# Distributed Event-Based System — AMQP, MQTT, and Kafka

Denis Conan

September 2024

# Outline

1. Motivations and objectives/requirements
2. Definition of Event-Based Systems
3. Which Topic-based filtering DEBS?
4. 1st tech.: Topic-based filtering w/ OASIS AMQP v.0.9.1
5. 2nd tech.: Topic-based filtering w/ OASIS MQTT, IoT requirements
6. 3rd tech.: Topic-based filtering w/ Apache Kafka, Event Sourcing
7. Conclusion

# Foreword

- The content of these slides is extracted from the following references:
  - P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec “The Many Faces of Publish/Subscribe”, ACM Computing Surveys, 35(2), June 2003.
  - G. Mühl, L. Fiege, and P. Pietzuch “Distributed Event-Based Systems”, Springer-Verlag, 2006.
  - E. Al-Masri, K.R. Kalyanam, J. Batts, J. Kim, S. Singh, T. Vo, and C. Yan. “Investigating Messaging Protocols for the Internet of Things (IoT)”, IEEE Access, pages 94880–94911, April 2020.
  - OASIS AMQP Consortium, “AMQP: Advanced Message Queuing Protocol”, Version 0-9-1, Protocol specification, OASIS Consortium, November 2008.
  - OASIS, “MQTT Version 5.0”, Standard, OASIS Consortium, March 2019.
  - `https://kafka.apache.org/documentation/`
  - B. Stopford, “Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka”, O’Reilly, 2018.

# 1 Motivations and objectives/requirements

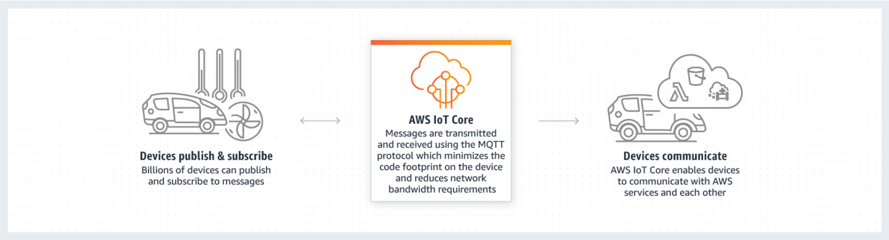
*Foreword:* We consider distributed architectures with application-layer messaging systems

# 1.1 E.g. IoT platforms I

- Communicate with lots of devices that are volatile

  $\Longrightarrow$ Scalability (#clients, #events)
  $+$ Space-, time-, and synchronisation-decoupling

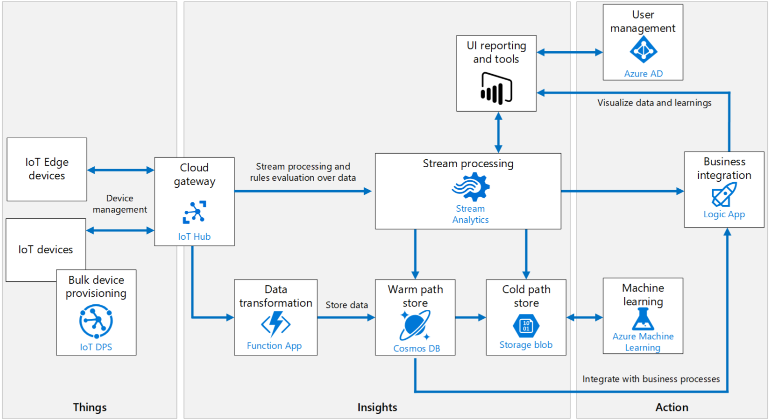
- E.g., Amazon IoT platform




https://aws.amazon.com/fr/iot-core/

# 1.1 E.g. IoT platforms II

- E.g. Microsoft Azure reference architecture

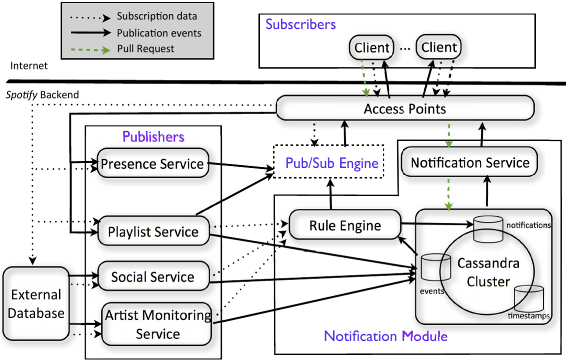



One of the previous version of

`https://docs.microsoft.com/fr-fr/azure/architecture/reference-architectures/iot`

# 1.2 E.g. Web services with "Event sourcing"

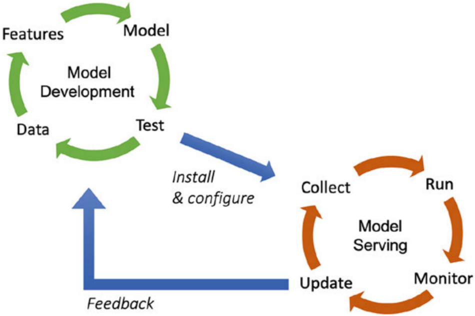
- Routing, event-driven for high performance,
  scalability (number of events per second, GB per second)




Old architecture from V. Setty, *et al.*, The Hidden Pub/Sub of Spotify (Industry Article). ACM DEBS'13, 2013

