Distributed Event-Based System — Basics

Master CILS — Module IAAIO, Revision : 2548

Denis Conan

Octobre 2019
Foreword

Most of the content of these slides is extracted from the following references:

Outline

1. Motivations and objectives/requirements

2. Definition of Event-Based Systems

3. Formal specification of simple event-based system

4. Notification filtering mechanisms

5. Topic-based filtering w/ AMQP v.0.9.1

6. Topic-based filtering w/ Apache Kafka

7. Topic-based filtering w/ OASIS MQTT v.3.1.1
1 Motivations and objectives/requirements

1.1 E.g. IoT platforms
1.2 E.g. Web services
1.3 E.g. Data-driven inference cycle
1.4 E.g. Autonomic computing—MAPE-K loop
1.5 Requirements
1.6 Example of unified architecture model: Context data distribution
1.1 E.g. IoT platforms I

- Communicate with lots of devices that are volatile
  \[\Rightarrow\] 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

https://docs.microsoft.com/fr-fr/azure/architecture/reference-architectures/iot/index
1.2 E.g. Web services

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

1.3 E.g. Data-driven inference cycle

Specify objectives

Acquire data

Compute

Incorporate result

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

1.5 Requirements

- Data production/consumption decoupling
  - Synchronisation decoupling: asynchronous and anonymous communication
  - Space decoupling: unknown producers and consumers
  - Time decoupling: production and consumption at different times

- Scalability: in messages per minute, in data per second, in clients (producers and consumers) at a given instant

- Data life-cycle management + filtering + aggregation

- Adaptation to mobile 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.

1.6 Example of unified architecture model: Context data distribution

- E.g. for the events from the Internet of Things

1.6.1 Classification of Data Routing

Routing Overlay

- Centralized Architecture
- Decentralized Architecture
  - Flat distributed
  - Hierarchical distributed

---

## Comparison I

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Architecture</td>
<td>-</td>
<td>• Context data access is always ensured</td>
<td>• Limited scalability and reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No management overhead for routing overlay maintenance</td>
<td>• Locality principles difficult to apply</td>
</tr>
<tr>
<td>Decentralized Architecture</td>
<td>Flat distributed architecture</td>
<td>• Increased scalability and reliability</td>
<td>• Context data access could not be always ensured</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Locality principles easy to apply</td>
<td>• Additional management overhead for routing overlay maintenance</td>
</tr>
<tr>
<td></td>
<td>Hierarchical distributed architecture</td>
<td>• Increased scalability and reliability</td>
<td>• Context data access could not be always ensured</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Locality principles easy to apply</td>
<td>• Additional management overhead for routing overlay maintenance</td>
</tr>
</tbody>
</table>

---

1.6.3 Classification of Context Data Dissemination

## Comparison I

<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-category</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor Direct Access</td>
<td></td>
<td>• No state on mobile nodes</td>
<td>• Strong coupling between sources/sinks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Low network overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sink always receive interesting data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dissemination reaches only interested nodes</td>
<td></td>
</tr>
<tr>
<td>Flooding-based</td>
<td>Data flooding</td>
<td>• Low state on mobile nodes</td>
<td>• High network overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loose coupling between sources/sinks</td>
<td>• Dissemination can reach not-interested nodes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sink always receive interesting data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subscription flooding</td>
<td>• Loose coupling between sources/sinks</td>
<td>• High state on mobile nodes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Sink always receive interesting data</td>
<td>• High network overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dissemination reaches only interested nodes</td>
<td></td>
</tr>
</tbody>
</table>

## Comparison II

| Selection-based | System wide scope | • Loose coupling between sources/sinks  
  • Sink always receive interesting data  
  • Dissemination reaches only interested nodes | • Medium state on mobile nodes  
  • Medium network overhead |
|-----------------|-------------------|----------------------------------------------------------------------------------|
|                 | Limited scope     | • Loose coupling between sources/sinks  
  • Dissemination reaches only interested nodes | • Sink could miss interesting data  
  • Medium state on mobile nodes  
  • Low network overhead |
| Gossip-based    | Context-oblivious | • Low state on mobile nodes  
  • Loose coupling between sources/sinks  
  • Low network overhead | • Sink could miss interesting data  
  • Dissemination can reach not-interested nodes |
|                 | Context-aware     | • Low network overhead  
  • Loose coupling between sources/sinks | • Medium state on mobile nodes  
  • Sink could miss interesting data  
  • Dissemination can reach not-interested nodes |

