# Pub Sub Protocols

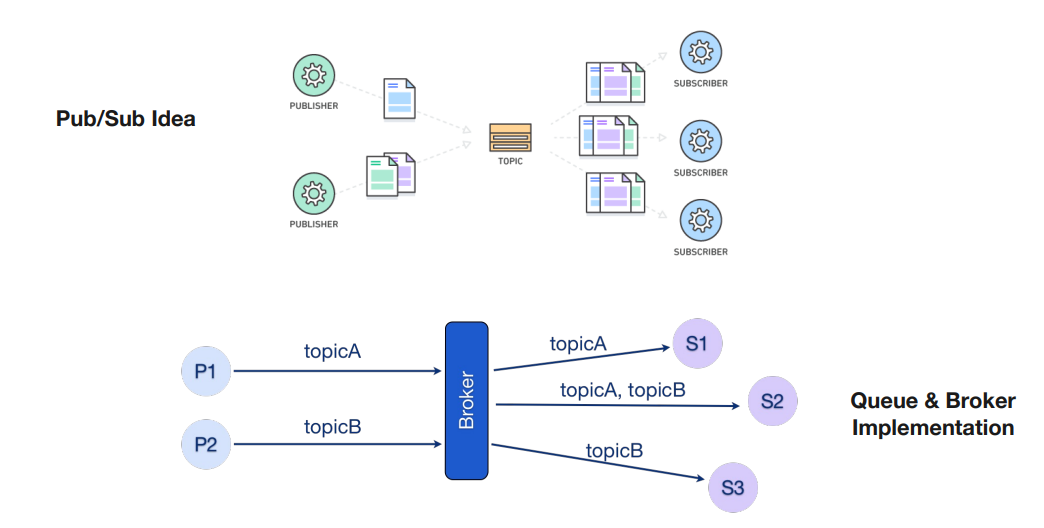
Two Main paradigms:

* Request/Reponse
* Publish/Subscribe

Pub/Sub is a message oriented middleware providing distributed, asynchronous , loosely coupled communication between message producers and message consumers

Pub/sub offers offers 3 types of deployments

* Message Producers and consumers are decoupled in time.
* Messages are not explicitily addressed to a specific consumer
* Messaging is synchronous



* Publishers send message ( tagged in a topic ) into a queue server (broker)
* Subscribers send registration request to the broker for a given topic
* The broker routes messages to subscribers

**QoS Semantics:** quality of service levels determine if is tolerable loss of data.

**Topology**: may be different according to the application requirements and the target architecture:

* **Centralized**: the broker forward the messages based on the requested filters
* **Distributed**: In order to reduce the load for the broker some device can communicate with each other
* **Hybrid**: producers and consumers use a central broker, while the actual payload is transferred directly

**Message Format:** the payload must be agnostic. Must be Binary.

Advantages:

* No Coupling: publishers do not know who will receive the message or even if they exist
* Scalability: since brokers only need to route messages they can be replicated easily to support higher volumes of data being transferred

Disadvantages:

* No content-type negotiation
* Long term evolution is hard to achieve
* Complex and open system with different semantics can be difficult to manage
* Within centralized approached it’s hard to support an end-to-end security.

**Frameworks:**

**Zero MQ** is a distributed socket system and does not mandatory require a broker to work.

* Runs on embedded device
* All unprocessed messages are store and managed in generalized sockets above the transport layer
* To establish a middleware communication, all participants have to know each other
* If the payload’s size exceeds maximum frame size can be split in two message

**Data Distribution Service:** is a data-centric publish-subscribe middleware for highly dynamicly distributed systems.

* DDS is optimized for distributed processing ( Directly connecting sensors, device and applications to each other without any dependence on centralized IT infrastructure
* It is standardized by OMG
* Data is published into the DDS domain and subscribers can subscribe data from that domain without knowing where the information came from or how It is structured (already describe itself)
* Support dynamic discovery and provide levels of QoS

**Apache Kafka -> open source**

* Broker based -> highly scalable and runs on distributed systems
* Every message is on the local hard drive by using a simple storage concept. (Log file)
* Each consumer has its own reading offset.
* Meets the requirements for secure messaging and is widely used and supported

**MQTT 🡪 Message Queue Telemetry Transport**

Is a lightweight **open-source** TCP based Pub/Sub Protocol

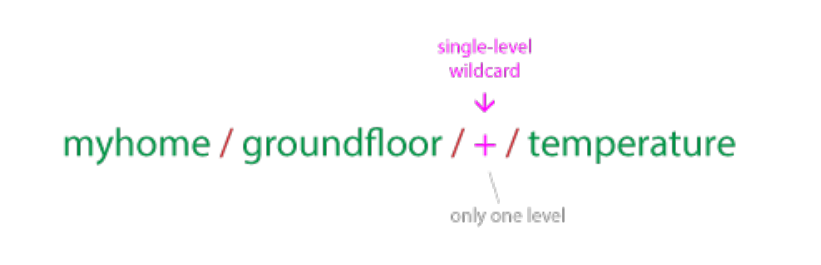
It’s standardized by **OASIS**

Target environment where devices and network are **constrained** and where protocol overhead and message size should be minimal.