# 1.3 E.g. Life-cycle of data-driven machine learning applications

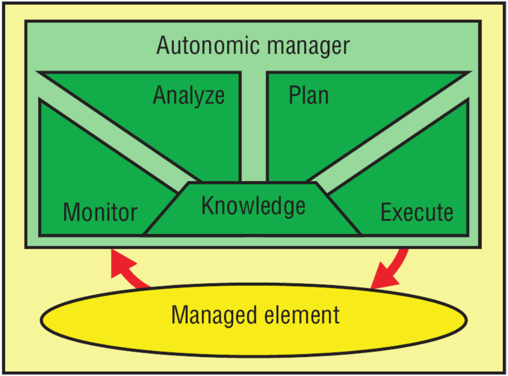
- On the right, execution of the application on a target machine
  - Prediction on collected data (real and not annotated)




P. Lalanda. Edge Computing and Learning. In M. Kirsch Pinheiro *et al* (editors), The Evolution of Pervasive Information Systems, Springer, 2023

# 1.4 E.g. Autonomic computing—MAPE-K loop

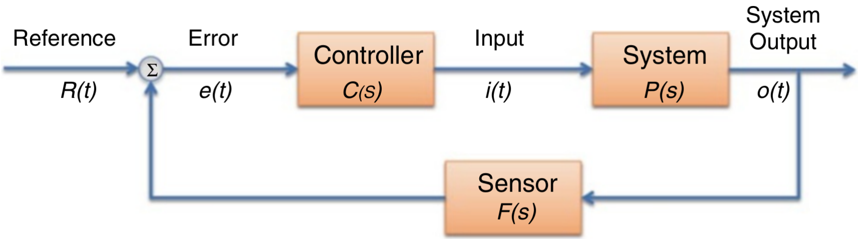
- Model of the architecture at runtime for self-management:
  i.e. self-configuration, self-optimization, self-healing, and self-protection
- MAPE-K: Monitor, Analyze, Plan, Execute, Knowledge




J. Kephart *et al.* The Vision of Autonomic Computing. IEEE Computer, 36(1):41–50, 2003.

# 1.5 E.g. Control theory—SISO loop

- E.g., General model of a Single-Input Single-Output feedback control system




P. Lalanda, J.A. McCann, and A. Diaconescu. Autonomic Computing. Springer, 2014.

A. Filieri, *et al.*. Software Engineering Meets Control Theory. In Proc. of the IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pages 71–82, May 2015.

# 1.6 Requirements

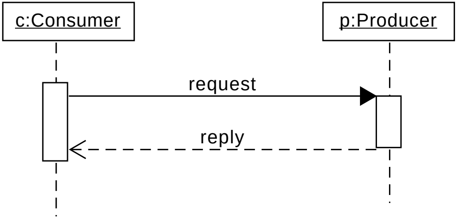
- Data production/consumption decoupling
  - Space decoupling: producers and consumers are distributed
  - Synchronisation decoupling: asynchronous and anonymous communication
  - Time decoupling: production and consumption at different times
- Scalability: in messages per second, in data per second, in clients (producers and consumers) at a given instant
- Data life-cycle management + filtering
  - Aggregation is out of the scope (it is called complex event processing and streaming)
- Adaptation to mobile, volatile, and heterogeneous environments

- One name for many “technologies”: distributed event-based systems, distributed publish-subscribe systems, distributed messaging service, message-oriented middleware, active databases, etc.

# 2 Definition of Event-Based Systems

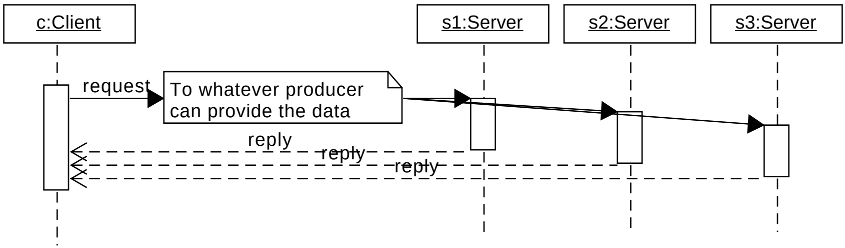
# 2.1 Models of interaction and EBS

# 2.1.1 “Request/Reply”



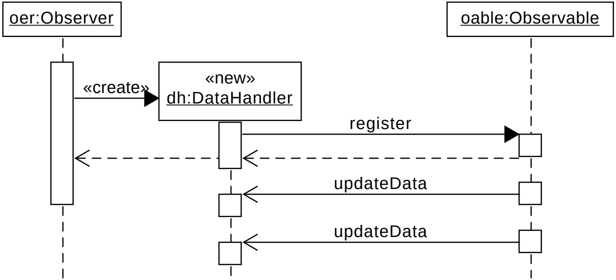

- The consumer initiates the interaction
- The consumer knows the address of the producer for issuing the request
- The consumer waits for the reply: the call is synchronous
- The producer knows the address of the consumer

# 2.1.2 “Anonymous Request/Reply”



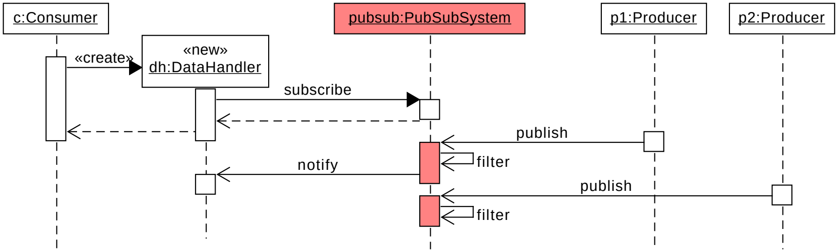

- The consumer initiates the interaction without knowing the address of the potential producers: there is an intermediate “entity” or “mechanism”
- The producers that can provide the requested data receive the request
- The producers reply to the consumer,
  i.e. they know the address of the consumer
- The consumer is willing to receive several replies

