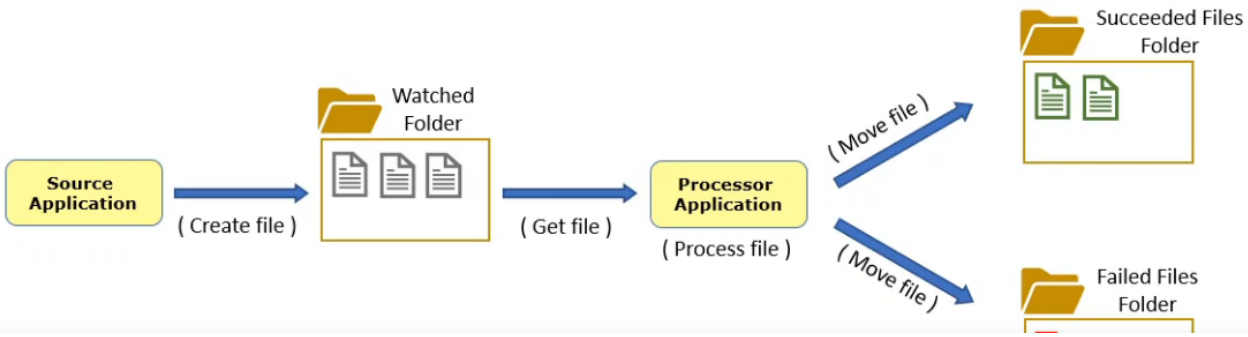
RabitMQ

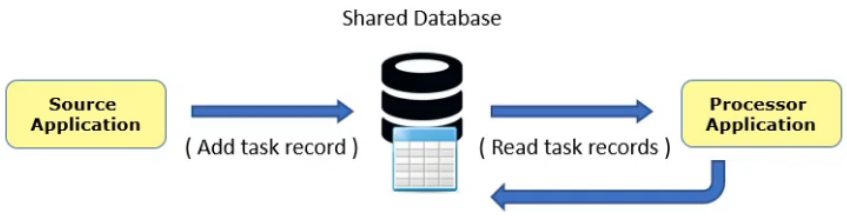
1. Application of queue

- One of the applications create files that need to be processed and put them into a well-known folder.
- Other application watches that folder for new files, and if a new file appears in the folder, it is automatically grabbed and processed.



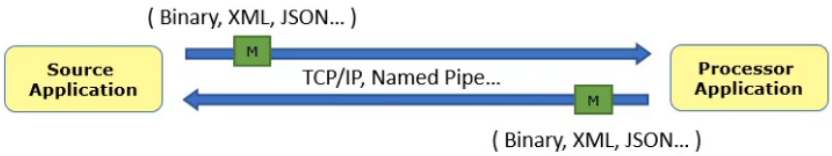
- Processing File

- One of the applications adds records, to a task table, in a shared database.
- Other application, watches that table for new entries, and if there is a new or unprocessed reads it, processes it and finally marks it as processed.
- It may also write the result to a column in the same table or to a different table.



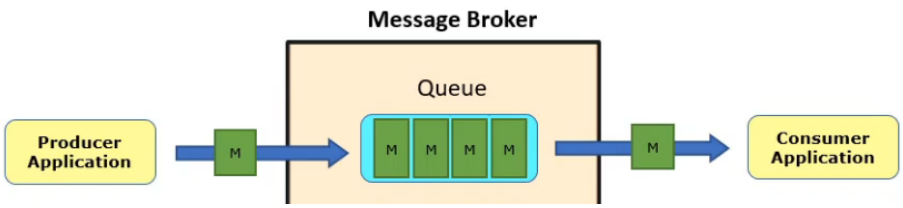
- Sync data

- Applications connect and send messages to each other directly.
- Type of the connection may be a TCP/IP connection, or a named pipe connection.
- After initiating the connection, they start sending messages to each other, directly.
- Format of the message may be anything. It can be binary or text based, like XML or JSON.



- Intergrate Service

- Applications send messages to each other in any format.
- But this time, with the help of an intermediate application.
- Those applications are mostly called as Message Brokers or Message Buses.
- Message Brokers receive messages from the source or producer applications, and forward them target, or consumer applications.

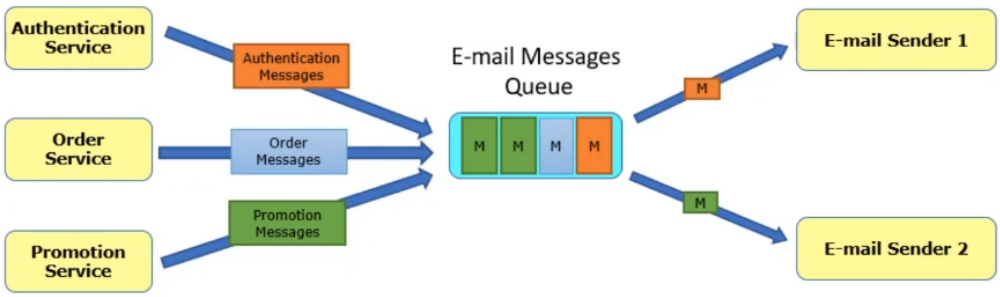


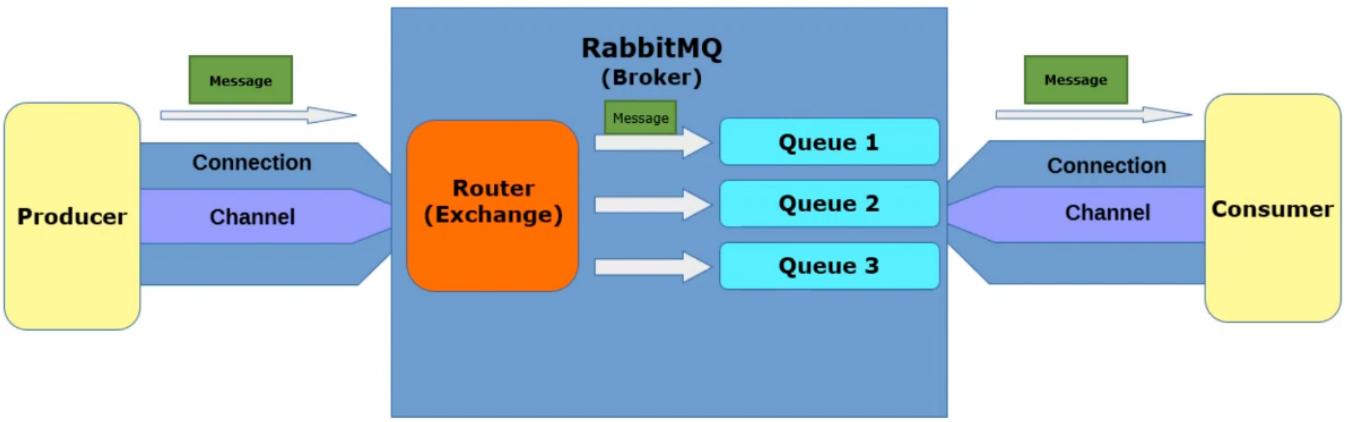
Advantage of queue:

- Decouples publishers and consumers.
- Provides a way for reliable, asynchronous task processing and communication.
- Provides a way for implementing horizontally scalable architectures.
- More efficient than data store polling.

E-commerce website needs to send e-mails to its customers asynchronously for various reasons.

- ✓ To validate customers e-mail address
- ✓ To reset password
- ✓ To inform that an order has been created successfully,
- ✓ To inform that the status of an order has been changed,
- ✓ To inform about the current or upcoming promotions

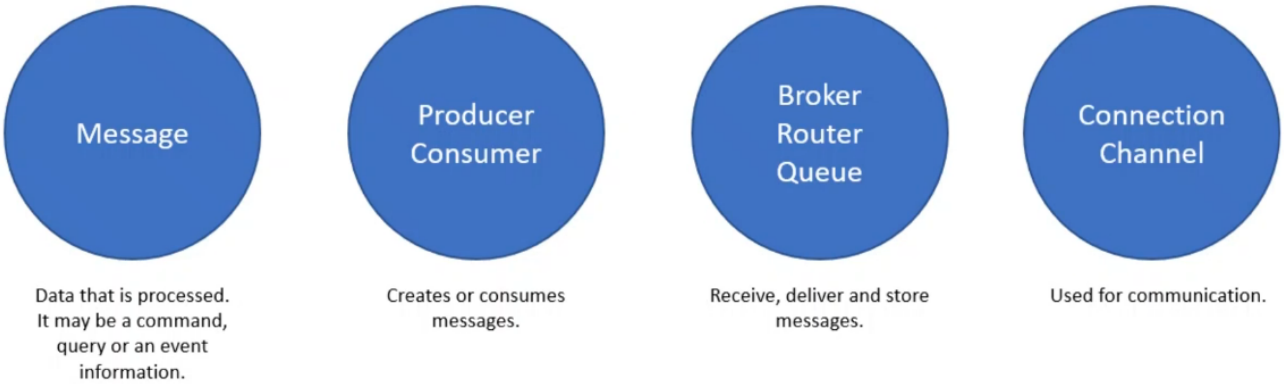




2. Install RabitMQ

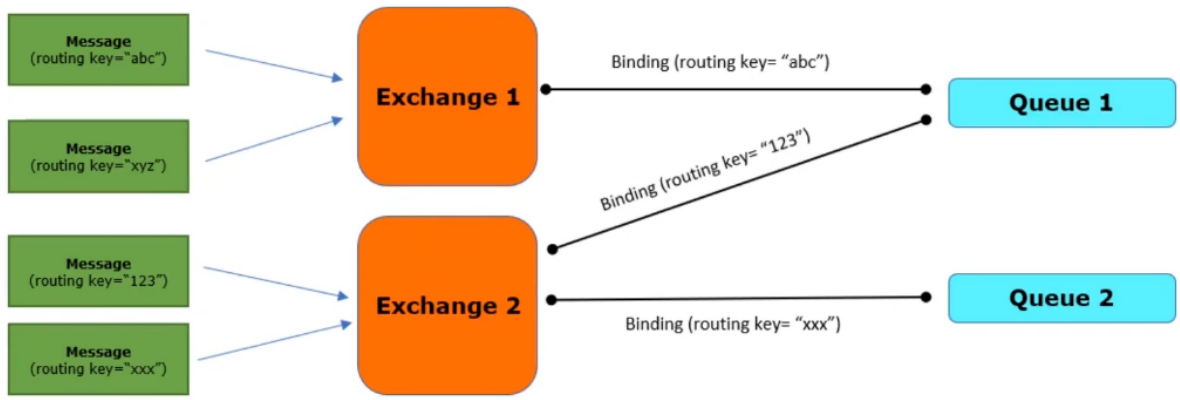
```
docker run -d --name some-rabbit -p 5672:5672 -p 5673:5673 -p 15672:15672 rabbitmq:3-management
```

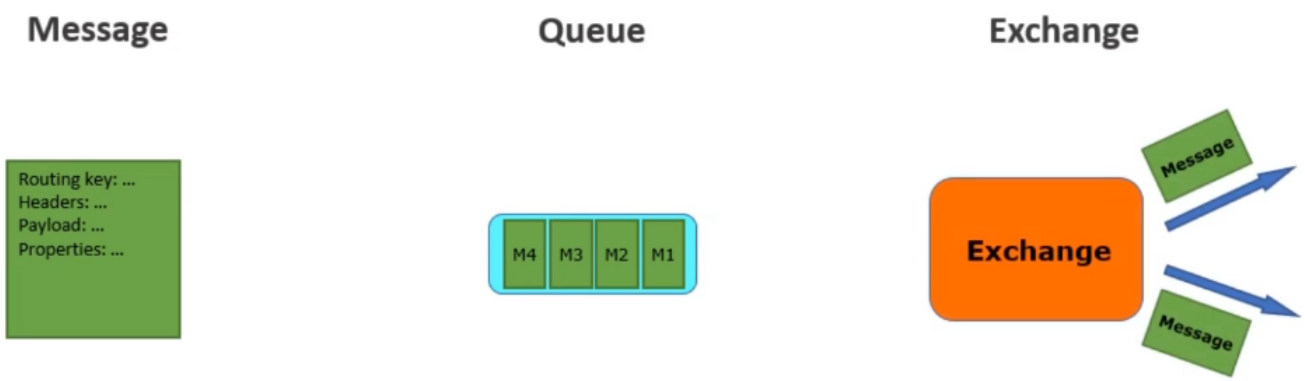
3. RabitMQ Core



- Producer
- Consumer
- Message: include two parts: routing + payload
- Queue

Binding: Defines the relationship between an exchange and a queue. Binding definition may contain arguments like “routing key” and “headers” that are used to filter messages that will be sent to the bound queue.





Components of a messages:

- **Routing Key:** Single or multiple words that are used when distributing a message to the queues.
- **Headers:** Collection of key-value pairs that are also used for routing messages and passing additional information.
- **Payload:** Actual data that a message carries.
- **Publishing Timestamp:** Optional timestamp provided by the publisher.
- **Expiration:** Life time for the message in a queue. After this time, the message considered to be "dead" which means the message is undeliverable. Unit is milliseconds.
- **Delivery Mode:** It can "persistent" or "transient". Persistent messages are written to disk and if RabbitMQ service restarts, they are re-loaded whereas transient messages are lost.
- **Priority:** Priority of the message, between 0-255.
- **Message Id:** Optional unique message id set by the publisher, to distinguish a message.
- **Correlation Id:** Optional id for matching a request and a response in remote procedure call (RPC) scenarios.

Components of a queue

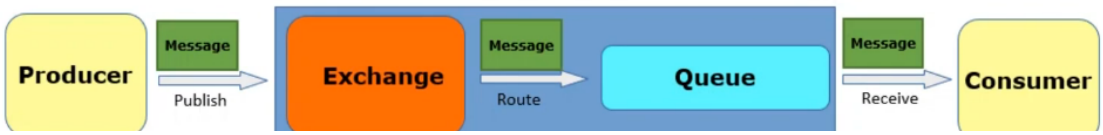
- **Name:** Unique queue name, max 255 characters UTF-8 string.
- **Durable:** Whether to preserve or delete this queue when RabbitMQ restarts.
- **Auto Delete:** Whether to delete this queue if no one is subscribed to it.
- **Exclusive:** Used only by one connection and deleted when the connection is closed.
- **Max Length:** Maximum number of waiting messages in a queue. Overflow behavior can be set as drop the oldest message or reject the new messages.
- **Max Priority:** Maximum number of priority value that this queue supports.(Up to 255)
- **Message TTL:** Life time for each message that is added to this queue. If both message and queue has a TTL value, the lowest one will be chosen.
- **Dead-letter Exchange:** Name of the exchange that expired or dropped messages will be automatically sent.
- **Binding Configurations:** Associations between queues and exchanges. A queue must be bound to an exchange, in order to receive messages from it.

