**Microservices** are a group of components that communicate using a centralized service, like a Message Queue.

The opposite of a Microservice System is referred to as a **Monolith System** and is instead communicating directly with other parts of the system. For that reason, microservice applications are often *"Decoupled"* while monolith applications are *"Highly Coupled"*.

Also, using a message broker in your system means that you’re making the communication between the components *Asynchronous*. It’s much like when we’re sending an email, the response is sent when it's convenient.

The opposite of this is direct communication or *"Synchronous"* communication, which is more similar to when we’re chatting on the phone. The response is immediate and, when you hang up, the conversation is over.

Asynchronous communication and decoupling within systems can be very beneficial. Here follows some advantages that show the benefits of having a Message Queue as part of the architecture.

The microservice system is decoupled and divided into two or more small services, where each microservice is developed separately. The sum of all microservices is what makes up the total application. When a system is decoupled, every microservice operates individually and doesn't have to relate to the others.

Because of that, different microservices can be written in different languages, be developed by different teams, and evolve independently.

If one of the microservices goes down or for another reason is unavailable, the rest of the services in the system will still proceed with their tasks independently. This makes the application fault-tolerant and stable.

In a monolith system, on the other hand, a failure of one single service can cause problems or even make the entire application go down.

A system using a Message Queue offers temporary storage for messages waiting to be handled by a consumer. Publishers can, therefore, continue to produce messages without having to worry about the message itself. As long as messages hit the Queue, the message is safe within the Broker and suffers a minimal risk of getting lost.

If the consumer is not able to handle a message, the same message can be re-delivered multiple times until the consumer can process it successfully.

Let’s give another common example; almost every IT system includes a database in one way or another. In a tightly coupled architecture, the application must wait for the database server to respond, before it can finish a transaction. If the database crashes, the application will potentially crash as well, and, why crash a complete application due to a small update in the database? By decoupling the database from the application and adding a Message Queue in between, the consumer can pick up the data from the Broker and perform the database action at a later time - making the application more fault-tolerant.

A highly scalable system means that it can be customized to handle the amount of workload required, at every given moment. When the system is under heavy load: you scale up, and if the workload decreases: you scale down. Much like when you go grocery shopping and the checkout queue is growing long, more cashiers are added to help out.

Adding more consumers to an existing microservice system is helping to keep the queues short during peak hours, and you can remove consumers when they are not needed. You don’t have to build an over-dimensional system just to be able to handle peaks.

Retailers and media-streaming services are frequent users of scaling since they often know specific moments when their applications are under heavy load.

A decoupled system is a resilient system meaning that it can recover quickly from difficult conditions. Since the use of Message Queues decouples your services, leaving them unaware and unbound of each other, the impact of a failure of one service won't affect the others.

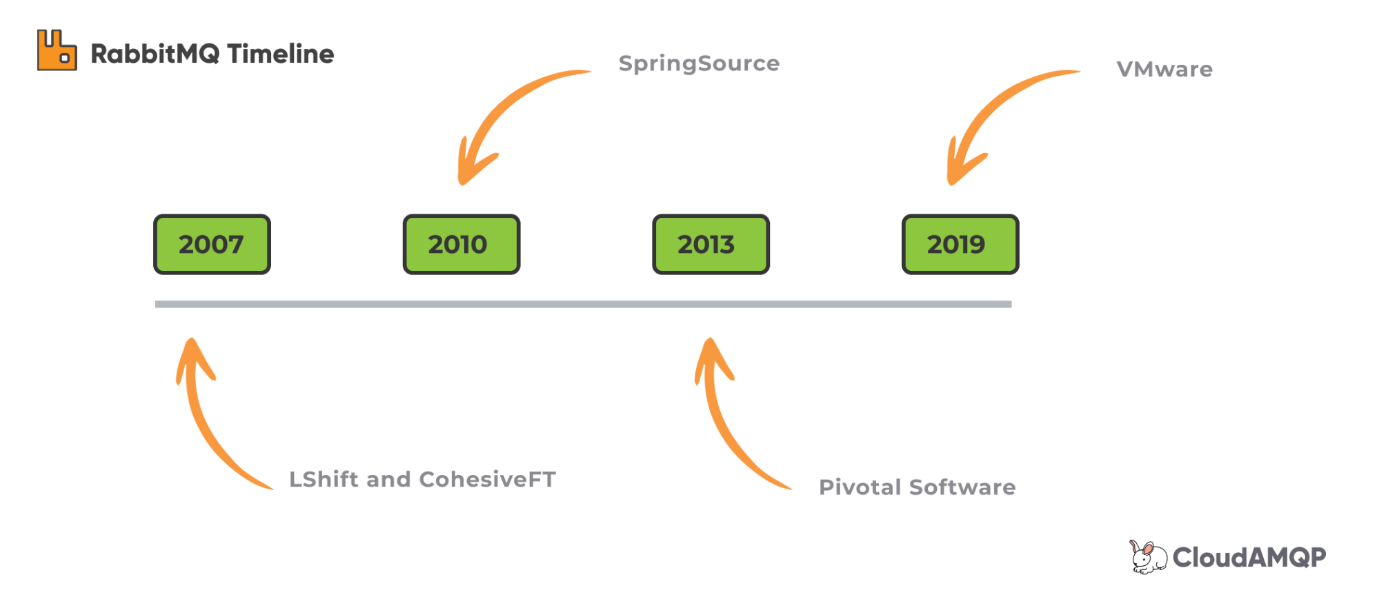
A service can still publish messages to the Queue, and the messages will stay in the Queue until the failing service is up and running. This also applies when a service needs upgrading or if developers wish to add a new service to an already existing system.

Asynchronous communication within Message Queuing means that messages in the Queue don’t have to be processed immediately.Instead, it will be processed as soon as the assigned service is ready. This allows the system to carry on, serving you or other users, with the assurance that the task will be completed not immediately, but as soon as possible.

To clarify the extremes between synchronous and asynchronous communication: When shopping online, an asynchronous system will let you continue navigating the web shop after you place the order and hit send. The payment will be processed in the background and the order confirmation will be sent to your email whenever possible.

On the other hand, the synchronous system would temporarily lock the web shop up until the system approves of all the series of events.

* RabbitMQ persists messages until they are dropped on the acknowledgment of receipt
* Available monitoring through a built-in UI
* Most languages are supported
* Supports multiple messaging protocols
* Supports standard authentication and OAuth2
* RabbitMQ allows clients to connect over a range of different open and standardized protocols such as [AMQP](https://www.cloudamqp.com/docs/amqp.html), [HTTP](https://www.cloudamqp.com/docs/http.html), [STOMP](https://www.cloudamqp.com/docs/stomp.html), [MQTT](https://www.cloudamqp.com/docs/mqtt.html) and [WebSockets/Web-Stomp](https://www.cloudamqp.com/docs/web_stomp.html" \t "_blank)

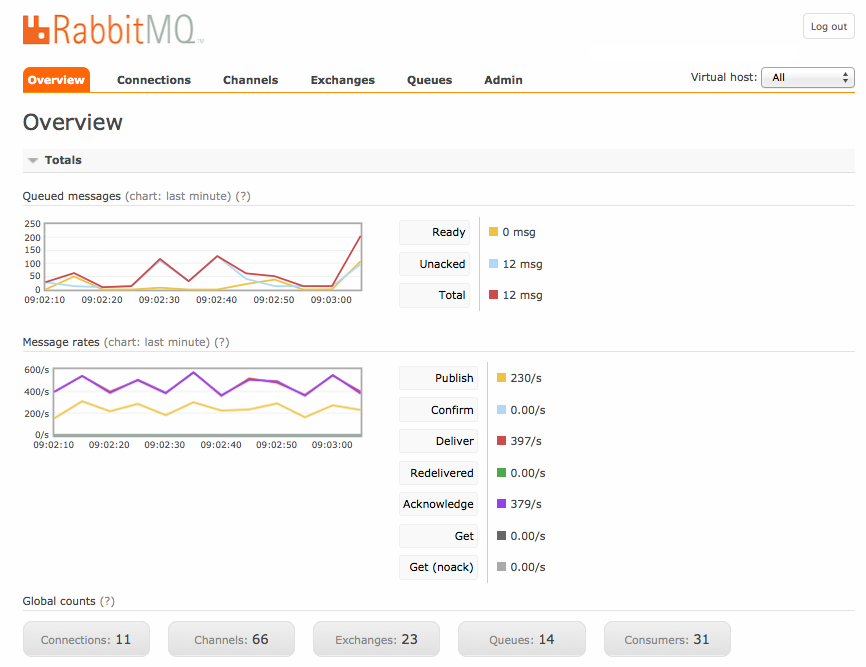


RabbitMQ was originally developed by both LShift and CohesiveFT as a joint project back in 2007. In 2010 RabbitMQ was picked up by the VMware division SpringSource and later, in 2013, the project became part of Pivotal Software.

**Easy To Use**

RabbitMQ Management is a user friendly interface that lets you monitor and handle your RabbitMQ server from a web browser. Among other things queues, connections, channels, exchanges, users, and user permissions can be handled - created, deleted, and listed - in the browser. You can monitor message rates and send/receive messages manually. This section gives information about the different views found in the RabbitMQ Management interface.

The RabbitMQ Management interface is a plugin that can be enabled for RabbitMQ. It gives a single static HTML page that makes background queries to the HTTP API for RabbitMQ. Information from the management interface can be useful when you are debugging your applications or when you need an overview of the whole system. If you see that the number of unacked messages starts to get high, it could mean that your consumers are getting slow. If you need to check if an exchange is working, you can try to send a test message through the interface.



RabbitMQ implements a standard protocol called AMQP, short for Advanced Message Queuing Protocol. Over the years RabbitMQ has been extended to support additional protocols such as STOMP, MQTT, and more.

AMQP is still most widely used and will be the focus of this course. A protocol is basically a specification, or a standard, which makes it possible for electronic devices to communicate efficiently with each other.

This is achieved by declaring how data is sent according to a set of rules. These rules may include what commands should be used when sending or receiving data, what type of data can be transmitted, and how to confirm that data has been transferred successfully.

We are relying on standard protocols every day when we are using the internet, without even reflecting on it. Protocols are what makes it possible for a Mac computer to access a website hosted on a Unix-based server (HTTP). It also makes it possible for an iPhone to send emails to an Android device (SMTP).

Essentially, the protocols allow connected devices to communicate with each other, regardless of any differences in their internal processes, structure, or design.

But let’s get back to AMQP. AMQP is a protocol specifically for sending messages to a Queue. In order for RabbitMQ to be compatible it has to live up to rules of how the messages are routed and stored within the Broker.