**Topics** are used as filters on the message

Messages are **delivered to all clients** that have subscribed with a **matching topic filter**.

**MQTT Wildcards:**

Single level wildcards replaces one topic level.

Is substitute of an arbitrary string.

In the example can be:

Livingroom, kitchen ecc.

Multi Level Wildcard:

Covers many topic level, only at the end.

A client that subscrive at the example topic will receive every message that start with muhome/groundfloor/

**Quality of Service Mqtt Levels**

0 🡪 at most once

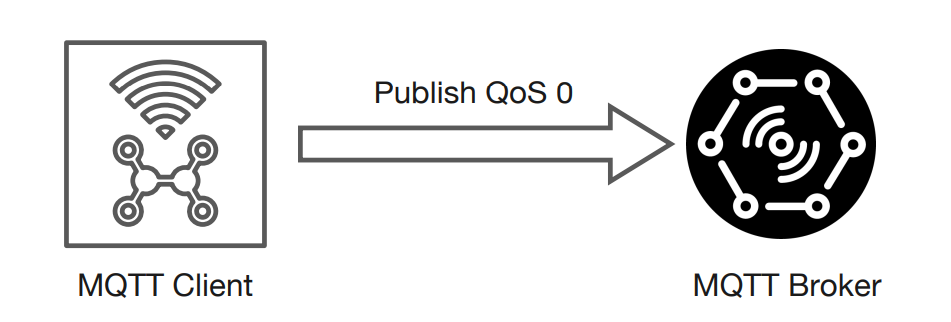
1 🡪 at least once

2 🡪 exactly once

The client publish to the broker with that defines the QoS level

The Broker will respond from now on with the client defined QoS

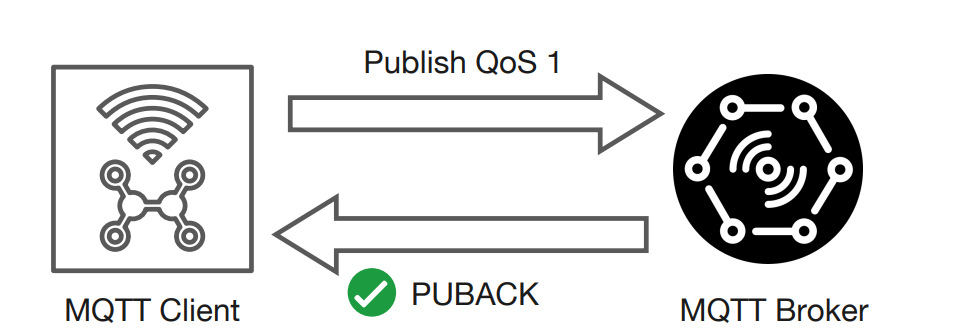
**Level 0 🡪 Best effort delivery** (Fire and forget)

****

No guarantees of the correct delivery

**Level 1 🡪 Guarantees that a message is delivered at least one time to the receiver**

The sender store the message util it receive a PUBACK packet from the receiver

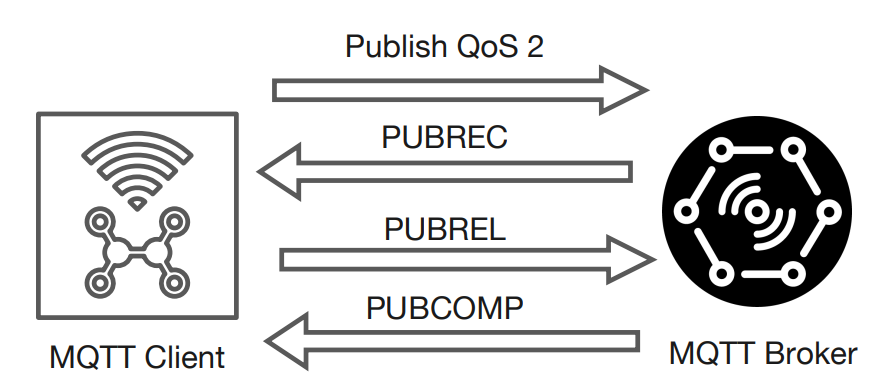


If the publisher send again the message set a DUP flag.

