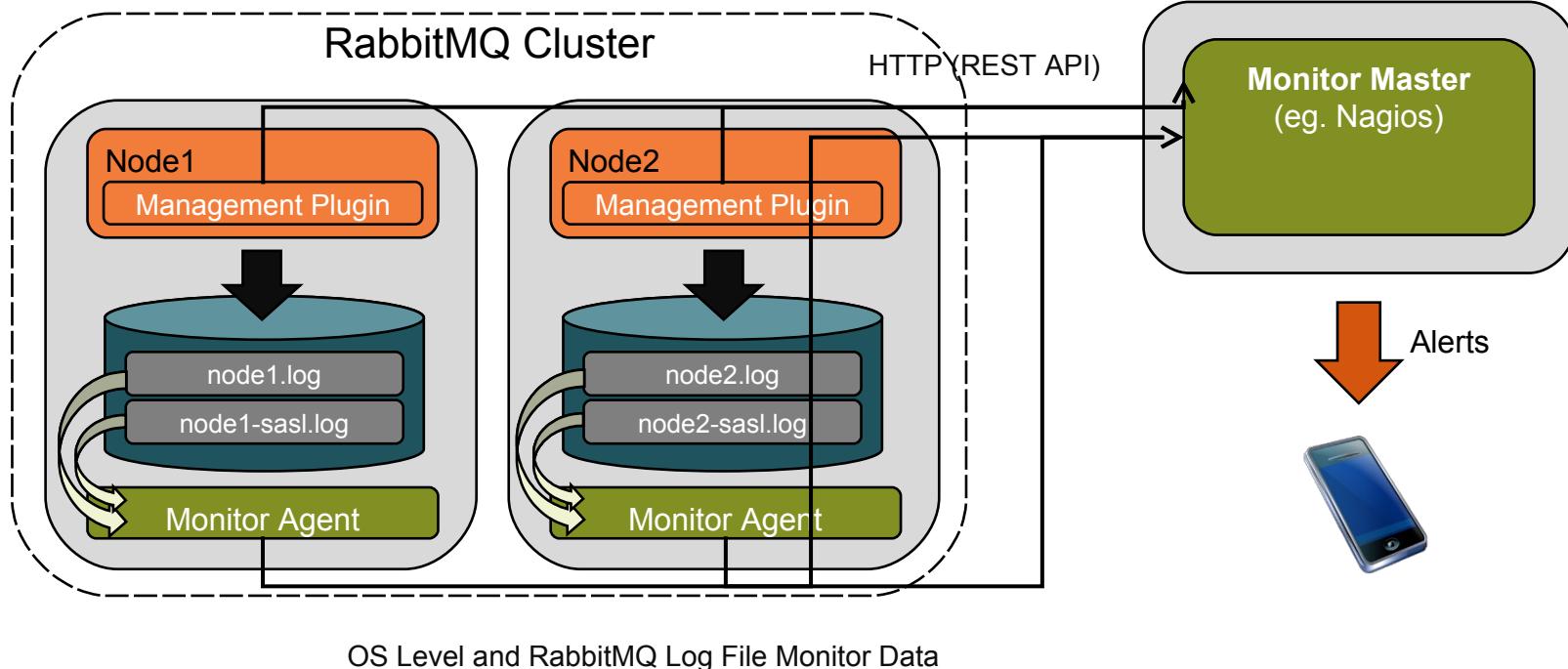
Agenda

- Operations
- **Monitoring**
- How to monitor with the HTTP API
- Support for third-party monitoring tools

What to Monitor?

- System metrics (memory, file system, CPU usage)
- Global state of the cluster
 - Each node is responding
 - Network partitions
 - Message rates in the cluster
 - Message rates in exchanges and queues
 - Size of queues
 - Clients connections
- RabbitMQ log files
 - Erlang "System Application Support Libraries" (SASL) log file: "`=CRASH REPORT=`"
 - Node log file: "`=ERROR REPORT=`"

Typical Monitoring Configuration



Queue size monitoring

- Queues are meant to be always empty
- Strive to keep them empty!
- Nevertheless, messages will be queued
 - Peak of traffic, consumers failure
- Set up an alert when queue size exceeds a limit
 - Limit depends on the application
- Ensure you can catch up with the backlog
 - It shouldn't take hours or days to unqueue the messages
- Ensure you can absorb messages at much higher rate than usual, nominal rate
 - E.g. 10 times as much

Agenda

- Operations
- Monitoring
- **How to monitor with the HTTP API**
- Support for third-party monitoring tools

RabbitMQ HTTP REST API

- The best way to monitor a cluster
 - Is cluster aware
 - Provides messaging metrics, as well as some system metrics
 - Allows managing resources
- Plug your favorite third-party monitoring tool onto it
- Access to help page from all web console pages

The screenshot shows the RabbitMQ Management UI's "Import / export definitions" page. At the top, there is a table with two rows:

Context	Bound to	Port	SSL	Path
RabbitMQ Management	0.0.0.0	15672	<input type="radio"/>	/
Redirect to port 15672	0.0.0.0	55672	<input type="radio"/>	/

Below the table is a section titled "Import / export definitions". It contains fields for "Export" and "Import".