---

2 Definition of Event-Based Systems

2.1 Models of interaction and EBS
2.2 Constituents of EBS
2.1 Models of interaction and EBS

2.1.1 Reminder: “Request/Reply”
2.1.2 Reminder: “Callback”
2.1.3 Reminder: “Anonymous Request/Reply”
2.1.4 New in this presentation: “Event-Based”
2.1.5 Recap: Models of interaction and EBS
2.1.1 Reminder: “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
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 $\implies$ multi-threading
2.1.3 Reminder: “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—they know the address of the consumer
- The consumer is willing to receive several replies
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 and notifies 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.
- **The trade-off is between the simplicity of request/reply and the flexibility of event-based interaction.**

<table>
<thead>
<tr>
<th>Adressee</th>
<th>Initiator</th>
<th>Consumer</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Initiator</td>
<td>Request/Reply</td>
<td>Callback</td>
</tr>
<tr>
<td>Indirect</td>
<td>Initiator</td>
<td>Anonymous Request/Reply</td>
<td>Event-Based</td>
</tr>
</tbody>
</table>
2.2 Constituents of EBS

- **Event Consumer**
  - F : Filter
  - N : Notification

- **Producer**
  - 1.a. advertise (F)
  - 2. publish (N)
  - 1.b. subscribe (F)
  - 3. notify (N)

- **Notification Service**
- **Publish/Subscribe interface**
- **Communication Implementation**
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
2.2.1 Terminology II

Event notification service:

- Is the mediator in event-based systems that decouples producers from consumers
- Implements a publish/subscribe interface

Operations:

- **publish(n)**: a producer pushes notification \( n \) to the notification service
- **advertise(F)**: a producer advertises that it will send notifications which match the filter \( F \)
- **subscribe(F)**: a consumer subscribes to receive notifications which match the filter \( F \)
- **notify(n)**: the notification service delivers the notification \( n \) to those consumers that have a matching subscription

- Maintains published events and subscribers interests
3 Formal specification of simple event-based system

3.1 Formal background—Temporal logic
3.2 Changes of the state caused by interface operations (w/o advertisements)
3.3 Trace-based specification of simple event-based system (w/o advertisements)
3.4 Changes of the state caused by the adding advertisements
3.5 Safety specification of simple event system with advertisements
3.6 Liveness specification of simple event system with advertisements
3.1 Formal background—Temporal logic

- **Trace**: a sequence of states: \( \sigma = s_0, s_1, s_2, \ldots \)
  - **Subtrace**: \( \sigma|_i \) is the trace \( s_i, s_{i+1}, s_{i+2}, \ldots \)

- Atomic predicate \( P \) is true for every trace whose first state satisfies \( P \)

- Formula \( \Psi \): with quantifiers (\( \forall, \exists \)) and logical operators (\( \lor, \land, \Rightarrow, \neg \))

- **Temporal operators**:
  - \( \Box \) ("always")
    - \( \Box \Psi \) is true for traces \( \sigma \) iff \( \forall i \geq 0, \Psi \) is true for \( \sigma|_i \)
    - \( \Box P \) means \( P \) always holds, i.e. for all subtraces
  - \( \Diamond \) ("eventually")
    - \( \Diamond \Psi \) is true for traces \( \sigma \) iff \( \exists i \geq 0 : \Psi \) is true for \( \sigma|_i \)
    - \( \Diamond P \) means \( P \) will hold eventually, i.e. there exists a subtrace for which \( P \) holds
  - \( \circ \) ("next")
    - \( \circ \Psi \) is true for traces \( \sigma \) iff \( \Psi \) is true for \( \sigma|_1 \)
    - \( \circ P \) means \( P \) holds for the subtrace starting at the second place of the trace
3.1.1 Exercise

? □◊P

? ◊□P

? □[P → □P]

? □[P → ◊Q]

? □[P → ◊□¬P]

? P → ◊□Q

? □¬P ∨ □¬Q ≡ ¬(◊P ∧ ◊Q)
3.2 Changes of the state caused by interface operations (w/o advertisements)