# 2.1.3 "Callback"




- This is the Observable Observer design pattern
- The consumer creates a data handler to manage registration and receptions
- The consumer knows the address of the producer and registers with it
- The producer sends the data updates to the consumer
- Consumer and DataHandler in the same process $\Longrightarrow$ multi-threading

# 2.1.4 Studied in this lecture: "Event-Based"




- This is the Publish Subscribe design pattern
- The consumer and the producers know the address of the PubSubSystem
- The consumer subscribes a filter to the PubSubSystem
- The producers publish data to the PubSubSystem
- The PubSubSystem applies subscription filters to route data and notify the relevant consumers

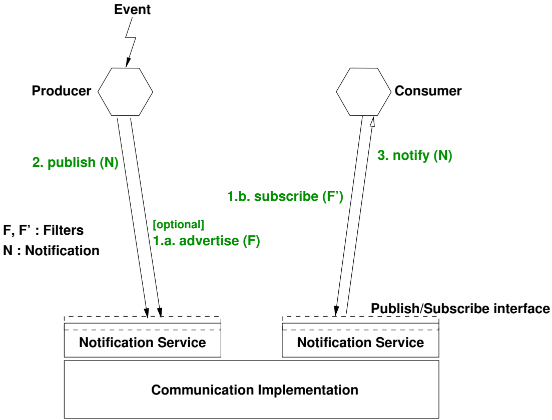
# 2.1.5 Recap: Models of interaction and EBS

- Initiator: describes whether the consumer or the provider initiates the interaction
- Addressing: indicates whether the addressee of the interaction is known or unknown at the beginning of the interaction

| | | Initiator | |
|---|---|---|---|
| | | Consumer | Provider |
| Adressee | Direct | Request/Reply | Callback |
| | Indirect | Anonymous Request/Reply | Event-Based |

- The trade-off is between the simplicity of request/reply and the flexibility of event-based interaction

# 2.2 Constituents of an EBS




We do not detail the `advertise` operation in this lecture.

## 2.2.1 Terminology

- Event : any happening of interest that can be observed from within a computer
  - Event example: physical event, timer event, etc.
- Notification: an object that contains data describing the event
- Producer: a component that publishes notifications
- Consumer: a component that reacts to notifications delivered to them by the notification service
- Subscription: describes a set of notifications a consumer is interested in
- Advertisement: is issued by a producer to declare the notifications it is willing to send

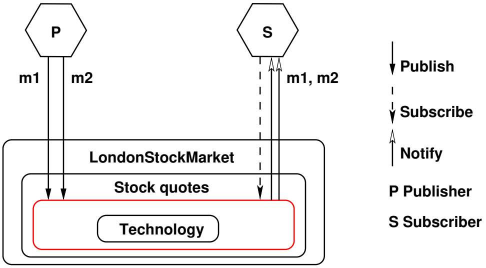
## 2.2.2 Publish/subscribe interface

- Specifies the functionalities for decoupling producers from consumers
- Proposes the following operations:
  - `publish(n):` a producer pushes notification `n` to the notification service
  - `advertise(F):` a producer advertises that it will send notifications that match the filter `F`
    - We do not detail the `advertise` operation in this lecture.
  - `subscribe(F):` a consumer subscribes to receive notifications that match the filter `F`
  - `notify(n):` the notification service delivers the notification `n` to those consumers that have a matching subscription

# 2.3 Notification filtering mechanisms

# 2.3.1 Channels-based filtering

- Producers select a channel into which a notification is published
- Consumers select a channel and will get all notifications published therein
- The message is “opaque” to the event-based service
- Framework examples: CORBA Event Service, CORBA Notification Service, OASIS AMQP standard v 0.9.1 (emulated in exchange mode “*fanout*”)...

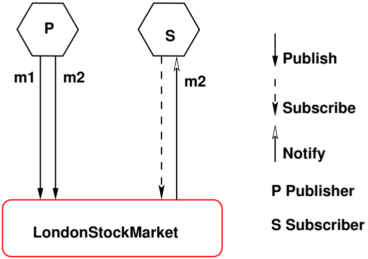


# 2.3.2 Topic-based (a.k.a. subject-based) filtering

- Uses string matching for notification selection with jokers
- Each notification and subscription is defined as a rooted path in a tree of topics
- Example:
  - A stock exchange application publishes new quotations of FooBar under the topic: /Exchange/Europe/London/Technology/FooBar
  - Consumers subscribe for /Exchange/Europe/London/Technology/* to get all technologies quotations
- The subject or topic is in message header, the content is "opaque"
- Example of solution: OASIS AMQP standard version 0.9.1 (exchange mode "*topic*"), OASIS MQTT standard version 3.1.1, TIBCO Rendezvous, JMS (Java Message Queue), WebSphere MQ Publish/Subscribe (WMQPS), Apache Kafka, Apache Qpid, Spring/Pivotal RabbitMQ, Amazon IoT Core, Microsoft Azure IoT Hub...

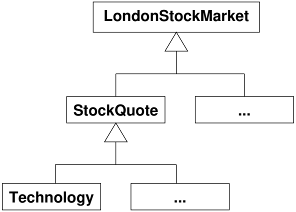
# 2.3.3 Content-based filtering

- Filters are evaluated on the whole content of notifications
- Example solutions: template matching, extensible filter expressions on name value pairs, XPath expressions on XML schemas, etc.
- Example for the structured-record data model:
  A publication is a set of pairs: $m_1 = \{(company, \text{“Telco”}), (price, 120)\}$
  $m_2 = \{(company, \text{“Telco”}), (price, 90)\}$
  A filter is a conjunction of triples: $F = \{(company, =, \text{“Telco”}), (price, <, 100)\}$



# 2.3.4 (Type-based filtering)

- Uses subtype inclusion to select notifications
- If a consumer subscribes to the type `StockQuote`, it will receive `Technology` quotations and other notifications that are sub-types of `StockQuote`