Export:
Filename for download:
 [\(?\)](#)

Import:
Definitions file:
 No file chosen [\(?\)](#)

[Download broker definitions](#) [\(?\)](#) [Upload broker definitions](#) [\(?\)](#)

A red circle highlights the "HTTP API | Command Line" link at the bottom of the page.

HTTP API help page

RabbitMQ Management HTTP API

Introduction

Apart from this help page, all URIs will serve only resources of type `application/json`, and will require HTTP basic authentication (using the standard RabbitMQ user database). The default user is guest/guest.

Many URIs require the name of a virtual host as part of the path, since names only uniquely identify objects within a virtual host. As the default virtual host is called "`/`", this will need to be encoded as "`%2f`".

PUTing a resource creates it. The JSON object you upload must have certain mandatory keys (documented below) and may have optional keys. Other keys are ignored. Missing mandatory keys constitute an error.

Since bindings do not have names or IDs in AMQP we synthesise one based on all its properties. Since predicting this name is hard in the general case, you can also create bindings by POSTing to a factory URI. See the example below.

Many URIs return lists. Such URIs can have the query string parameters `sort` and `sort_reverse` added. `sort` allows you to select a primary field to sort by, and `sort_reverse` will reverse the sort order if set to `true`. The `sort` parameter can contain subfields separated by dots. This allows you to sort by a nested component of the listed items; it does not allow you to sort by more than one field. See the example below.

You can also restrict what information is returned per item with the `columns` parameter. This is a comma-separated list of subfields separated by dots. See the example below.

URIs which return some forms of numerical data (such as message rates and queue lengths) can return historical samples. To return samples you need to set an age and an increment for the samples you want. The end of the range returned will always correspond to the present. Use `msg_rates_age` and `msg_rates_incr` to return samples for messages sent and received, `data_rates_age` and `data_rates_incr` to return samples for bytes sent and received, and `lengths_age` and `lengths_incr` to return samples for queue lengths. For example, appending `?lengths_age=3600&lengths_incr=60` will return the last hour's data on queue lengths, with a sample for every minute.

Examples

A few quick examples for Windows and Unix, using the command line tool `curl`:

- Get a list of vhosts:

```
 :: Windows  
C:\> curl -i -u guest:guest http://localhost:15672/api/vhosts  
  
# Unix  
$ curl -i -u guest:guest http://localhost:15672/api/vhosts  
  
HTTP/1.1 200 OK  
Server: MochiWeb/1.1 WebMachine/1.10.0 (never breaks eye contact)  
Date: Mon, 16 Sep 2013 12:00:02 GMT  
Content-Type: application/json  
Content-Length: 30  
  
[{"name":"/","tracing":false}]
```

HTTP API reference

GET	PUT	DELETE	POST	Path	Description
X				/api/overview	Various random bits of information that describe the whole system.
X				/api/nodes	A list of nodes in the RabbitMQ cluster.
X				/api/nodes/name	An individual node in the RabbitMQ cluster. Add "?memory=true" to get memory statistics.
X				/api/extensions	A list of extensions to the management plugin.
X		X		/api/definitions /api/all-configuration (deprecated)	The server definitions - exchanges, queues, bindings, users, virtual hosts, permissions and parameters. Everything apart from messages. POST to upload an existing set of definitions. Note that: <ul style="list-style-type: none">The definitions are merged. Anything already existing is untouched.Conflicts will cause an error.In the event of an error you will be left with a part-applied set of definitions. For convenience you may upload a file from a browser to this URI (i.e. you can use <code>multipart/form-data</code> as well as <code>application/json</code>) in which case the definitions should be uploaded as a form field named "file".
X				/api/connections	A list of all open connections.
X	X			/api/connections/name	An individual connection. DELETEing it will close the connection. Optionally set the "X-Reason" header when DELETEing to provide a reason.
X				/api/channels	A list of all open channels.
X				/api/channels/channel	Details about an individual channel.
X				/api/exchanges	A list of all exchanges.
X				/api/exchanges/vhost	A list of all exchanges in a given virtual host.
X	X	X		/api/exchanges/vhost/name	An individual exchange. To PUT an exchange, you will need a body looking something like this: <pre>{"type": "direct", "auto_delete": false, "durable": true, "internal": false, "arguments": []}</pre> The <code>type</code> key is mandatory; other keys are optional.
X				/api/exchanges/vhost/name/bindings/source	A list of all bindings in which a given exchange is the source.
X				/api/exchanges/vhost/name/bindings/destination	A list of all bindings in which a given exchange is the destination.
		X		/api/exchanges/vhost/name/publish	Publish a message to a given exchange. You will need a body looking something like:

HTTP API reference

URL to call
(can have path variables)

GET	PUT	DELETE	POST	Path	Description
X				/api/overview	Various random bits of information that describe the whole system.
X				/api/nodes	A list of nodes in the RabbitMQ cluster.
X				/api/nodes/name	An individual node in the RabbitMQ cluster. Add "?memory=true" to get memory statistics.
X				/api/extensions	A list of extensions to the management plugin.
X		X	X	/api/definitions /api/all-configuration (deprecated)	<p>The server definitions - exchanges, queues, bindings, users, virtual hosts, permissions and parameters. Everything apart from messages. POST to upload an existing set of definitions. Note that:</p> <ul style="list-style-type: none">• The definitions are merged. Anything already existing is untouched.• Conflicts will cause an error.• In the event of an error you will be left with a part-applied set of definitions. <p>For convenience you may upload a file from a browser to this URI (i.e. you can use <code>multipart/form-data</code> as well as <code>application/json</code>) in which case the definitions should be uploaded as a form field named "file".</p>
X	X	X		/api/connections	A list of all open connections.
X				/api/connections/name	An individual connection. DELETEing it will close the connection. Optionally set the "X-Reason" header when DELETEing to provide a reason.
X				/api/channels	A list of all open channels.
X				/api/channels/channel	Details about an individual channel.
X				/api/exchanges	A list of all exchanges.
X				/api/exchanges/vhost	A list of all exchanges in a given virtual host.
X	X	X		/api/exchanges/vhost/name	An individual exchange. To PUT an exchange, you will need a body looking something like this: <code>{"type": "direct", "auto_delete": false, "durable": true, "internal": false, "arguments": []}</code> The <code>type</code> key is mandatory; other keys are optional.
X				/api/exchanges/vhost/name/bindings/source	A list of all bindings in which a given exchange is the source.
X				/api/exchanges/vhost/name/bindings/destination	A list of all bindings in which a given exchange is the destination.
		X		/api/exchanges/vhost/name/publish	Publish a message to a given exchange. You will need a body looking something like:

HTTP method to use
(only GET for monitoring)

RabbitMQ HTTP API

- Use any HTTP client – or a browser

```
$ curl -i -u guest:guest http://localhost:15672/api/aliveness-test/%2f
Server: MochiWeb/1.1 WebMachine/1.10.0 (never breaks eye contact)
Date: Thu, 28 Nov 2013 15:21:03 GMT
Content-Type: application/json
Content-Length: 15
Cache-Control: no-cache

{"status": "ok"}
```

Encoded default virtual host
(/ => %2f)

curl for Windows: <https://curl.haxx.se/download.html>

Aliveness test

- [`/api/aliveness-test/{vhost}`](#)
- Creates a queue, publishes and consumes a message
- Returns OK if successful
- Queue isn't deleted to prevent churn

Cluster messaging metrics

- [/api/overview](#)
- Returns message rates, queue totals
- Useful for a coarse-grained view

Nodes information

- [/api/nodes](#)
- An array of all the nodes
 - Are they all there?
 - "running" = true?
- System metrics and nodes settings
 - Used file descriptors, RAM or disc node, etc.

Nodes information

```
[  
  {  
    "partitions": [],  
    "os_pid": "23408",  
    "fd_used": 22,  
    "fd_total": 1024,  
    "sockets_used": 1,  
    "sockets_total": 829,  
    "mem_used": 40707184,  
    "mem_limit": 1617299046,  
    "mem_alarm": false,  
    "disk_free_limit": 50000000,  
    "disk_free": 27900448768,  
    "disk_free_alarm": false,  
    "proc_used": 194,  
    "proc_total": 1048576,  
    "statistics_level": "fine",  
    "uptime": 2035246,  
    "run_queue": 0,  
    "processors": 4,  
    ...  
  }]
```

The diagram illustrates the structure of node information by highlighting specific fields and connecting them to their respective meanings. The highlighted fields are:

- "partitions": A red oval surrounds this field, which is connected to a box labeled "Network partitions (should be empty)".
- "mem_alarm": A red oval surrounds this field, which is connected to a box labeled "Memory alarm".
- "disk_free_alarm": A red oval surrounds this field, which is connected to a box labeled "Disk usage alarm".