Component of an exchange:

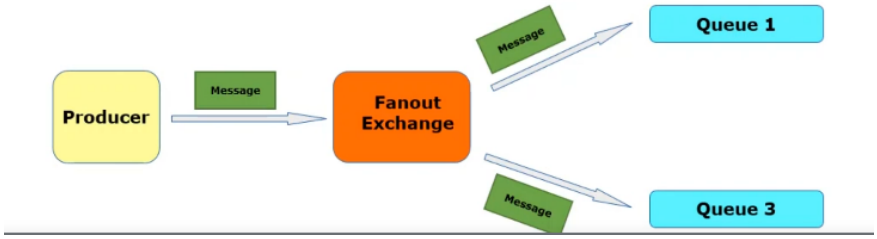
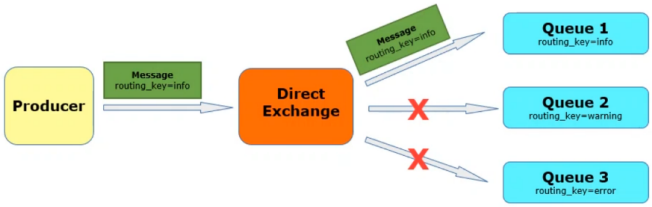
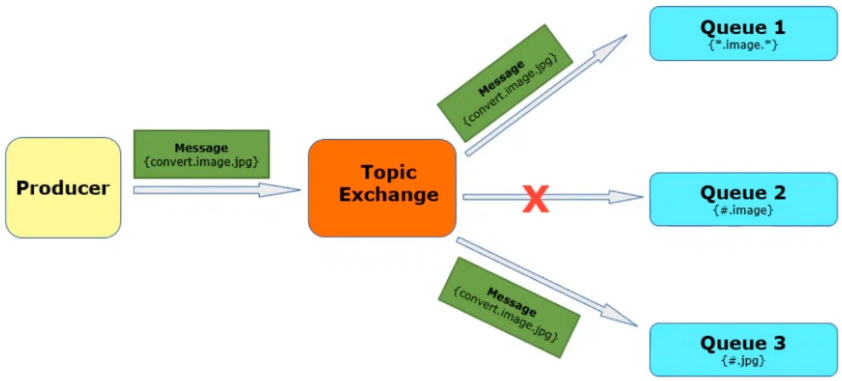
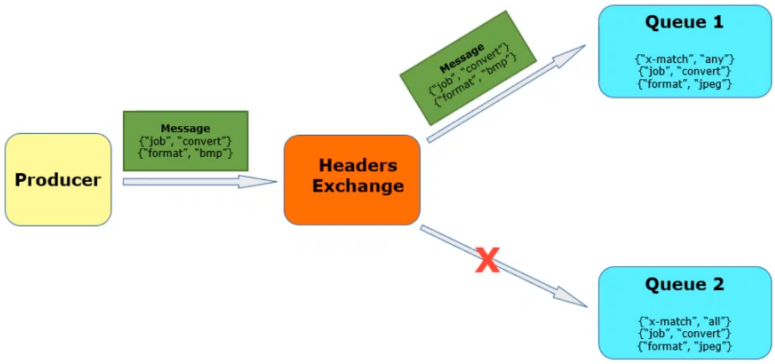
- **Name:** Unique name of the exchange.
- **Type:** Type of the exchange. It can be, “fanout”, “direct”, “topic” or “headers”.
- **Durability:** Same as queue durability. Durable exchanges survive after a service restart.
- **Auto Delete:** If “true”, exchange will be deleted when there is no bound queue left.
- **Internal:** Internal exchanges can only receive messages from other exchanges.
- **Alternate Exchange:** The name of the exchange that unroutable messages will be sent.
- **Other Arguments (x-arguments):** Other named arguments or settings that can be provided when creating an exchange. Their names start with “x-”, so these arguments are also called “x-arguments”.

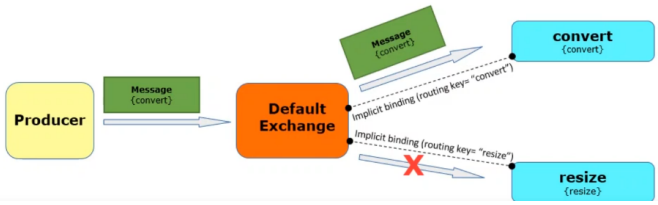
Components of Exchange:

- Exchanges are the message router elements of RabbitMQ.
- Producers does not send messages directly to queues, they send them to exchanges.
- Queues are bound to one or more exchanges with a binding definition or configuration.
- Exchanges receive messages from producers and route them to zero or more queues which are bound to them.
- Exchanges can route a message only to the queues that are bound to them.
- There are four exchange types;
  - Fanout
  - Direct
  - Topic
  - Headers
- There is at least one exchange in a RabbitMQ system. This predefined exchange is called “default exchange” and its type is “direct”. Every newly created queue is implicitly bound to this exchange.

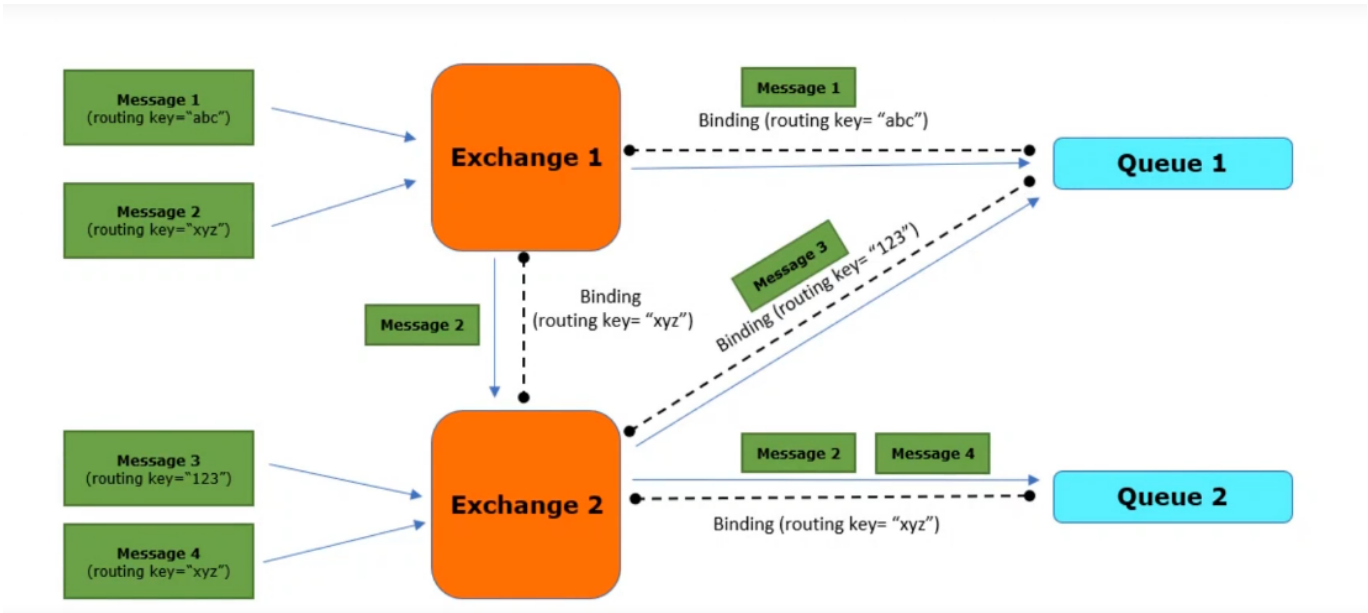


Types of exchange:

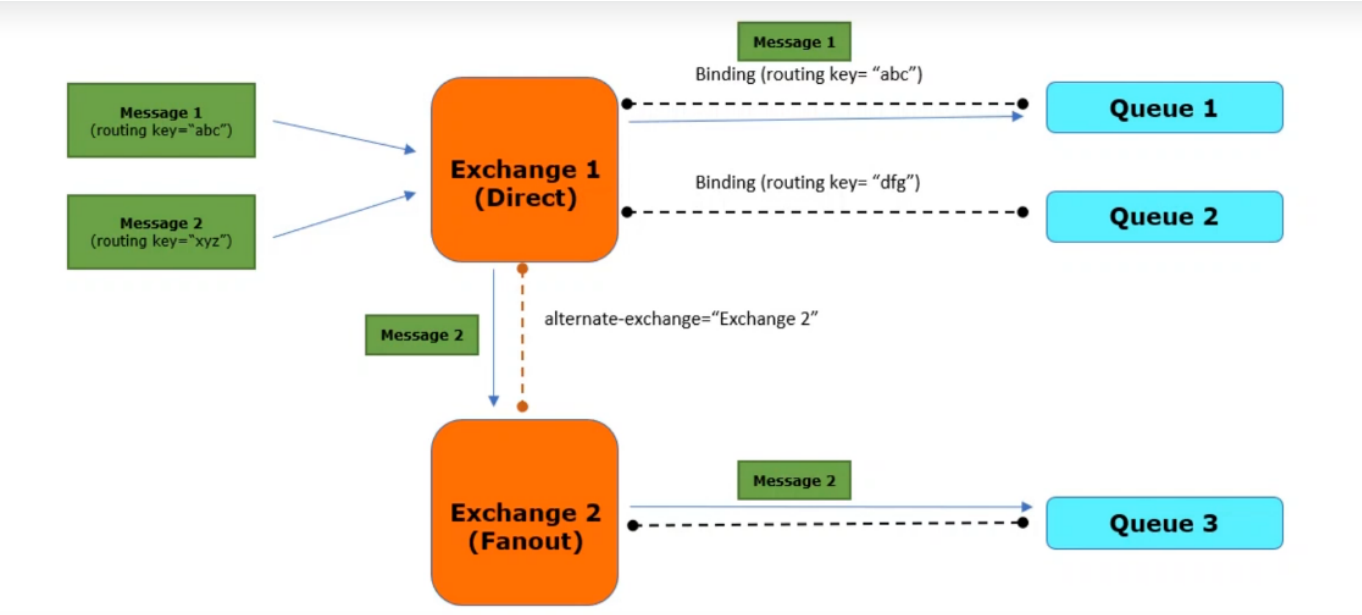
Type	
Fantout: send to all queue	
Direct exchange: routing queue of message and queue must exactly the same.	▪ Routes messages to the queues based on the "routing key" specified in binding definition. ▪ In order to send a message to a queue, routing key on the message and the routing key of the bound queue must be exactly the same. 
Topic exchange: base on pattern of routing key.	
Header exchange	

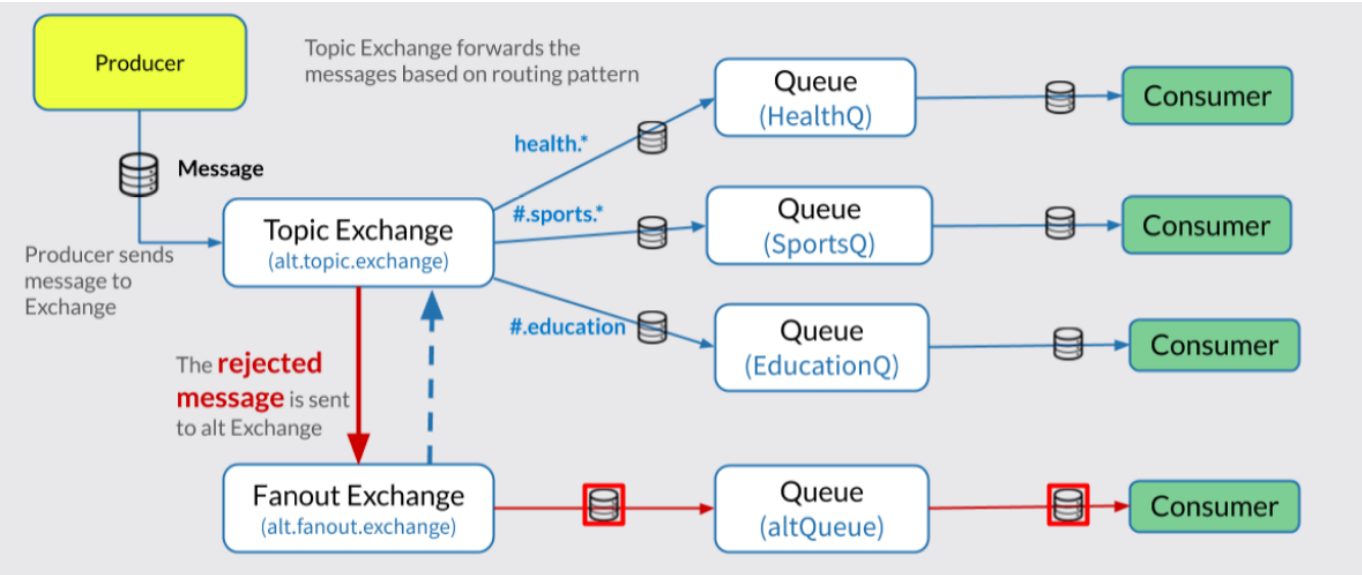
Type	
Default exchange	▪ When a new queue is created on a RabbitMQ system, it is implicitly bound to a system exchange called "default exchange", with a routing key which is the same as the queue name. ▪ Default exchange has no name (empty string). ▪ The type of default exchange is "direct". ▪ When sending a message, if exchange name is left empty, it is handled by the “default exchange”. 

We can bind exchange to another exchange:



Alternative exchange: exchange do not apply to any queue



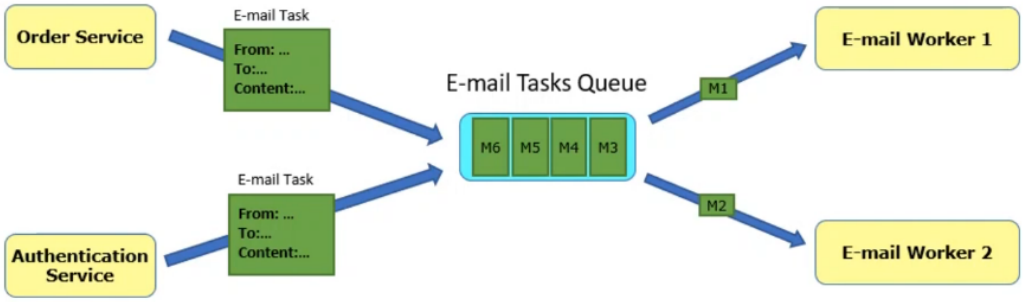


Distributed worker queue pattern

- Work queues are used to distribute tasks among multiple workers.
- Producers add tasks to a queue and these tasks are distributed to multiple worker applications.
- Pull or push models can be used to distribute tasks among the workers.

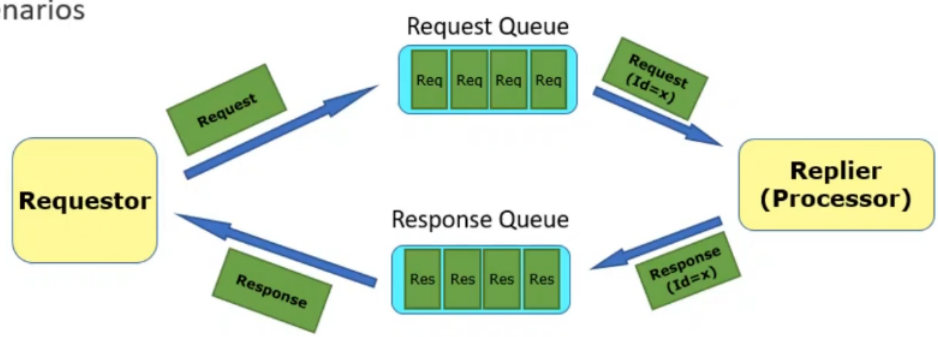
  Pull Model: Workers get a message from the queue when they are available to perform a task.

  Push Model: Message broker system sends (pushes) messages to the available workers automatically.
- Sample Scenario: Sending e-mails.



Request and reply Pattern

- Request-reply or request-response pattern is used when the publisher of the message, which is called requestor, needs to get the response for its message.
- The request message mostly contains a query or a command.
- Using request-reply pattern, remote procedure call (RPC) scenarios can also be implemented.
- In request-reply patterns, there are at least two queues;
  - 1 for the requests
  - 1 for the replies or responses. this queue is also named as the callback queue for RPC scenarios



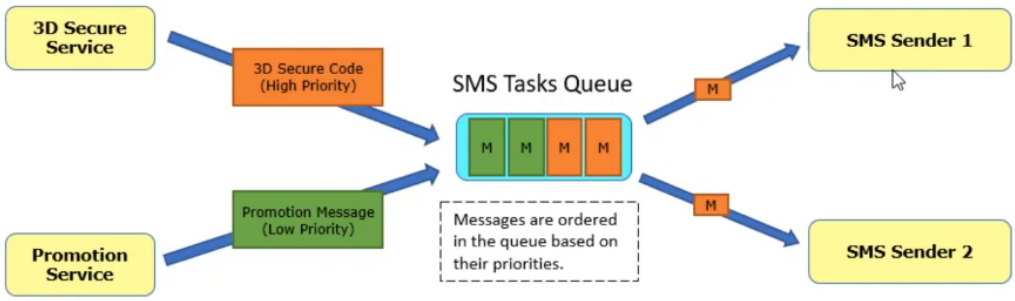
Priority queue:

- All the messages or tasks may not have the same urgency level.
- Some of them may be very urgent while others may be processed if there is no other message.