- All the producers and consumers must agree on the hierarchy of types
  $\Longrightarrow$ Not flexible at all, thus not used $\Longrightarrow$ the title in brackets

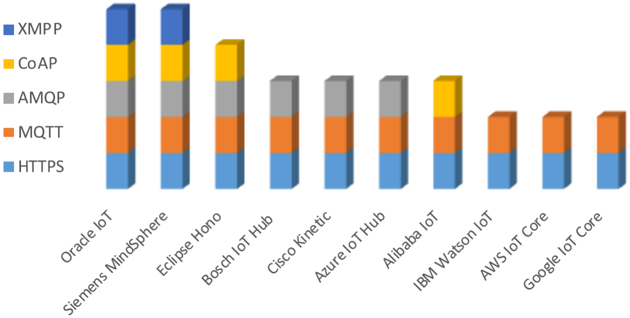
# 3 Which Topic-based filtering DEBS? I

- *Reminder:* We study application-layer distributed-event based systems
- Topic-based filtering = filtering currently used by IT industry
  - Channels-based filtering: previous middleware like CORBA
  - Type-based filtering: not usable
  - Content-based filtering: more for complex event processing and streaming

1. OASIS AMQP: introduce the concept of "broker"
2. OASIS MQTT: introduce constraints from the Internet of Things
3. Apache Kafka: introduce design patterns "Event Sourcing" and "Collaboration"

# 3 Which Topic-based filtering DEBS? II

In this slide, let's take the application domain of the IoT

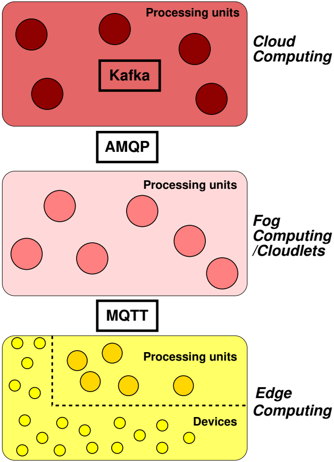



E. Al-Masri, K.R. Kalyanam, J. Batts, J. Kim, S. Singh, T. Vo, and C. Yan "Investigating Messaging Protocols for the Internet of Things (IoT)", IEEE Access, pages 94880–94911, April 2020.

Also, RabbitMQ is one of the engine of: Amazon MQ, the Google Cloud Platform through `bluemedora`, IBM Cloud in the context of Web and mobile applications.

# 3 Which Topic-based filtering DEBS? III

Overview of a prototypical software architecture with DEBS middleware



# 4 1st tech.: Topic-based filtering w/ OASIS AMQP v.0.9.1



Initially, a proposition made by JPMorgan Chase

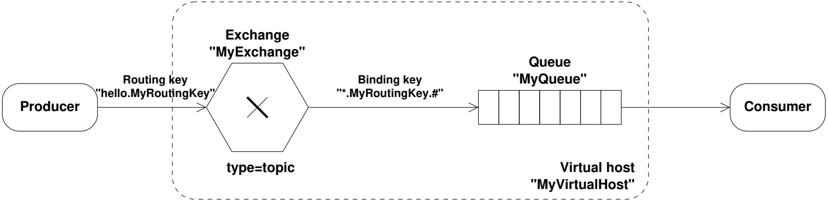
The content of this section is extracted from

http://www.amqp.org/specification/0-9-1/amqp-org-download

and from

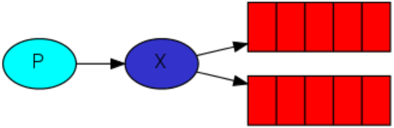
https://www.rabbitmq.com/getstarted.html.

# 4.1 Overview of topic-based filtering of AMQP



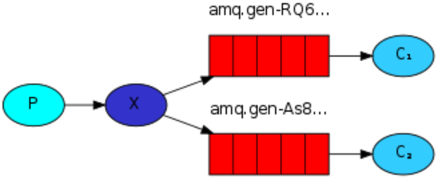

- Lots of implementations: RabbitMQ, Apache Qpid, Microsoft Azure IoT Hub, etc.
- We propose to follow a tutorial on RabbitMQ

# 4.2 Exchange, binding, and queue



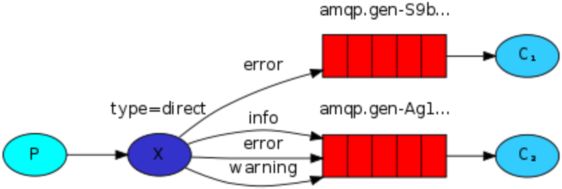

- Queue = name for a "post box" that lives inside the AMQP server
  - Messages are only stored inside a queue, never in exchanges
  - A queue is essentially a large message buffer
  - Many producers can send messages that go to one queue
  - Many consumers can try to receive data from one queue
- An exchange = A matching and routing engine
  - It inspects notifications (headers), and using its binding tables, decides how to forward these notifications to message queues or other exchanges
- A binding key = A criteria for notification routing
- A binding = A relationship (queue, exchange) with a binding key

# 4.2.1 Exchange of type “fan-out”



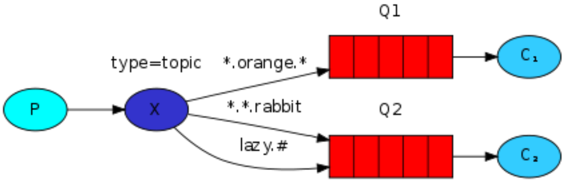

- The “fan-out” exchange type implements channel-based filtering
  - A message queue binds to the exchange with no arguments
    - Nothing on the arrow/binding from the exchange to the queue
  - A publisher sends notifications to the exchange
  - The notification is passed to the message queue unconditionally