Connections

- [`/api/connections`](#)
- All the connections, with host, port, client properties, received/sent octets, etc.
- Helps to detect clients that generate too much traffic
- Don't poll too often if you have lots of connections
 - Thousands or more

Channels

- [/api/channels](#)
- Message rates (ack, deliver), channel settings (transactional or not, prefetch count), etc.
- Useful to detect unusual behavior

Exchanges

- `/api/exchanges`
`/api/exchanges/{vhost}`,
`/api/exchanges/{vhost}/{name}`
- Settings, rates
- Useful to check appropriate rate of messages (not too many, but enough)

Queues

- `/api/queues`, `/api/queues/{vhost}`, `/api/queues/{vhost}/{name}`
 - memory
 - size
 - consumer count
 - idle since
- Thresholds may differ by application and queue
- The kind of resource to monitor!

Account Permissions for Monitoring – 1

- Separate account recommended
 - "monitoring" tag
- Read-only access to all resources in all vhosts
- For aliveness test, require:
 - write to *amq.default*
 - configure for "aliveness-test"

Account Permissions for Monitoring – 2

User: rmqmon

▼ Overview

Tags	monitoring
Can log in with password	•

▼ Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/	aliveness-test	amq.default	.*	<button>Clear</button>
/orchestration-integration	aliveness-test	amq.default	.*	<button>Clear</button>

Agenda

- Operations
- Monitoring
- How to monitor with the HTTP API
- **Support for third-party monitoring tools**

Third-party Monitoring Tools

- Third-party monitoring tools may provide RabbitMQ plugins
- New Relic, Nagios, Splunk, etc.
- Easy to write your own using the HTTP API
 - Use a scripting language such as Perl or Python

Nagios Support

- A set of Nagios checks
 - <https://github.com/jamesc/nagios-plugins-rabbitmq>
- PERL scripts, easy to customize

Nagios Checks – 1

- `check_rabbitmq_aliveness`
 - Issues a liveness test
- `check_rabbitmq_server`
 - Checks memory, processes, file descriptors, and sockets
- `check_rabbitmq_objects`
 - Checks number of virtual hosts, exchanges, bindings, queues, and channels

Nagios Checks – 2

- `check_rabbitmq_overview`
 - Checks number of messages (total, ready, and unacknowledged)
- `check_rabbitmq_queue`
 - Checks number of messages (total, ready, and unacknowledged) and of consumers for a queue
- `check_rabbitmq_watermark`
 - Checks if memory alarm or disk alarm has been triggered

Nagios Checks – Typical Usage

```
check_rabbitmq_queue -H localhost --port=15672 --queue=quotations -w 20,10,5 -c 30,20,15
```

Thresholds for total, ready,
unacknowledged messages

Levels (warning or critical)

New Relic RabbitMQ Plugin

- Cloud based monitoring
- Custom RabbitMQ plugin
 - Uses Ruby
 - https://github.com/gopivotal/newrelic_pivotal_agent

Plugin Configuration

```
...
# Configuration for RabbitMQ  rabbitmq:
# Must enable rabbit management plugin
# Note: When monitoring multiple nodes you should use the real hostnames instead of localhost

# Uncomment the appropriate line for your version
# RabbitMQ Default URL version 3.0
management_api_url: http://guest:guest@localhost:15672
# RabbitMQ Default URL versions prior to 3.0
#management_api_url: http://guest:guest@localhost:55672
#
# Set "debug: true" to see additional debug output
#debug: false
...
```

Set host URL

New Relic – Dashboard Homepage

The screenshot shows the New Relic Dashboard homepage for the RabbitMQ Monitoring Plugin. On the left sidebar, under the 'RabbitMQ' section, there is a 'NEW' badge. The main content area displays two instances of the plugin: '192.168.73.129:15672' and 'localhost:15672'. The 'localhost:15672' instance shows 0 messages and 0 messages/sec. A red circle highlights the 'Alert conditions for localhost:15672' modal window, which contains fields for setting thresholds for various metrics: Queued Messages Ready (Caution: 5.0, Critical: 7.0), Queued Messages Unacknowledged, Acknowledged Message Rate, and a 'Save settings' button. A callout box with the text 'Set thresholds for alerts' points to this modal. The top right corner of the dashboard shows 'Pivotal_10 Lite / Mobile Lite' and a notification count of 2.