**Level 2 🡪 Exacly one**

Safest and slowest qos

* Four part handshake



**Persistent session**

The purpose is to reduce the computational cost and avoid the re-subscription to the topic in case of interruption in the connection.

All message sent are queued for offline client until the client is available again.

The broker will store some information like:

* All client subscription,
* Qos
* Existence of a session
* All missed message of level 1 and 2

When the device will reconnect the messages will be immediately available.

**CleanSession Flag**

True 🡪 the client do not want a persistent session.

False 🡪 The broker will setup the session.

**Retained messages:**

The purpose is to have the new topic subscribers to have a old message ready to consume.

Obviously the new message overwrites the old retained message

**Last will and testament**

The purpose is that clients can define a message that the broker will **automatically release** on their behalf in case of unexpected disconnection of the client.

The potency is that the client can **signal his absence**

The broker will discard the LWT message if the device disconnect gracefully with the **disconnect message**

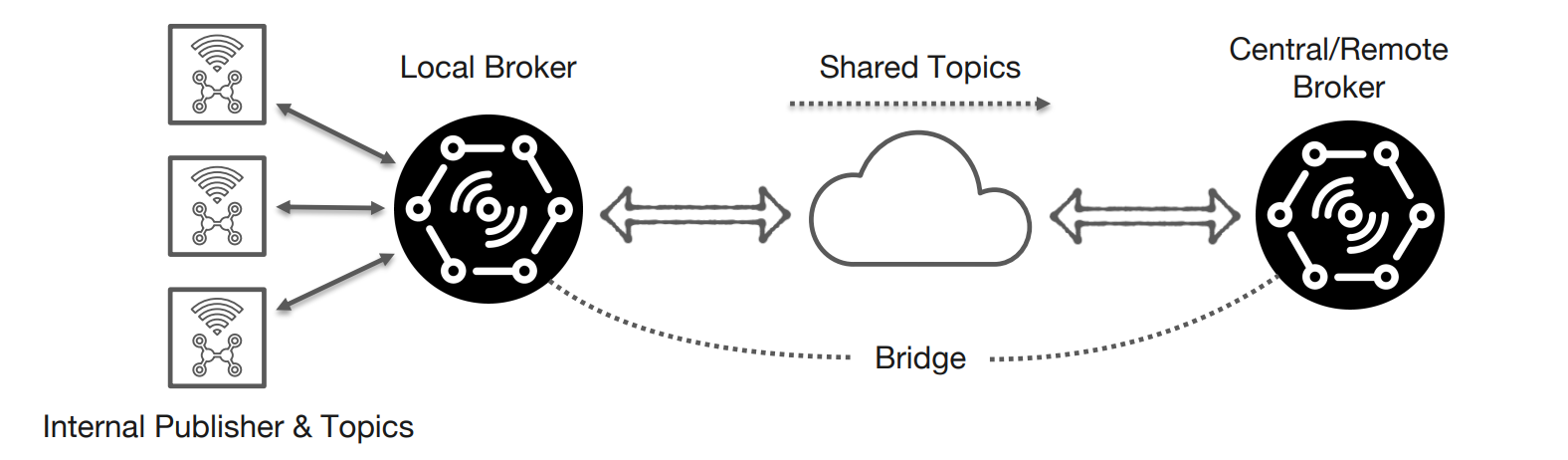
To configure the LWT message you have to include it in the **connect message**

**MQTT Bridge**

Let you connect two brokers together and they are generally used for sharing messages between systems

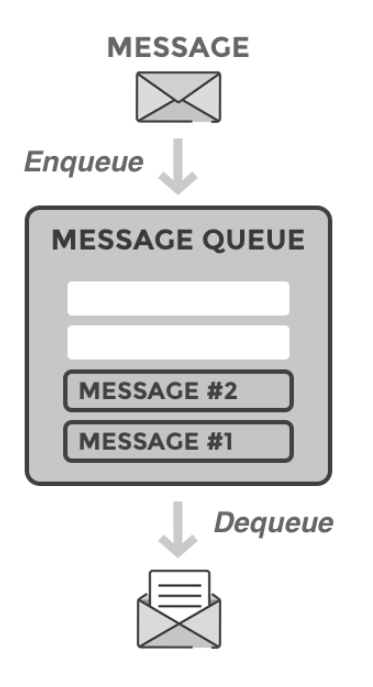
Example a remote origin server.

