Internet of Things

Publish & Subscribe Broker Architecture

(RabbitMQ)

AURIGA
the banking e-volution

THE # NEXTGENBANK

www.aurigaspa.com

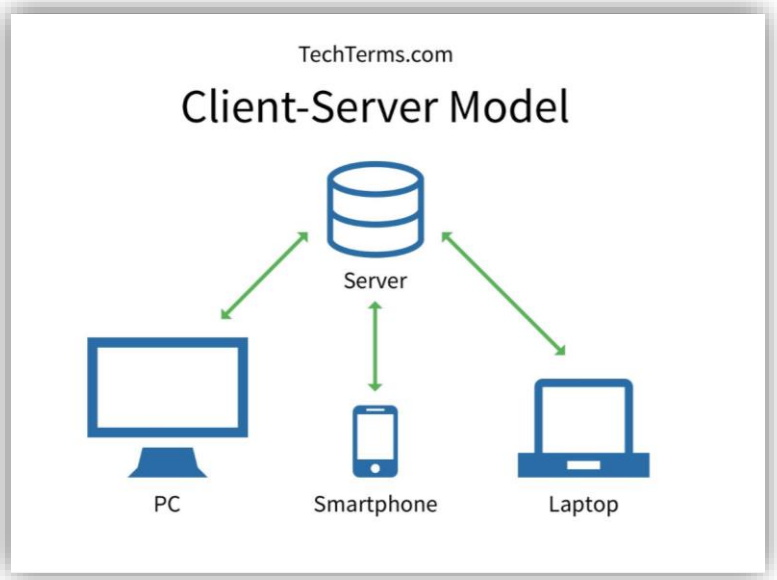
Summary

- Broker Pub/Sub Architecture
- RabbitMQ - Main features
- Quorum queues vs Classic mirrored queues
- Supported protocols
- Supported clients
- Database integrations

Broker Pub/Sub Architecture

Distributed Architecture - Issues

- Data is often stored on legacy on premise data centers.
- Applications distributed on multiple networks may go through unplanned downtime and maintenance.



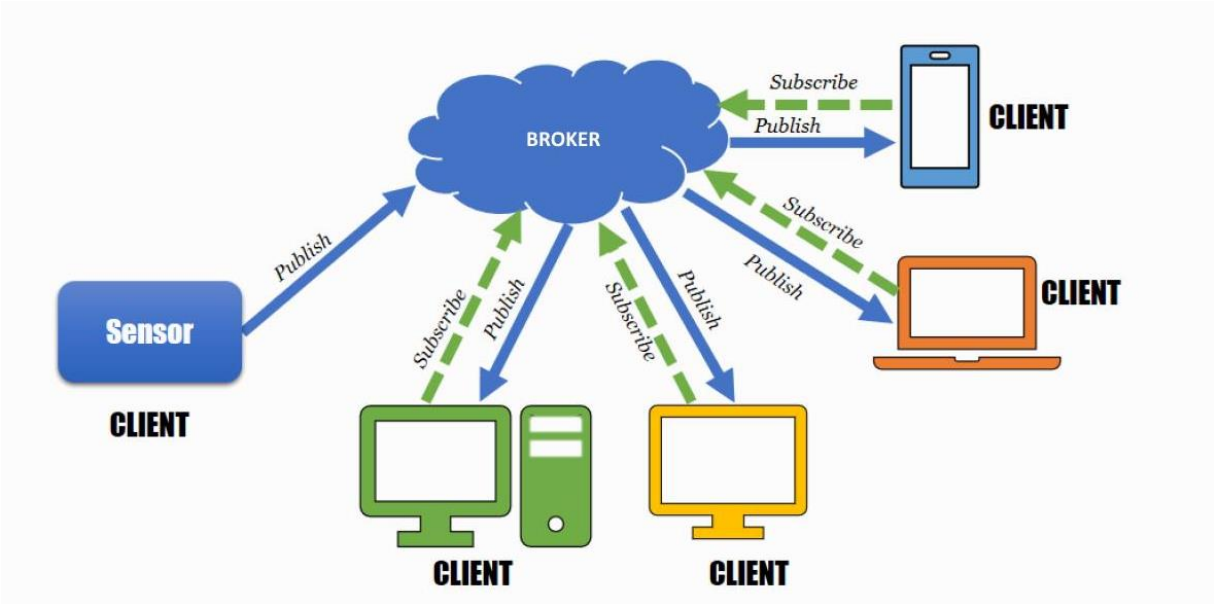
- Some processes may require multiple sub-processes (i.e. email/PDF generation, batch jobs, interface with 3rd party systems) and a simple HTTP REST call isn't enough to manage a delayed response.
- Gathering data for log processing, data analytics, and audit logging should ideally not have a side effect on the performance of the overall system.

Publish – Subscribe Architecture

Messaging pattern where **publishers** of messages categorize messages into classes without knowledge of which **subscribers** there may be.

Similarly, subscribers express interest in one or more classes and only receive messages that are of interest.

Broker: manages mechanism for copy of each message from publishers to subscribers.



From Client-Server to Publish-Subscribe - Advantages

Increased reliability:

- By decoupling different components with an additional messaging layer, the system becomes more fault-tolerant
- Should one system component go offline, the others can still continue to interact with the queues
- Queues can be replicated to add even more reliability

From Client-Server to Publish-Subscribe - Advantages

Better performance:

- Queues add asynchronicity to the application
- Consumers process messages only when available
- No system component is ever stalled waiting for another, optimizing data flow

Better scalability:

- based on the active load, different components of the message queue can be scaled accordingly

From Client-Server to Publish-Subscribe - Advantages

Easier to develop (or migrate to) micro services:

- Event-based micro services integration patterns can be used to optimize scalability and resiliency

Message queues can be used:

- To coordinate multiple micro services
- To notify micro services of data changes
- As an event *firehose* to process IoT/social/real-time data

RabbitMQ

List of Message Broker

- AWS Amazon MQ
- AWS Kinesis
- Apache ActiveMQ
- Apache Kafka
- Eclipse Mosquitto MQTT Broker (Eclipse Foundation)
- Google Cloud Pub/Sub
- HiveMQ MQTT Broker
- IBM MQ
- JBoss Messaging
- Microsoft Azure Service Bus
- Microsoft BizTalk Server
- Oracle Message Broker
- **RabbitMQ**
- WSO2 Message Broker
- ...