RabbitMQ Monitoring Plugin
2 instances

Name	Queued Messages Ready	Queued Messages Unacknowledged	Acknowledged Message Rate
192.168.73.129:15672	0 messages	0 messages	0 messages/sec
localhost:15672	0 messages	0 messages	0 messages/sec

PUBLISHED AND SUPPORTED BY
 Pivotal
[About us](#) | [Support site](#)

Recent RabbitMQ events 

Alert conditions for localhost:15672

Caution Critical

Queued Messages Ready	5.0	7.0
Queued Messages Unacknowledged		
Acknowledged Message Rate		

Save settings

Search our Documentation  Help Center Docs Privacy Terms Site status:  New Relic:  

Summary

- Important to tune RabbitMQ for the type of load expected
- Monitoring is supported through the HTTP REST API
- Support for third-party monitoring tools such as Nagios and New Relic is available

Finishing Up

Course Completed

What's Next?

What's Next

- Congratulations, we've finished the course
- What to do next?
 - Certification
 - Other courses
 - Resources
 - Evaluation

Certification

- Computer-based exam
- Check <https://academy.pivotal.io/> for details



Other Courses

- Many courses available
 - Pivotal Cloud Foundry
 - Spring
 - Gemfire
 - Data Science
 - Greenplum
 - Pivotal HDB
- See <http://pivotal.io/academy> for the most up-to-date list



Pivotal Support Offerings

- Support Portal: <https://support.pivotal.io>
- Global organization provides 24x7 support
- Premium and Developer support offerings:
 - <http://www.pivotal.io/support/offering>
 - <http://www.pivotal.io/support/oss>
- Both Pivotal and Open Source products
- Community forums, Knowledge Base, Product documents

Pivotal Consulting

- Custom consulting engagement?
 - Contact us at <https://pivotal.io/contact/spring-consulting>
 - Even if you don't have a support contract!
- Pivotal Labs
 - Agile development experts
 - <https://pivotal.io/labs>



Resources

- RabbitMQ product website
 - <http://www.rabbitmq.com/>
- Mailing list
 - <https://groups.google.com/forum/#!forum/rabbitmq-users>
- Want to contribute?
 - <http://www.rabbitmq.com/github.html>

Thank You!

- We hope you enjoyed the course
- Please fill out the course evaluation
 - Americas: <http://tinyurl.com/usa-eval>
 - EMEA: <http://tinyurl.com/emea-eval>
 - Asia-Pac: <http://tinyurl.com/apj-eval>
- Once you've done, login to *Pivotal Academy*
 - You can download your *Attendance Certificate*



Spring AMQP

Agenda

- **Introduction to Spring**
- Spring AMQP Overview
- Spring's AmqpTemplate
- Configuring AMQP Resources with Spring
- Sending Messages
- Receiving Messages

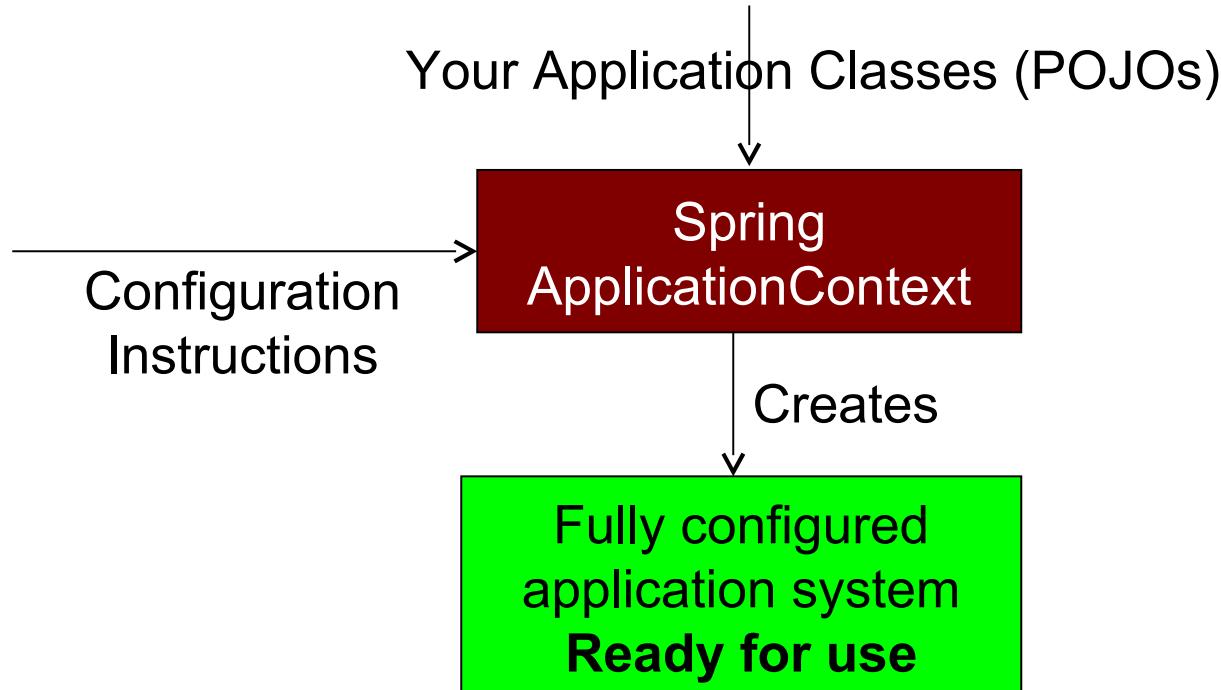
What is Spring?