Sample: Bank sends promotional, and 3D Secure confirmation messages to its customers.

✓ Promotional messages have low priority. They may be sent any time.

✓ 3D Secure confirmation code messages expire in a few minutes and have high priority. They must be sent as soon as possible.



- Configuring RabbitMQ queues and channels for message priorities;
  - ✓ Set the Max Priority value of the queue by setting the "x-max-priority" argument. It can be 0 to 255. 0 means queue does not support priorities. Values between 1 and 10 are recommended.
  - ✓ Set the priority of the message using the "basicProperties.Priority".
  - ✓ Messages must stay in the queue for a while, to be ordered properly.
  - ✓ Use "channel.BasicQos(...)" to set the "Prefetch Count" to 1.

    This will configure a worker's channel, not to send a new message until the worker finishes (acknowledges) the last message.
  - ✓ Set "Auto Ack" value to "false" while subscribing to a queue.
  - ✓ If you don't configure the channel for not sending a new message until the worker acknowledges the last one, every incoming message will be immediately sent to a worker.

    There will be no time to reorder them in the queue.

4. RabbitMQ client

Messages:

```
public class Message {

    private final MessageProperties messageProperties;

    private final byte[] body;

    public Message(byte[] body, MessageProperties messageProperties) {
        this.body = body;
        this.messageProperties = messageProperties;
    }

    public byte[] getBody() {
        return this.body;
    }

    public MessageProperties getMessageProperties() {
        return this.messageProperties;
    }
}
```

Exchange

```
public interface Exchange {

    String getName();

    String getExchangeType();

    boolean isDurable();

    boolean isAutoDelete();

    Map<String, Object> getArguments();

}
```

Queue

```java
public class Queue  {

    private final String name;

    private volatile boolean durable;

    private volatile boolean exclusive;

    private volatile boolean autoDelete;

    private volatile Map<String, Object> arguments;

    /**
     * The queue is durable, non-exclusive and non auto-delete.
     *
     * @param name the name of the queue.
     */
    public Queue(String name) {
        this(name, true, false, false);
    }

    // Getters and Setters omitted for brevity

}
```

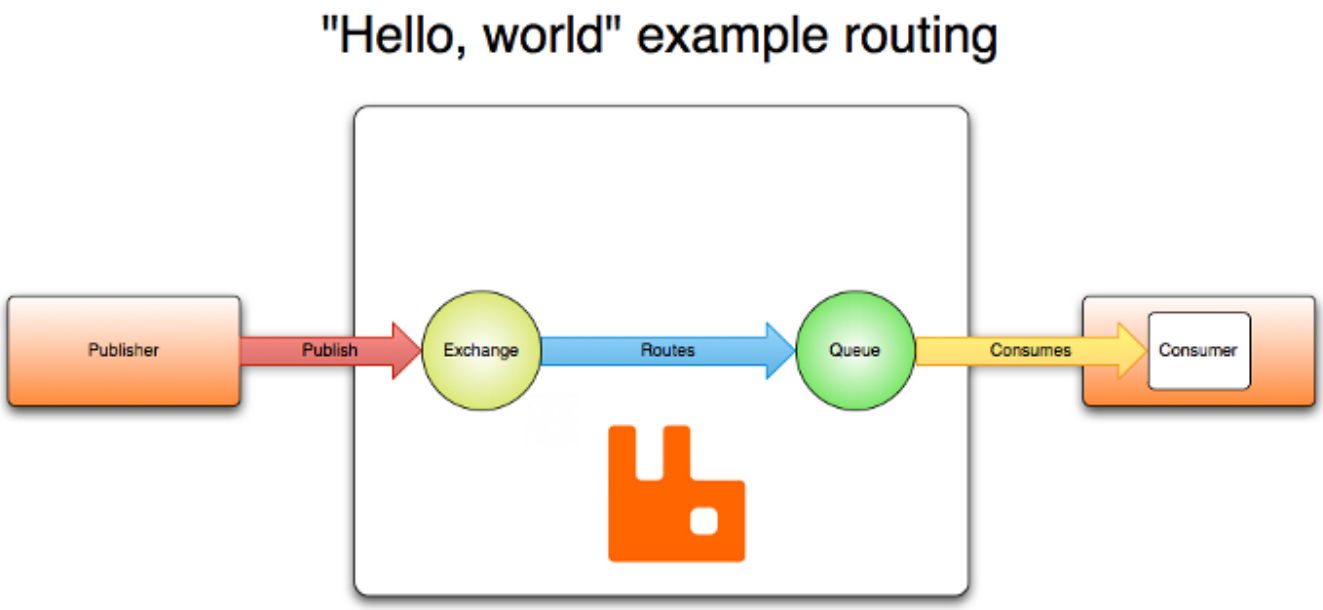
Connection and Channel Lifespan

Client connections are meant to be long-lived. The underlying protocol is designed and optimized for long running connections. That means that opening a new connection per operation, e.g. a message published, is unnecessary and strongly discouraged as it will introduce a lot of network roundtrips and overhead.

Channels are also meant to be long-lived but since many recoverable protocol errors will result in channel closure, channel lifespan could be shorter than that of its connection. Closing and opening new channels per operation is usually unnecessary but can be appropriate. When in doubt, consider reusing channels first.

Channel-level exceptions such as attempts to consume from a queue that does not exist will result in channel closure. A closed channel can no longer be used and will not receive any more events from the server (such as message deliveries). Channel-level exceptions will be logged by RabbitMQ and will initiate a shutdown sequence for the channel (see below).

AMQP:



Acknowledgement:

There are two acknowledgement modes:

- After broker sends a message to an application (using either basic.deliver or basic.get-ok method).
- After the application sends back an acknowledgement (using the basic.ack method).

The former choice is called the automatic acknowledgement model, while the latter is called the explicit acknowledgement model. With the explicit model the application chooses when it is time to send an acknowledgement. It can be right after receiving a message, or after persisting it to a data store before processing, or after fully processing the message (for example, successfully fetching a Web page, processing and storing it into some persistent data store).

Connection:

AMQP 0-9-1 connections are typically long-lived. AMQP 0-9-1 is an application level protocol that uses TCP for reliable delivery. Connections use authentication and can be protected using TLS. When an application no longer needs to be connected to the server, it should gracefully close its AMQP 0-9-1 connection instead of abruptly closing the underlying TCP connection.

Channels:

Some applications need multiple connections to the broker. However, it is undesirable to keep many TCP connections open at the same time because doing so consumes system resources and makes it more difficult to configure firewalls. AMQP 0-9-1 connections are multiplexed with *channels* that can be thought of as "lightweight connections that share a single TCP connection".

Spring AMQP:

• PooledChannelConnectionFactory • ThreadChannelConnectionFactory • CachingConnectionFactory

5 RabbitMQ in Action

5.1 Unsderstanding messaging

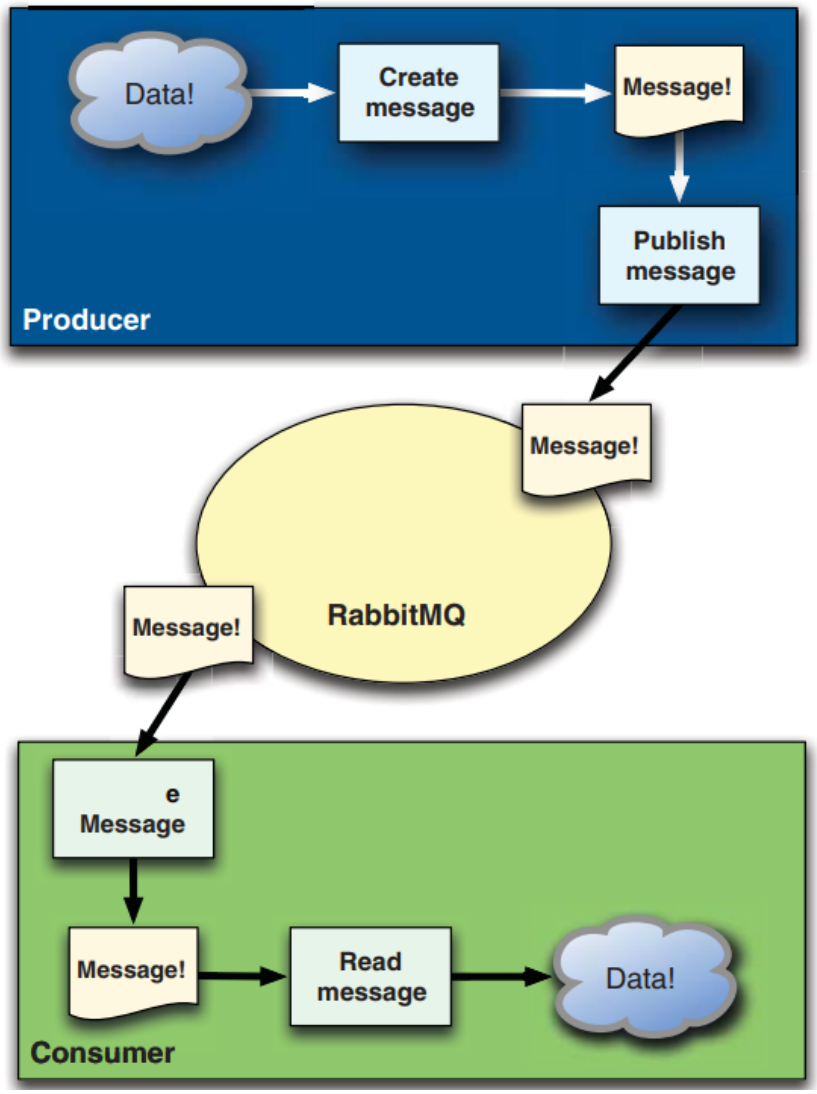
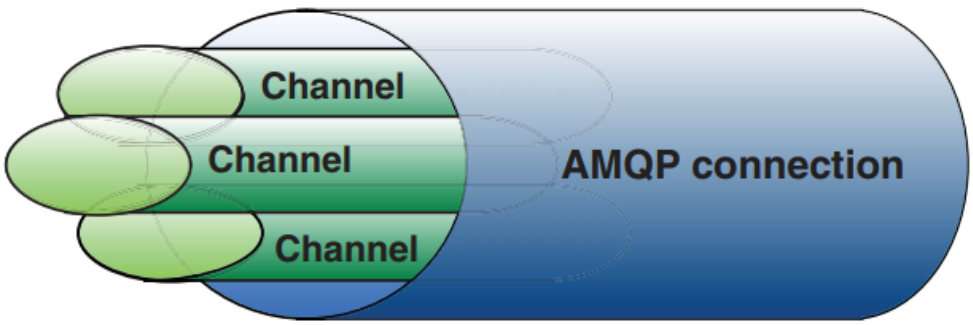


Figure 2.1 Message flow from producers to consumers

Before you consume from or publish to Rabbit, you first have to connect to it. By connecting, you're creating a TCP connection between your app and the Rabbit broker. Once the TCP connection is open (and you're authenticated), your app then creates an AMQP channel. This channel is a virtual connection inside the "real" TCP connection, and it's over the channel that you issue AMQP commands. Every channel has a unique ID assigned to it (your AMQP library of choice will handle remembering the ID for you). Whether you're publishing a message, subscribing to a queue, or receiving a message, it's all done over a channel. Why do we need channels, you might ask? Why not just issue AMQP commands directly over the TCP connection? The main reason is because setting up and tearing down TCP sessions is expensive for an operating system. Let's say your app consumes messages from a queue and spins threads up or down based on service demand. If all you had were TCP connections,

- Channel is a virtual connection over one TCP connection, mutilple channel will execute in multiple plexing in over one TCP connection
- If all you had were TCP connections, each thread would need its own connection to Rabbit, which could mean hundreds of connections per second during high load periods. Not only would this be a massive waste of TCP connections, but an operating system can only build so many per second. So you could quickly hit a performance wall.



With multiple channels, many threads can share the same connection simultaneously, meaning that responding to requests doesn't block you from consuming new requests and you don't waste a TCP connection for each thread. Sometimes you may choose to use only one channel, but with AMQP you have the flexibility to use as many channels as your app needs without the overhead of multiple TCP connections.

Queue:

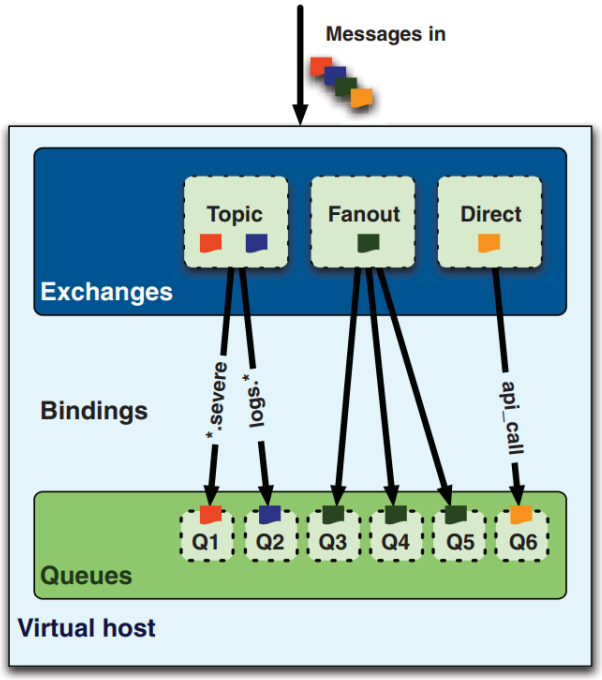


Figure 2.3 AMQP stack: exchanges, bindings, and queues

Get message:

- Using basic.consume: consume channel and receive message
- Using basic.get: pulling message from queue

Acknowledge when receive:

- By using basic.ack
- Or by setting auto ack when consuming

Message have no ack:

- RabbitMQ automatically requeue message to another consume
- Using basic.reject, it allow queue keep sending to another consumer (requeue.reject = true) if requeue.reject = false, it will reject message.

Declaring queue:

- queue.declare:
- If declare a queue exactly created queue, rabbit mq do nothing

Exchanges and binding:

accomplish—with a broker that only allows you to publish directly to queues.

As you've seen, the broker will route messages from exchanges to queues based on routing keys, but how does it handle delivery to multiple queues? Here come into play the different types of exchanges provided by the protocol. There are four: direct, fanout, topic, and headers. Each implements a different routing algorithm. We'll go over all of them except the headers exchange, which allows you to match against a header in the AMQP message instead of the routing key. Other than that, it operates identically to the direct exchange but with much worse performance. As a result, it doesn't provide much real-world benefit and is almost never used. Let's see each of the other exchange types in detail.