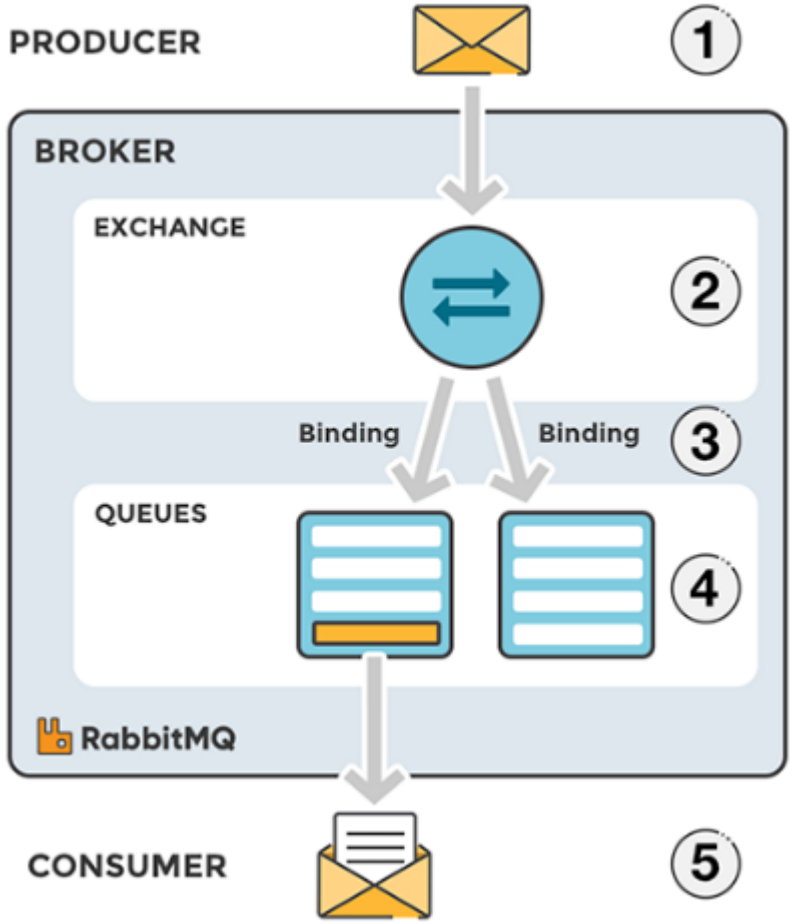
Choosing the right message broker for an application largely depends on the business case of the application.

Broker is NOT a black box

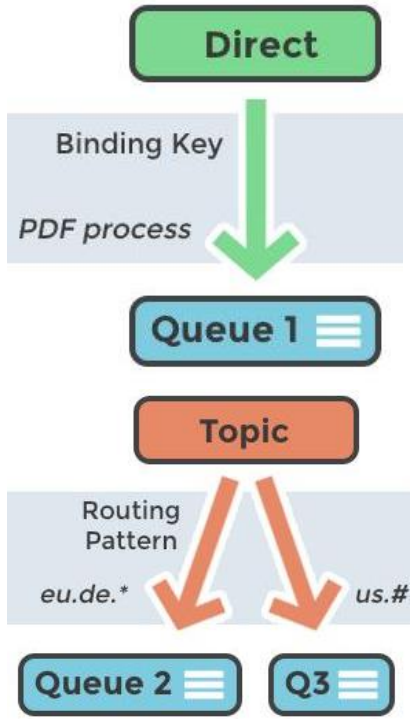
In the publish-subscribe model, subscribers (also known as *Consumers*) typically receive only a subset of the total messages published from *Producers*. The process of selecting messages for reception and processing is called *filtering*.

A broker in RabbitMQ is composed of:

- **Exchanges:** are responsible for delivering messages to bound queues.
- **Queues:** data structures with FIFO order of elements.