# 4.2.2 Exchange of type “direct”




- The “direct” exchange type implements a simplistic form of topic-based filtering
  - A message queue binds to the exchange using a routing key K (a string)
  - A publisher sends to the exchange a notification with the routing key R
  - The notification is passed to the message queue if K = R

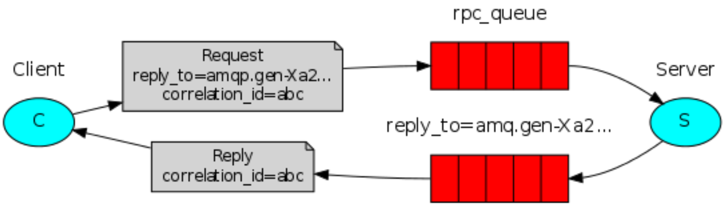
## 4.2.3 Exchange of type "topic"




- The "topic" exchange type works as follows:
  - A queue binds to the exchange using a binding key $B$ as the routing pattern
  - A publisher sends to the exchange a notification with the routing key R
  - The notification is passed to the message queue if R matches B
- Routing key used for a topic exchange = 0 or more words delimited by dots
- Each word may contain `[A–Z]`, `[a–z]`, and `[0–9]`, or be equal to a joker ("*" or "#")
- The binding key follows the same rules as the routing key with: "*" that matches a single word and "#" that matches 0 or more words

# 4.2.4 Message properties and emulation of RPC

- Using message properties
  - The AMQP 0-9-1 protocol defines a set of 14 message properties
  - "`deliveryMode`": Marks a message as persistent or transient
  - "`contentType`": Used to describe the mime-type of the encoding (e.g. `application/json`)
  - For a RPC-like call:
    - "`replyTo`": Commonly used to name a callback queue
    - "`correlationId`": Useful to correlate RPC responses with requests



# 4.3 Message and queue properties

- Message acknowledgement
  - What happens if a consumer fails while treating a message?
  - Consumer can choose to `autoAck` or not
    1. `autoAck=true`: Once delivered, the server immediately marks the message for deletion
       $\Longrightarrow$ May be lost if the consumer fails
    2. `autoAck=false`: The server waits for an explicit acknowledgement
       $\Longrightarrow$ Memory leakage if the consumer forgot to send the acknowledgement
- Message durability and queue persistence
  - When the server quits/crashes it forgets queues and messages unless told to do so
  - Two properties to make nearly sure that messages aren't lost:
    1. Mark both the queue and messages as "durable"
    2. Mark messages of queue as "persistent"

# 4.3.1 More about message reliability

- A server forgets the queues and messages unless it is told not to
- Message reliability capabilities in a continuum:
  1. Mark queues and messages as durable = eventually stored in database
     - But, e.g., RabbitMQ doesn't do `fsync(2)` for every message
       - Messages may be just saved to cache and not really written to the disk
  2. Clustering = Replicate broker for highly available queues (active replication)
     - Not in the AMQP specification, but provided in RabbitMQ for instance
  3. Publisher confirms =
     - Consumers acknowledge the treatment of a message
     - The broker sends a confirm message to the publisher
       when all the clients have acknowledged

# 5 2nd tech.: Topic-based filtering w/ OASIS MQTT, IoT requirements



Initially, a proposition supported by IBM

The content of this section is extracted from

https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

# 5.1 MQTT features

- Initially, a proposition made by IBM
- MQTT v.3.1: an OASIS standard in Oct. 2014
  Then, MQTT v.3.1.1: an ISO/IEC standard (20922:2016[1]) in June 2016
  Today, MQTT v.5.0: OASIS Standard, March 2019
- It runs over TCP/IP, or over other network protocols that provide ordered, lossless, bidirectional connections
  - MQTT-SN was proposed using UDP for sensor networks in which these network conditions cannot be assumed
- Topic-based filtering with 3 levels of Quality of Service / message reliability
- Concept of sessions, in addition to connections
- Popular implementations: Eclipse Mosquitto and Paho, Amazon IoT Core, BevyWise, HiveMQ, Microsoft Azure IoT Hub, VerneMQ, etc.

1. https://www.iso.org/standard/69466.html

# 5.2 Topic filters w. wildcards and topic names I

- The forward slash ("/") is used to separate each level within a topic tree and provide a hierarchical structure to the topic names
- Topic filter = an expression contained in a subscription
  - ≈ AMQP binding key
  - "#,+" can be used in topic filters similarly to AMQP's "*" and "#"
- Topic name = the label attached to a message that is matched against the subscriptions
  - ≈ AMQP routing key
  - A broker can change the topic name of a published packet

# 5.2 Topic filters w. wildcards and topic names II

- The plus sign ("+") matches only one topic level
  - The single-level wildcard can be used at any level in the Topic Filter, including first and last levels
  - Where it is used it must occupy an entire level of the filter
  - E.g.
    - "`sport/tennis/+`" matches "`sport/tennis/player1`" and "`sport/tennis/player2`", but not "`sport/tennis/player1/ranking`"
    - "`sport/+`" does not match "`sport`" but it does match "`sport/`"
    - "`+`" and "`+/tennis/#`" are valid
    - "`sport+`" is not valid
    - "`/finance`" matches "`+/+`" and "`/+`", but not "`+`".
  - See the discovery Lab: Step 2.a, script `run_example_topics_sign_plus.sh`