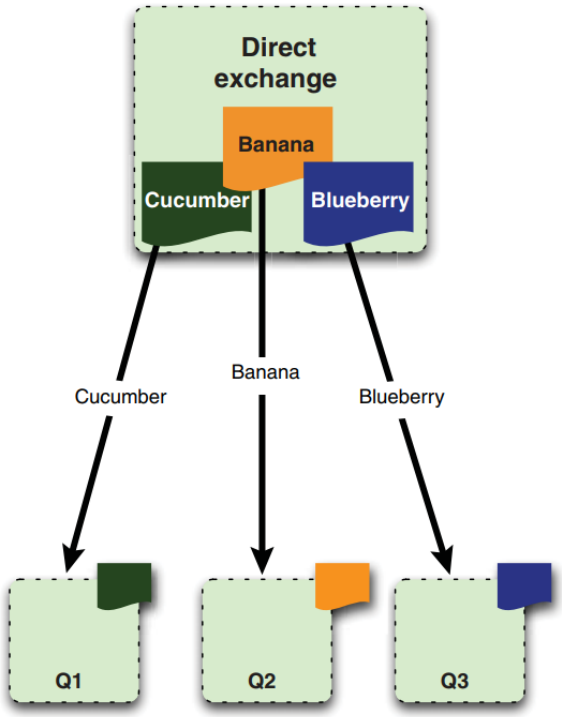
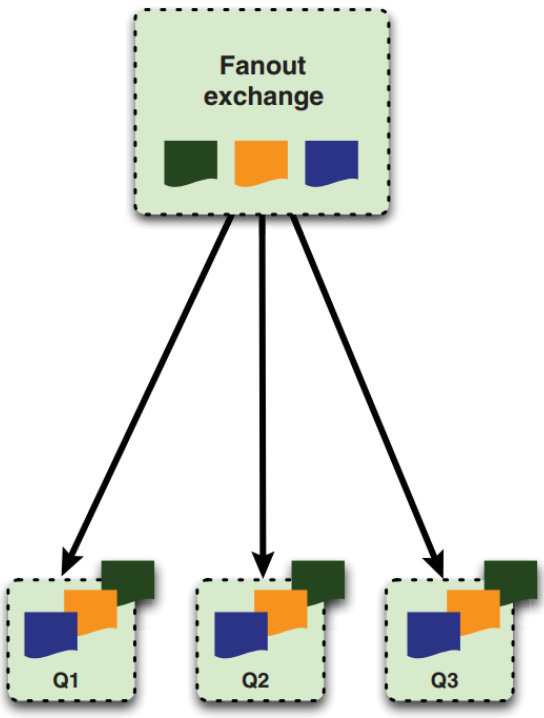
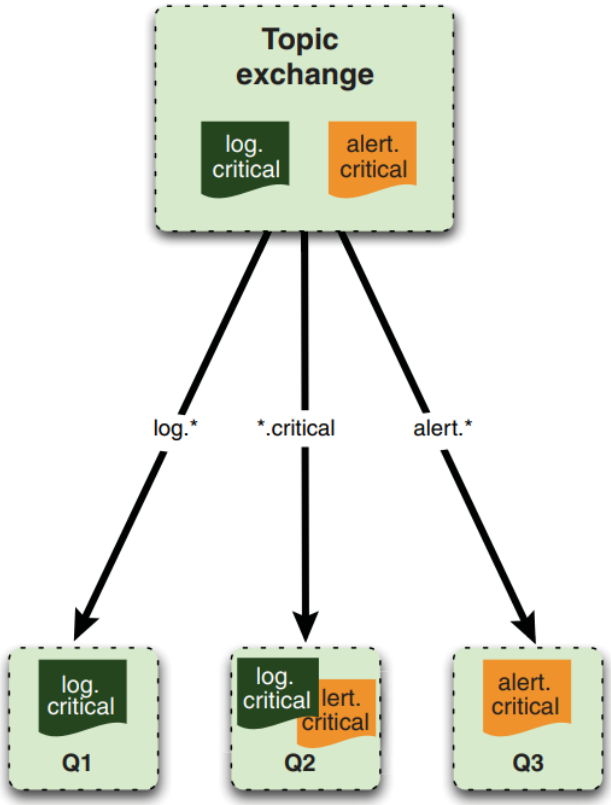


Figure 2.4 **Direct exchange message flow**





Durability of queue and exchange:

- Control by **durable** parameter

Durability of message:

- Control by delivery mode = 2

For a message that's in flight inside Rabbit to survive a crash, the message must:

- Have its delivery mode option set to 2 (persistent)
- Be published into a durable exchange
- Arrive in a durable queue

The way that RabbitMQ ensures persistent messages survive a restart is by writing them to the disk inside of a persistency log file. When you publish a persistent message to a durable exchange, Rabbit won't send the response until the message is committed to the log file. Keep in mind, though, that if it gets routed to a nondurable queue after that, it's automatically removed from the persistency log and won't survive a restart. When you use persistent messages it's crucial that you make sure *all* three elements required for a message to persist are in place (we can't stress this enough).

You might be thinking that you should use persistent messaging for all of your messages. You could do that, but you'd pay a price for ensuring your messages survive Rabbit restarts: performance. The act of writing messages to disk is much slower than

A better way to ensure deliery message is: **puplisher confirm**

- Enable this by setting channel to confirm mode, every message will wait for all consumer receive and publish a confirm message.
- This lets the producer know the message has been safely queued at all of its destinations.

Using publisher confirms to verify delivery

So now you know how to write a basic consumer and producer, but you might be wondering how publisher confirms and transactions fit into the mix. Let's take a look at how you can upgrade your Hello World producer to take advantage of publisher confirms to track message delivery.[3] Before we dive into updating the Hello World producer to use publisher confirms, look at figure 2.8, which is an illustration of how message IDs are assigned.

We said that every message published gets a unique ID if the channel is in confirm mode. This might make you think that `basic_publish` will suddenly start returning a message ID, but that's not how message IDs work. Since a channel can only be used by a single thread, you can be assured that all publishes using that channel are sequential. As a result, RabbitMQ makes a simple assumption: the first message published on any channel

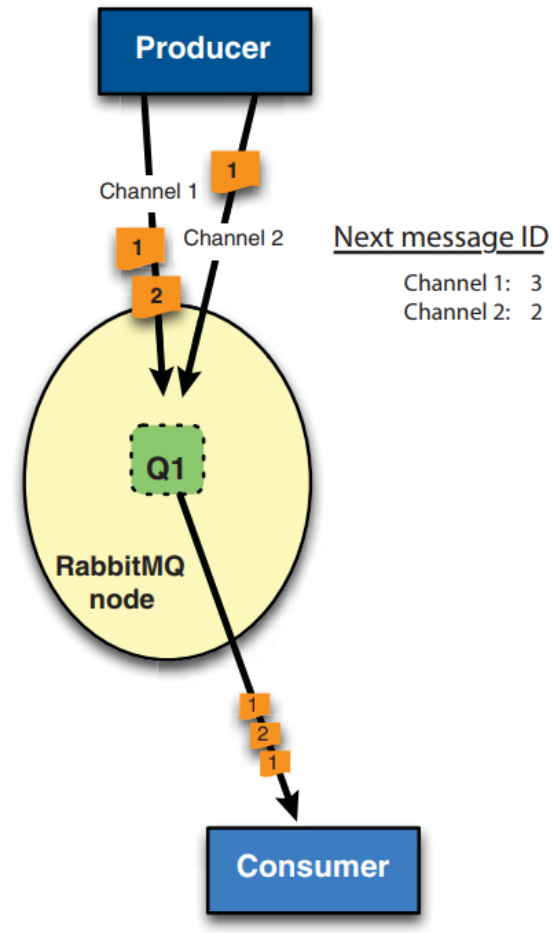


Figure 2.8 **Publisher confirm message IDs**

- Channel is single thread, message come to channel is sequential, web application keep tracking its own message ID.

5.2 RabbitMQ design and pattern

Two general types of jobs fit into this pattern:

- *Batch processing*—Work and transformations that need be completed on a large data set. This can be structured as a single job request or many jobs operating on individual parts of the data set.
- *Notifications*—A description of an event that has occurred. This can be anything from a message to be logged, to an actual alert that should be sent to another program or an administrator.

Fire and forger model

- Sending alerts

Consumer:

```
def send_mail(recipients, subject, message):
    """E-mail generator for received alerts."""
    headers = ("From: %s\r\nTo: \r\nDate: \r\n" + \
               "Subject: %s\r\n\r\n") % ("alerts@ourcompany.com",
                                         subject)

    smtp_server = smtplib.SMTP()
    smtp_server.connect("mail.ourcompany.com", 25)
    smtp_server.sendmail("alerts@ourcompany.com",
                         recipients,
                         headers + str(message))
    smtp_server.close()

#/(asc.5) Notify Processors
def critical_notify(channel, method, header, body):
    """Sends CRITICAL alerts to administrators via e-mail."""

    EMAIL_RECIPS = ["ops.team@ourcompany.com",]

    #/(asc.6) Decode our message from JSON
    message = json.loads(body)

    #/(asc.7) Transmit e-mail to SMTP server
    send_mail(EMAIL_RECIPS, "CRITICAL ALERT", message)
    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    #/(asc.8) Acknowledge the message
    channel.basic_ack(delivery_tag=method.delivery_tag)

def rate_limit_notify(channel, method, header, body):
    """Sends the message to the administrators via e-mail."""

    EMAIL_RECIPS = ["api.team@ourcompany.com",]

    #/(asc.9) Decode our message from JSON
    message = json.loads(body)

    #/(asc.10) Transmit e-mail to SMTP server
    send_mail(EMAIL_RECIPS, "RATE LIMIT ALERT!", message)

    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    #/(asc.11) Acknowledge the message
    channel.basic_ack(delivery_tag=method.delivery_tag)


if __name__ == "__main__":
    #/(asc.0) Broker settings
```

```
AMQP_SERVER = "localhost"
AMQP_USER = "alert_user"
AMQP_PASS = "alertme"
AMQP_VHOST = "/"
AMQP_EXCHANGE = "alerts"

#/(asc.1) Establish connection to broker
creds_broker = pika.PlainCredentials(AMQP_USER, AMQP_PASS)
conn_params = pika.ConnectionParameters(AMQP_SERVER,
                                        virtual_host = AMQP_VHOST,
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()

#/(asc.2) Declare the Exchange
channel.exchange_declare( exchange=AMQP_EXCHANGE,
                          type="topic",
                          auto_delete=False)

#/(asc.3) Build the queues and bindings for our topics
channel.queue_declare(queue="critical", auto_delete=False)
channel.queue_bind(queue="critical",
                   exchange="alerts",
                   routing_key="critical.*")

channel.queue_declare(queue="rate_limit", auto_delete=False)
channel.queue_bind(queue="rate_limit",
                   exchange="alerts",
                   routing_key="*.rate_limit")

#/(asc.4) Make our alert processors

channel.basic_consume( critical_notify,
                       queue="critical",
                       no_ack=False,
                       consumer_tag="critical")

channel.basic_consume( rate_limit_notify,
                       queue="rate_limit",
                       no_ack=False,
                       consumer_tag="rate_limit")

print "Ready for alerts!"
channel.start_consuming() channel.basic_ack(delivery_tag=method.delivery_tag)
```

Publisher:

```
#/(asp.0) Read in command line arguments
opt_parser = OptionParser()
opt_parser.add_option("-r",
                      "--routing-key",
                      dest="routing_key",
                      help="Routing key for message (e.g. myalert.im)")
opt_parser.add_option("-m",
                      "--message",
                      dest="message",
                      help="Message text for alert.")

args = opt_parser.parse_args()[0]

#/(asp.1) Establish connection to broker
creds_broker = pika.PlainCredentials("alert_user", "alertme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()

#/(asp.2) Publish alert message to broker
msg = json.dumps(args.message)
msg_props = pika.BasicProperties()
msg_props.content_type = "application/json"
msg_props.durable = False

channel.basic_publish(body=msg,
                      exchange="alerts",
                      properties=msg_props,
                      routing_key=args.routing_key)

print ("Sent message %s tagged with routing key '%s' to " + \
       "exchange '/'.") % (json.dumps(args.message),
                           args.routing_key)
```

- RPC call

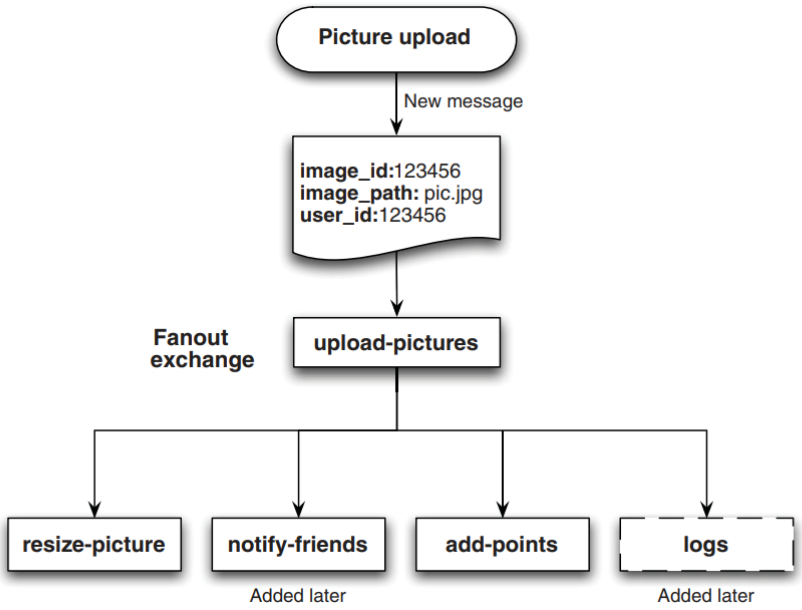
Publisher-client:

```
#/(rpcc.0) Establish connection to broker
creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

#/(rpcc.1) Issue RPC call & wait for reply
msg = json.dumps({"client_name": "RPC Client 1.0",
                  "time" : time.time()})

result = channel.queue_declare(exclusive=True, auto_delete=True)
msg_props = pika.BasicProperties()
msg_props.reply_to=result.method.queue

channel.basic_publish(body=msg,
                      exchange="rpc",
                      properties=msg_props,
                      routing_key="ping")

print "Sent 'ping' RPC call. Waiting for reply..."

def reply_callback(channel, method, header, body):
    """Receives RPC server replies."""
    print "RPC Reply --- " + body
    channel.stop_consuming()



channel.basic_consume(reply_callback,
                      queue=result.method.queue,
                      consumer_tag=result.method.queue)

channel.start_consuming()
```

Consumer, server:

```
#/(apiserver.0) Establish connection to broker
creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

#/(apiserver.1) Declare Exchange & "ping" Call Queue
channel.exchange_declare(exchange="rpc",
                         type="direct",
                         auto_delete=False)
channel.queue_declare(queue="ping", auto_delete=False)
channel.queue_bind(queue="ping",
                   exchange="rpc",
                   routing_key="ping")

#/(apiserver.2) Wait for RPC calls and reply
def api_ping(channel, method, header, body):
    """'ping' API call."""
    channel.basic_ack(delivery_tag=method.delivery_tag)
    msg_dict = json.loads(body)
    print "Received API call...replying..."
    channel.basic_publish(body="Pong!" + str(msg_dict["time"]),
                          exchange="",
                          routing_key=header.reply_to)

channel.basic_consume(api_ping,
                      queue="ping",
                      consumer_tag="ping")

print "Waiting for RPC calls..."
channel.start_consuming()
```

Parallel processing:



RPC over RabbitMQ and waiting for answers

RabbitMQ can be use to tranfer RPC call

On every AMQP message header is a field called reply_to. Within this field the producer of a message can specify the queue name they'll be listening to for a reply.