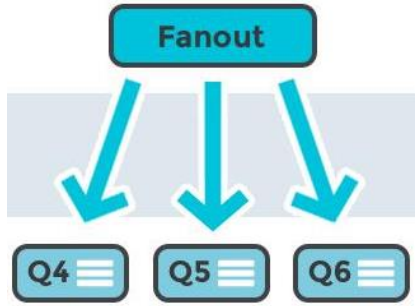


About Exchanges



Direct Exchange

Topic Exchange



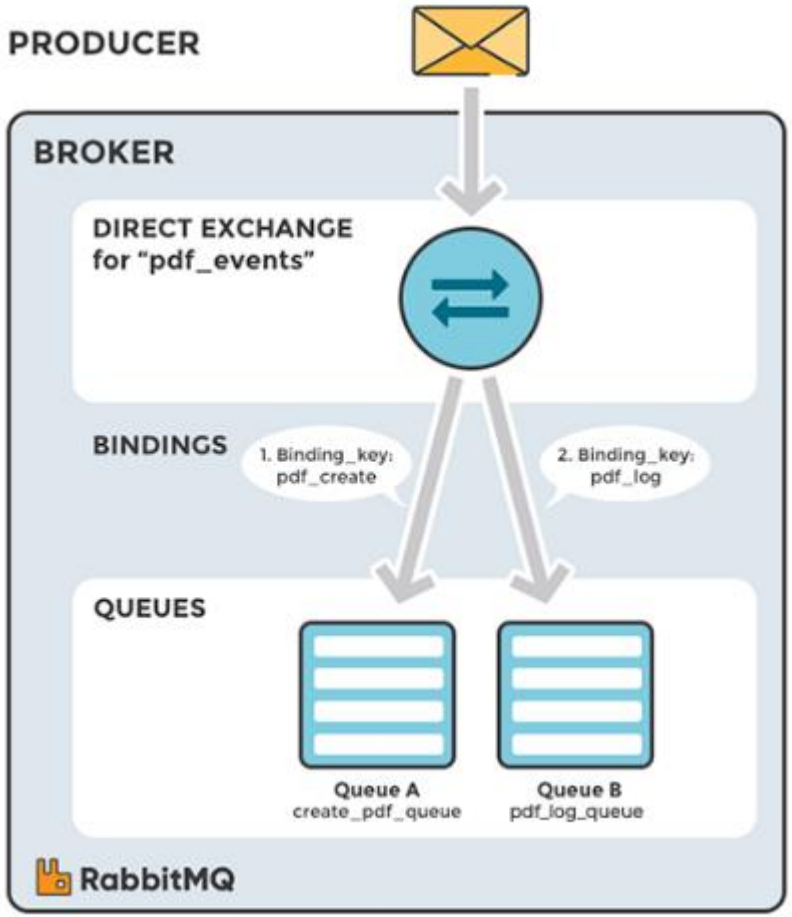
Fanout Exchange

An other special exchange is the *Header Exchange*.

Direct Exchange

A **direct exchange** will deliver the incoming message to any queue whose binding key exactly matches the routing key of the message.

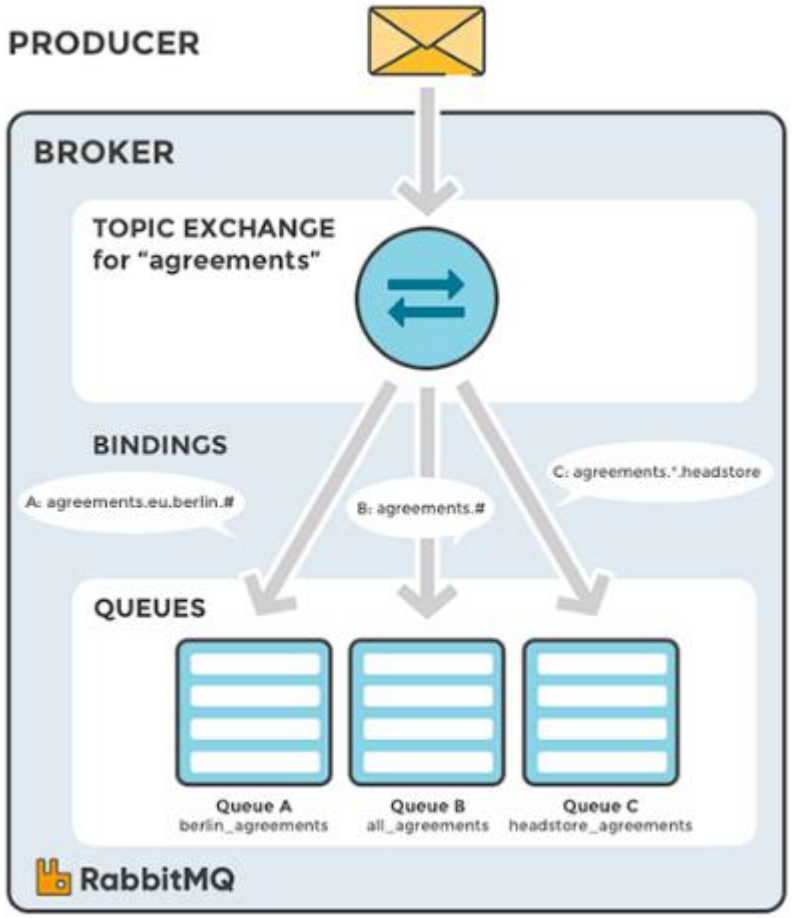
Direct exchanges are often used to distribute tasks between multiple workers/consumers in a round robin manner.



Topic Exchange

Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange.

The topic exchange type is often used to implement various pub/sub pattern variations.

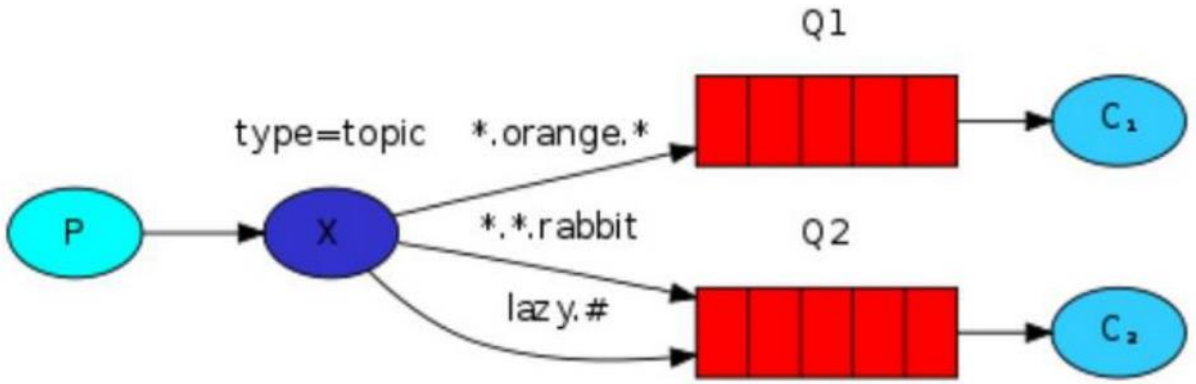


Topic Exchange

Topic exchanges are commonly used for multicast routing. Whenever a problem involves multiple consumers/applications that selectively choose which type of messages they want to receive, the use of topic exchanges should be considered.