# 5.2 Topic filters w. wildcards and topic names III

- The number sign ("#") matches any number of levels within a topic
  - The multi-level wildcard represents the parent and any number of child levels
  - "#" must be specified either on its own or following a topic level separator
  - "#" must be the last character specified in the topic filter
  - E.g.
    - "`sport/tennis/player1/#`" matches "`sport/tennis/player1`", "`sport/tennis/player1/ranking`", and "`sport/tennis/player1/score/wimbledon`"
    - "`sport/#`" matches "`sport`", since "`#`" includes the parent level
    - "`sport/tennis#`" is not valid
    - "`sport/tennis/#/ranking`" is not valid

  - See the discovery Lab: Step 2.b, script `run_example_topics_sign_number.sh`

# 5.2 Topic filters w. wildcards and topic names IV

- Special character "$"
  - Broker implementations may use topic names that start with a leading "$" character for other purposes
    - E.g. "`$SYS/`" has been widely adopted as a prefix to topics that contain server-specific information or control APIs
  - The broker must not match topic filters starting with a wildcard character ("#" or "+") with topic names beginning with "$"
  - The broker should prevent clients from using such topic names to exchange messages with other Clients

# 5.3 QoS—Message reliability

- Published messages have associated quality of service (QoS)
  - QoS0/"At most once": best efforts of the operating environment
    - Message loss can occur
    - Level used for example with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after
  - QoS1/"At least once": assured to arrive but duplicates can occur
  - QoS2/"Exactly once": assured to arrive exactly once

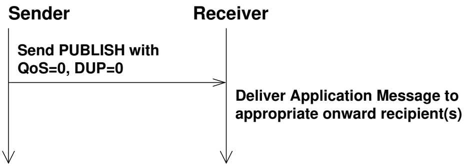
$\Longrightarrow$ Client and broker store session state in order to provide QoS levels 1 and 2

- A subscription contains a topic filter and a maximum QoS
  - The broker might grant a lower maximum QoS than the subscriber requested
  - When filters overlap, the delivery respects the maximum QoS of all the matching subscriptions

# 5.3.1 Delivery of QoS0/“At most once” messages

- QOS 0
  - No storage of the message is performed by the sender
  - No acknowledgment is sent by the receiver
  - No retry is performed by the sender

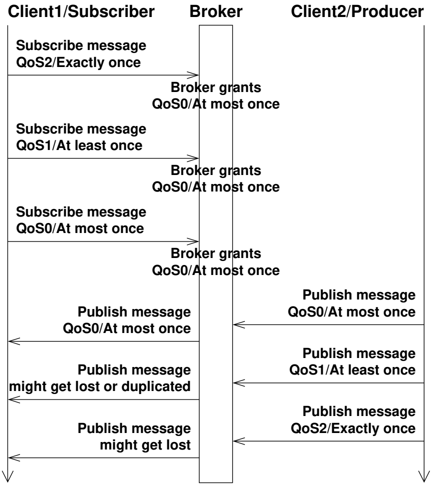
- The sender sends a publish packet with `QoS=0, DUP=0`[2]
- The receiver accepts ownership of the message when it receives the publish packet




1. The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver
2. `DUP` is set to 1 when the sender knows it is a duplicate

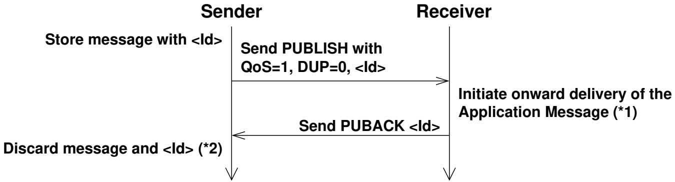
# 5.3.2 Subscription and publication with QoS0/“At most once”

- In this scenario, let us consider that the broker grants a maximum QoS0
- A QoS1/“At least once” message might either get lost or duplicated
- A QoS2/“Exactly once” message might get lost
  but the broker should never send a duplicate
- See the discovery Lab: Step 3.a, script `run_example_qos_0.sh`



# 5.3.3 Delivery of QoS1/“At least once” messages

- A QoS1 publish packet has an `Id` and is acknowledged
- The sender may resend the message if no acknowledgement is received
- The Sender:
  1) assigns an `Id` and sends a publish packet containing `Id, QoS=1, DUP=0`
- The Receiver:
  1) acknowledges, having accepted ownership of the message
  2) treats any incoming publish packet with same `Id` as being a new publication, then forwarding it if the receiver is a broker

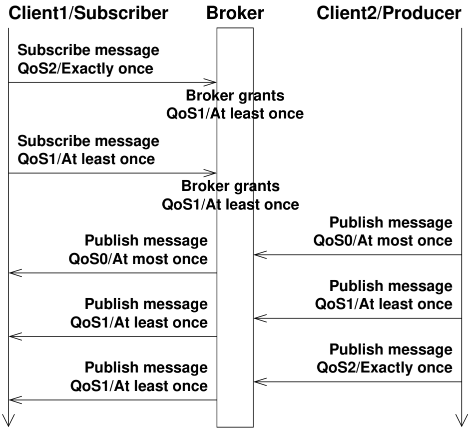



(*1) The receiver is not required to complete the delivery before sending the PUBACK

(*2) The sender knows that ownership of the message is transferred to the receiver

# 5.3.4 Subscription and publication with QoS1/"At least once"

- The server grants a maximum QoS1
- A QoS0 message matching the filter is delivered at QoS0/"At most once"
- A QoS2 message published to the same topic is downgraded by the server to QoS1
  - Client might receive duplicate copies of the message
- See the discovery Lab: Step 3.b, script `run_example_qos_1.sh`