The receiving RPC server can then inspect this reply_to field and create a new message containing the response with this queue name as the routing key

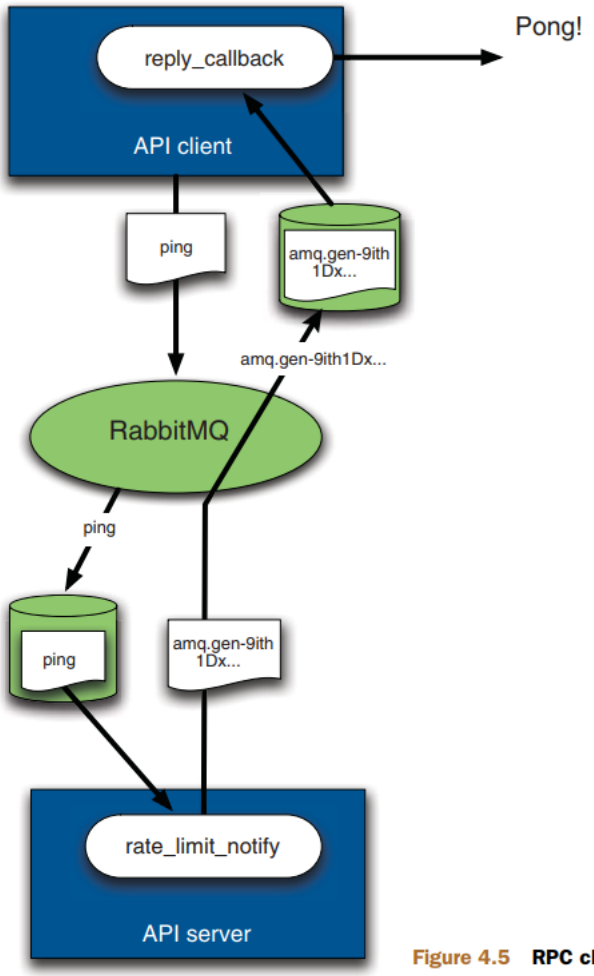
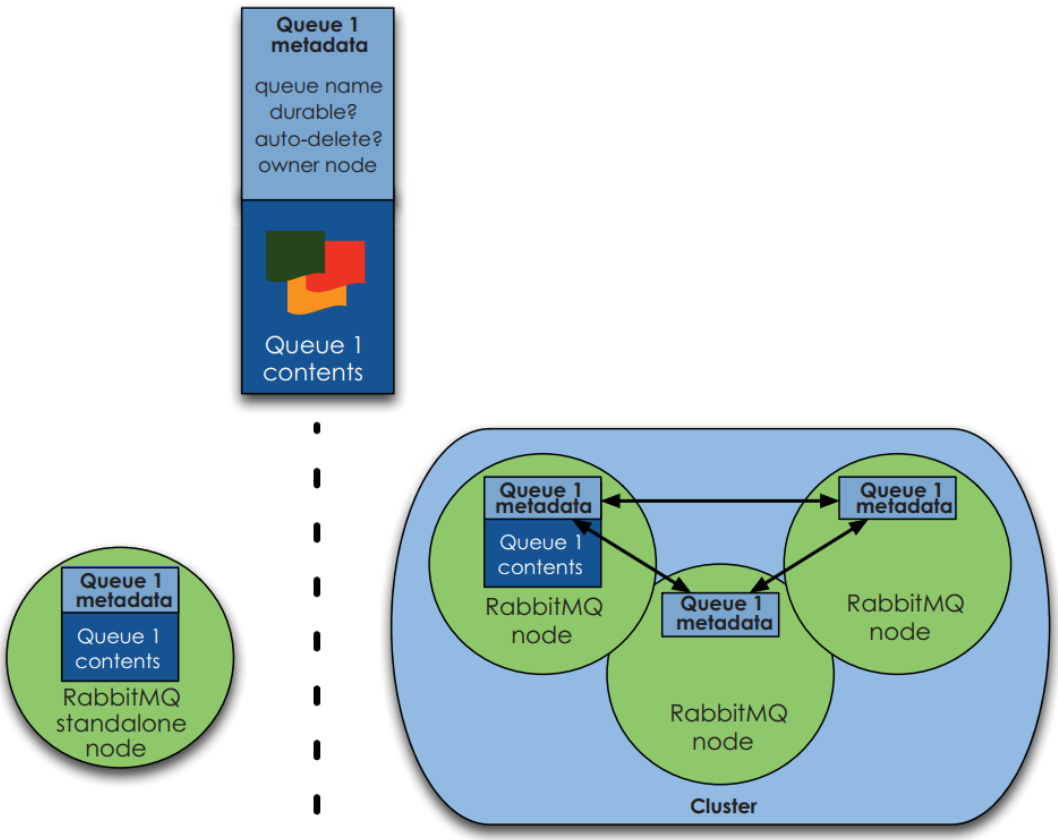


Figure 4.5 **RPC client and server flow**

5.3 RabbitMQ clustering

Architecture of a cluster, cluster of RabbitMQ keep tracking data of:

- Queue metadata
- Exchange metadata
- Binding metadata
- Vhost metadata



In order to avoid message lost, using publisher confirm to checking which message is lost.

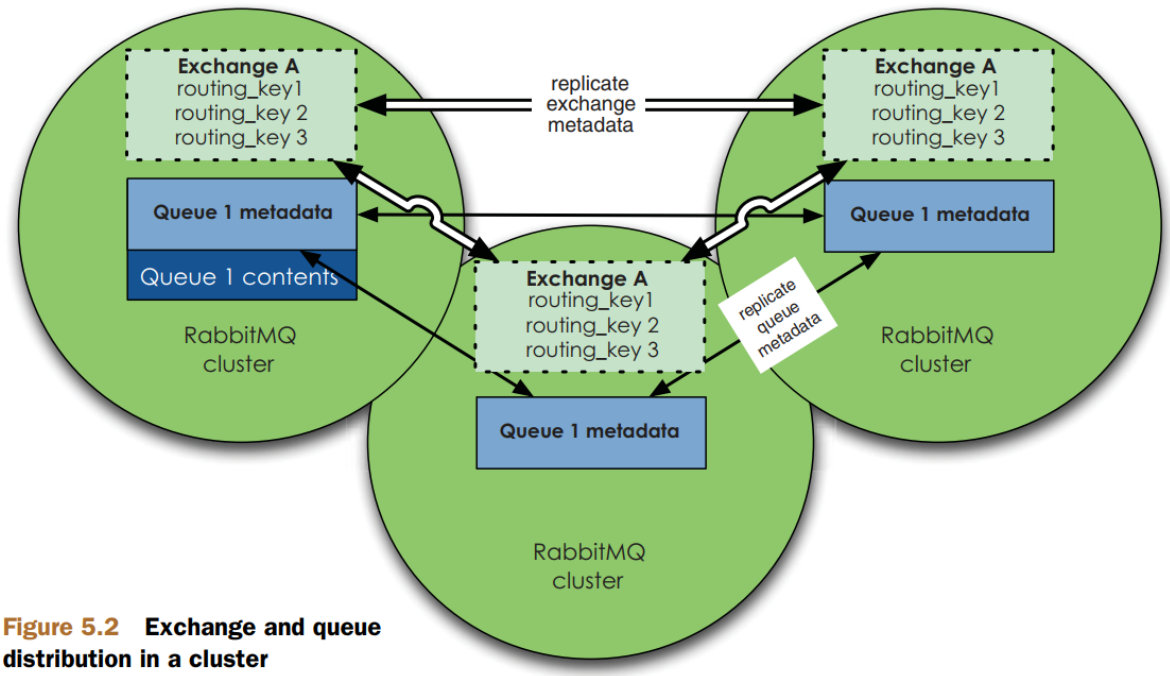


Figure 5.2 Exchange and queue distribution in a cluster

blocks until the message is routed to a queue, or to use *publisher confirms* to keep track of which messages are still unconfirmed when the connection to a node dies. Both

Disk vs RAM mode

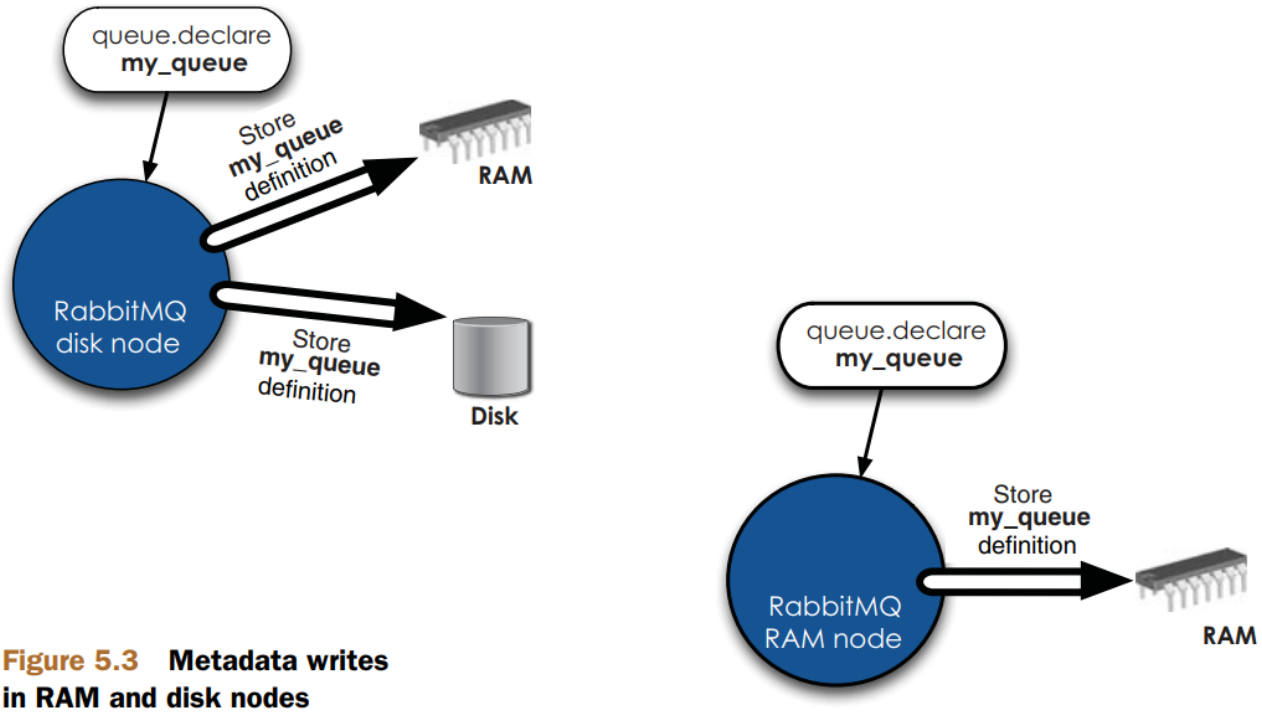


Figure 5.3 Metadata writes in RAM and disk nodes

what if you're doing heavy RPC? If every RPC client is creating and destroying hundreds of reply queues every second, you start to see where disk nodes can kill performance. So how do you balance the performance of RAM nodes against the need to have some disk nodes that allow the cluster configuration to survive cluster restarts?

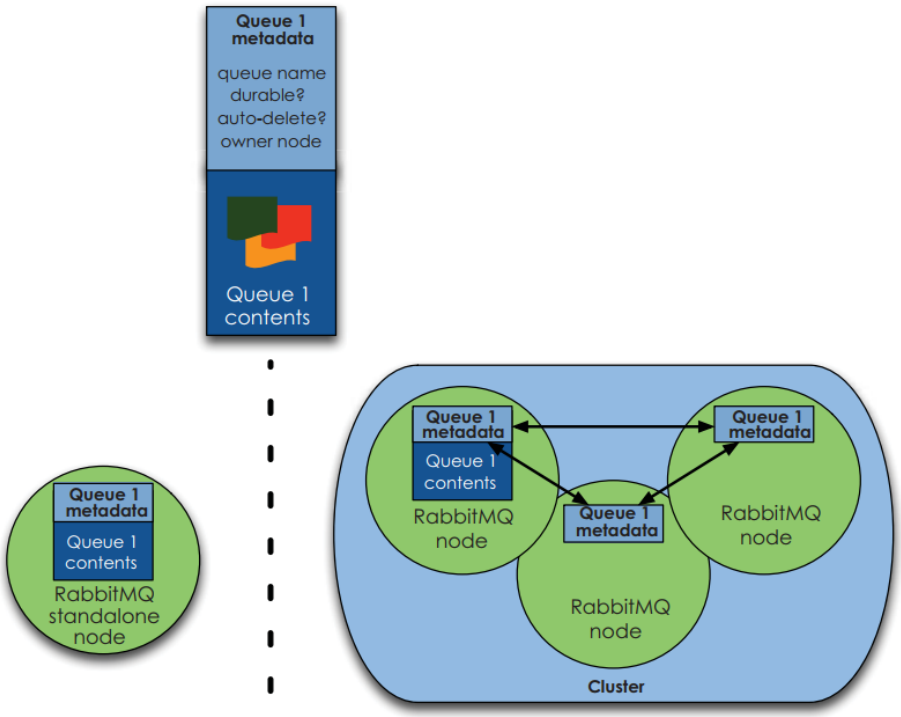
RabbitMQ only requires that one node in a cluster be a disk node. Every other node can be a RAM node. Keep in mind that when nodes join or leave a cluster, they need to be able to notify at least one disk node of the change. If you only have one disk node and that node happens to be down, your cluster can continue to route messages but you can't do any of the following:

- Create queues
- Create exchanges
- Create bindings
- Add users
- Change permissions
- Add or remove cluster nodes

In other words, your cluster can keep running if its sole disk node is down but you'll be unable to change anything until you can restore that node to the cluster. The solution is to make two disk nodes in your cluster so at least one of them is available to persist metadata changes at any given time. The only operation *all* of the disk nodes need to be online for is adding or removing cluster nodes. When RAM nodes restart, they connect to the disk nodes they're preconfigured with to download the current copy of the cluster's metadata. If you only tell a RAM node about one of your two disk nodes and that disk node is down when the RAM node's power cable gets knocked loose, the RAM node won't be able to find the cluster when it reboots. So when you join RAM nodes, make sure they're told about all of your disk nodes (the only metadata RAM nodes

5.4 Reliable RabbitMQ

Queues in a cluster



- In cluster, only owner node hold the content of queue.

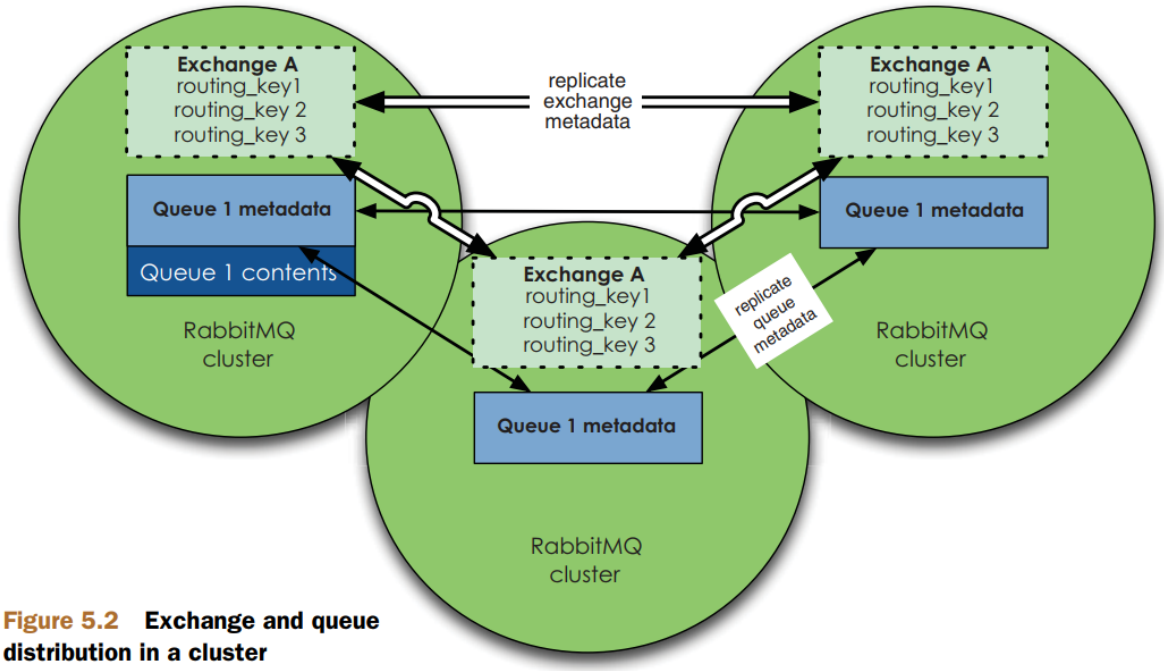


Figure 5.2 Exchange and queue distribution in a cluster

- Exchange is replicate among cluster

Ram and disk

- RabbitMQ require at least one queue is disk persitence.