Examples

- Distributing data relevant to specific geographic location
- Background task processing done by multiple workers (capable of handling specific set of tasks)
- Stocks price updates (and updates on other kinds of financial data)
- News updates that involve categorization or tagging
- Orchestration of services of different kinds in the cloud



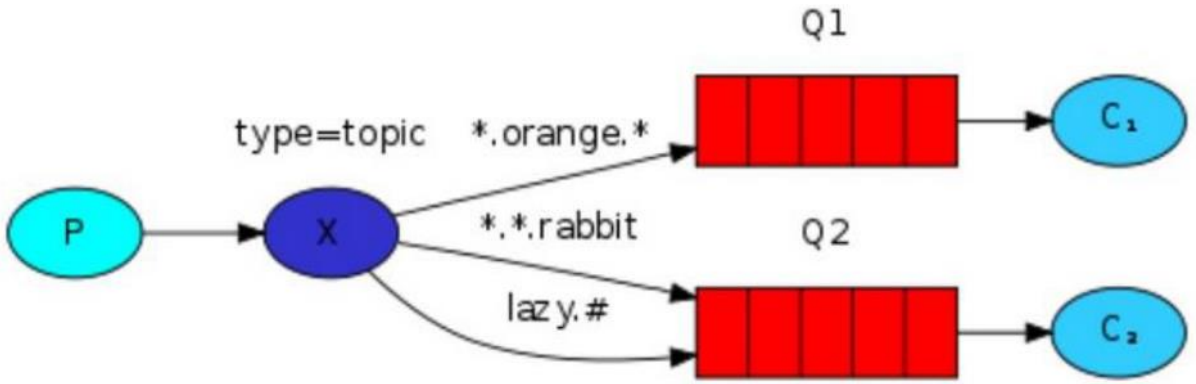
Topic Exchange – Use Case

Messages sent to a topic exchange have a special routing key consisting of a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message.

Examples:

"*stock.usd.nyse*", "*nyse.vmw*", "*quick.orange.rabbit*".

NOTE: The binding key must also be in the same format.

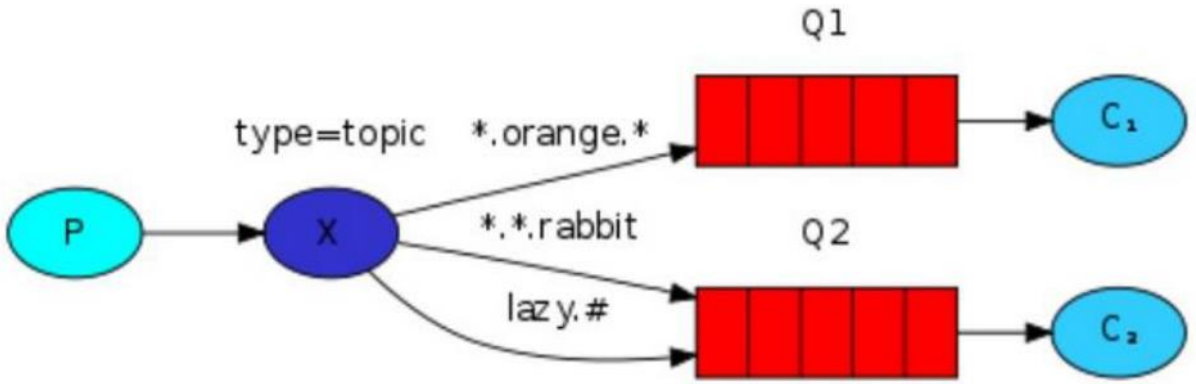


Topic Exchange – Use Case

A *topic exchange* is similar to a *direct exchange*. A message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key.

However there are two important special cases for *binding keys*:

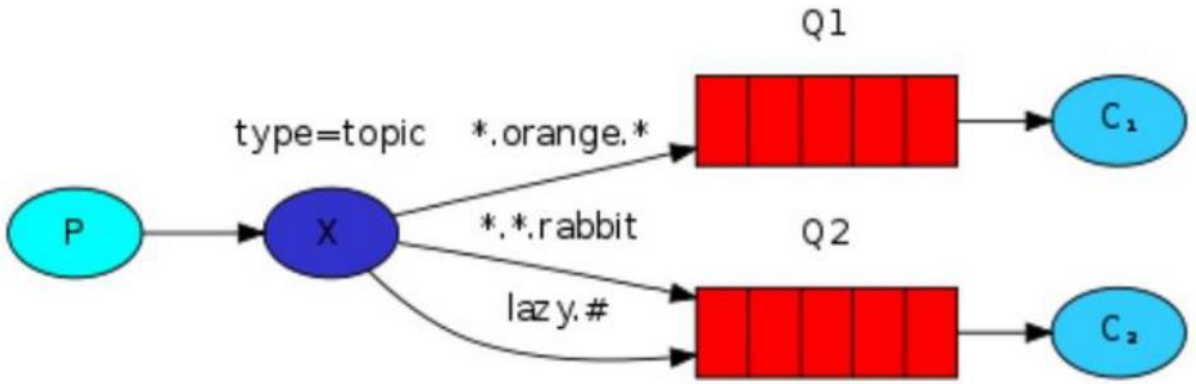
- *** (star)**: can substitute for exactly one word
- **# (hash)**: can substitute for zero or more words.



Topic Exchange – Use Case

In the following example, messages related to animals will be sent to an exchange with a routing key that consists of a 3-level topic: "**<celerity>.<color>.<species>**".