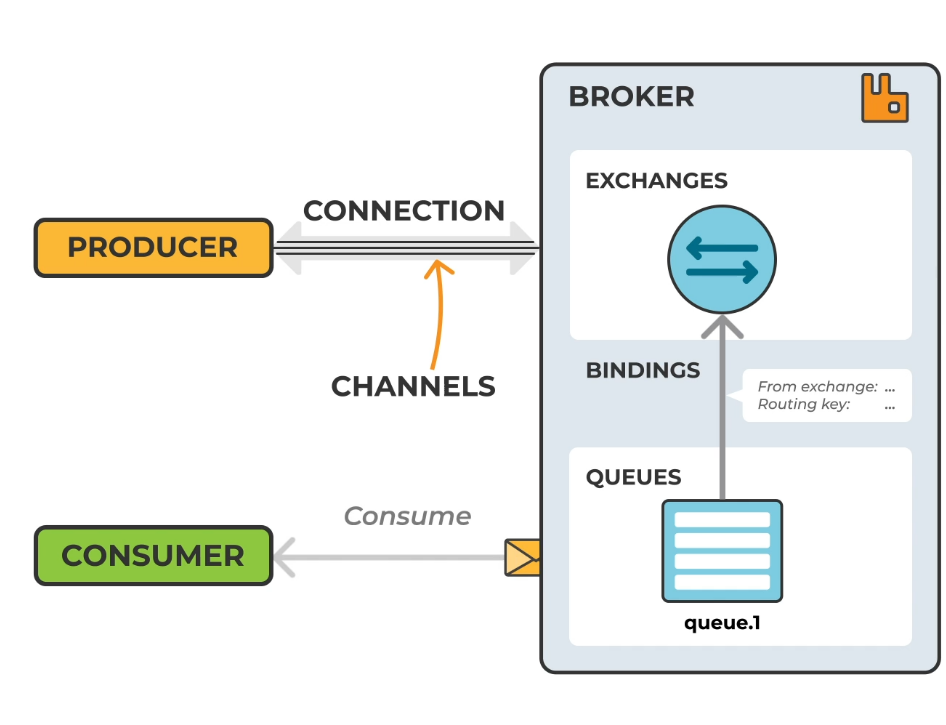
The protocol is for example very specific on how connections are made, the behaviour of the Broker and how you declare the Queues. Other defining features of AMQP include specification of reliability and security.

Now when we understand the difference between RabbitMQ (the software) and AMQP (the protocol), it’s time to dive a bit deeper.

1. Message Brokers acts as a middleman between services who wish to communicate asynchronously.
2. Message send by published and consumed by consumer
3. Message can have multiple fields like payload, routing\_key, headers.
4. Message from producer are place in a message queue in FIFO order inside the broker before it gets consumed by Consumer.
5. Message is sent from producer to queue via a connection and a channel
6. Exchanges ensures that a message endsup in the correct queue.
7. Binding connects the exchange and the queue.
8. Bindings are defined while creating an exchange.
9. Publish message is available inside the exchange.
10. VHost provides a way to segregate applications using the same RabbitMQ instance.
11. Consumers Fetch messages from the queue and process them in sequential order.

A diagram of a product

Description automatically generated



A diagram of a blockchain

Description automatically generated

**Queues**

The queue is the place where messages are stored until they are consumed by the consumer, or in other ways removed from the queue. Queues have properties that define how they behave, and these properties are passed to the broker when the queue is declared.

* A queue has some required properties and some optional. A queue always has a name, so that services can reference them. A queue declared with no name, is given a random name by most client libraries.
* A queue can be marked as durable, which specifies if the queue should survive a broker restart.
* A queue can be exclusive, which specifies if the queue can be used by only one connection. An exclusive queue is deleted when that connection closes.
* A queue can also be declared with the auto-delete property, meaning that a queue that has had at least one consumer is deleted when the last consumer unsubscribes.
* There are also some optional properties used by plugins and broker-specific features, like TTL, which is telling an unused queue when to expire after a period of time.

Before a queue can be used it has to be declared. Declaring a queue will cause it to be created if it does not already exist.

Let’s follow the life-cycle for a temporary message queue.

1. The client creates the message queue (Declare). The server confirms (Declare-Ok).
2. The client starts a consumer on the message queue.
3. The client cancels the consumer, either explicitly or by closing the channel and/or connection.
4. When the last consumer disappears from the message queue, the server deletes the message queue.

**Exchanges**

Theoretically, the exchange is the first entry point for a message entering the message broker. Messages are not published directly to a queue. Instead, the producer sends messages to an exchange, which you can think of as a message routing agent. An exchange is responsible for routing messages to different queues with the help of header attributes, bindings, and routing keys.

Exchanges can, as with queues, be configured with parameters such as durable, temporary, and auto-delete upon creation. Durable exchanges survive server restarts and exist until they are explicitly deleted. Temporary exchanges exist until RabbitMQ is shut down. Auto-deleted exchanges are removed once the last bound object is unbound from the exchange.

Let’s follow the life-cycle for an exchange.

1. The client asks the server to make sure the exchange exists (Declare). The exchange is created if it does not already exist.
2. The client publishes messages to the exchange.
3. The client may choose to delete the exchange (Delete).

In RabbitMQ, there are four different types of exchanges that route the message differently using different parameters and bindings setups. Clients can create their own exchanges or use the predefined default exchanges which are created when the server starts for the first time.

channel.exchange\_declare(exchange='direct\_exchange', exchange\_type='direct')

**Bindings**

A binding is an association, or relation between a queue and an exchange. It describes which queue is interested in messages from a given exchange. Bindings can take an extra parameter called routing\_key. If you remember, a routing key can also be sent with a message. The routing key on the binding sometimes called a binding key, and the routing key in the message are the things the exchange is looking at while delivering messages.

**Exchange Types**

In RabbitMQ there are four main types of Exchanges:

* Direct
* Topic
* Fanout
* Headers

Existing exchanges and types can be seen in the management interface or through rabbitmqadmin.

**Taxi Company Example**

We will use an example from a taxi company to explain the different exchange types. Every request for a taxi is made through an app that communicates with an application service that uses RabbitMQ.

**Direct Exchange**

Direct exchange directs the message to a specific queue by looking at the routing key. The routing key in the message is compared for equality with routing keys on bindings.

In our example the direct exchange is used when a user requests a specific taxi, like their favorite driver.

**Topic Exchange**

Topic exchange routes messages to one or many queues by looking at the routing key. The routing key in the message is compared for matches with routing key-patterns on the bindings.

In our example a customer with a group of friends ask for a large environmentally-friendly taxi. This order is routed through an exchange bound to taxis of this type.

The routing key must be a list of words delimited by a period. The topic exchange supports "strict" routing key-matching, like a direct exchange, but will also perform "wildcard" matching using star (\*) and hash (#) as placeholders.

In another example a customer orders a large taxi but does not care about the taxi type. The topic exchange routes this message to all taxis bound as a large taxi.

**Fanout exchange**

Fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys. A provided routing key is simply ignored.

In our example the Fanout Exchange is used when the taxi coordinators inform all taxi drivers about a blocked road.

**Headers Exchange**

Headers exchanges are very similar to topic exchanges but route messages based on the header values instead of routing keys.

Headers exchanges are not very common, but in our example it is used by a reporting service at the taxi company. Taxi car status data is sent to the exchange every now and then and this data is used to build reports by other parts of the system.

A special argument named "x-match" added in the binding between exchange and queue, specifies if headers must match "all" or "any".

In this example one service writes a report for all taxis around Manhattan, New York, keeping track of fuel consumption and miles traveled. This data is included in the message. When "x-match" is set to "any", together with the following arguments, the New York report receives all messages from taxis with trips from, to, or within New York. A trip that starts in New York and ends in Jersey will therefore be included.

Another report may be interested in trips within New York alone. This can be achieved by simply setting the "x-match" to "all". The new report will then only get messages where "from" and "to" are both set to New York - trips that never leaves the city.

**Consumer Acknowledgements and Publisher Confirm**

Messages in transit might get lost in the event of a connection failure and need to be retransmitted. Acknowledgments let the server and clients know when to retransmit messages. The client can either ack the message when it receives it, or when the client has completely processed the message. Publish confirm is the same concept but for publishing. The server confirms when it has received a message from a publisher.

A message can be considered successfully delivered either immediately once it is sent out, once it’s written to a TCP socket, OR when an explicit acknowledgment is received from the client. The manually sent acknowledgment can be positive or negative.

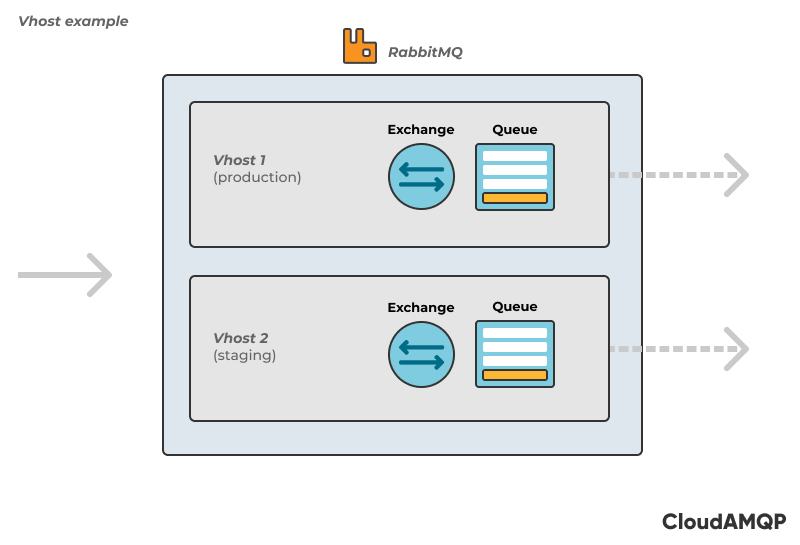
Once an acknowledgment is received, the message can be discarded from the queue. If the consumer cannot process a message, the desired outcome could be to requeue it and let another consumer receive and handle it, or to retry processing at a later time. Such deliveries can be discarded by the broker or requeued. This behavior is controlled by the requeue field. When the field is set to true, the broker will requeue the delivery. A requeued message is placed back into its original position in its queue, if possible, and can therefore immediately be ready for redelivery. If all consumers requeue because they cannot process a delivery, they create a requeue/redelivery loop. It is possible to track the number of redeliveries and reject messages for good (discard them) or schedule a requeue after a delay.

**Virtual Hosts**

Virtual hosts (or vhosts) in RabbitMQ provide a way to segregate applications using the same RabbitMQ instance. RabbitMQ vhosts create a logical group of connections, exchanges, queues, bindings, user permissions, etc. within an instance.

Think of vhosts as individual, uniquely named containers. Inside each vhost container is a logical group of exchanges, connections, queues, bindings, user permissions, and other system resources.