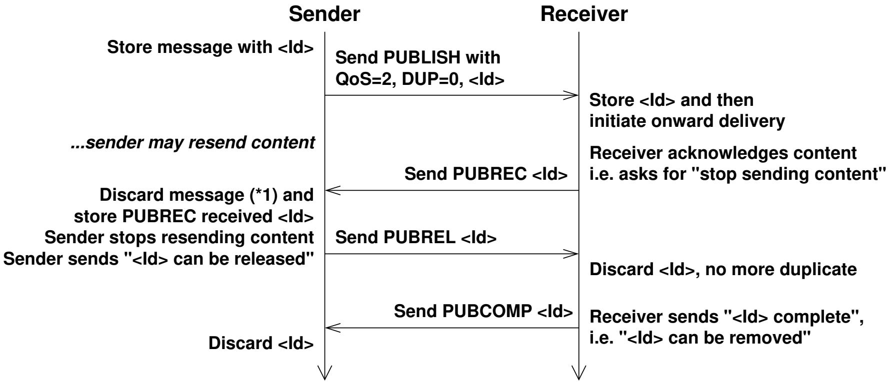
# 5.3.5 Delivery of QoS2/"Exactly once" messages I

- The receiver acknowledges receipt with a two-step acknowledgement process

- The Sender:
  1) assigns an `Id` and sends a publish packet containing `Id, QoS=2, DUP=0`
  3) treats the publish packet as "unack" until it has received the `PUBREC`
  4) sends a `PUBREL` (release) packet when it receives a `PUBREC` packet
  5) treats the `PUBREL` packet as "unack" until it has received the `PUBCOMP` (complete)
  6) do not re-send the publish packet once it has sent the `PUBREL`

- The Receiver:
  1) responds with a `PUBREC`, having accepted ownership of the message
  2) until it has received the corresponding `PUBREL` packet, acknowledges any subsequent publish packet with the same `PUBREC`
  3) responds to a `PUBREL` packet by sending a `PUBCOMP`

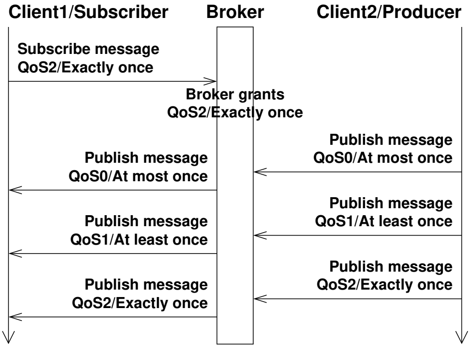
# 5.3.5 Delivery of QoS2/“Exactly once” messages II




(*1) The sender knows that ownership of the message is transferred to the receiver

# 5.3.6 Subscription and publication with QoS2/"Exactly once"

- The broker grants a maximum QoS2
- A topic filter at QoS 2 = delivery of a message at the QoS with which it were published
- See the discovery Lab: Step 3.c, script `run_example_qos_2.sh`



# 5.4 Disconnections

## 5.4.1 Sessions

- Session = A stateful interaction between a client and a broker
- Some sessions last only as long as the network connection, others can span multiple consecutive network connections
- When a client connects with `CleanStart` set to 0, it is requesting that the broker maintain its state after disconnection
- When a client has determined that it has no further use for the session, it should connect with `CleanStart` set to 1 and then disconnect
- A broker is permitted to disconnect a client that it determines to be inactive or non-responsive at any time

- See the discovery Lab: Step 4, scripts `run_example_clean_start_true.sh`, `run_example_clean_start_false.sh`, and `run_example_clean_start_false_qos_1.sh`

## 5.4.2 RETAIN flag in a publish packet

- On a publish, if the `RETAIN` flag is set to 1, the broker must store the message (and its QoS) so that it can be delivered to future subscribers whose subscriptions match its topic
  - The client can mix publishing with and without the `RETAIN` flag set
  - The retained message on the broker is the last received with the `RETAIN` flag set
  - `RETAIN` set + empty payload $\Longrightarrow$ broker removes previously retained message
- When a new subscription is established, the last retained msg (if any) is sent to the subscriber as it were the first message

- See the discovery Lab: Step 5, script `run_example_retained_flag.sh`

## 5.4.3 Message ordering

- When a client reconnects with `CleanStart` set to 0 when connecting, both the client and broker must re-send any unacknowledged publish packets (where QoS$>$0) and `PUBREL` packets using their `Ids`
- A broker must by default treat each topic as an "Ordered Topic"
  - It may provide an administrative or other mechanism to allow one or more topics to be treated as an "Unordered Topic"

# 6 3rd tech.: Topic-based filtering w/ Apache Kafka, Event Sourcing



See `https://kafka.apache.org/`
and `https://kafka.apache.org/quickstart`

# 6.1 Cluster-based architecture

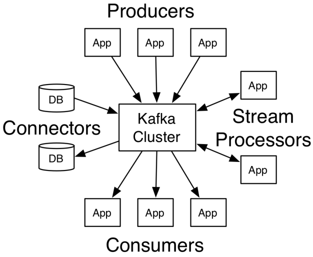



Image extracted from `https://kafka.apache.org`

- Kafka is run as a cluster of servers that can span multiple datacenters
- The Kafka cluster stores streams of records in categories called topics
- Producers publish a stream of records to one or more Kafka topics
- Consumers consume an input stream from one or more topics

# 6.2 Topics as structured commit logs

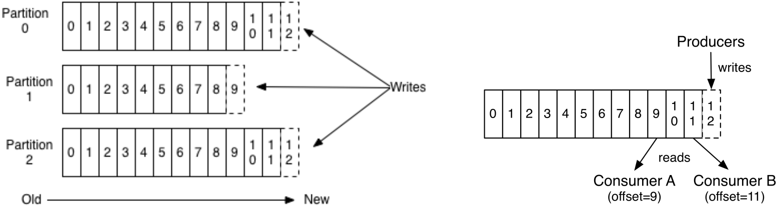



Image extracted from `https://kafka.apache.org`

- A topic = stream of records = partitioned log = structured commit log
- Records are assigned a sequential id. number called the offset
- Each partition is an ordered, immutable sequence of records that is continually appended to
- A partition must fit on the server that hosts it
- A topic may have many partitions, each one acting as the unit of parallelism
- Consumers can consume records in any order they like, but usually of the time in ascending order