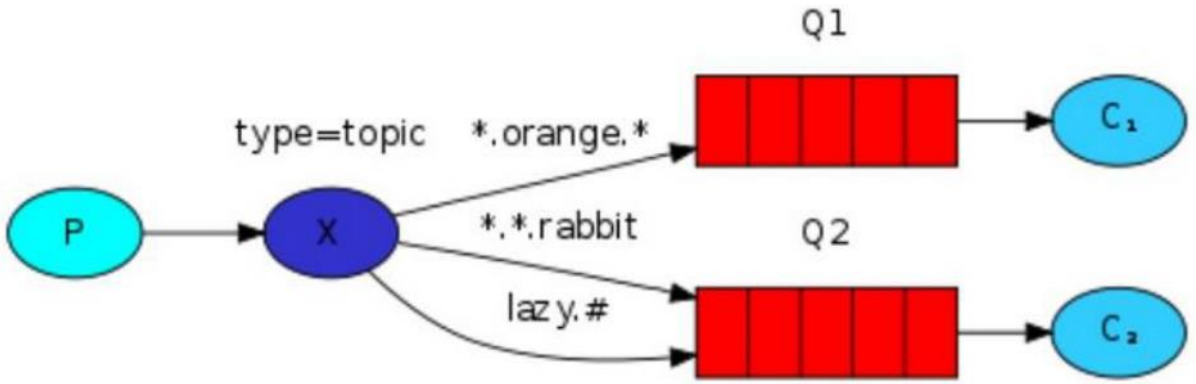
The first level in the routing key will describe the celerity of the animal, the second its color and the third one its species.



Topic Exchange – Examples

Examples:

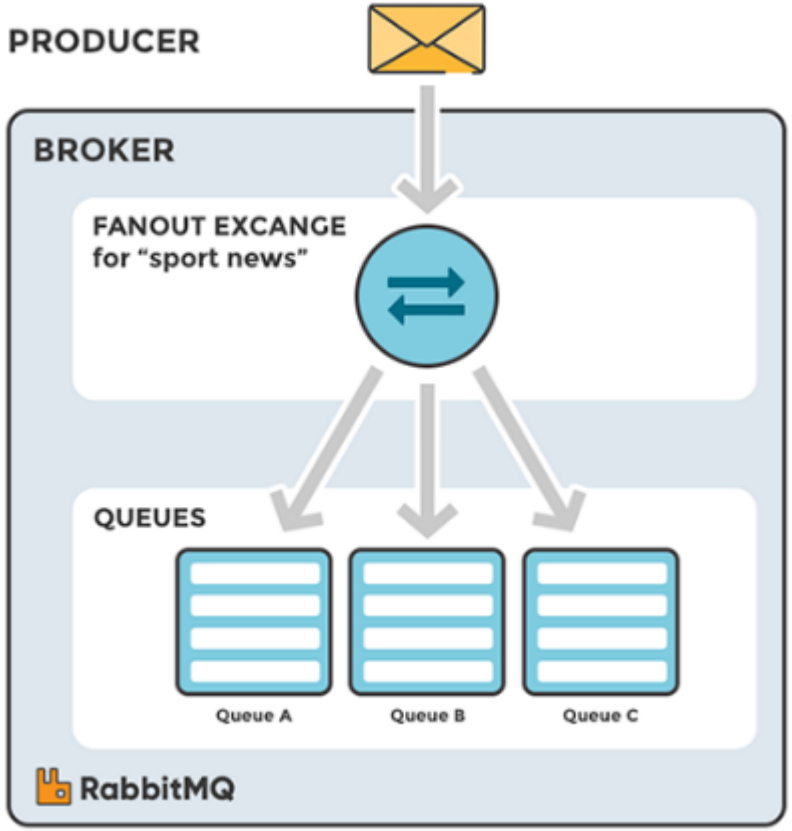
- "**quick.orange.rabbit**" will be delivered to both queues.
- "**lazy.orange.elephant**" will be delivered to both queues.
- "**quick.orange.fox**" will only go to the Q1.
- "**lazy.brown.fox**" only to Q2.
- "**lazy.pink.rabbit**" will be delivered to Q2 only once, even though it matches 2 bindings.
- "**quick.brown.fox**" does not match any binding so it will be discarded.



Fanout Exchange

A **fanout exchange** routes messages to all of the queues that are bound to it and the routing key is ignored.

If N queues are bound to a fanout exchange, when a new message is published to that exchange a copy of the message is delivered to all N queues.

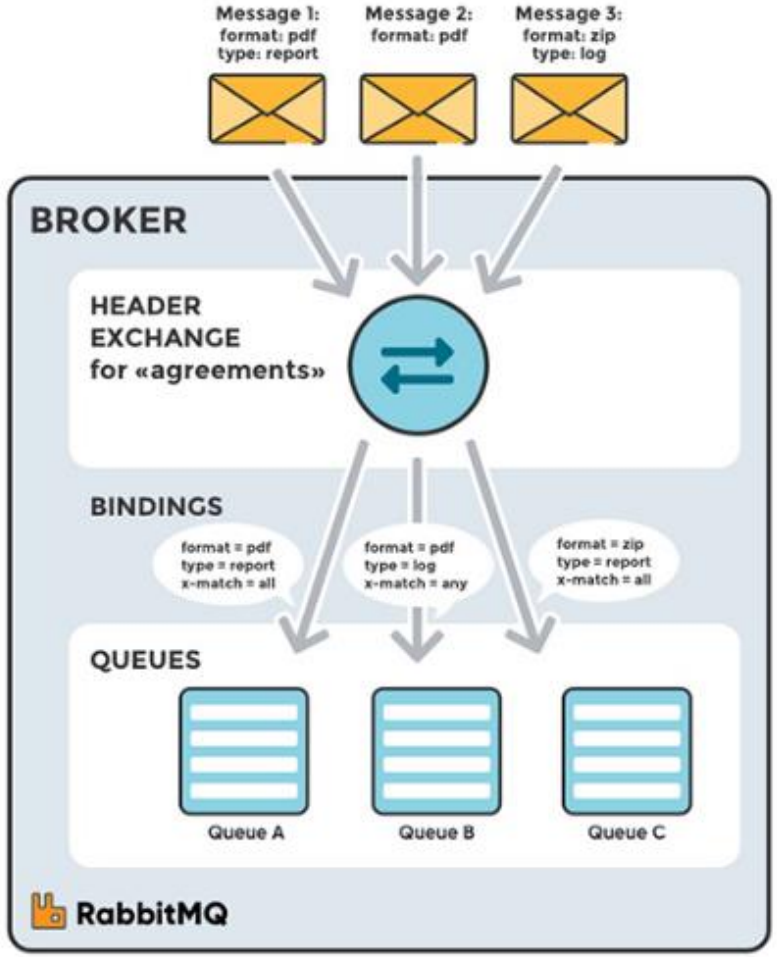


Header Exchange

A **headers exchange** is designed for routing on multiple attributes that are more easily expressed as message (custom) headers than a routing key.