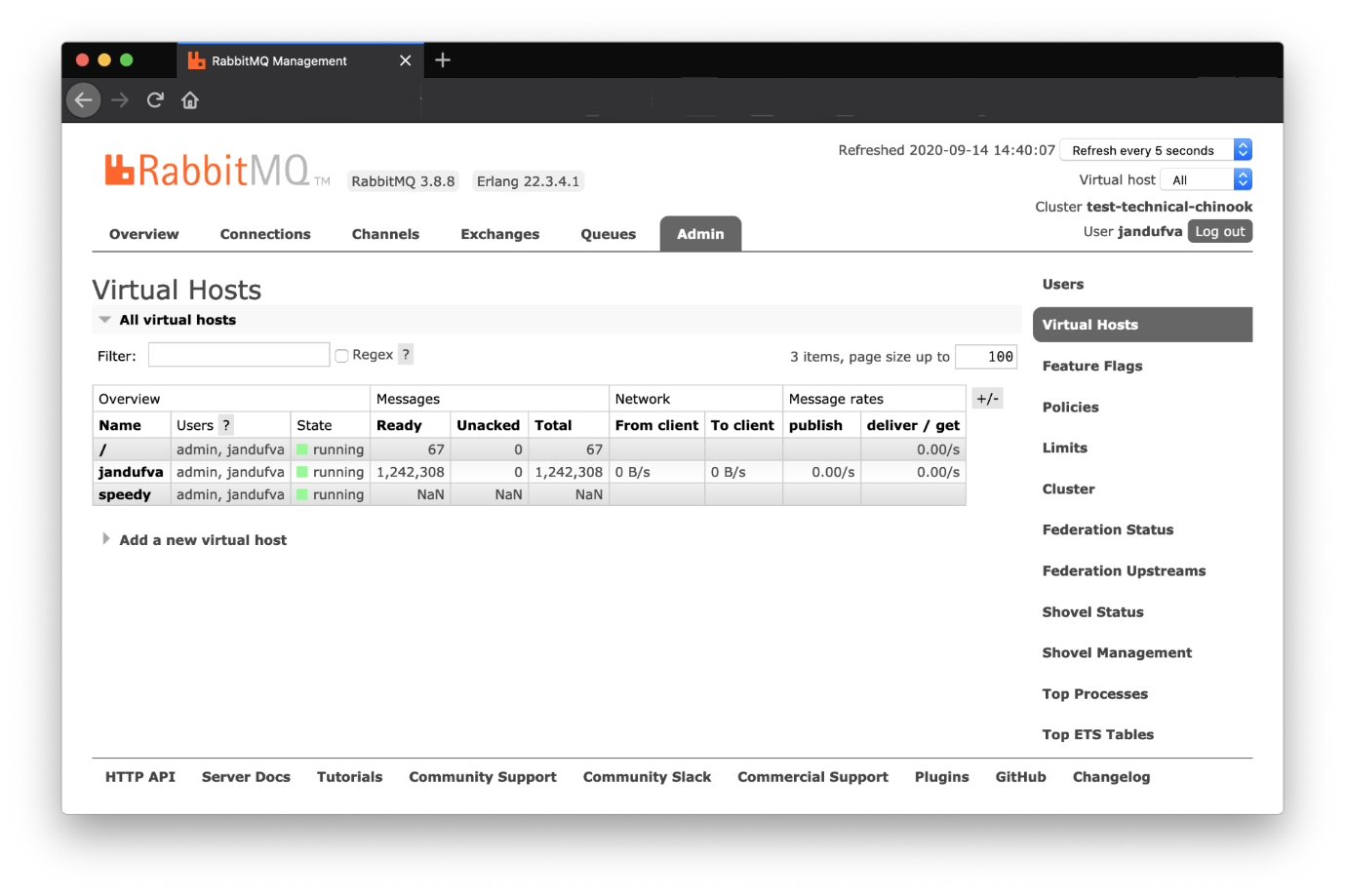
Different users can have different permissions to different vhost and queues and exchanges can be created, so they only exist in one vhost. When a client establishes a connection to the RabbitMQ server, it specifies the vhost within which it will operate.



Resources such as exchanges and queues are named resources inside the vhost container, making each vhost essentially a RabbitMQ mini-server. When configuring RabbitMQ, at least one vhost is needed, which in default is just a slash “/”.

Virtual hosts can be created through the management portal, through the HTTP API, or via rabbitmqctl. View vhosts by entering the admin tab and select the Virtual Hosts. Select the Add New Virtual Host option to create a new vhost.

The permissions within the vhost and the users assigned to it depend on your system requirements, and it’s up to you to assign users to the vhost. A newly created vhost always has a default set of exchanges, but no other entities and no user permissions.



Virtual hosts do not share exchanges or queues between them, and users, policies, etc. are unique to each vhost. Essentially, RabbitMQ vhosts are like a virtual machine for a physical server, allowing for multiple secure application operations through virtual rather than physical separation. As the separation is virtual, it is important to remember that the vhosts are not physically separated from each other and therefore they might affect each other’s performance.

The same broker can be used on parts of different applications. You can separate environments, e.g. production to one vhost and staging to another vhost, within the same broker instead of setting up multiple brokers.

If one service is experiencing a traffic spike or a code bug, this may cause problems for other services and affect their performance. A vhost can be created for each service to hold all of the logical infrastructure, which also allows for better individual topology management.

**Mirrored Queues**

In the past, using classic Mirrored Queues was the way to create highly available (HA) queues in RabbitMQ. Now, Quorum queues should be the default choice for a replicated queue type. Alternatively, Streams is a messaging data structure available as of RabbitMQ version 3.9, and is also replicated.

**Classic queue mirroring** will be removed in a future version of RabbitMQ, and will remain a supported non-replicated queue type. So, this is just a section telling you to **not** use mirrored queues anymore.

**Quorum Queues**

Quorum queues ensure that the cluster is up to date by agreeing on the contents of a queue. By doing so, quorum queues avoid losing data. Quorum queues are available as of RabbitMQ 3.8.0.

All communication is routed to the queue leader, which means the queue leader locality has an effect on the latency and bandwidth requirement of the messages.

In quorum queues, the leader and replication are consensus-driven, which means they agree on the state of the queue and its contents. Quorum queues will only confirm when the majority of its nodes are available, which thereby avoids data loss.

Declare a quorum queue using the following command:

rabbitmqadmin declare queue name=<name> durable=true arguments='{“x-queue-type”:“quorum”}'

These queues must be durable and instantiated by setting the *x-queue-type* header to quorum. If the majority of nodes agree on the contents of a queue, the data is valid. Otherwise, the system attempts to bring all queues up to date.

Quorum queues have support for the handling of poison messages, which are messages that are never consumed completely or positively acknowledged. The number of unsuccessful delivery attempts can be tracked and displayed in the *x-delivery-count* header. A poison message can be dead-lettered when it has been returned more times than configured.

**Introducing RabbitMQ Streams**

*RabbitMQ streams is a new feature available from RabbitMQ version 3.9, and is a completely new part of RabbitMQ, making RabbitMQ an option for a lot of new use cases. We will touch more on this later in the course.*

There are two components of stream functionality, stream queues, and the stream protocol. Stream queues are persistent and replicated, and work similarly to an append-only log. Stream queues can be used with traditional AMQP clients, i.e. they don’t have to use the stream protocol. Messages in stream queues are not removed when consumed, instead, they can be consumed over and over again using an offset or a timestamp, similar to functionalities that can be found in Apache Kafka. RabbitMQ Streams also comes with its very own stream protocol, which has shown to be much faster than AMQP in RabbitMQ.

**Benefits of Using RabbitMQ Streams**

* Large number of consumers can easily consume the same message.
* Messages can be consumed multiple times.
* Messages will stay in queues until they are expired with retention policies.
* High throughput when using the stream protocol.
* Streams can easily store millions of messages without issues (which is not always the case with traditional queue-type messages in RabbitMQ). Alternative to Apache Kafka.

**Lazy Queues**

Queues can become long for various reasons including consumer maintenance or the arrival of large batches of messages. While RabbitMQ can support millions of messages, keeping queues as short as possible is recommended by most experts. Messages are stored in memory by default. RabbitMQ then flushes messages (page out) to free up the RAM usage when the queue becomes too long for the underlying instance to handle. Storing messages in RAM enables faster delivery of messages to consumers than storing them to disk.

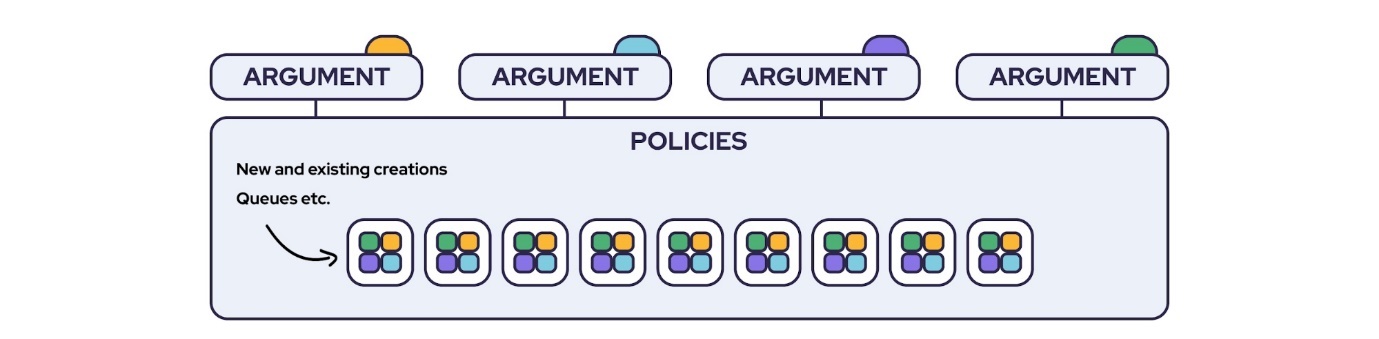
The page out function usually takes time and often stops the queue from processing messages, which deteriorates the queue speed. For this reason, queues that contain a lot of messages can have a negative impact on the broker's performance. Additionally, it takes a lot of time to rebuild the index after a cluster is restarted and to sync messages between Nodes.

Beginning with RabbitMQ version 3.6, a policy called lazy queues was added to enable the storage of messages to disk automatically in order to minimize RAM usage. Lazy queues can be enabled by setting the mode via the queue.declare arguments or by applying a policy to all queues.

**Policies**

In order to ensure uniformly configured queues and exchanges, RabbitMQ (AMQP) includes the ability to define Policies and Arguments. Policies can be advantageously used to apply queue or exchange arguments to more than one created queue/exchange. Policies are created per vhost, with a pattern that defines where it will be applied and a parameter that defines what the policy will do.