# 6.3 Consumer groups

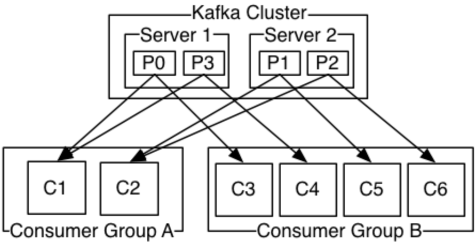



Image extracted from `https://kafka.apache.org`

- Consumers join groups, which are labelled with a consumer group name
- Typically, applications are structured/programmed as follows:
  - Each record published to a topic is delivered to one consumer within each subscribing consumer group
    - If all the consumers are in the same group, then records are load balanced
    - If all the consumers are in different groups, then records are replicated

# 6.4 Fault tolerance

- Each partition is replicated across a configurable number of hosts
- One host acts as the "leader" and the others act as "followers"
  - Usually, each host acts as a leader for some of its partitions and as a follower for others
- The process of maintaining membership in the group is handled by Kafka dynamically
  - If an instance joins a group, it takes over partitions from existing instances
  - If an instance dies, its partitions are distributed to the remaining instances
- Total order over records within a partition, not between different partitions in a topic

# 6.5 From Event Collaboration to CQRS

# 6.5.1 Design pattern "Event Collaboration"

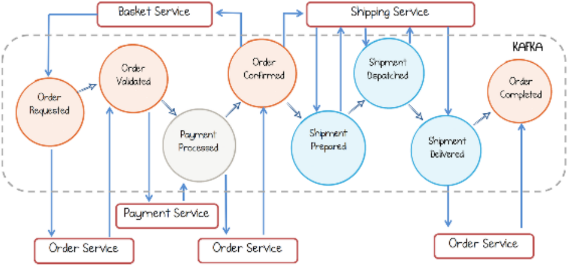
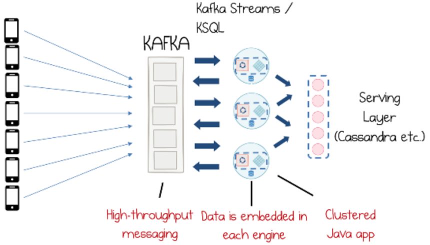



Image from B. Stopford, "Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka", O'Reilly, 2018

- https://martinfowler.com/eaaDev/EventCollaboration.html
- Each (micro-)service listens events and creates new events
- No service knows the other services nor owns the entire workflow
  - This is called a choregraphy
    - $\neq$ An orchestration, in which a process controls the whole workflow

# 6.5.2 Design pattern “Event Sourcing”




B. Stopford, “Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka”, O’Reilly, 2018.

- https://martinfowler.com/eaaDev/EventSourcing.html
- Use Kafka as a data store of the events in the order of their creation
  - Make the events “the source of truth”: include commands into Kafka log
- Fault-tolerance using passive replication by rollback recovery
  - Consider (micro-)services that have a pseudo-deterministic execution
    - Any state of the execution can be computed from an initial state and the sequence of events that leads to this state
  - Periodic creation of snapshots + replay of events in order

# 6.5.3 Design pattern "Command Query Responsability Segregation"

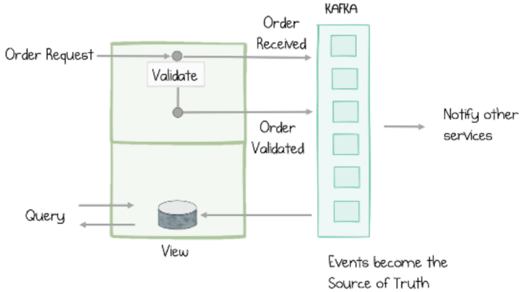



Image from B. Stopford, "Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka", O'Reilly, 2018

- `https://martinfowler.com/eaaDev/EventSourcing.html`
- Separate the write path from the read path and links them with an asynchronous channel
- Provide adequate view(s) of the (micro-)service and query the view(s)

# 7 Conclusion I

- Distributed Event-Based Systems for acquiring data
- Other names of this architectural style: Distributed Publish Subscribe System, Distributed Messaging Service
- Interaction mode = event-based
  - Producers initiate the exchanges of data (push mode)
  - Producers do not know the potential consumers when pushing
- Properties of this architectural style =
  - Space decoupling: Producers and consumers do not know each others
  - Time decoupling: Producers and consumers do not need to be active at the same time
  - Synchronisation decoupling: asynchronous communication (producers and consumers are not blocked while producing or being notified, respectively)

# 7 Conclusion II

- In the order of the presentation
  - AMQP and MQTT = server-based architecture using topic-based filtering
  - AMQP proposes three types of exchanges:
    - "fan-out" = broadcast functionality
    - "direct" = string equality as a very simple matching algorithm
    - "topic" = topic-based filtering with meta-characters to match a single word or more words
  - MQTT comes in addition with QoS:
    - 0/"at most once" = best effort
    - 1/"at least once" = assured to arrive but duplicates can occur
    - 2/"exactly once" = assured to arrive exactly once
  - Kafka looks more like a distributed commit logging system
    - A topic is a set of partitions, which are append-only files
    - More stream-oriented than topic-based

# 7 Conclusion III

- Kafka for stream processing, data integration, stable network and good infrastructure
- AMQP for AMQP consortium, high throughput, high availability
- MQTT for ISO standard, lightweight, poor connectivity, high latency, disconnections and reconnections