Headers exchanges ignore the routing key attribute. Instead, the attributes used for routing are taken from the headers attribute.

A message is considered matching if the value of the header equals the value specified upon binding.

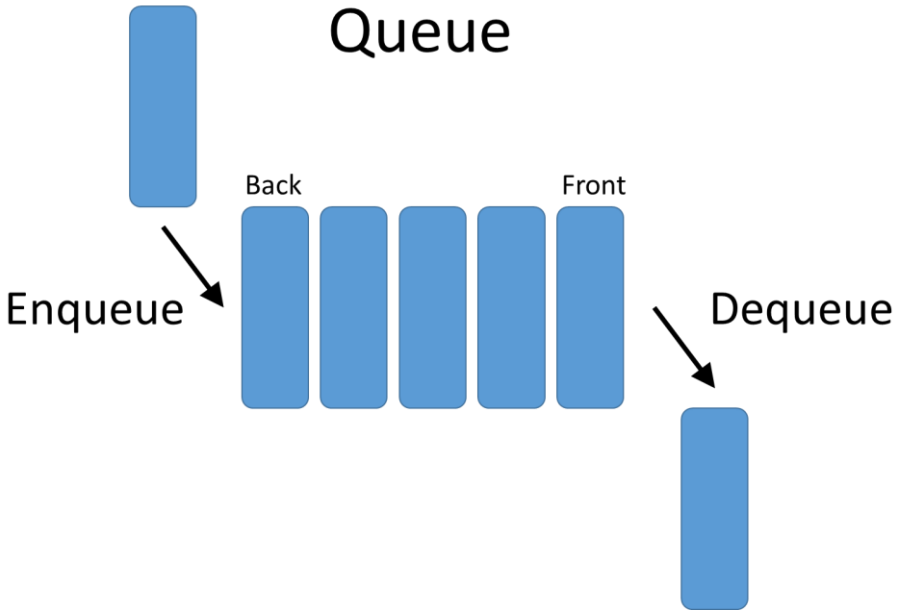


Queues

A **queue** is a sequential data structure with 2 primary operations: an item can be queued (added) at the tail and dequeued (consumed) from the head.

Properties that can be defined when declaring a queue:

- **Name**
- **Durable** (the queue will survive a broker restart)
- **Exclusive** (used by only one connection and the queue will be deleted when that connection closes)
- **Auto delete** (queue that has had at least 1 consumer is deleted when last consumer unsubscribes)
- **Arguments** (optional; used by plugins and broker specific features)



Queues – Message ordering

Messages are queued and dequeued (delivered to consumers) in the **FIFO** manner.

Ordering also can be affected by the presence of:

- multiple competing consumers
- consumer priorities
- message redeliveries

Applications can assume messages published on a single channel will be queued in publishing order in all the queues they get routed to.

When publishing happens on multiple connections or channels, their sequences of messages will be routed concurrently and interleaved.

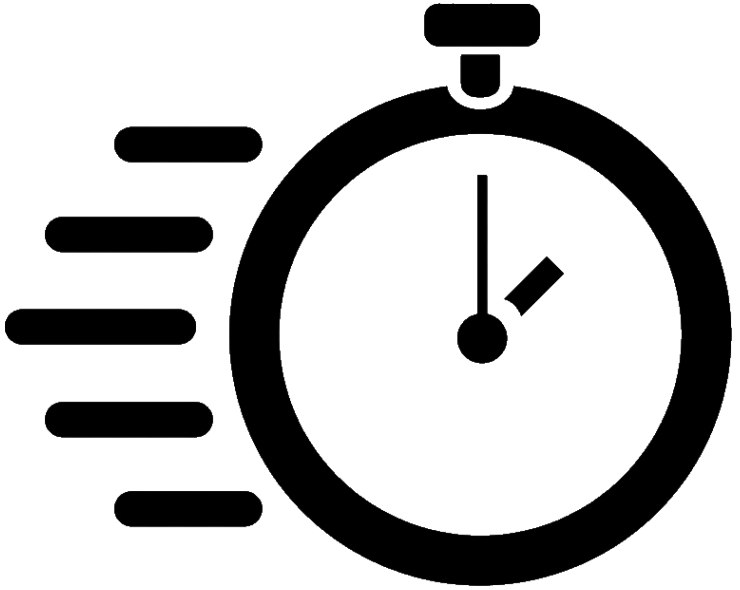
If all of the consumers have equal priorities, they will be picked on a round robin basis.

Queues - Durability

Queues can be durable or transient.

Metadata of a durable queue is stored on disk, while metadata of a transient queue is stored in memory when possible.

- **Transient queues** will be deleted on node boot. Messages in transient queues will also be discarded.
- **Durable queues** will be recovered on node boot, including messages in them published as persistent.