The specifications of the AMQP protocol (0.9.1) enable support for various features, called Arguments. Depending on which argument you implement, changes can be made to their settings after the queue declaration. Arguments define certain configurations, such as message and queue TTL, different consumer priorities, and queue length limit. Policies make it possible to configure arguments for one or many queues and exchanges at once, and the queues/exchanges will all be updated when the policy definition is updated. Policies can be changed at any time, and changes will affect all matching queues and exchanges.



When using policies, argument configurations and updates don’t have to be done for every single queue or exchange. Policies simply ensure that all matching queues or exchanges come with the desirable preset arguments, suitable for their purpose, and also ensure that updates of one single argument are applied on all queues or exchanges bound to it.

A policy can be set when you want to apply a TTL on a set of queues. Policies could be used when you want to delete single or multiple queues at once, or when you want to delete all messages from a queue.

**Policies in RabbitMQ**

A policy is applied when the pattern, a [regular expression](https://en.wikipedia.org/wiki/Regular_expression), matches a queue or exchange. As soon as a policy is created it will be applied to the matching queues and/or exchanges and its arguments will be amended to the definitions. As the match occurs continuously changes can easily be applied to multiple queues that are up and running. For example, if a TTL is to be set on a group of queues, or if multiple queues are to be deleted or purged at once. A policy is also applied every time an exchange or queue is created if a match exists. Only one policy can be matched to every queue or exchange at once, but one policy may be set to apply multiple arguments.

Policies are created per vhost, with a pattern that defines where it will be applied and a parameter that defines what the policy will do. The parameter is entered as a key (the parameter name) and a value (the parameter value), also called a key-value pair. Policies can be set from a terminal using rabbitmqctl or by using the HTTP API or Web Management Interface.

**Create and View Policies**

To create a policy, define the following:

* Name of the policy
* The vhost it lives in, default is /
* A pattern or exact match to determine which queues or exchanges it applies to
* A definition consisting of one or several key-value pairs
* A priority, how it will be applied in relation to other policies, default is 0

Policies can be viewed and created from either of the following:

* The RabbitMQ management interface
* A terminal using rabbitmqctl
* The HTTP API

**Policies in the RabbitMQ Management Interface**

Policies are listed under “policies” in the RabbitMQ menu. On the same page in the Add/update section, a new policy can be created.

The name, patterns, and definition fields are mandatory. Below the definitions box, a selection of keys is listed that can be added to the policy by clicking them. Values have to be added to the definitions box for every key added.

A priority should be used if multiple policies are used where the patterns overlap.

**Policies in rabbitmqctl**

Policies can be listed and created via rabbitmqctl. Here are some examples:

**List Policies**

List policies by using the command:

rabbitmqctl list\_policies -p vhostname

**Create Policies**

Create a policy by using the command

rabbitmqctl set\_policy <name> <pattern> <definition>

**Example**

rabbitmqctl set\_policy Policy2 '.\*' '{"message-ttl": 60}'

To specify vhost, priority, and exchange/queue-application use the following syntax

rabbitmqctl set\_policy <name> <pattern> <definition> -p <vhost> --apply-o <queues|exchanges> --priority 8

**Example**

rabbitmqctl set\_policy Policy5 'queue\_A' '{"message-ttl": 60}' -p zyxhvjpx --apply-to queues --priority 10'

**Delete Policies**

Deleting policies is done with the command:

rabbitmqctl clear\_policy <name>

**Example**

sudo rabbitmqctl clear\_policy 'Policy1'

**Policies With the HTTP API**

The following commands are available for policies via RabbitMQ HTTP API.

**List Policies**

Policies can be listed with the following API call

curl -u USERNAME:PASSWORD -X GET https://SERVERNAME.amq.cloudamqp.com/api/policies

**Create Policies**

Policies can be added with the following API call

curl -XPUT -u USERNAME:PASSWORD --header "Content-Type: application/json" --data '{"pattern":"[PATTERN]","definition":{"key":value},"apply-to":"[queues|exchanges]"}' https://host/api/policies/[VHOST]/[POLICYNAME]

**Delete Policies**

Policies can be deleted with the following API call

curl -u USERNAME:PASSWORD -X DELETE https://host/api/policies/[VHOST]/[

**Consumer Acknowledgment**

Messages in transit between RabbitMQ and the consumer might get lost and have to be re-sent. This can occur when a connection fails or during other events that disconnect the receiving service (consumer) from RabbitMQ. The use of consumer acknowledgements assures that messages have been delivered. When it comes to message publishing, a publish confirm is the same concept.

Consumer acknowledgements (acks) provide notice if and when messages between RabbitMQ and the consumer have been sent to or received and handled by the intended destination. The response of a consumer acknowledgement setting can also trigger actions that may re-queue and resend messages if needed. There are many ways of using consumer acks, depending on the desired result.

If for some reason, the consumer cannot process a message, the use of consumer acknowledgement can prevent a message loss by letting another available consumer receive and handle it, or retrying the process at a later time. Once an acknowledgement is received by RabbitMQ, the message can be discarded from the queue.

For RabbitMQ to securely deliver messages to a consumer, the RabbitMQ broker needs to understand when a message is considered successfully sent. A message can be considered successfully delivered in two ways, **Manual Acknowledgement**, and **Automatic Acknowledgement**.

**Manual Acknowledgement in RabbitMQ**

One way for RabbitMQ to receive message delivery confirmation is when an explicit acknowledgment is received from the consumer. This means that RabbitMQ considers the message to be successfully sent once it has been consumed and acked by a consumer. Manual Acks are the default setting for consumer acknowledgements in RabbitMQ.

**Automatic Acknowledgment in RabbitMQ**

Another way for RabbitMQ to be noticed about message delivery is via Automatic Acknowledgement. This means that Acks are received once the message is sent out and written to a TCP socket. RabbitMQ considers the message to be successfully sent once it has left the broker.

Notice that Automatic Acknowledgement should be considered unsafe, as an unexpected message loss in case of connection failure to the consumer might result in a message being dropped after leaving RabbitMQ. Because of this, this setup is not suitable for all workloads.

**Publisher Confirm**

Messages in transit might get lost if the broker goes down before RabbitMQ has the chance to receive it. This can happen in the event of a connection failure or during a broker restart. Sometimes you want these messages to be retransmitted from the publisher to the RabbitMQ server. *Publisher Confirms* informs the client of when to retransmit messages.

RabbitMQ can acknowledge that a message has been received, with the use of Publisher Confirms. This should be used when you cannot afford to lose messages and when you want to ensure that RabbitMQ has received the message sent by the publisher.

**When to Use RabbitMQ Streams**

Queues in RabbitMQ are great! They anchor the communication between producers and consumers. Replicated queues (e.g. [Quorum Queues](https://training.cloudamqp.com/course/44)) even orchestrate communication with reliability and data safety. However, there are scenarios where queues fall flat or crawl on their knees. What scenarios?

Queues are limited in the following scenarios:

* They deliver the same message to multiple consumers by binding a dedicated queue for each consumer. Clearly, this could create a scalability problem.
* They erase read messages making it impossible to re-read(replay) them or grab a specific message in the queue.
* They perform poorly when dealing with millions of messages because they are optimized to gravitate toward an empty state.

The RabbitMQ team introduced Streams in [RabbitMQ 3.9](https://www.rabbitmq.com/streams.html) to mitigate the above-listed challenges. But what are RabbitMQ Streams? This article will explore Streams in RabbitMQ, and how they solve the problems described above. The subsequent post in this series will also look at how to get started with Streams.

**What are RabbitMQ Streams?**

RabbitMQ Streams basically perform the same tasks as queues in that they buffer messages from producers that are read by consumers. However, Streams differ from queues in two ways:

* How producers write messages to them
* And how consumers read messages from them

Under the hood, Streams model an append-only log that's immutable. In this context, this means messages written to a Stream can't be erased, they can only be read. A more scholarly description would be to call this behavior of Streams “non-destructive consumer semantics”.

To read messages from a Stream in RabbitMQ, one or more consumers subscribe to it and read the same message as many times as they want. Additionally, Streams are always persistent and replicated.

Like queues, consumers talk to a Stream via AMQP-based clients and, by extension, use the AMQP protocol. Alternatively, consumers can also connect to a Stream via the binary stream protocol. The stream protocol fosters faster message flow when working with RabbitMQ Streams.

All these unique sets of characteristics compound to make Streams in RabbitMQ a dramatic shift from queues. The Stream wasn’t created to replace queues but to complement them. Streams open up endless possibilities for new RabbitMQ use cases like the scenarios identified earlier.

Let’s explore these use cases a little bit deeper.

**When to Use RabbitMQ Streams**

The use cases where streams shine include:

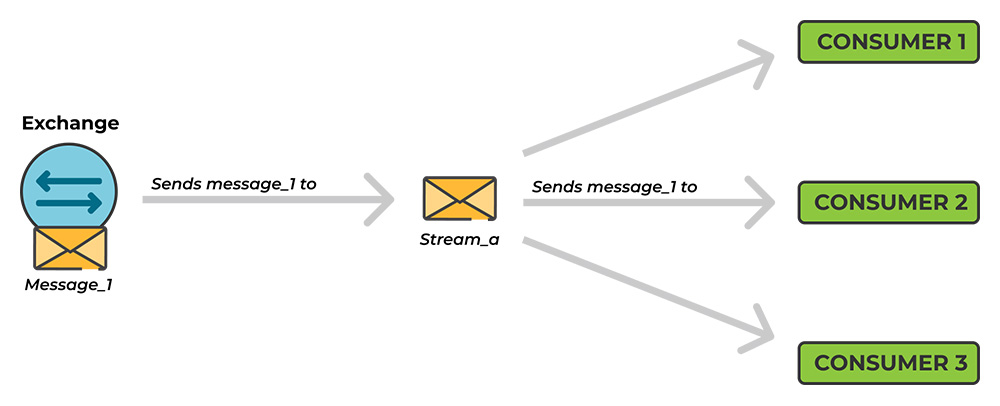
* **Fan-out architectures**: Where many consumers need to read the same message
* **Replay & time-travel**: Where consumers need to read and reread a message from any point in the stream.
* **Large Volumes of Messages**: Streams are great for use cases where large volumes of messages need to be persisted.
* **High Throughput**: RabbitMQ Streams process relatively higher volumes of messages per second.