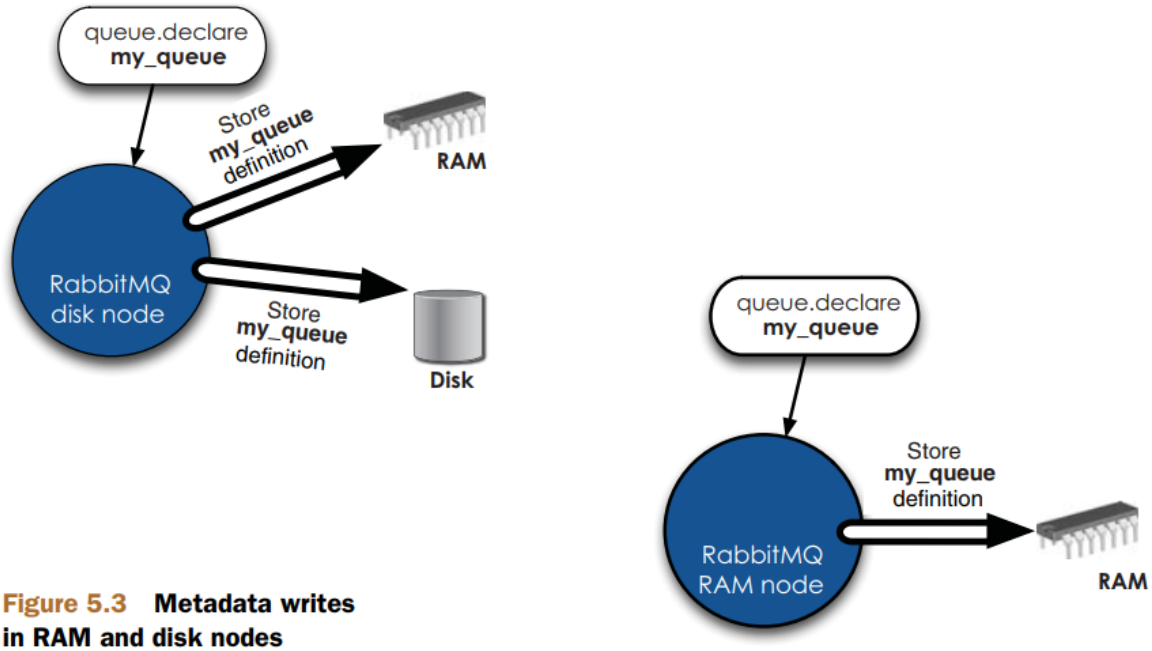
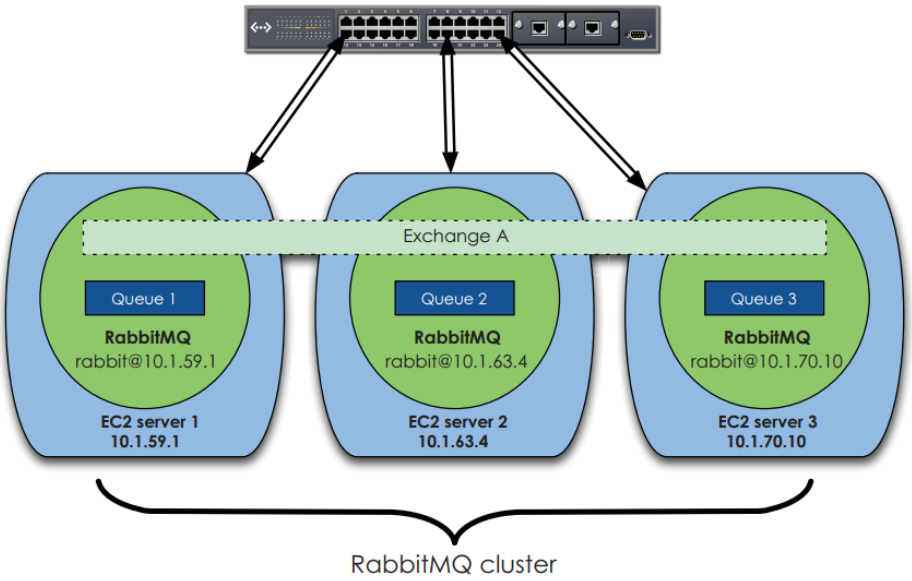


Figure 5.3 Metadata writes in RAM and disk nodes

Distributing the nodes to more machines



Upgrading cluster nodes

- First you'll need to back up the current configuration via the RabbitMQ management plugin.
- Then shut down any producers and wait for your consumers to drain all of the queues (use rabbitmqctl to watch queue statuses until all of them are empty).
- Now, shut down the nodes and unpack the new version of RabbitMQ into your existing installation directories.
- At this point, select one of the disk nodes to be your upgrader node.
- When it starts, this node will upgrade the persisted cluster data to the new version.
- Then you can start the remaining cluster disk nodes, which will acquire the upgraded cluster data.
- Finally, start the cluster RAM nodes and you'll have your cluster running the shiny new version of RabbitMQ and all of your metadata/configuration will have been retained

Mirroring queue:

- As with many aspects of AMQP, your application defines a queue as being mirrored rather than rabbitmqctl.
- x-ha-policy to the queue.declare call

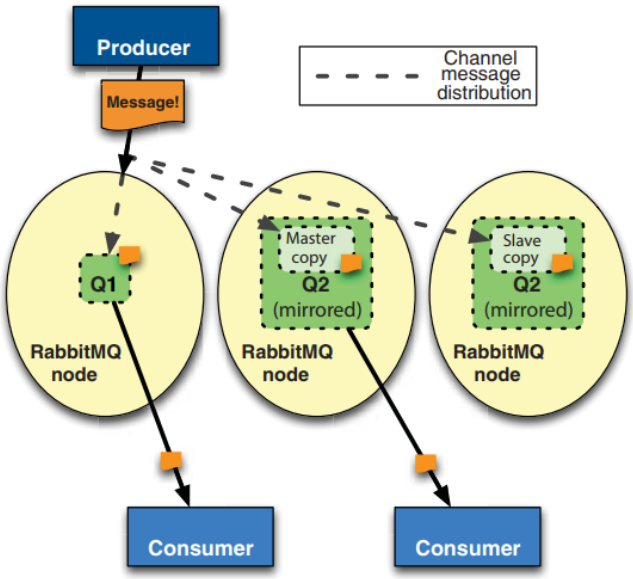
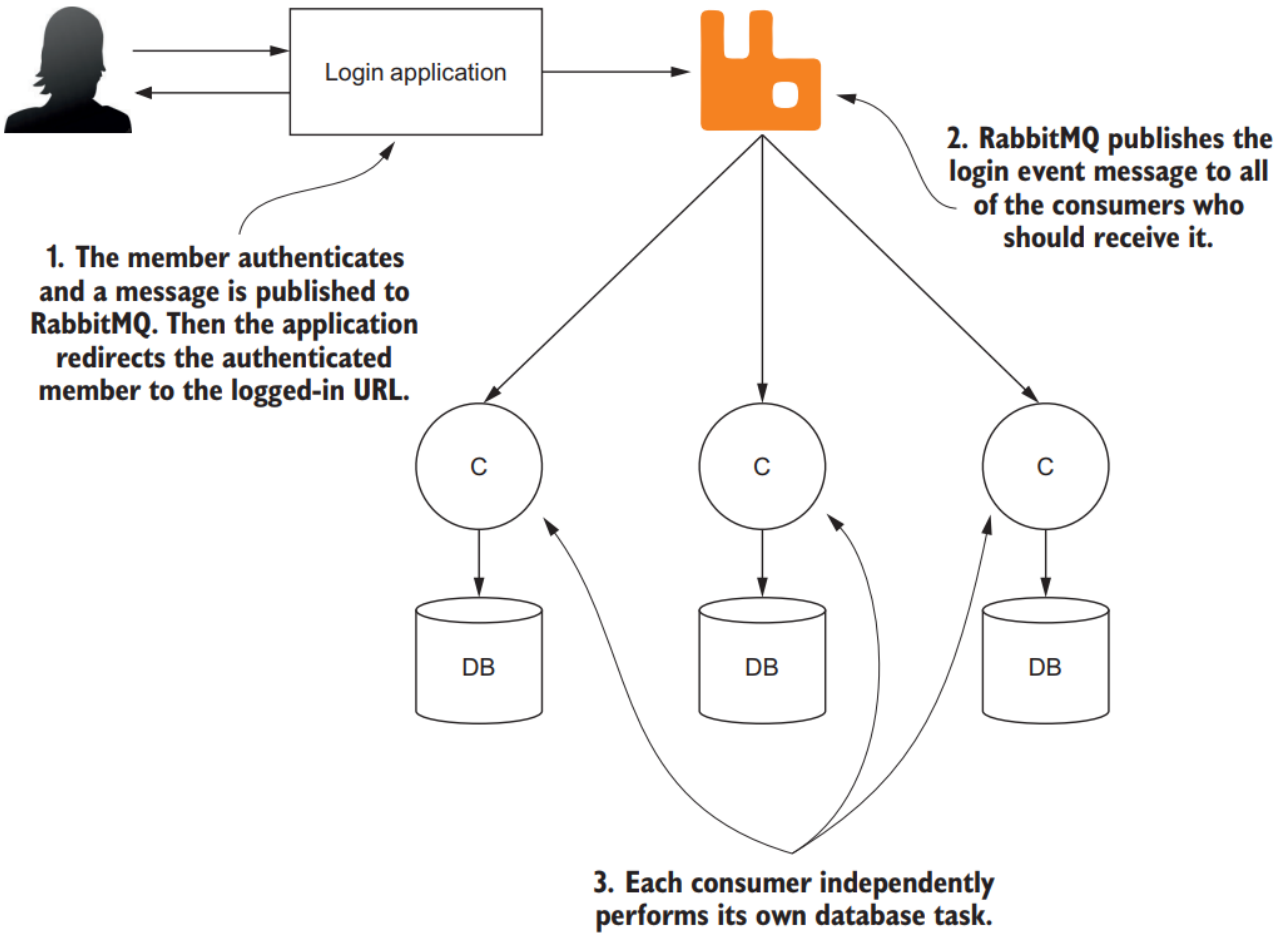
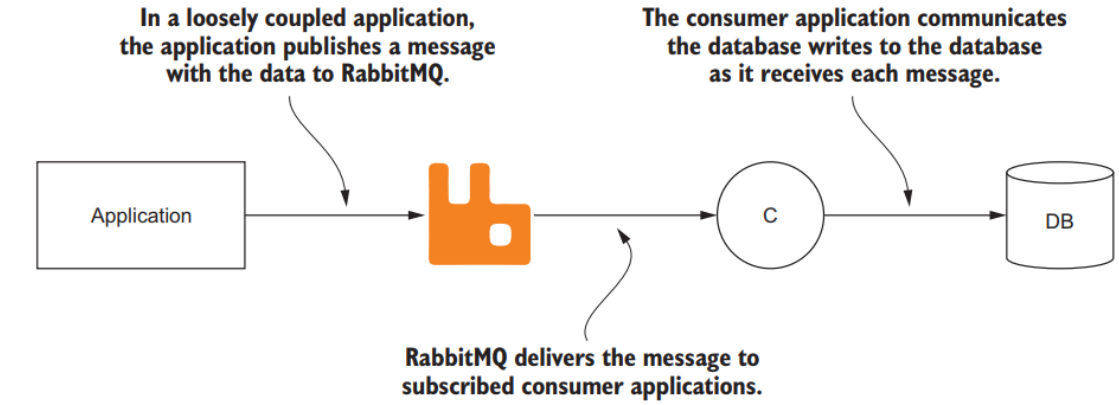


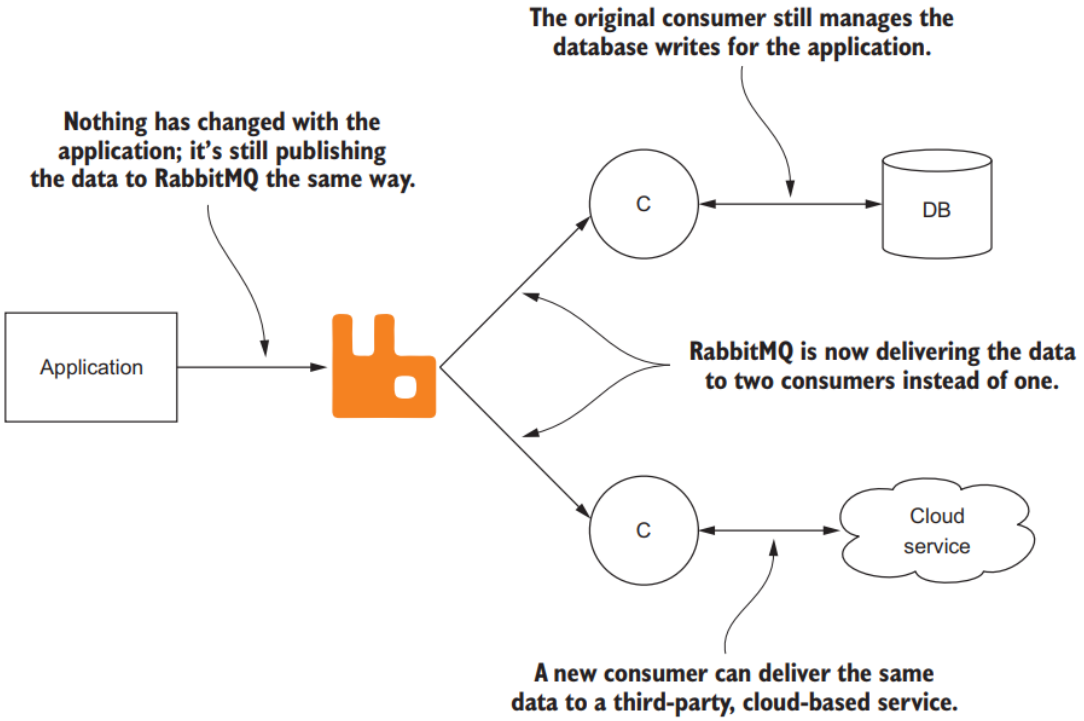
Figure 5.5 **Mirrored queue behavior**

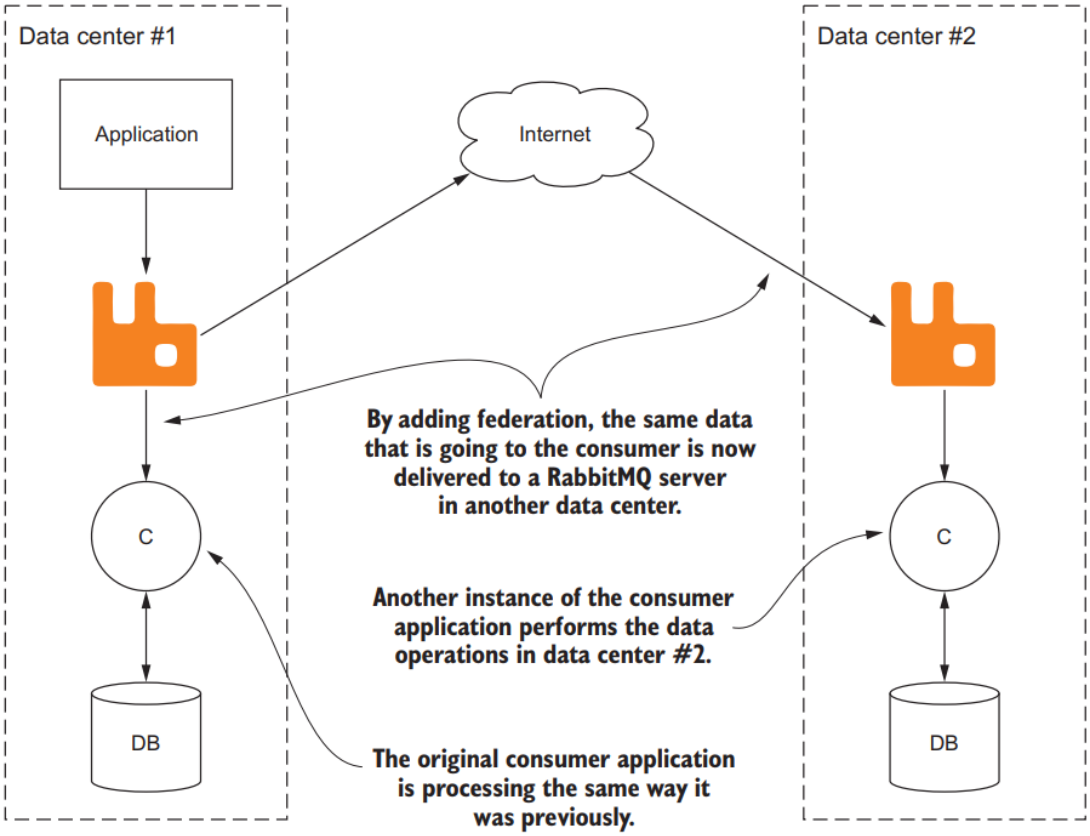
- With mirror queue, producer publish to both the main queue and the mirror queue

5.5 Rabbit MQ in depth




In a loosely coupled application, the application publishes a message with the data to RabbitMQ.
The consumer application communicates the database writes to the database as it receives each message.
Application
C
DB
RabbitMQ delivers the message to subscribed consumer applications.