Only a broker need to be configured as a bridge the next will act as a normal broker



**AMQP**

Stands for  **Advanced Message Queue Protocol** (OASIS Standard)

* Asynchronous complement to HTTP
* Not an actual Pub/Sub but a specification for interoperable messaging for Message-Oriented-Middleware (**MOM**)
* AMQP Clients can work with any MOM that implements itself
* ****The routing key supports a dot-separated syntax that allows for different levels of expressiveness.

**Persistent queues:**  sore messages until the last consumer pull

Queue can be created by a consumer and destroyed at the disconnection.

**Main components:**

Producer 🡪 Application that sends the messages

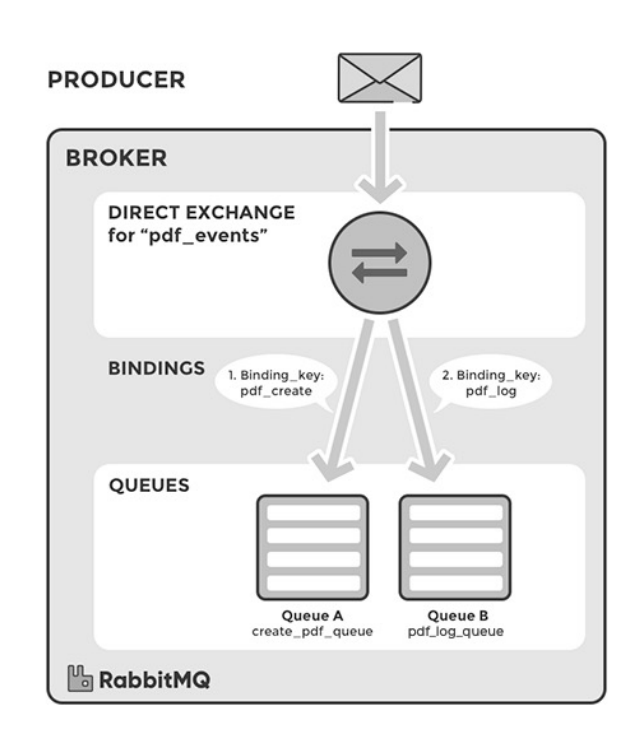
Consumer 🡪 Application that receive the messages

Queue 🡪 Buffer that store messages.

Message 🡪 Information that is sent from the producer to a consumer through RabbitMQ

Exchange: Receive messages from producers and pushes them to queue depending on the exchange type

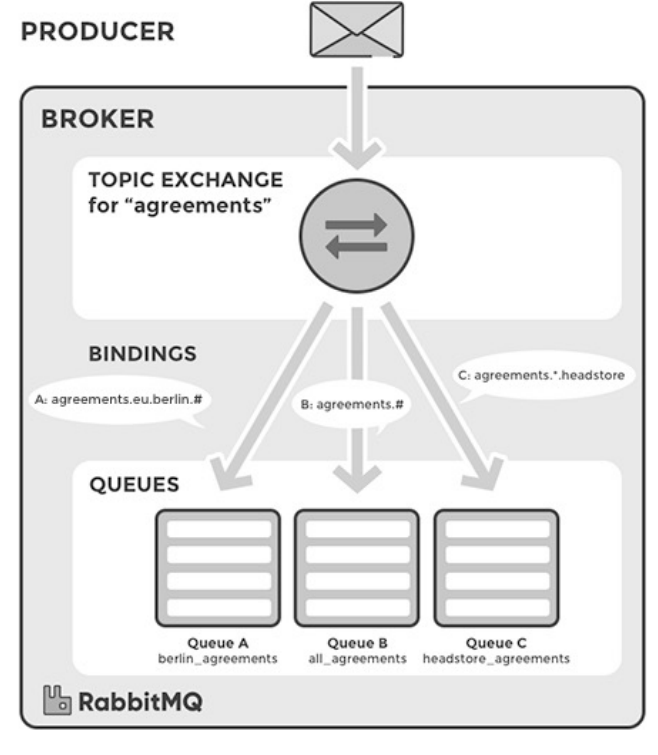
Binding: A link between a link and a Exchange

Routing Key: explain the route between a exchange and a queue

**Direct Exchange** 🡪 delivers messages to queue based on a message **Routing key** (on the producer header)

A message goes to the queue(s) with the binding key that exactly matches the routing key.