**Fan-out Architectures**

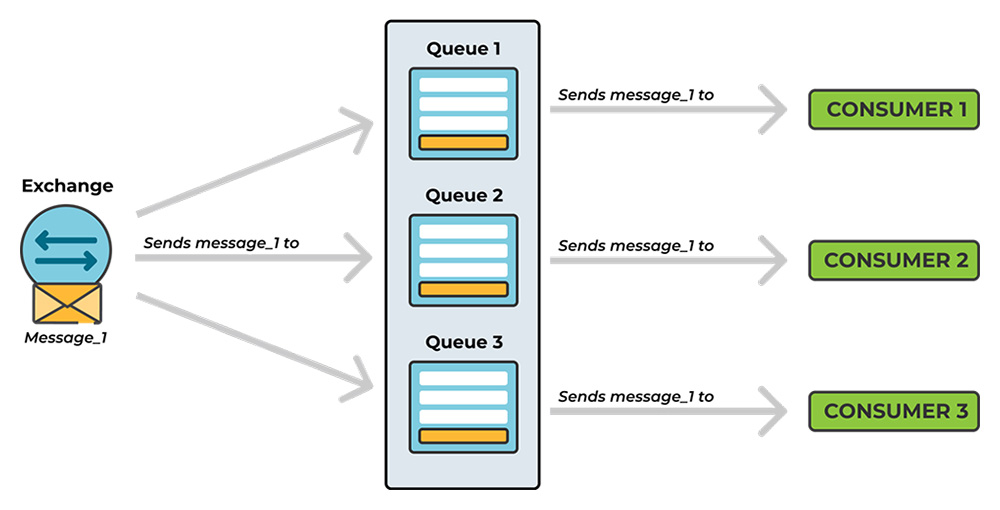
A fan-out architecture is where multiple consumers read the same message. As mentioned earlier, implementing this sort of architecture with queues isn’t optimal. Having to add queues for every added consumer is resource intensive, which gets worse when dealing with queues that need to persist data.

Streams in RabbitMQ make implementing fan-out architectures a breeze. Because consumers read messages from a Stream in a non-destructive manner, a message will always be there for the next consumer to access it. In essence, to implement a fan-out architecture, just declare a RabbitMQ Stream and bind as many consumers as needed.

The image below depicts what implementing a fan-out with a Stream would look like.



Conversely, trying to achieve the same thing with queues would look like what’s shown in the image below.

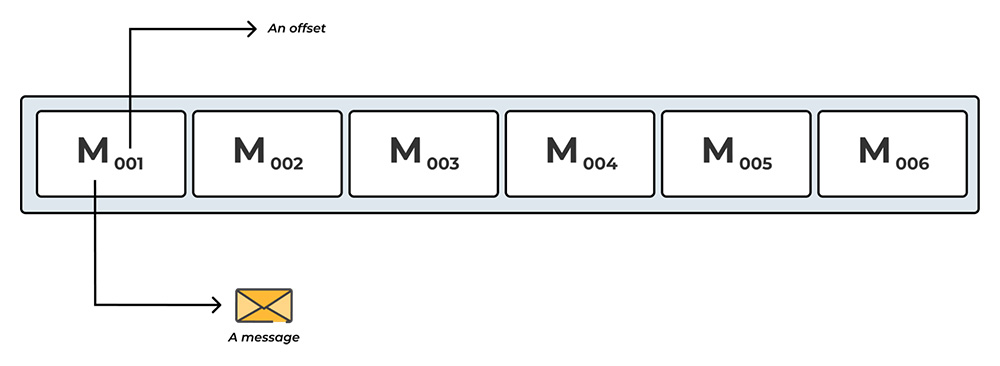


**Replay & Time Travel**

RabbitMQ Streams are also fit for use cases where a consumer needs to re-read the same message, something that isn't possible with queues. Aside from re-reading the most recent message, it is also possible to re-read a message from any point in the Stream.

This easy replay and time-travel feature of Streams is made possible with offsets. Offsets are to Streams what indexes are to arrays or keys to hash maps. To grab a specific message in a Stream, just specify its offset in the consumer query.

For example, this is what messages and their corresponding offsets would look like in a given Stream.



**Large Volumes of Messages**

RabbitMQ Streams are perfect when persisting large volumes of messages. Streams shine in this area because they store messages on the file system. As a result, a Stream in RabbitMQ could grow indefinitely until the host disk space runs out.

As running out of disk might not be a desirable behavior, RabbitMQ Streams allow setting a maximum log data size. When the upper limit is reached, the oldest messages are discarded preventing the Stream from consuming the entire disk space.

**High Throughput**

If a RabbitMQ use case requires processing high volumes of messages per second, then using a Stream is the best option.

In a comparison Quorum queues handled about 40,000 messages per second. Streams, on the other hand, handled around 64,000 messages per second when used with AMQP protocol, and over 1 million messages per second when used with native Stream protocol.

This high throughput is a testament to the simplicity of the Stream data structure and the Stream protocol itself. For example, since the Stream protocol doesn’t handle things like routing messages, de-queueing messages, etc., it technically does less work, and this translates to higher performance.

**Summary**

This is a basic understanding of how the streaming and replay features that come with Streams in RabbitMQ 3.9 open up possibilities for new RabbitMQ use cases. They make it easier to implement fan-out architectures and to read and re-read old messages, amongst other things.

[Shovel](https://www.cloudamqp.com/docs/shovel.html) is a plugin for RabbitMQ that enables you to define replication relationships between brokers. It can, for example, be used to balance the load of a queue or when you need to be able to take messages out of one RabbitMQ broker and insert them into another.

Shovel repeatedly pushes messages from a source to a destination using a configurable number of consumers. A backup can take the place of an unreachable broker with minimal loss of data.

Federation is used to transfer messages between brokers. It will transfer messages from one broker called the upstream to another broker called the downstream.

The RabbitMQ Federation plugin has several use-cases; collect messages from multiple clusters to a central cluster, distribute the load of one queue to multiple other clusters, and when migrating to another cluster without stopping all producers/consumers while doing so.

There are two ways of configuring a federation, via queue federation or exchange federation.

* **Queue federation** is used when you want to transfer messages from a queue to another queue on another node or cluster, i.e. messages are moved and not copied. With queue federation, messages are only transferred where there are underutilized consumers, always preferring local consumers to remote ones. With exchange federation messages are always federated, no matter where the consumers are connected.
* **Exchange federation** is used when you want to copy messages from an exchange to another exchange on another node or cluster. Exchange federation will pass on bindings from the downstream to the upstreams when possible, whereas queue federation will not.

**Consistent Hash Exchange Type**

The consistent hash exchange type is an exchange type available in RabbitMQ. It distributes messages among queues bound to it, while still ensuring that messages of a certain character are sent to the same queue, assuming no bindings have changed.

Message distribution depends on the computed hash of the routing key or header value. Queues are bound with an integer-based weight rather than a routing key or header values. This weight is used as part of the algorithm that determines the delivery of the message.

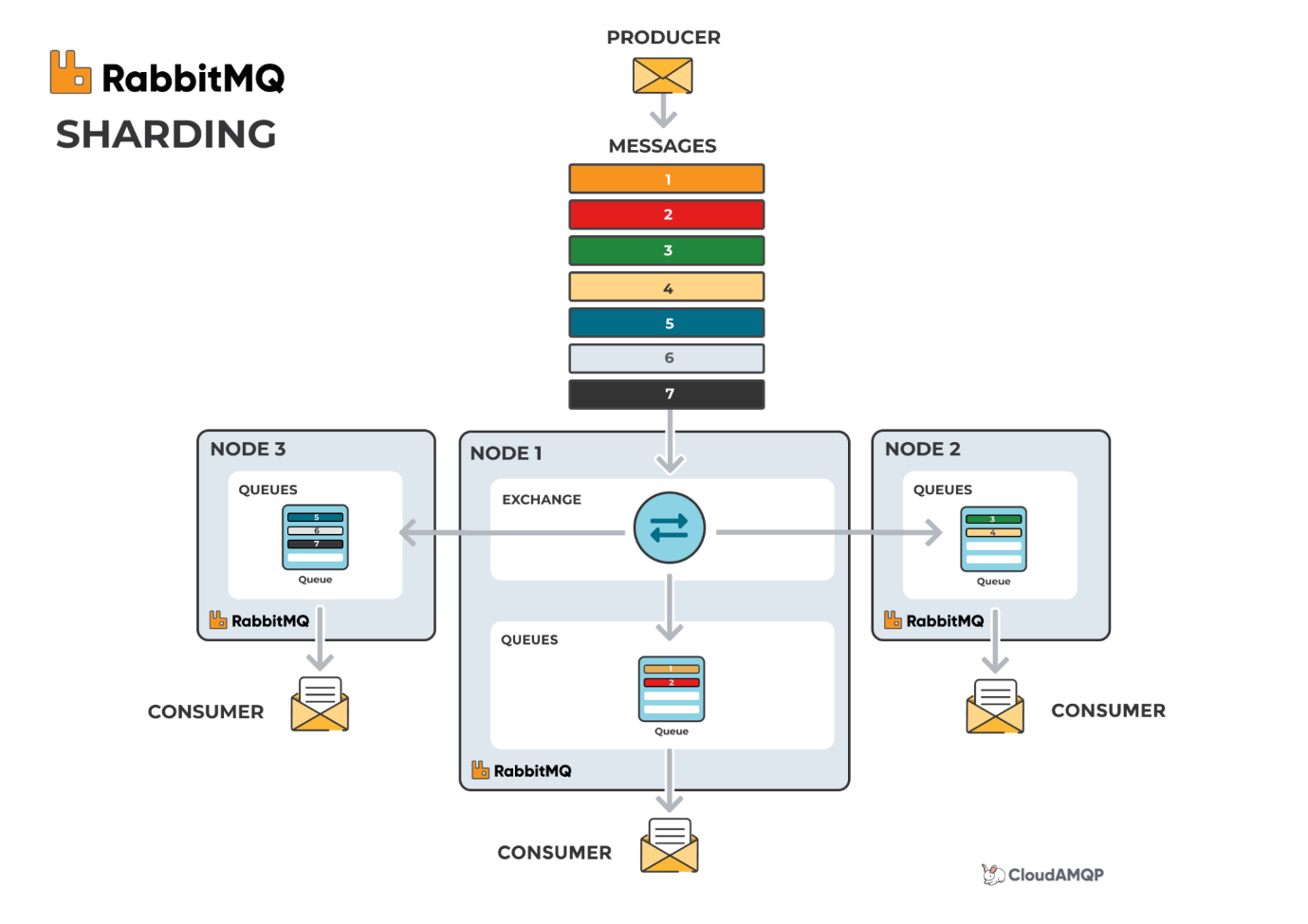
With the [RabbitMQ Consistent Hash Exchange Type Plugin](https://github.com/rabbitmq/rabbitmq-server/tree/main/deps/rabbitmq_consistent_hash_exchange), the broker itself is used as the unit of parallelism instead of the consumers.

The consistent hash exchange type can be used to load-balance messages between queues when causal message ordering is still needed. Manually distributing messages can be hard, since publishers aren’t connected with the information on the number of queues and bindings.