- $X$: a component of a system (being a producer and/or a consumer)
- $C$: the set of all the components
- $S_X$: a set of active subscriptions for component $X$
- $P_X$: a set of published notifications by component $X$
- $D_X$: a set of delivered notifications to component $X$
- $N$: the set of all the notifications; $N \subseteq N$: a set of notifications
  - $n \in N(S_X)$: $X$ has a subscription that matches $n \in N$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub($X, F$)</td>
<td>Component $X$ subscribes to filter $F$</td>
<td>$S_X' = S_X \cup {F}$</td>
</tr>
<tr>
<td>unsub($X, F$)</td>
<td>Component $X$ unsubscribes to filter $F$</td>
<td>$S_X' = S_X \setminus {F}$</td>
</tr>
<tr>
<td>pub($X, n$)</td>
<td>Component $X$ publishes $n$</td>
<td>$P_X' = P_X \cup {n}$</td>
</tr>
<tr>
<td>notify($X, n$)</td>
<td>Component $X$ is notified about $n$</td>
<td>$D_X' = D_X \cup {n}$</td>
</tr>
</tbody>
</table>

- « ′ » indicates the state of a variable after the execution of the interface operation
3.2.1 Exercise

\[ \diamond \text{notify}(X, n) \]

\[ \Box \neg \text{unsub}(X, F) \]

\[ \Box [\text{notify}(X, n) \implies \Diamond \Box \neg \text{notify}(X, n)] \]

\[ \Box [\text{notify}(X, n) \implies n \in N(S_X)] \]

\[ \Box [\text{notify}(Y, n) \implies n \in \bigcup_{X \in C} P_X] \]
3.3 Trace-based specification of simple event-based system (w/o advertisements)

- A component receives
  
  (a) only notifications it is currently subscribed to
  
  (b) only notifications that have previously been published
  
  (c) a notification at most once
  
  (d) all future notifications matching one of its active subscriptions

- **Safety**: demands that “something irremediably bad” will never happen

  \[ \Box \left[ \text{notify} (Y, n) \implies [n \in N(S_Y)] \right] \quad (=a) \]

  \[ \land [n \in \bigcup_{X \in C} P_X] \quad (=b) \]

  \[ \land [\Box \Box \neg \text{notify} (Y, n)] \quad (=c) \]

- **Liveness**: requires that “something good” will eventually happen

  \[ \Box \left[ \Box (F \in S_Y) \implies \Diamond \Box \left[ \text{pub}(X, n) \land n \in N(F) \implies \Diamond \text{notify} (Y, n) \right] \right] \quad (=d) \]
3.4 Changes of the state caused by the adding advertisements

- $A_X$: set of all active advertisements of component $X$
- $n \in N(A_X)$: $X$ has an advertisement that matches $n \in N$

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$adv(X, F)$</td>
<td>Component $X$ advertises filter $F$</td>
<td>$A'_X = A_X \cup {F}$</td>
</tr>
<tr>
<td>$unadv(X, F)$</td>
<td>Component $X$ unadvertises filter $F$</td>
<td>$A'_X = A_X \setminus {F}$</td>
</tr>
</tbody>
</table>
3.5 Safety specification of simple event system with advertisements

(a) + (b) + (c) + 

(e) If a notification is published that does not match any of the active advertisements of the publishing component, the notification should not be delivered to any component

\[ \Box \left[ \text{notify}(Y, n) \implies \Box \Box \neg \text{notify}(Y, n) \right] \quad (=c) \]

\[ \land \left[ \text{notify}(Y, n) \implies n \in \bigcup_{X \in C} P_X \cap N(S_Y) \right] \quad (=b,a) \]

\[ \land \left[ \text{pub}(X, n) \land n \notin N(A_X) \implies \Box \neg \text{notify}(Y, n) \right] \quad (=e) \]
3.6 Liveness specification of simple event system with advertisements

f1) If a client $Y$ is always subscribed to $F$ and a client $X$ always advertises $G$

f2) then there exists a future time where a notification $n$ published by $X$ matches $F$ and $G$

f3) will lead to the delivery of $n$ to $Y$.

$$\square \left[ \square (F \in S_Y) \land \square (G \in A_X) \right] \implies \left[ \Diamond \square (\text{pub}(X, n) \land n \in N(F) \cap N(G)) \implies \Diamond \text{notify}(Y, n) \right]$$
4 Notification filtering mechanisms

4.1 Channels-based filtering
4.2 Subject-based (a.k.a. topic-based) filtering
4.3 Type-based filtering
4.4 Content-based filtering
4.1 Channels-based filtering

- Producers select a channel into which a notification is published.
- Consumers select a channel and will get all notifications published therein.
- Channel identifier is only the visible message part to the event-based service.
- **Framework examples:** CORBA Event Service, CORBA Notification Service, AMQP standard version 0.9.1 (Advanced Message Queuing Protocol).