- At it's core, a widely used Java framework
- Supports dependency injection: decouples application from implementation code
 - Facilitates multiple implementations
 - Separates service definition from configuration
- Speeds application development
 - Relieves developer from tedious boilerplate code for things such as connection retries, transactions, etc.
- Plethora of projects

Spring Projects

Spring AMQP	≡	Spring for Android	≡	Spring Batch	≡
Spring Data JPA	≡	Spring Data Commons	≡	Spring Data JDBC Extensions	≡
Spring Data MongoDB	≡	Spring Data Neo4J	≡	Spring Data Redis	≡
Spring Data REST	≡	Spring Data Solr	≡	Spring Flex	≡
Spring Framework	≡	Spring Data GemFire	≡	Spring for Apache Hadoop	≡
Spring HATEOAS	≡	Spring Integration	≡	Spring LDAP	≡
Spring Mobile	≡	Spring Roo	≡	Spring Security	≡
Spring Security OAuth	≡	Spring Shell	≡	Spring Social	≡
Spring Social Facebook	≡	Spring Social Twitter	≡	Spring Web Flow	≡
Spring Web Services	≡				

How Spring Works



Steps

1. Define classes
2. Define configuration
 - Using Java, annotations, or XML
3. Create application code to load dependencies defined in configuration

Your Application Classes

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Configuration

```
@Configuration
public class ApplicationConfiguration {

    @Bean TransferService transferService() {
        return new TransferServiceImpl(accountRepository());
    }

    @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource());
    }

    @Bean DataSource dataSource() {
        org.apache.tomcat.jdbc.pool.DataSource ds = new org.apache.tomcat.jdbc.pool.DataSource();
        ds.setDriverClassName("org.postgresql.Driver");
        ds.setUrl("jdbc:postgresql://localhost/transfer");
        ds.setUsername("transfer-app");
        ds.setPassword("secret45");
        return new ds;
    }
}
```