**Sharding Plugin**

The [RabbitMQ Sharding Plugin](https://github.com/rabbitmq/rabbitmq-server/tree/master/deps/rabbitmq_sharding) plugin makes it easy to scale messages among queues. It allows the number of messages in your system to grow by distributing a queue among multiple nodes and cores. Sharding is performed by exchanges. Exchanges can be defined as sharded and then the routing keys ensure an even distribution of messages among queues. The plugin hashes the incoming routing key sent to a new x-modulus-hash exchange. A modulus function applied to the resulting hash determines which shard receives the incoming message.

A queue is spread among multiple actual queues. The plugin expects you to run a consumer per shard with new nodes being automatically incorporated. Note that it’s important to consume from all queues.



*Image description: Sharding takes place in the exchange. Messages sent from a producer are, via the exchange and the routing key, distributed equally among queues in other nodes. The messages are then handled by separate consumers.*

**Priority Queues**

Priority Queues is a plugin useful for exactly what it sounds like: setting different priorities and letting each message be handled according to its importance.

A queue's priority range needs to be applied when the queue is created. A priority queue is declared via the *x-max-priority* queue argument, where this argument should be an integer between 1 and 255.

The priority of each message is specified when the message is published in the queue. Larger numbers indicate higher priority, and messages with higher priority will be delivered before messages with less priority. Publishers specify message priority using the priority field in message properties. Messages without a priority property are treated as if their priority were 0.

channel.basic\_publish(properties=pika.BasicProperties(priority=your\_priority),

exchange=...,

routing\_key=...,

body=...)

**Queue Performance Optimization**

The following are some advise on what you should try to do, in order to optimize the performance of your RabbitMQ queues:

**1. Use Quorom Queues**

Historically, classic queues supported replication via classic [mirrored queues](https://training.cloudamqp.com/course/43), but this should be avoided. Using a variant of the [Raft protocol](https://www.cloudamqp.com/blog/quorum-queues-internals-a-deep-dive.html#introduction-to-raft), a distributed consensus algorithm, [**quorum queues**](https://training.cloudamqp.com/course/44)**are both safer and achieve higher throughput than mirrored queues**.

**2. Keep Queues Short**

**Short queues are the fastest**. A message published to an empty queue will go straight to the consumer, as soon as the queue receives the message. However, keep in mind that a persistent message in a durable queue will go to the disk. **The recommendation is to keep fewer than 10,000 messages in one queue**.

Queues should be kept short because many messages in a queue can put a heavy load on RAM usage. In order to free up RAM, RabbitMQ starts flushing (*page out*) messages to disk. The process usually takes time and blocks the queue from processing messages when there are many messages to page out. A large number of messages might have a negative impact on the broker since the process deteriorates queuing speed.

It is also time-consuming to restart a cluster with many messages because the index must be rebuilt and it takes time to sync messages between nodes in the cluster.

**3. Enable Lazy Queues for Long Queues**

Queues can become long for various reasons; consumers might crash, they might be offline due to maintenance or they might simply be working slower than usual. [**Lazy queues**](https://training.cloudamqp.com/course/46)**support long queues with millions of messages**. They were introduced in [RabbitMQ version 3.6](https://github.com/rabbitmq/rabbitmq-server/releases/tag/rabbitmq_v3_6_0) and they write messages to disk immediately. Thus spreading the work over time instead of taking the risk of a performance spike somewhere down the line. It provides a more predictable and smooth performance without sudden drops, but at a cost. Messages are only loaded into memory when needed, thereby minimizing the RAM usage but increasing the throughput time.

**Lazy queues gives you a more predictable performance and we recommend using them when you know that you will have large queues from time to time.** Situations where, many messages are sent at once (e.g. when processing batch jobs) or where there is a risk that the consumers will not be able to keep up with the speed and consistency of the publishers, is best handled using lazy queues.

**Disable lazy queues if high performance is required, if queues are always short, or if a max-length policy exists**.

**4. Limit Queue Size With Max-length or TTL**

Another recommendation for applications that often get hit by spikes of messages, and where throughput is more important than anything else, is to **set a max-length on the queue**. This keeps the queue short by discarding messages from the head of the queue so that it never becomes larger than the max-length setting.

**You can also limit queue size with TTL (Time-To-Live)**. Declaring a queue with the x-message-ttl property means that messages will be discarded from the queue if they haven’t been consumed within the specified time.

**5. Consider the Number of Queues**

Queues are single-threaded in RabbitMQ. Achieve better throughput on a multi-core system using multiple queues and multiple consumers. **Optimal throughput is achieved with as many queues as cores on the underlying node(s)**.

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster. This might slow down the server if there are thousands upon thousands of active queues and consumers. The CPU and RAM usage may also be negatively affected by too many queues.

**6. Split Queues Over Different Cores**

A RabbitMQ queue is bounded to a single core. **Better queue performance is achieved if your queues are split over different cores and into different nodes**. It is possible to manually split queues evenly between nodes, but the downside is having to keep track of where each queue is located. Two plugins that will help you spread messages are:

**RabbitMQ Consistent Hash Exchange Type**

The [RabbitMQ Consistent Hash Exchange Type Plugin](https://training.cloudamqp.com/course/31) allows an exchange to load balance messages between queues. Messages sent to the exchange are distributed consistently and equally across many queues based on the routing key of the message. The plugin creates a hash of the routing key and spreads the messages between queues that have a binding to that exchange. Performing this manually would be almost impossible.

The plugin is also used to get the maximum use of many cores in the cluster. *Note that it is important to consume from all queues*.

**RabbitMQ Sharding**

The [RabbitMQ Sharding Plugin](https://training.cloudamqp.com/course/32) automatically *shards* (spreads) messages over queues. Sharding is performed by specifying an exchange as sharded. Messages sent to that exchange will be partitioned across "shard"-queues. By sharding, the partitioning over queues will be done automatically for you. The supporting queues are then automatically created on every cluster node, and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer, but many queues could be running in the background.

The plugin is a centralized place to send messages with the goal of *load balancing*, by adding queues to the other nodes in the cluster.

**7. Don’t Set Names on Temporary Queues**

Setting a queue name is important when sharing the queue between producers and consumers, but it is not when using temporary queues. Instead, **allow the server to choose a random queue name or modify the RabbitMQ policies**.

A queue name starting with amq. is reserved for internal use by the broker.

**8. Auto-delete Unused Queues**

Client connections can fail and potentially leave unused queues behind, which might affect the performance of the system. **There are three ways to have queues deleted automatically**:

* The first option is to set **a TTL policy** on the queue. For example, a TTL policy of 28 days will delete queues with no messages consumed from them in the last 28 days.
* Another option is **an auto-delete queue**, that gets deleted when its last consumer has canceled or when the channel/connection is closed, alternatively, when it has lost the TCP (Transmission Control Protocol) connection with the server.
* Finally, **an exclusive queue** is only used (consumed from, purged, deleted, etc.) by its declaring connection. Exclusive queues are deleted when their declaring connection is closed or gone due to underlying TCP connection loss or other circumstances.

**9. Set Limited Use on Priority Queues**

Queues can have zero or more priority levels. Remember that each priority level uses an internal queue on the Erlang VM, which means that it takes up resources. **In most use cases, having no more than five priority levels is sufficient**, which keeps resource use manageable.

**10. Consider the Payload**

*"How do we handle the payload size of messages sent to RabbitMQ?"*, is a common question developers ask. The answer is, of course, to avoid sending huge files in messages, but also to keep in mind that the rate of messages per second can be a larger bottleneck than the message size itself. Sending multiple small messages might be a bad alternative. The better approach could be to bundle the small messages into one larger message and let the consumer split them up. However, bundling multiple messages might affect the processing time. If one of the bundled messages fails, will all of them need to be reprocessed? **Bandwidth and architecture should dictate the best way to set up message queues, while considering the payload**.

**Connections and Channels**

These are some tips on how to handle connections and channels to avoid bad performance and strain on the RabbitMQ server:

**1. Don’t Use Too Many Connections or Channels**

**Try to keep connection/channel count low**. Each connection uses about 100 KB of RAM, even more if TLS (Transport Layer Security) is used. Thousands of connections can become a heavy burden on a RabbitMQ server. In a worst-case scenario, the server can crash due to running out of memory.

**Avoid connection and channel leaks**. Connection leaks can cause RabbitMQ to run out of memory. Make sure your clients are not leaking connections. *If you have more than 10 connections from the same host, you may have a connection leak*.

Also, a large number of connections and channels can have an negative effect on the performance of the RabbitMQ management interface and make it slow to work with. Metrics must be collected, analyzed, and displayed for every connection and channel that consumes server resources.

**2. Don’t Share Channels Between Threads**

Most clients don't make channels thread-safe because it would have a severe negative impact on performance, which is why **it is not recommended to share channels between threads**.

**3. Don’t Open and Close Connections or Channels Repeatedly**

**Repeatedly opening, and closing channels will create a higher latency because more TCP packets have to be sent and received**. The handshake process for an AMQP connection is actually quite involved and requires at least seven TCP packets, even more if TLS is used. It is recommended that each process only create one TCP connection with multiple channels, in that connection, for different threads. Connections should be long-lived so channels can be opened and closed more frequently if required. Even channels should be long-lived if possible.

Do not open a channel every time a message is published. **Best practice includes reusing connections and multiplexing a connection between threads with channels, when possible**.

RabbitMQ is optimized for long lived connections. Each connection establishment is pretty heavy and uses many TCP packets. Keep connections open if you are able to. If you have a client that is unable to keep connection long lived you can use AMQP Proxy.

The amount of TCP packets that will be sent/recieved in different situations:

* **AMQP connections:** 7 packets
* **AMQP channel:** 2 packets
* **AMQP publish:** 1 packet (more for larger messages)
* **AMQP close channel:** 2 packets
* **AMQP close connection:** 2 packets

**» Total 14-19 packets (+ Acks)**

**4. Separate Connections for Publisher and Consumer**

**For the highest throughput, you should separate the connections for publisher and consumer**. Unless the connections are separated between publisher and consumer, messages may not be consumed. This is especially true if the connection is in flow control, constricting the message flow even more.

Another thing to remember is that RabbitMQ may cause back pressure on the TCP connection when the publisher sends too many messages to the server. When consuming on the same TCP connection, the server might not receive the message acknowledgments from the client, affecting the performance of message consumption and the overall server speed.

**5. Consider VPC Peering**

Connecting to RabbitMQ over AMQPS, wraps the AMQP protocol in TLS. TLS has a performance impact since all traffic must be encrypted. **For maximum performance, use**[**VPC (Virtual Private Cloud) peering**](https://www.cloudamqp.com/blog/amazon-vpc-peering.html) instead, which encrypts the traffic without involving the AMQP client/server.

CloudAMQP configures the RabbitMQ servers to accept and prioritize fast, but secure, encryption ciphers.