---

**Diagram:**

```
+----------------------------------+
|   P                             |
|   m1                             |
|   +-------------------------------+
|   m2                             |
|   +-------------------------------+
|   LondonStockMarket             |
|   Stock quotes                  |
|   Technology                    |
|   +-------------------------------+
|   S                             |
|   m1, m2                        |
|   +-------------------------------+
|   Publish                       |
|   Subscribe                     |
|   Notify                        |
|   +-------------------------------+
|   P Publisher                   |
|   S Subscriber                  |
|   +-------------------------------+
```

---

37/73 10/2019

Denis Conan

Distributed Event-Based System — Basics
4.2 Subject-based (a.k.a. topic-based) filtering

- Uses string matching for notification selection
- Each notification and subscription is defined as a rooted path in a tree of subjects
- Example:
  - A stock exchange application publishes new quotations of FooBar under the subject: `/Exchange/Europe/London/Technology/FooBar`
  - Consumers subscribe for `/Exchange/Europe/London/Technology/*/` to get all technologies quotations
- The subject/topic is in message header, the content is “opaque”
- Example of solution: AMQP standard version 0.9.1 (Advanced Message Queuing Protocol), 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
4.3 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
4.4 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

m1: {"company": "Telco", "price": 120}
m2: {"company": "Telco", "price": 90}
Filter F: {"company": "Telco", "price": < 100}
5 Topic-based filtering w/ AMQP v.0.9.1

5.1 Overview of a topic-based filtering system
5.2 Producer, queue, and consumer
5.3 Exchange
5.4 Message delivery properties

The content of this section is extracted from
http://www.amqp.org/specification/0-9-1/amqp-org-download and from
5.1 Overview of a topic-based filtering system

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

- **Producer** = A program that sends messages to a queue
- **Queue** = name for a “post box” that lives inside the AMQP server
  - Messages are only stored inside a queue
  - 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
- **Consumer** = A program that waits to receive messages
5.3 Exchange

- An exchange = A matching and routing engine
  - It inspects notifications (headers)
  - Using its binding tables, decides how to forward these notifications to message queues or other exchanges

- A binding = A relationship (queue, exchange) with a binding key

- A binding key = A notification routing criteria

- An exchange never store notifications

- An exchange receives notifications from publishers and routes them to message queues based on binding key/criteria
5.3.1 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
5.3.2 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 $K = R$
5.3.3 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]

- The binding key follows the same rules as the routing key with:
  - “*” that matches a single word and “#” that matches 0 or more words
5.3.4 Emulation of RPC-like call

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)
- "replyTo": Commonly used to name a callback queue
- "correlationId": Useful to correlate RPC responses with requests
5.4 Message delivery 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
       \[\implies\] May be lost if the consumer fails
    2. `autoAck=false`: The server waits for an explicit acknowledgement
       \[\implies\] Memory leakage if the consumer forgot to send the acknowledgement

- **Message durability**
  - 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”
5.4.1 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
6 Topic-based filtering w/ Apache Kafka

6.1 Cluster-based architecture
6.2 Topics as structured commit logs
6.3 Consumer groups
6.4 Fault tolerance

All the content of this section is extracted from https://kafka.apache.org.
6.1 Cluster-based architecture

- Kafka is run as a cluster on one or more servers that can span multiple datacenters.
- The Kafka cluster stores streams of records in categories called topics.
- Each record consists of a key, a value, and a timestamp.
- 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

- A topic = stream of records = partitioned log = structured commit log
- Each partition is an ordered, immutable sequence of records that is continually appended to—i.e. a structured commit log
- A partition must fit on the server that hosts it
- A topic may have many partitions ⇒ an arbitrary amount of data
- Partitions act as the unit of parallelism
- Records are assigned a sequential id number called the offset
- Consumers can consume records in any order they likes
6.3 Consumer groups