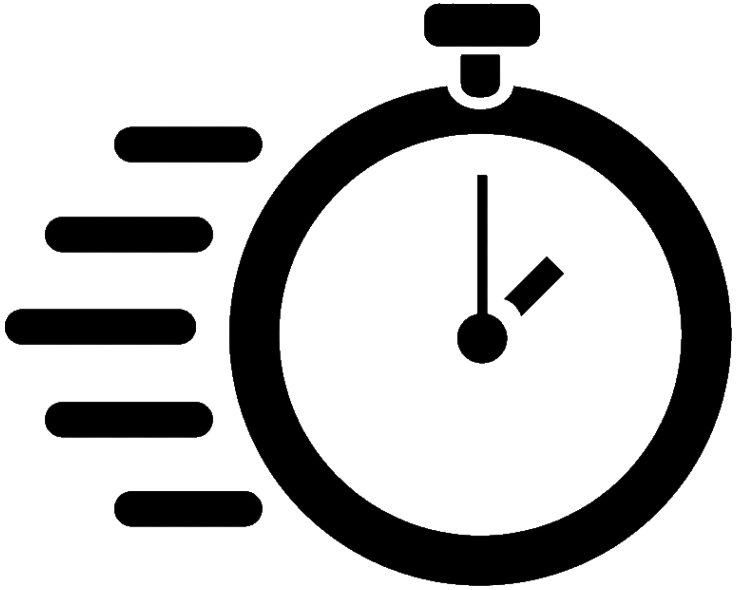


Queues - Durability

In most cases, throughput and latency of a queue are not affected if a queue is durable or not.

Only environments where queues are deleted and re-declared hundreds or more times a second will see latency increment for some operations, namely on bindings.

Temporary queues can be a reasonable choice for workloads with transient clients (i.e. temporary WebSocket connections, mobile apps and devices that are expected to go offline or use switch identities)



Queues - Exclusive

An exclusive queue can only be used (consumed from, purged, deleted, etc.) by its declaring connection. An attempt to use an exclusive queue from a different connection will result in a channel-level exception RESOURCE_LOCKED with an error message that says 'cannot obtain exclusive access to locked queue'.

Exclusive queues are deleted when their declaring connection is closed or gone (e.g. due to underlying TCP connection loss). They therefore are only suitable for client-specific transient state.

It is common to make exclusive queues server-named.

Quorum vs Classic mirrored queues

Replicated and Distributed Queues

Queues can be replicated to multiple cluster nodes and across loosely coupled nodes or clusters.

There are two replicated queue types provided:

- **Quorum queues**: implementing a durable, replicated FIFO queue based on the Raft consensus algorithm
- **Classic queues** with mirroring enabled (which will be removed in a future version of RabbitMQ)

Quorum queues is the recommended option for most workloads and use cases.

Quorum Queues – When to use

Quorum queues are designed for topologies where:

- Queues exist for a long time
- Queues are critical to certain aspects of system operation

Therefore fault-tolerance and data safety is more important than low latency and advanced queue features.

Quorum (in a distributed system) system): an agreement between the majority of nodes ((N/2) + 1 where N = total number of system participants).

Examples:

- incoming orders in a sales system
- votes cast in an election system
- Where potentially losing messages would have a significant impact on system correctness and function

Quorum Queues – When NOT to use

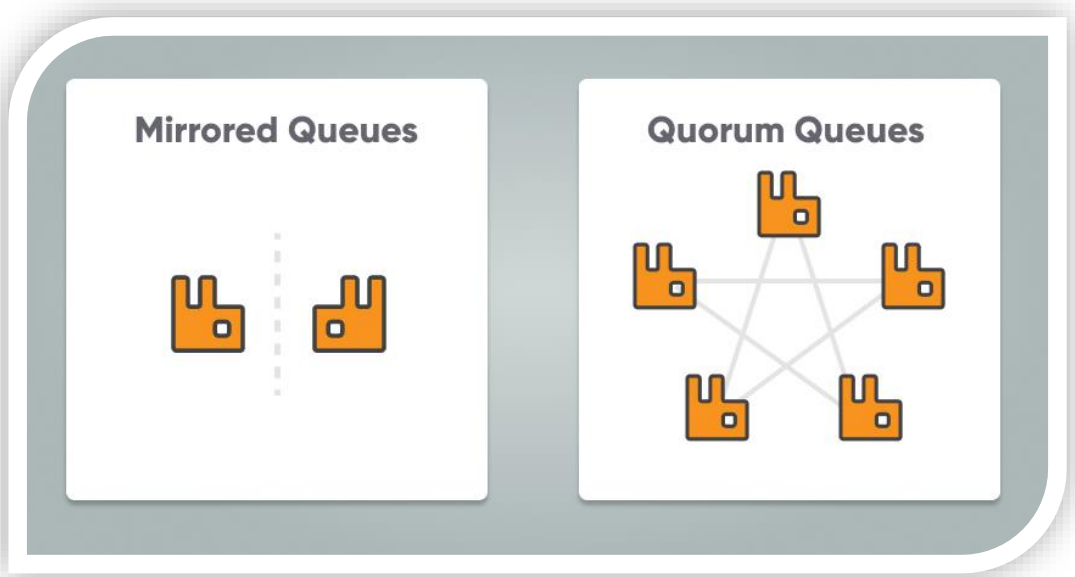
- **Temporary nature of queues**: transient or exclusive queues, high queue churn (declaration and deletion rates)
- **Low latency**: the underlying consensus algorithm has an inherently higher latency due to its data safety features.
- **When data safety is not a priority**: e.g. applications do not use manual acknowledgements and publisher confirms are not used
- **Very long queue** "*backlogs:quorum*" queues currently keep all messages in memory at all times, up to a limit