**Dead Lettering**

Some messages become undeliverable or unhandled even when received by the broker. This can happen when the amount of time the message has spent in a queue exceeds the time to live, TTL, when the queue reaches its capacity, or when a message is negatively acknowledged by the consumer. Such a message is called a dead message.

But don’t worry, there is no need to lose messages entirely. Setting up a RabbitMQ dead letter exchange and a dead letter queue allows orphaned messages to be stored and processed. Dead letter handling is a key part of almost any messaging system, and RabbitMQ is no different.

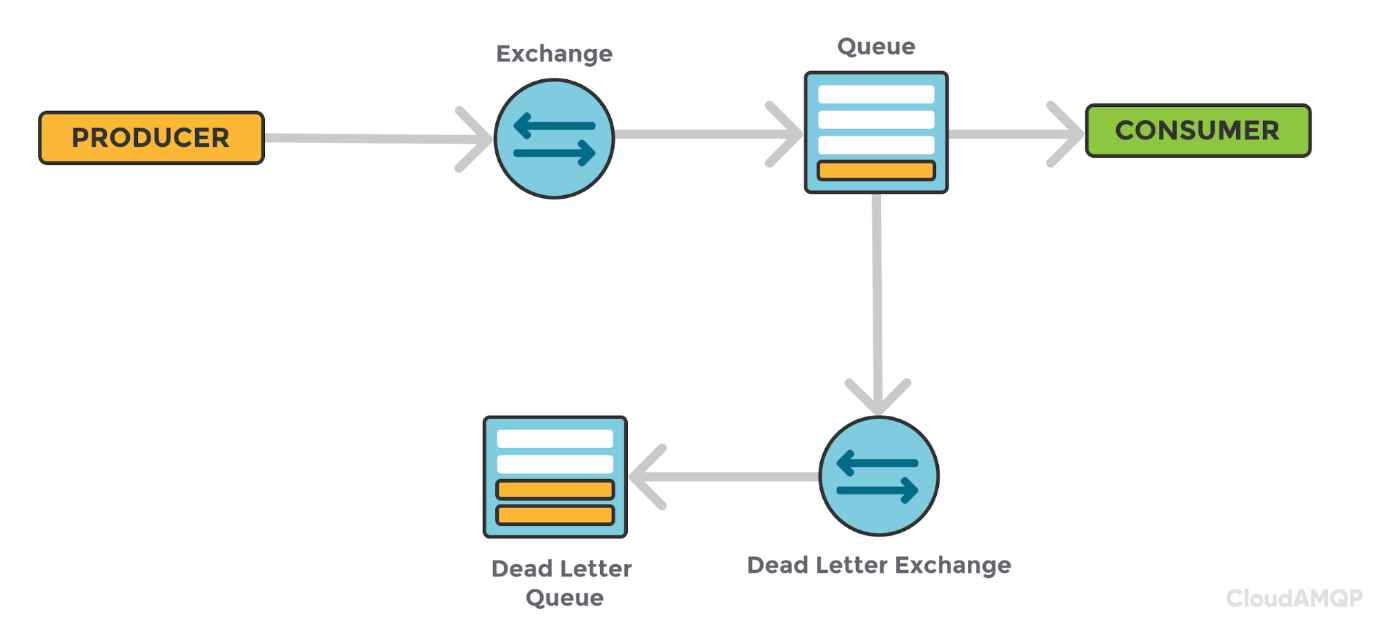
**When Is a Message Considered Dead?**

There are three identified situations where a message becomes undeliverable after reaching RabbitMQ:

* A message is negatively acknowledged by the consumer
* The TTL of a message expires
* The queue reaches capacity

By default, the broker drops these messages. Publishing is successful, however, the RabbitMQ consumer never handles or has a chance to handle the message successfully.

Please note that messages that are dropped by the exchange or messages that are not routed need to be handled by the alternate exchange.



**Why Use a Dead Letter Exchange?**

Queues attached to a dead letter exchange collect dropped messages, with the next steps determined by you. In other words, it's up to you to decide how to handle messages in the dead letter queue. When implemented correctly, information is almost never lost.

Attach a dead letter exchange when you know that you have messages that might be nacked, but still need to be handled. When you can’t lose messages with an expiring TTL or when the queue might reach its capacity.

**Setting up a RabbitMQ Dead Letter Exchange**

Dead letter exchanges are no different than other exchanges.

Simply specify the exchange normally and declare it as a backup for a queue:

channel.exchange\_declare("dlx\_exchange", "direct")

channel.queue\_declare("test\_queue", arguments={

"x-dead-letter-exchange": "dlx\_exchange", "x-dead-letter-routing-key": "dlx\_key"})

The x-dead-letter-exchange parameter tells the *test\_queue* to use the *dlx\_exchange* for dead messages. Notice how the exchange is not dedicated to a single queue.

**Creating and Binding RabbitMQ Dead Letter Queues**

Just as with a dead letter exchange, a dead letter queue is a regular queue in RabbitMQ; it is just attached to the exchange.

Create the queue normally and attach it to the exchange:

channel.queue\_declare("dead\_letter\_queue")

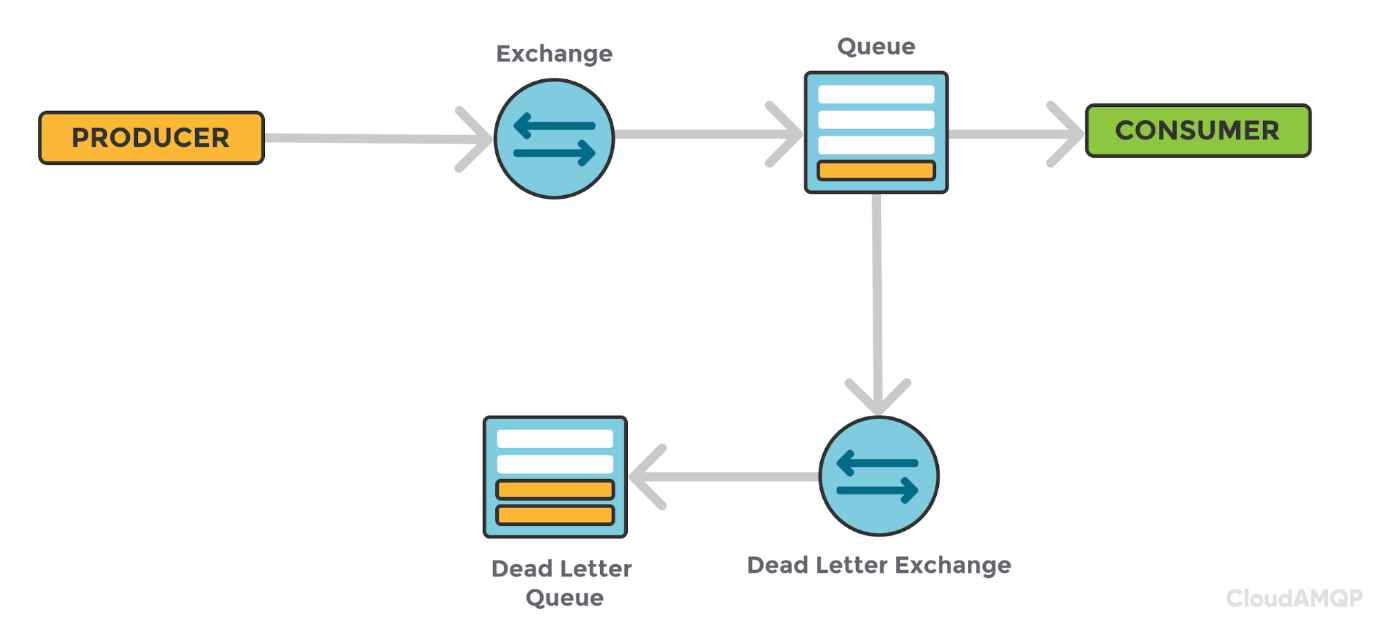
If desired, specify a new routing key:

channel.queue\_bind("dead\_letter\_queue", "dlx\_exchange", "dlx\_key")

The x-dead-letter-routing-key changes the routing key from the one used in the original message to *dlx\_key*. This allows you to redirect dead letter messages from multiple queues to the same dead letter queue.

**Retrying Dead Letters**

You can create a redelivery mechanism using dead letter queues by attaching the original exchange to the dead letter queue. However, it is possible to create an endless cycle of redelivered messages which could clog dead letter queues if left unchecked.



Create a separate dead letter queue that stores messages before pushing them to an exchange or routing them back to the original queue.

To further improve the system, include an incremented property in the message body indicating the number of times the message was received. This requires handling dead letters in a separate consumer but allows you to eventually drop messages or push them to storage.

**Alternate Exchange**

No matter how careful you are, mistakes can happen. For example, a client may accidentally or maliciously route messages using non-existent routing keys. To avoid complications from lost information, collecting unroutable messages in a RabbitMQ alternate exchange is an easy, safe backup.

**Issues With Unrouted Messages**

Unroutable messages are an issue. They slow down processing times if applications make multiple attempts at delivery or are kept busy by continually logging them. Additionally, there is the problem of what to do with unrouted messages, which must eventually be handled or dropped.

Whether messages return or not, unroutable messages can:

* Be returned to a broken application that constantly resends them
* Be the result of malicious activity aimed at causing RabbitMQ to become unresponsive
* Cause the loss of mission-critical data

As an example, think of a hospital that uses RabbitMQ to help store patient information in a database. Medical devices typically push vital data to a small server for processing, which then forwards the information to the cloud for storage. In this case, RabbitMQ can be used to push data to the cloud and be employed to facilitate streaming.

Should messages drop before they end up in RabbitMQ, doctors could lose vital information. This loss could impact diagnosis and treatment. The situation could become even worse if the system breaks and overloads the resources handling messages related to more than one patient.