Creating and Using the Application

```
// Create the application from the configuration
ApplicationContext context =
    new AnnotationConfigApplicationContext(ApplicationConfiguration.class);

// Look up the application service interface
TransferService service = context.getBean(TransferService.class);

// Use the application
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

Agenda

- Introduction to Spring
- **Spring AMQP Overview**
- Spring's AmqpTemplate
- Configuring AMQP Resources with Spring
- Sending Messages
- Receiving Messages

Spring AMQP features

- RabbitTemplate for sending and receiving messages
- Sophisticated listener container for async consumption
- Auto-declaration of configured resources with RabbitAdmin
- Java/XML configuration and programmatic use as well
- Auto-recovery after node failure
- Queue affinity handling for the best performance
- And more...

NOTE

This module does not cover all these features!

Agenda

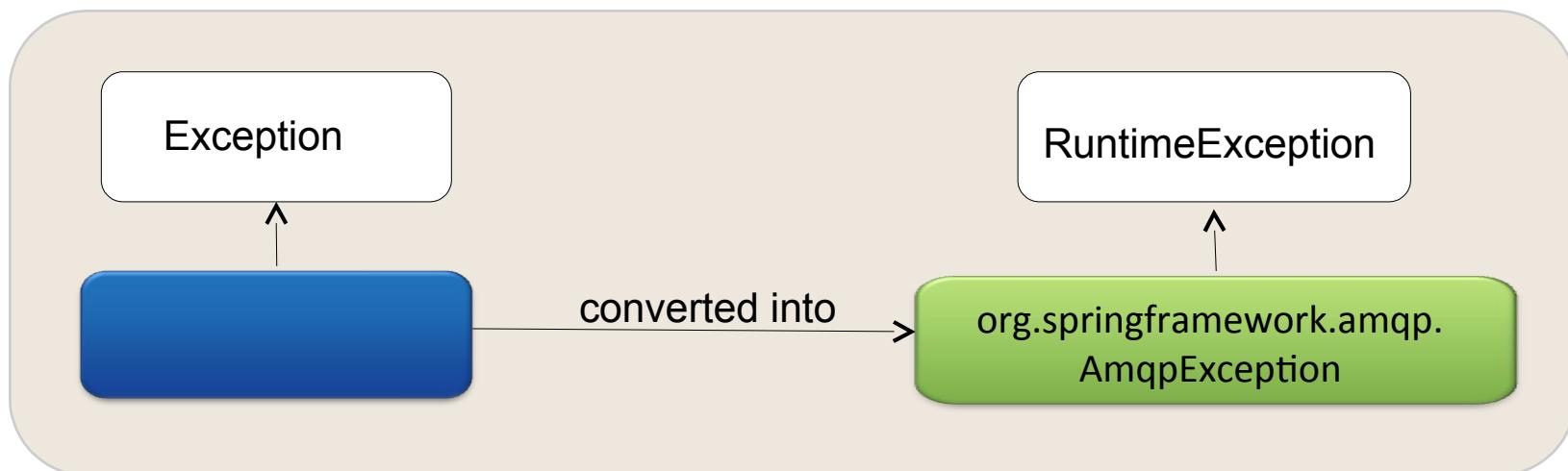
- Introduction to Spring
- Spring AMQP Overview
- **Spring's AmqpTemplate**
- Configuring AMQP Resources with Spring
- Sending Messages
- Receiving Messages

Spring's AmqpTemplate

- Interface that defines main operations, such as sending/receiving messages
- RabbitTemplate is currently the only implementation
 - Specifically depends on RabbitMQ Java-client
- Simplifies usage of the API
 - Reduces boilerplate code
 - Manages resources transparently
 - Converts checked exceptions to runtime equivalents
 - Provides convenience methods and callbacks

Exception Handling

- Exceptions in AMQP checked by default
- AmqpTemplate converts checked exceptions to runtime equivalents



Agenda

- Introduction to Spring
- Spring AMQP Overview
- Spring's AmqpTemplate
- **Configuring AMQP Resources with Spring**
- Sending Messages
- Receiving Messages

ConnectionFactory Configuration

- Typically use the CachingConnectionFactory

```
@Bean ConnectionFactory connectionFactory() {  
    CachingConnectionFactory connectionFactory =  
        new CachingConnectionFactory("localhost");  
    connectionFactory.setUsername("guest");  
    connectionFactory.setPassword("guest");  
    return connectionFactory;  
}
```

RabbitTemplate Configuration

- *Must* provide reference to ConnectionFactory
 - via either constructor or setter injection
- Optionally provide delegates to handle some of the work
 - Message Converter

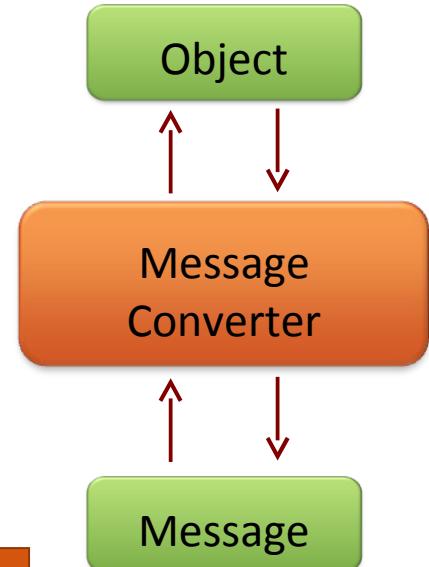
```
@Bean RabbitTemplate rabbitTemplate() {  
    RabbitTemplate tpl = new RabbitTemplate(  
        connectionFactory()  
    );  
    tpl.setMessageConverter(new JsonMessageConverter());  
    return tpl;  
}
```

MessageConverter

- RabbitTemplate uses **MessageConverter** to convert between objects and messages
 - You only send and receive objects
 - Decouples application from AMQP transport
- Default **SimpleMessageConverter** handles basic types
 - text-based content
 - serialized Java objects
 - simple byte arrays

NOTE

It is possible to implement custom converters by implementing the MessageConverter interface



JSON MessageConverter

- Avoid Java serialization
 - Otherwise could only support Java clients
- JSON: common language-independent message payload
- JsonMessageConverter: one implementation available

```
@Bean RabbitTemplate rabbitTemplate() {  
    RabbitTemplate tpl = new RabbitTemplate(  
        connectionFactory()  
    );  
    tpl.setMessageConverter(new JsonMessageConverter());  
    return tpl;  
}
```

CachingConnectionFactory settings

- CachingConnectionFactory has many settings
 - E.g. channel cache size, default is 1
- Override defaults during bean creation