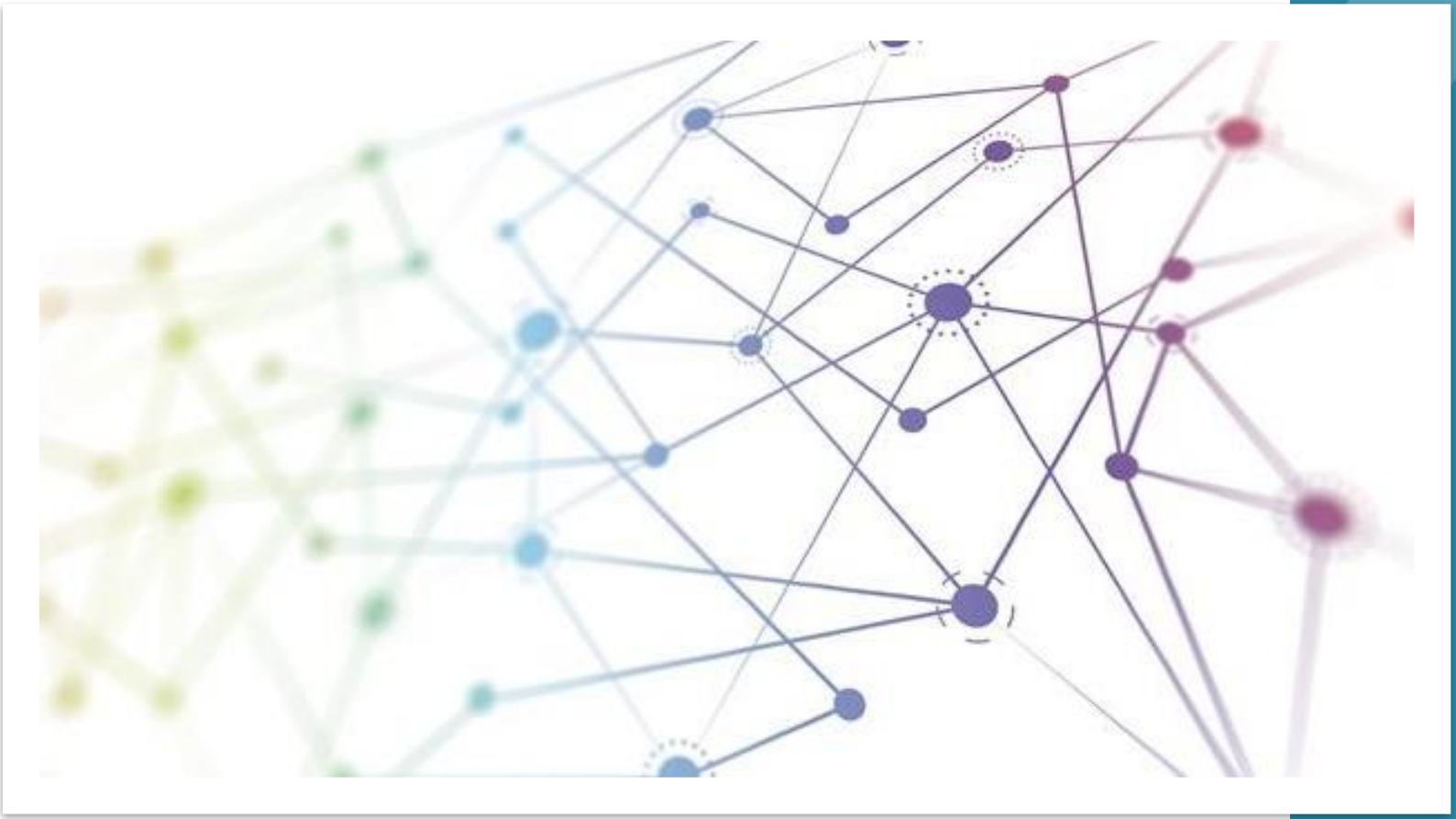
Quorum Queues – Fault Tolerance

NODE COUNT	TOLERATED FAILURES
1	0
2	0
3	1
4	1
5	2

RabbitMQ clusters with less than 3 nodes do not benefit fully from the quorum queue guarantees.

Therefore, the recommended number of replicas for a quorum queue is the quorum of cluster nodes (but no fewer than 3).





Supported Protocols

Supported Protocols

PROTOCOLS	DESCRIPTION
AMQP 0-9-1	• Core protocol supported by RabbitMQ • Binary protocol, with strong messaging semantics • Large number of client libraries available for many different programming languages and environments
STOMP	• Simple (or Streaming) Text Orientated Messaging Protocol • Text based messaging protocol emphasizing (protocol) simplicity • RabbitMQ supports STOMP (all current versions) via a plugin
MQTT	• Binary protocol for lightweight pub/sub messaging • RabbitMQ supports MQTT 3.1 via a plugin
AMQP 1.0	• New version of the AMQP protocol with fewer semantic requirements • RabbitMQ supports AMQP 1.0 via a plugin

Supported Protocols

Furthermore, RabbitMQ can transmit messages over HTTP in three ways:

- **Web STOMP plugin**: supports STOMP messaging to the browser using WebSockets
- **Web MQTT plugin**: supports MQTT messaging to the browser using WebSockets
- **Management plugin**: supports a simple HTTP API to send and receive messages.
  - **NOTE**: This is primarily intended for diagnostic purposes but can be used for low volume messaging without reliable delivery.

Supported Clients

Supported Clients

RabbitMQ is officially supported on a number of operating systems and has several official client libraries.

In addition, the RabbitMQ community has created numerous clients, adaptors and tools.

Supported Clients

- Java
- Spring Framework
- .NET
- Ruby
- Python
- PHP
- JavaScript, Node
- Objective-C, Swift
- Rust
- Crystal
- C, C++
- Go
- Erlang
- Haskell
- Ocaml
- COBOL
- Scala
- Groovy, Grails
- Clojure
- Jruby
- Android
- iOS
- Unity 3D

Database Integrations

Database Integrations

DBMS	Documentation
Oracle	Oracle Stored Procedures for RabbitMQ
SQL Server (SSIS)	RabbitMQ component for SQL Server Integration Services (SSIS)
PostgreSQL	RabbitMQ integration with PostgreSQL
Riak	RabbitMQ RiakExchange

COPYRIGHT

THANKS FOR YOUR ATTENTION

Narracci Domenico

domenico.narracci@aurigaspa.com

AURIGA
the banking e-volution

THE # NEXTGENBANK

www.aurigaspa.com