The original consumer still manages the database writes for the application.
Nothing has changed with the application; it's still publishing the data to RabbitMQ the same way.
C
DB
Application
RabbitMQ is now delivering the data to two consumers instead of one.
C
Cloud service
A new consumer can deliver the same data to a third-party, cloud-based service.

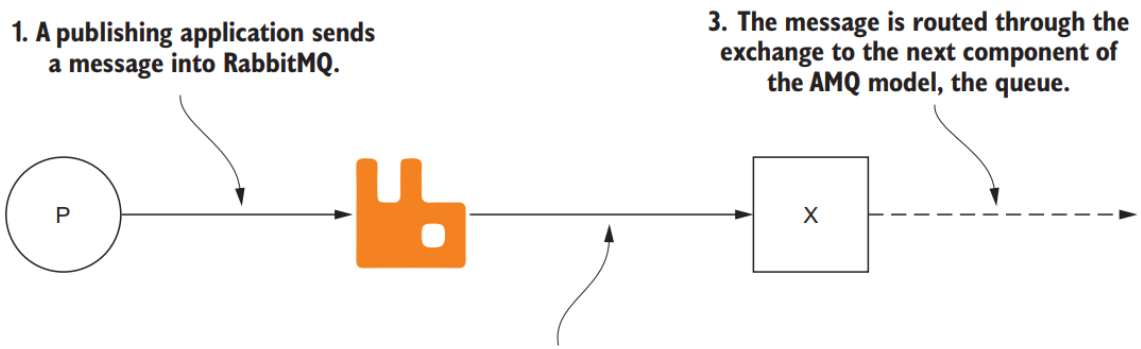


Messaging model:

- EXCHANGES

Exchanges are one of three components defined by the AMQ model. An exchange receives messages sent into RabbitMQ and determines where to send them. Exchanges define the routing behaviors that are applied to messages, usually by examining data attributes passed along with the message or that are contained within the message's properties.

RabbitMQ has multiple exchange types, each with different routing behaviors. In addition, it offers a plugin-based architecture for custom exchanges. Figure 1.9 shows a logical view of a publisher sending a message to RabbitMQ, routing a message through an exchange.



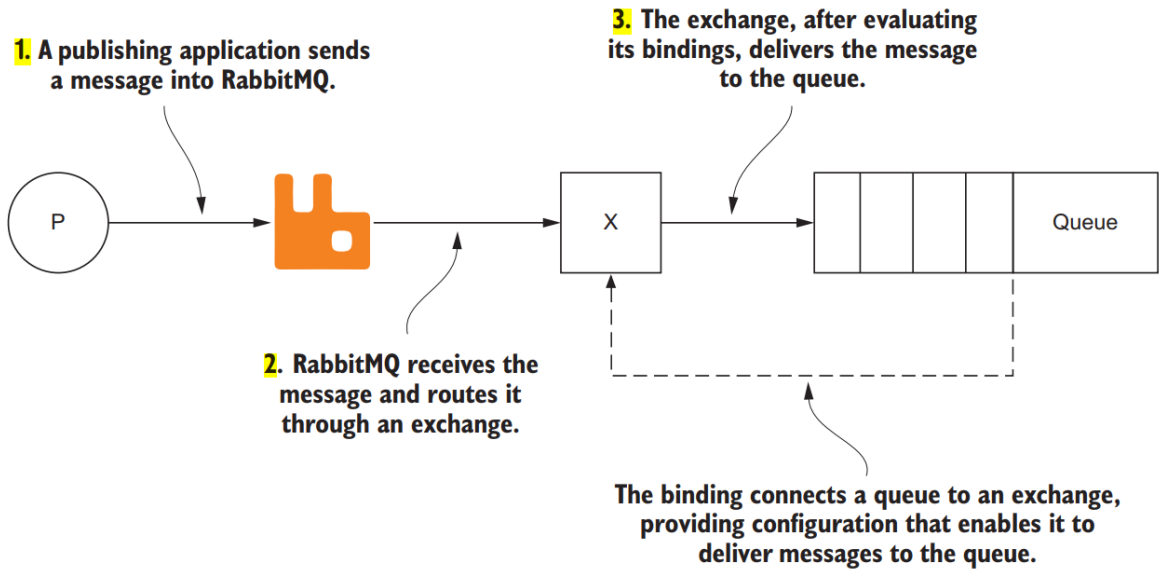
- QUEUES

A queue is responsible for storing received messages and may contain configuration information that defines what it's able to do with a message. A queue may hold messages in RAM only, or it may persist them to disk prior to delivering them in first-in, first-out (FIFO) order.

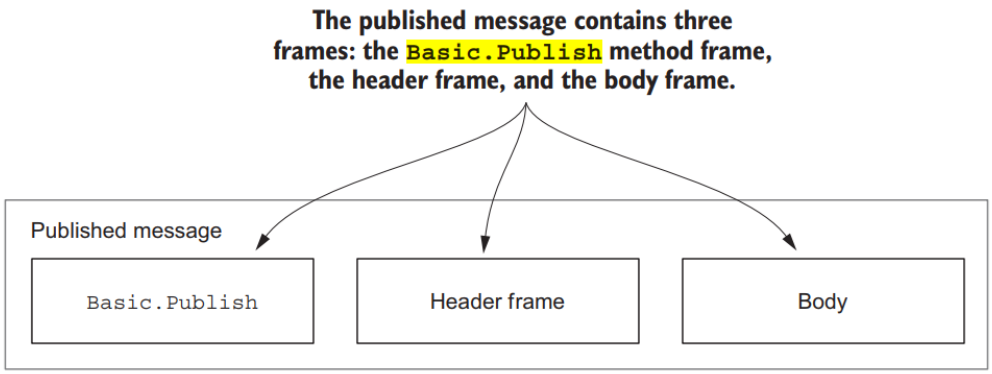
- BINDINGS

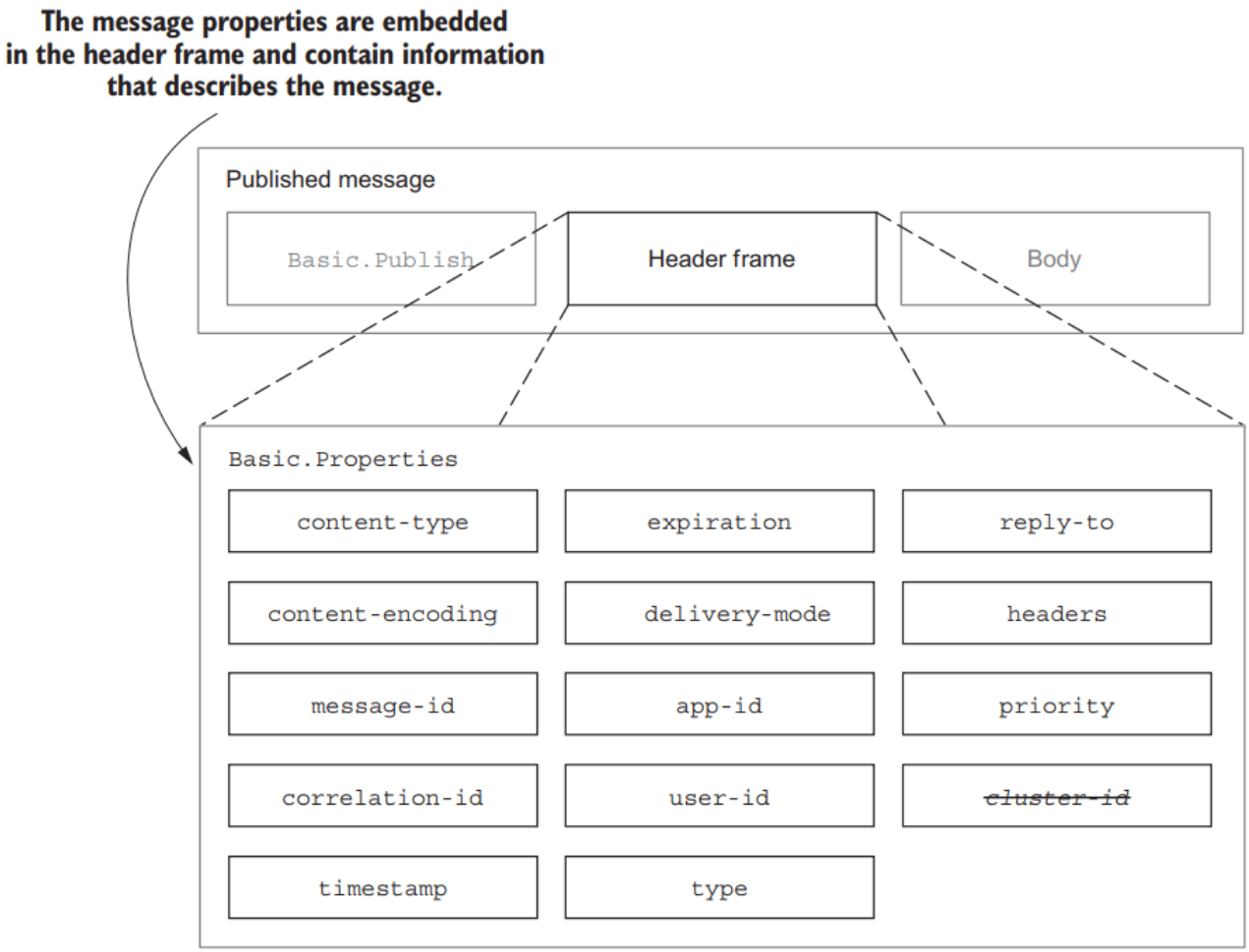
To define a relationship between queues and exchanges, the AMQ model defines a *binding*. In RabbitMQ, bindings or *binding keys*, tell an exchange which queues to deliver messages to. For some exchange types, the binding will also instruct the exchange to filter which messages it can deliver to a queue.

When publishing a message to an exchange, applications use a *routing*-key attribute. This may be a queue name or it may be a string that semantically describes the message. When a message is evaluated by an exchange to determine the appropriate queues it should be routed to, the message's routing key is evaluated against the binding



Message Properties





- content-type
- content-encoding
- message-id: unique id of message
- correlation-id: use for correlate, can be indentical to previous sent message
- timestamp
- delivery-mode: write to disk or not
- app-id
- user-id
- reply-to: specify channel to response
- header: for custom header
- priority: 0 - 255

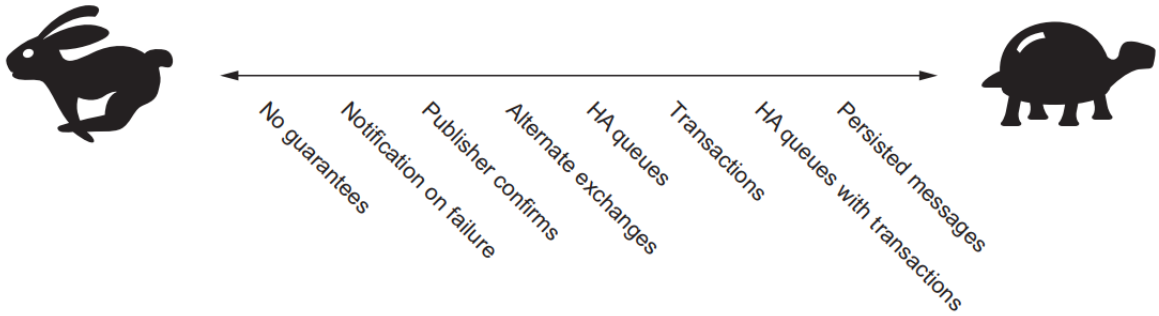
Property	Type	For use by	Suggested or specified use
app-id	short-string	Application	Useful for defining the application publishing the messages.
content-encoding	short-string	Application	Specify whether your message body is encoded in some special way, such as zlib, deflate, or Base64.
content-type	short-string	Application	Specify the type of the message body using mime-types.
correlation-id	short-string	Application	If the message is in reference to some other message or uniquely identifiable item, the correlation-id is a good way to indicate what the message is referencing.
delivery-mode	octet	RabbitMQ	A value of 1 tells RabbitMQ it can keep the message in memory; 2 indicates it should also write it to disk.
expiration	short-string	RabbitMQ	An epoch or Unix timestamp value as a text string that indicates when the message should expire.
headers	table	Both	A free-form key/value table that you can use to add additional metadata about your message; RabbitMQ can route based upon this if desired.
message-id	short-string	Application	A unique identifier such as a UUID that your application can use to identify the message.
priority	octet	RabbitMQ	A property for priority ordering in queues.
timestamp	timestamp	Application	An epoch or Unix timestamp value that can be used to indicate when the message was created.
type	short-string	Application	A text string your application can use to describe the message type or payload.
user-id	short-string	Both	A free-form string that, if used, RabbitMQ will validate against the connected user and drop messages if they don't match.

Tips for enchancement of rabbitmq performance

Performance trade-offs in publishing

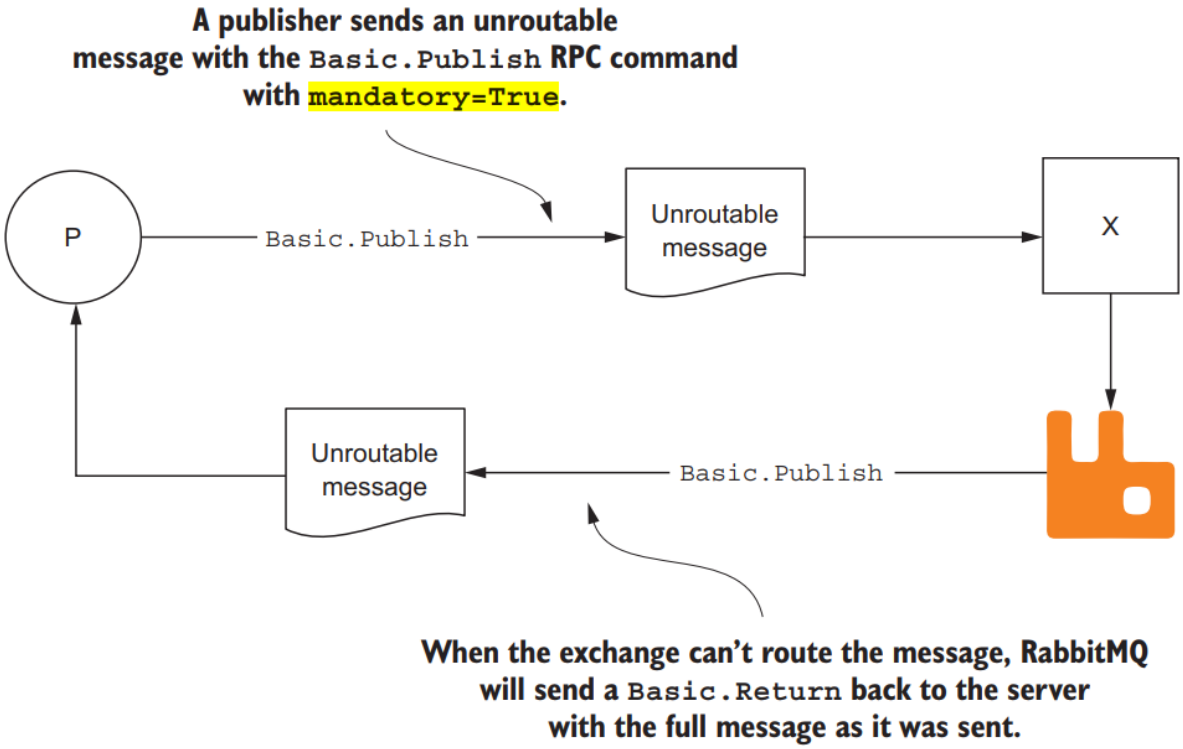
Balancing delivery speed with guaranteed delivery

When it comes to RabbitMQ, the *Goldilocks Principle* applies to the different levels of guarantees in message delivery. Abstracted as a takeaway from the "Story of the Three Bears," the Goldilocks Principle describes where something is *just right*. In the case of reliable message delivery, you should apply this principle to the trade-offs encountered when using the delivery guarantee mechanisms in RabbitMQ. Some of the features may be too slow for your application, such as the ability to ensure messages survive the reboot of a RabbitMQ server. On the other hand, publishing messages without asking for additional guarantees is much faster, though it may not provide a safe enough environment for mission-critical applications (figure 4.1).