```
@Bean ConnectionFactory connectionFactory() {  
    CachingConnectionFactory connectionFactory =  
        new CachingConnectionFactory("localhost");  
    connectionFactory.setChannelCacheSize(10);  
    return connectionFactory;  
}
```

Agenda

- Introduction to Spring
- Spring AMQP Overview
- Spring's AmqpTemplate
- Configuring AMQP Resources with Spring
- **Sending Messages**
- Receiving Messages

Sending Messages

- Template provides several options
 - One line methods leveraging template's MessageConverter
 - Callback-accepting methods that reveal more of the AMQP Java-client API
- Use simplest option for the task at hand

Sending a POJO

- Message can be sent in one single line

```
public class OrderManagerImpl implements OrderManager {  
  
    @Autowired RabbitTemplate rabbitTemplate;  
    private final String ORDERS_EXCHANGE = "Orders";  
  
    public void placeOrder(Order order) {  
  
        String routingKey = order.getSourceRegion();  
        // Use message converter  
        rabbitTemplate.convertAndSend(ORDERS_EXCHANGE,  
            routingKey, order);  
  
    }  
}
```

Agenda

- Introduction to Spring
- Spring AMQP Overview
- Spring's AmqpTemplate
- Configuring AMQP Resources with Spring
- Sending Messages
- **Receiving Messages**

Synchronous Message Reception

- AmqpTemplate can also receive messages, but methods are blocking (with optional timeout)
 - receive()
 - receive(String queueName)
- MessageConverter can be leveraged for message reception as well

```
Object someSerializable =
    template.receiveAndConvert();

Object someSerializable =
    template.receiveAndConvert(someQueue);
```

Synchronous Request-Reply

- RPC pattern using messaging
- Convenience `convertSendAndReceive()` method deserializes reply message payload into object

```
...
String orderId = "12345";
OrderDetailsRequest req = new OrderDetailsRequest(orderId);

String exchange = "order-services";
String routingKey = "get-order-details-req.1_0";
Order order = (Order)template.convertSendAndReceive(
    exchange, routingKey, req);
...
```

AMQP Asynchronous Listener Interfaces

- MessageListener
- ChannelAwareMessageListener (if access to channel required)

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.ChannelAwareMessageListener;
import com.rabbitmq.client.Channel;

public class GetOrderDetails implements ChannelAwareMessageListener {

    public void onMessage(Message msg, Channel channel) {
        // Process message...
    }
}
```

Spring's MessageListener Container

- Required for asynchronous listeners
- Default implementation: SimpleMessageListenerContainer
- Uses AMQP Java client
- Configurable
 - concurrency
 - acknowledgements
 - transactional channel
 - etc.

Defining a Simple AMQP Message Listener

- Use SimpleMessageListenerContainer

```
@Bean SimpleMessageListenerContainer messageListenerContainer() {  
    SimpleMessageListenerContainer container =  
        new SimpleMessageListenerContainer();  
    container.setConnectionFactory(connectionFactory());  
    container.setQueueNames("quotes");  
    container.setMessageListener(quoteRequestAmqpEndpoint());  
    return container;  
}
```

- Listener must implement MessageListener or ChannelAwareMessageListener

Spring vs AMQP Acknowledgement Modes

- Caution: Spring-AMQP's acknowledgement mode terminology differs from AMQP!

Spring-AMQP Ack Mode	AMQP Auto-Ack	Description
None	true	No acknowledgements in either case.
Auto	false	Spring container is responsible for acknowledging the message.
Manual	false	Developer must explicitly acknowledge the message.

Spring's Message-Driven POJO

- Spring also allows you to specify a plain Java object that can serve as a listener
 - MessageConverter provides parameter
 - Any return value sent to response-exchange after conversion

```
public class OrderService {  
    public OrderConfirmation order(Order o) {}  
}
```

```
SimpleMessageListenerContainer container = (...);  
container.setQueueNames("queue.orders");
```

```
MessageListenerAdapter adapter = new MessageListenerAdapter(orderService);  
adapter.setDefaultListenerMethod("order");  
adapter.setResponseExchange("exchange.confirmations");  
container.setMessageListener(adapter);
```

1

2

3

Summary

- Spring AMQP simplifies and reduces boilerplate code
- AmqpTemplate defines main operations, such as sending/receiving messages
- MessageConverter's simplify serialization/deserialization of Java objects
- Both synchronous and asynchronous message consumers are supported

Using the Spring AMQP Template