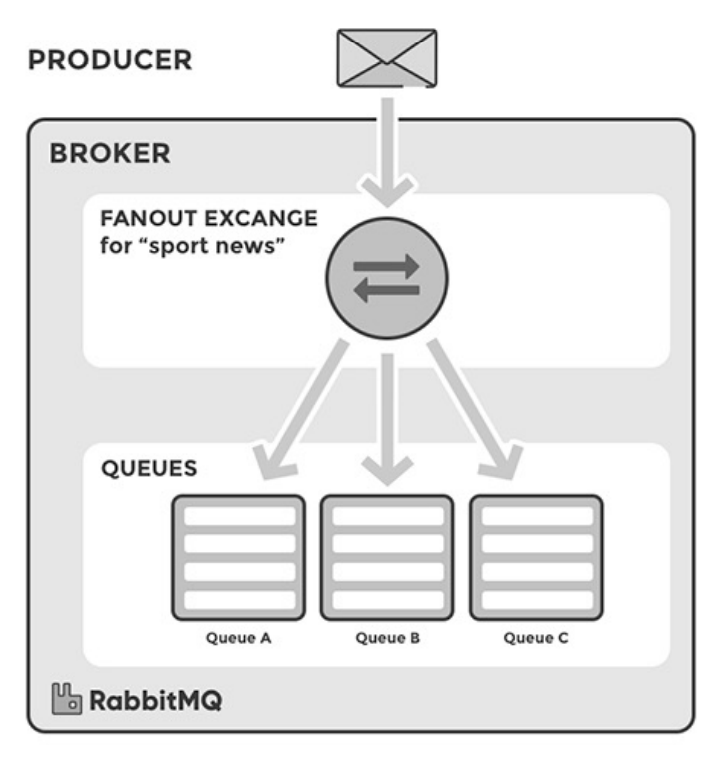
**Default exchange** is a pre-declared direct exchange with no name (empty string)

**Topic Exchange** route messages to queue based on wildcard matches between the routing key and the routing pattern

Messages are routes to on or many queue based on matching.

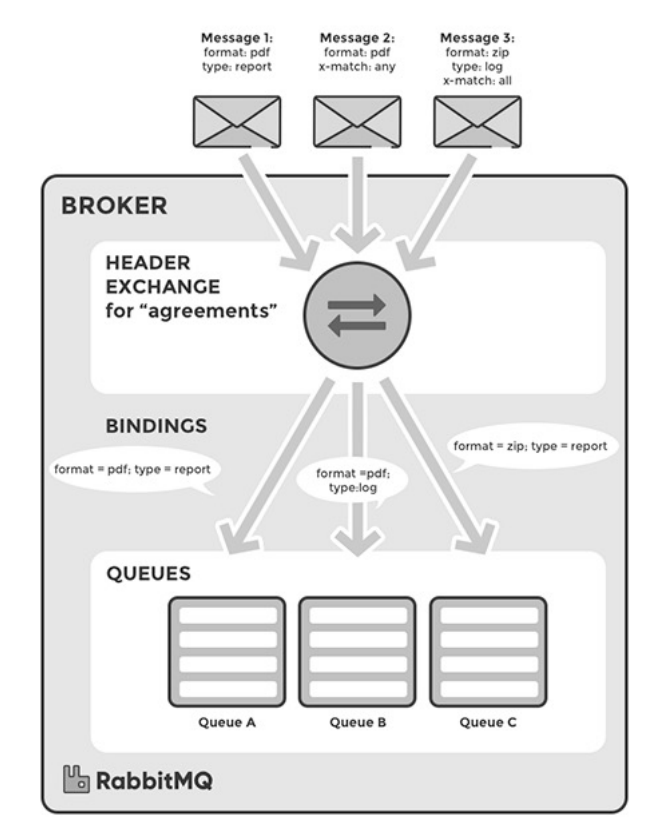
The routing key must be a list of words (separated by “.”)

* “\*” Stand for one word
* “#” Stands for 0 or more Words The consumers creates a queue and sets up a binding with given routing pattern.



**Fanout Exchange** 🡪 copies and routes a received message to all queue that are bound to it regardless of the key.

The key exist, is ignored

Can be useful when message needs to be sent to more queues.

**Headers Exchange** 🡪 Exchange routes messages based on arguments containing headers and optional values.

A message matches if the value of the header equals the value specified upon binding.

Argument “x-match”, added in the binding between exchange in queue, specifies if all headers must match or just one

Can be “any” or “all”