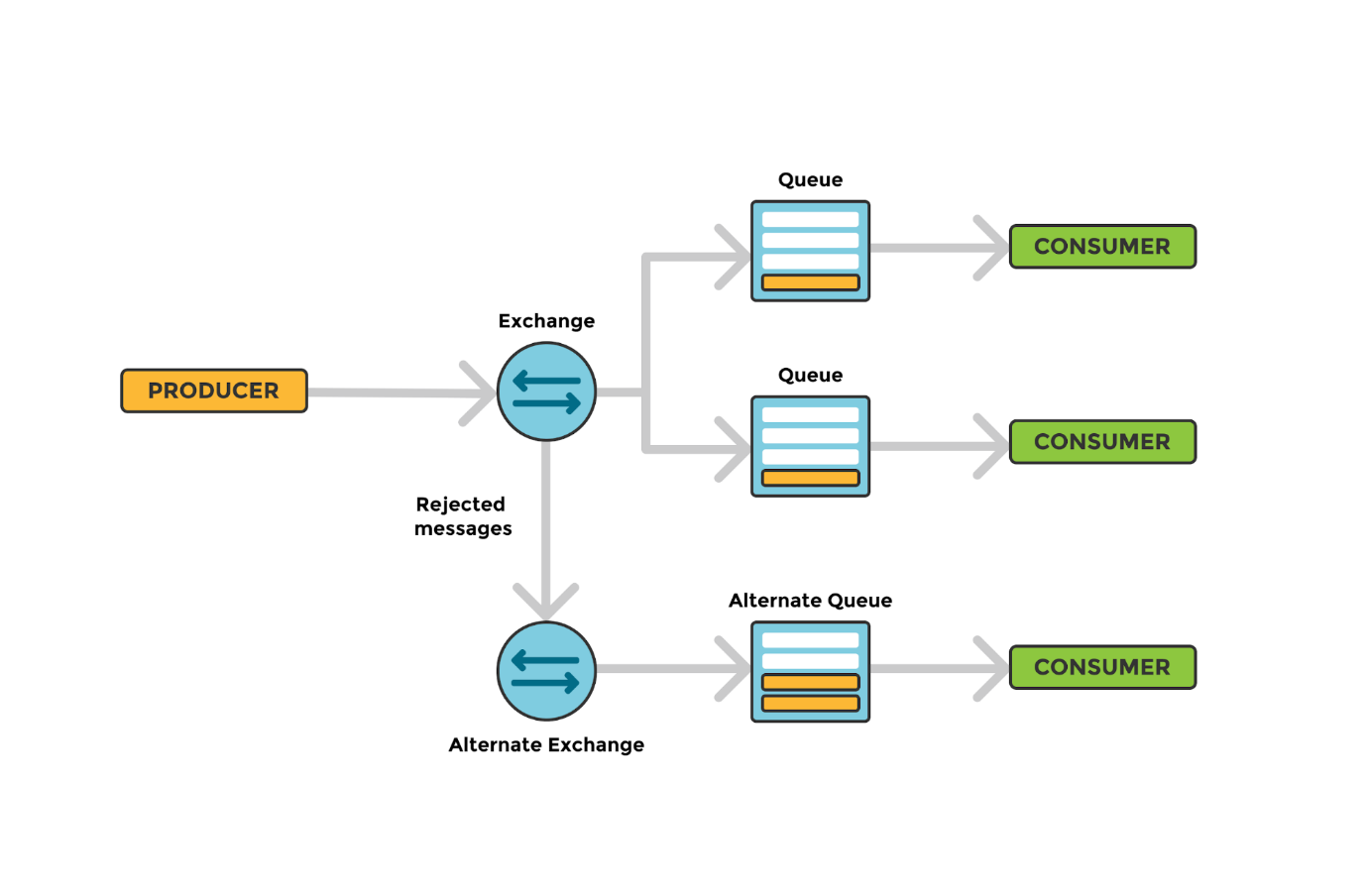
**What Happens to Unroutable Messages in RabbitMQ?**

It is possible to avoid the complete loss of a message. RabbitMQ handles unroutable messages in two ways based on the mandatory flag within the message header. The server either returns the message when the flag is set to *true* or silently drops the message when set to *false*.

Applications can log returned messages, but logging does not provide a mechanism for dealing with an unreachable exchange or queue.

**Avoid Data Loss With a RabbitMQ Alternate Exchange**

RabbitMQ also lets you define an alternate exchange to apply logic to unroutable messages. Set an alternate exchange using policies or within arguments when declaring an exchange.



The alternate exchange attaches to one or more primary exchanges. [Set the exchange type to fanout](https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html#fanout-exchange) to ensure that rejected messages always route to the alternate queue. Be advised that there is no way to catch invalid exchange names outside of the mandatory flag.

**Creating the Alternate Exchange**

Defining an alternate exchange is no different than defining any other part of your topology. The primary exchange forwards the message to the alternate exchange, which sends it to the alternate queues.

Create a fanout exchange and attach a queue:

conn\_str = os.environ["AMQP\_URI"]

params = pika.URLParameters(conn\_str)

connection = pika.BlockingConnection(params)

channel = connection.channel()

channel.exchange\_declare("alt\_exchange", "fanout")

channel.queue\_declare("alt\_queue")

channel.queue\_bind("alt\_queue", "alt\_exchange")

Make sure that the alternate queue is attached to the exchange through the RabbitMQ API. You can use the API provided by RabbitMQ to list bindings:

Open <http://localhost:15672/api/exchanges/test_vhost/primary_exchange> in your browser

Change the URL to match your host and port. Additionally, enable the RabbitMQ management console.

Any message sent to the alt\_exchange winds up in the alt\_queue. Since we use a fanout exchange, the message ends up in all queues attached to alt\_exchange.

**Attaching the Alternate Exchange**

Use policies, the UI, or message headers to set the alternate exchange. You must use the command line to attach an alternate exchange after declaring the primary exchange.

You can use message arguments to set the exchange:

args = {"alternate-exchange": "alt\_exchange"}

channel.exchange\_declare("primary\_exchange", "direct", arguments=args)

Alternately, use the command line to specify the alternate exchange:

rabbitmqctl set\_policy AE “primary\_\*” ‘{“alternate-exchange”: “alt\_exchange”}’

RabbitMQ sends all messages from the primary\_exchange to the alt\_exchange when the routing key is invalid. Unrouted messages end up in the alt\_queue.

**What Happens When the Mandatory Flag Is Set With an Alternate Exchange?**

There is still a chance that messages won’t be routed if an alternate exchange is provided. The service may be unreachable, or the alternate queue may not be specified correctly. You might accidentally specify a non-existent exchange, for example.

Setting the mandatory flag will catch these unrouted messages. Keep in mind that, if the message routes to the alternate exchange, RabbitMQ marks the message as delivered.

**Prefetch**

Understanding how to optimize the RabbitMQ prefetch count maximizes the speed of the system.

The RabbitMQ prefetch value is used to specify how many messages are being sent at the same time.

Messages in RabbitMQ are pushed from the broker to the consumers. The RabbitMQ default prefetch setting gives clients an unlimited buffer, meaning that RabbitMQ, by default, sends as many messages as it can to any consumer that appears ready to accept them. It is, therefore, possible to have more than one message "in-flight" on a channel at any given moment.



Messages are cached by the RabbitMQ client library (in the consumer) until processed. All pre-fetched messages are invisible to other consumers and are listed as unacked messages in the RabbitMQ management interface.

An unlimited buffer of messages sent from the broker to the consumer could lead to a window of many unacknowledged messages. Prefetching in RabbitMQ simply allows you to set a limit of the number of unacked (not handled) messages.

There are two prefetch options available, *channel prefetch count* and *consumer prefetch count*.

**Channel Prefetch Count and Consumer Prefetch Count**

The channel prefetch value defines the maximum number of unacknowledged deliveries that are permitted on a [channel](https://www.rabbitmq.com/confirms.html#channel-qos-prefetch). Setting a limit on this buffer caps the number of received messages before the broker waits for an acknowledgment.

Because a single channel may consume from multiple queues, coordination between them is required to ensure that they don’t pass the limit. This can be a slow process especially when consuming across a cluster, and it is not the recommended approach.

The best practice is to set a [consumer](https://www.rabbitmq.com/consumer-prefetch.html) prefetch by setting a limit on the number of unacked messages at the client.

Please note that the prefetch value does not have an impact if you are using the Basic.get request, as we explained in the article: [RabbitMQ Basic Consume vs. RabbitMQ Basic Get](https://www.cloudamqp.com/blog/2020-07-14-rabbitmq-basic-consume-vs-rabbitmq-basic-get.html).

**Set the Prefetch Count**

RabbitMQ uses AMQP version 0.9.1 by default. The protocol includes the quality of service method Basic.qos for setting the prefetch count. RabbitMQ allows you to set either a channel or consumer count using this method.

Consider the following Pika example:

connection = pika.BlockingConnection()

channel = connection.channel()

channel.basic\_qos(10, global=False)

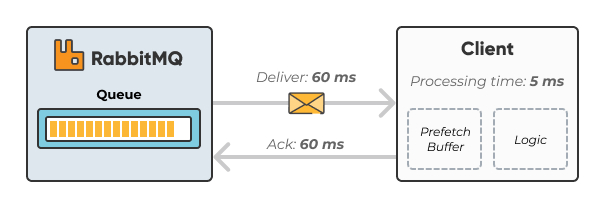
The basic\_qos function contains the global flag. Setting the value to false applies the count to each new consumer. Setting the value to true applies a channel prefetch count to all consumers. **Most APIs set the global flag to false by default**.

Optimizing the prefetch count requires that you consider the number of consumers and messages your broker handles. There is a negligible amount of additional overhead. The broker must understand how many messages to send to each consumer instead of each channel.

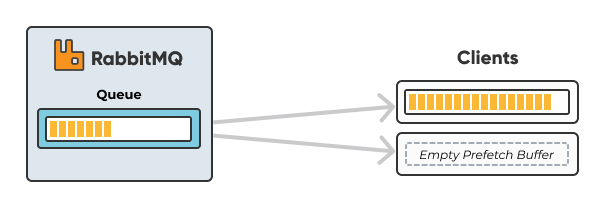
**Optimum Consumer Prefetch Count**

A larger prefetch count generally improves the rate of message delivery. The broker does not need to wait for acknowledgments as often and the communication between the broker and consumers decreases. Still, smaller prefetch values can be ideal for distributing messages across larger systems. Smaller values maintain an even rate of message consumption. A value of one helps ensure equal message distribution.

A prefetch count that is set too small may hurt performance since RabbitMQ might end up in a state where the broker is waiting to get permission to send more messages. The image below illustrates a long idling time. In the figure, we have a QoS prefetch setting of one (1). This means that RabbitMQ won't send out the next message until after the round trip completes (deliver, process, acknowledge). Round-trip time in this figure is in total 125ms with a processing time of only 5ms.



A large prefetch count, on the other hand, could take lots of messages off the queue and deliver all of them to one single consumer, keeping the other consumers in an idling state, as illustrated in the figure below:



**How To Set the Correct Prefetch Value**

If you have one single or only a few consumers processing messages quickly, we recommend prefetching many messages at once to keep your client as busy as possible. If you have about the same processing time all the time and network behavior remains the same, simply take the total round trip time and divide by the processing time on the client for each message to get an estimated prefetch value.

In a situation with many consumers and short processing time, we recommend a lower prefetch value. A value that is too low will keep the consumers idling a lot since they need to wait for messages to arrive. A value that is too high may keep one consumer busy while other consumers are being kept in an idling state.

If you have many consumers and/or long processing time, we recommend setting the prefetch count to one (1) so that messages are evenly distributed among all your consumers.

Please note that if your client auto-acks messages, the prefetch value will have no effect.

Avoid the usual mistake of having an unlimited prefetch, where one client receives all messages and runs out of memory and crashes, causing all the messages to be re-delivered.