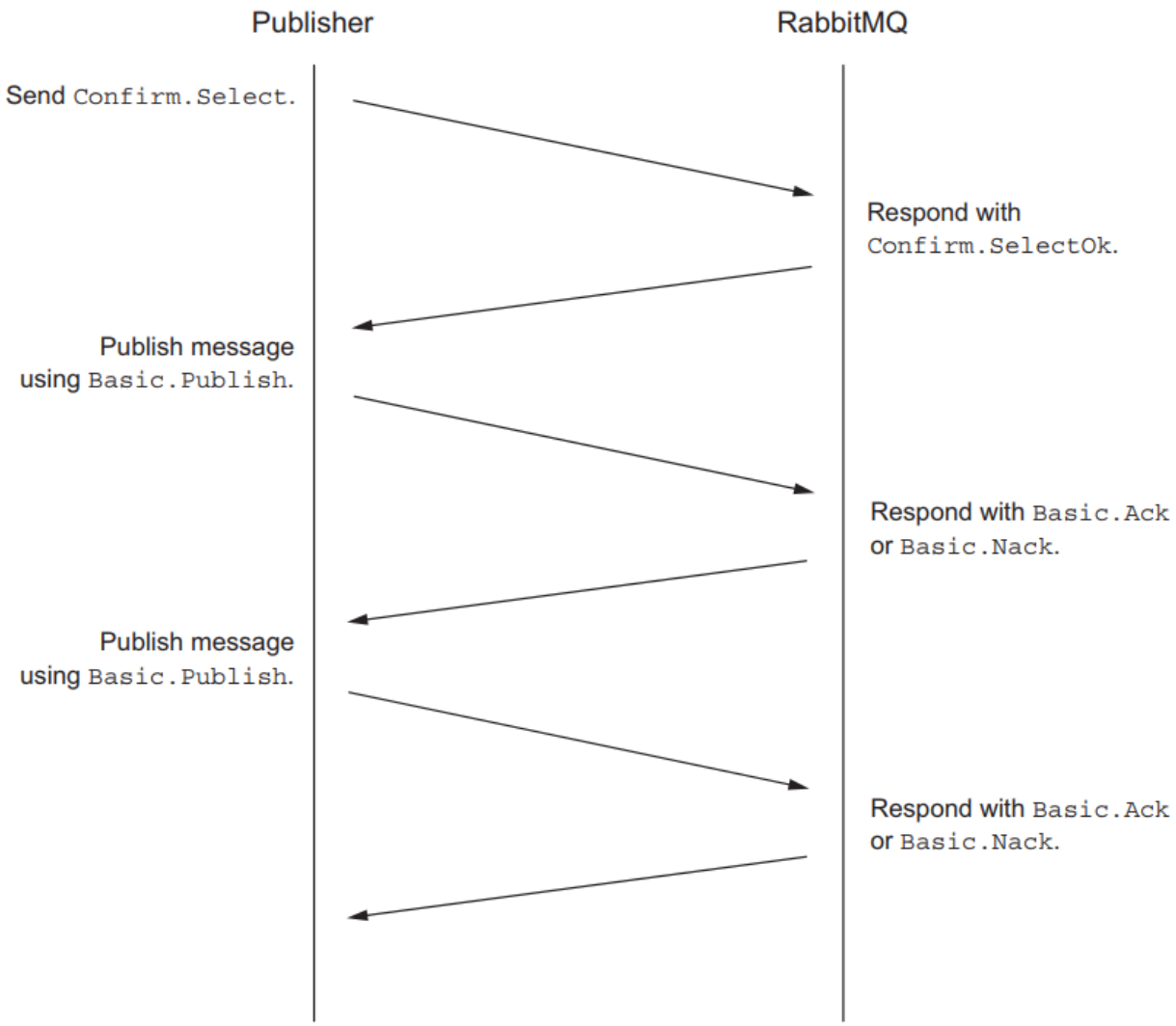


Mandatory message:

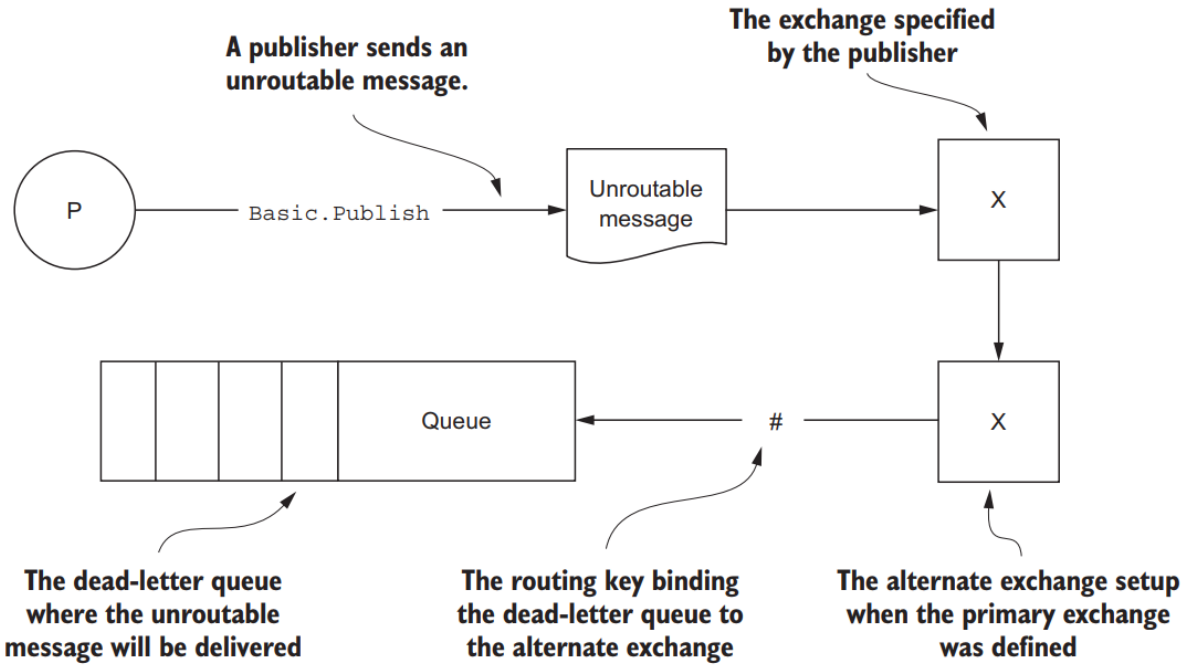
With mandatory message, when message can not be routed, rabbit-mq return message with the same content to publisher



Publisher Confirms as a lightweight alternative to transactions



Using alternate exchanges for unroutable messages



6. Reliable message

6.1 Publisher confirm

> Depending on the acknowledgement mode used, RabbitMQ can consider a message to be successfully delivered either immediately after it is sent out (written to a TCP socket) or when an explicit ("manual") client acknowledgement is received. Manually sent acknowledgements can be positive or negative and use one of the following protocol methods:
>
> - `basic.ack` is used for positive acknowledgements
> - `basic.nack` is used for negative acknowledgements (note: this is a RabbitMQ extension to AMQP 0-9-1)
> - `basic.reject` is used for negative acknowledgements but has one limitation compared to `basic.nack`
>
> How these methods are exposed in client library APIs will be discussed below.
>
> Positive acknowledgements simply instruct RabbitMQ to record a message as delivered and can be discarded. Negative acknowledgements with `basic.reject` have the same effect. The difference is primarily in the semantics: positive acknowledgements assume a message was successfully processed while their negative counterpart suggests that a delivery wasn't processed but still should be deleted.

Positive confirm from consumer:

```
long deliveryTag = envelope.getDeliveryTag();
            // positively acknowledge a single delivery, the message will
            // be discarded
            channel.basicAck(deliveryTag, false);
```

Negative Acknowledgement and Requeuing of Deliveries

Reject and discard

```
long deliveryTag = envelope.getDeliveryTag();
            // negatively acknowledge, the message will
            // be discarded
            channel.basicReject(deliveryTag, false);
```

Reject and requeue

```
long deliveryTag = envelope.getDeliveryTag();
            // requeue the delivery
            channel.basicReject(deliveryTag, true);
```

basic.qos: number of allow message in buffer to be unacknowledge, when message in buffer cap this number, channel will stop receive new message until the current one is done

Channel is multiplexing mechanism of rabbit MQ on the same channel

Each rabbitMQ consumer have one channel

```
@RabbitListener(queues = {DirectConfiguration.D_QUEUE}, // One channel
messageConverter = "consumerJackson2MessageConverter") // Mapping JSON
public void listenToQueue(Message<?> message)
{

 }


@RabbitListener(queues = {DirectConfiguration.D_QUEUE}, // Second channel
        messageConverter = "consumerJackson2MessageConverter") // Mapping JSON
public void listenToQueueX(Message<?> message){}
```

If not specify producer will send in round robin.

Pubisher confirm: ensure each message send to queue is ackknowledge, consumer can reject each message and require to requeue message.

- basic.ack: deliveried and delete from queue
- basic.nack: undeliveried and reenqueue for re-delivery.

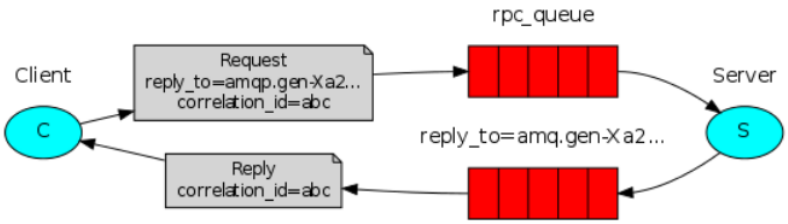
template.setReturnsCallback: use for unroutable message

template.setConfirmCallback: use for confirm message, (correct routing)

Deduplication:

Callback:

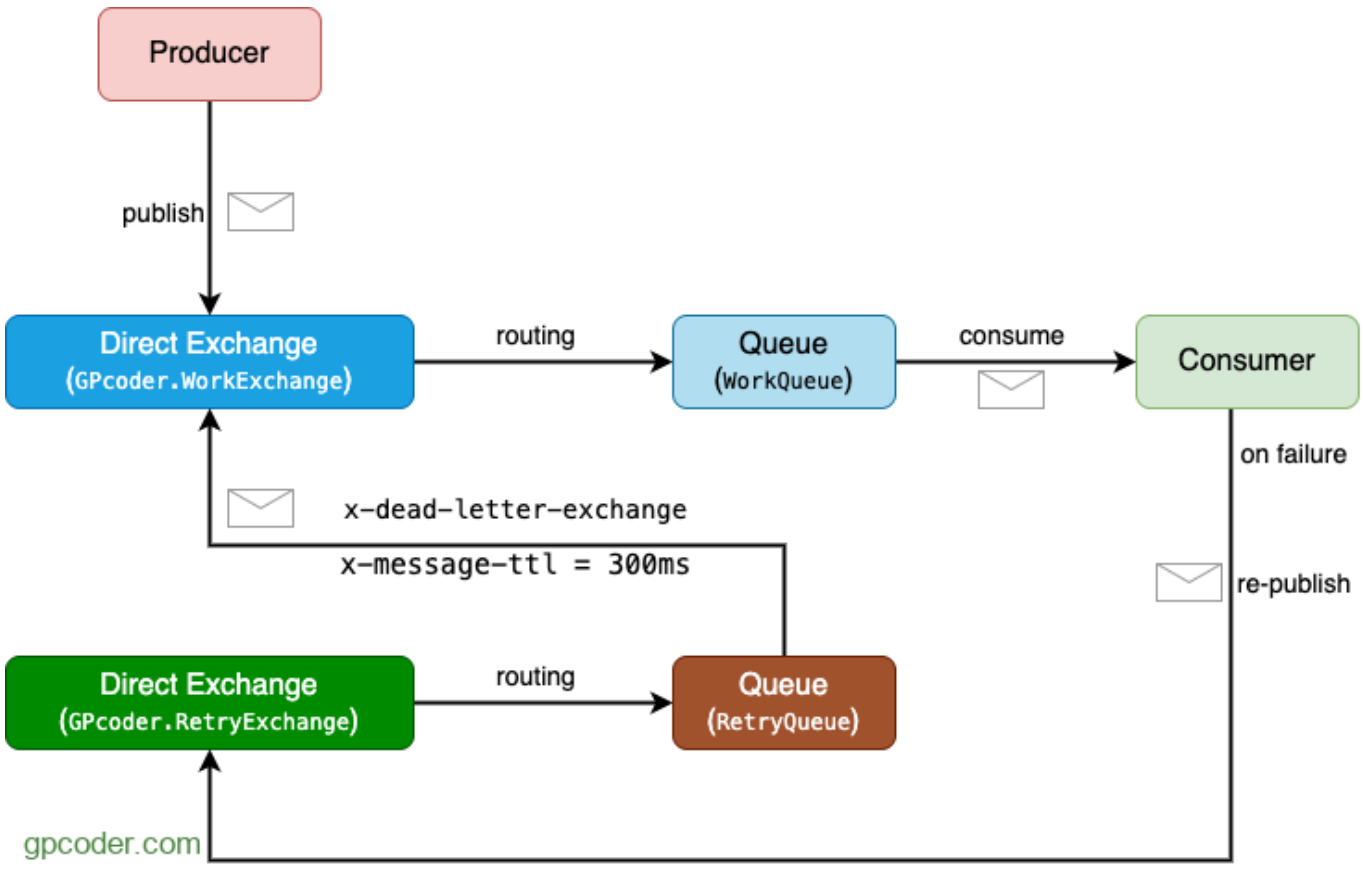
- DeliveryMode: persistence vs transient
- ContentType:
- ReplyTo:
- CorrelationId:



Our RPC will work like this:

- The `Tut6Config` will setup a new `DirectExchange` and a client
- The client will leverage the `convertSendAndReceive` method, passing the exchange name, the routingKey, and the message.
- The request is sent to an RPC queue `tut.rpc`.
- The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it performs the task and sends a message with the result back to the client, using the queue from the `replyTo` field.
- The client waits for data on the callback queue. When a message appears, it checks the `correlationId` property. If it matches the value from the request it returns the response to the application. Again, this is done automagically via the `RabbitTemplate`.

Dead exchange:



1. Tạo một WorkQueue và bind đến WorkExchange.
2. Tạo một RetryQueue và bind đến RetryExchange.
    - Gán agruments: **x-dead-letter-exchange** đến WorkExchange
    - Gán agruments: **x-message-ttl** là 300 ms.

3. Producer publish một Message đến WorkExchange. Sau đó, WorkExchange sẽ chuyển Message đến WorkQueue.
4. Consumer nhận Message từ WorkQueue và cố gắng xử lý nó.
5. Trường hợp xử lý thất bại, Consumer sẽ publish Message đó đến RetryExchange. Sau đó, RetryExchange sẽ chuyển Message đến RetryQueue.
6. Message sẽ lưu tại RetryQueue trong 300 ms.
7. Khi Message bị expire, nó sẽ được chuyển đến WorkExchange và được chuyển đến WorkQueue.
8. Khi đó Consumer có thể nhận lại Message từ WorkQueue và xử lý lại.