- Consumers join groups, which are labelled with a consumer group name
- Consumer instances can be in separate processes or on separate machines
- Each record published to a topic is delivered to one consumer instance within each subscribing consumer group
  - If all the consumer instances are in the same group, then records are load balanced
  - If all the consumer instances are in different groups, then records are replicated
- There cannot be more consumer instances in a group than partitions
### 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”.
- Each host acts as a leader for some of its partitions and 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.
7 Topic-based filtering w/ OASIS MQTT v.3.1.1

7.1 MQTT features
7.2 Topic filters w. wildcards and topic names
7.3 QoS—Message reliability
7.4 Disconnections

The content of this section is extracted from
7.1 MQTT features

- Initially, a proposition made by IBM

- MQTT v.3.1: an OASIS standard in Oct. 2014
  Then, MQTT v.3.1: an ISO/IEC standard (20922:2016) in June 2016
  Today, MQTT v.3.1.1 Plus Errata 01: an OASIS standard in Dec. 2015

- It runs over TCP/IP, or over other network protocols that provide ordered, lossless, bidirectional connections
  - MQTT-SN was proposed for sensor networks in which these network conditions cannot be assumed, for instance using UDP

- 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, Microsoft Azure IoT Hub
7.2 Topic filters w. wildcards and topic names

- **Topic filter** = an expression contained in a subscription

- **Topic name** = the label attached to a message which is matched against the subscriptions
  - A server can change the topic name of a publish packet

- “/,#,+” can be used in topic filters, but not within topic names

- The forward slash (“/”) is used to separate each level within a topic tree and provide a hierarchical structure to the topic names
7.2 Topic filters w. wildcards and topic names II

- 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"
  - E.g. "sport/#" matches "sport", since "#" includes the parent level
  - E.g. "sport/tennis#" is not valid
  - E.g. "sport/tennis/#/ranking" is not valid
7.2 Topic filters w. wildcards and topic names III

- 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”
  - E.g. “sport/+” does not match “sport” but it does match “sport/”
  - E.g. “+” and “+/tennis/#” are valid
  - E.g. “sport+” is not valid
  - E.g. “/finance” matches “+/+” and “/+”, but not “+”. 
Special character “$”

- Server 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 server must not match topic filters starting with a wildcard character (“#” or “+”) with topic names beginning with “$”
- The server should prevent clients from using such topic names to exchange messages with other Clients
7.3 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
    - This level could be 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

⇒ Client and server store session state in order to provide QoS levels 1 and 2

- A subscription contains a topic filter and a maximum QoS
  - The server might grant a lower maximum QoS than the subscriber requested
  - QoS properties are associated with a single session (possibly several connections)
  - When filters overlap, the delivery respects the maximum QoS of all the matching subscriptions
7.3.1 Subscription and publication with QoS0/“At most once”

- The server 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 server should never send a duplicate
7.3.2 Delivery of QoS0/“At most once” messages

- The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver.

- No response is sent by the receiver and no retry is performed by the sender.

- The sender sends a publish packet with QoS=0, DUP=0¹

- The receiver accepts ownership of the message when it receives the publish packet.

---

¹ DUP is set to 1 when the sender knows it is a duplicate
7.3.3 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
7.3.4 Delivery of QoS1/“At least once” messages

- A QoS1 publish packet has an Id and is acknowledged.

- 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, irrespective of the setting of its DUP flag

Sender

Receiver

Send PUBLISH with QoS=1, DUP=0, <Id>

Initiate onward delivery of the Application Message (*1)

Send PUBACK <Id>

Discard message (*2)

(*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
7.3.5 Subscription and publication with QoS2/“Exactly once”

- The server grants a maximum QoS2
- A topic filter at QoS 2 = delivery of a message at the QoS with which it were published
7.3.6 Delivery of QoS2/“Exactly once” messages

- 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
7.3.6 Delivery of QoS2/“Exactly once” messages

Sender

- Store message

Receiver

- Send PUBLISH with QoS=2, DUP=0, <Id>
  - Method A: Store message
  - Method B: Store <Id> and then initiate onward delivery of the Application Message (*1)
- Send PUBREC <Id>
- Send PUBREL <Id>
- Send PUBCOMP <Id>

Discard message (*2) and store PUBREC received <Id>

Discard stored data

(*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
7.4 Disconnections

7.4.1 Sessions
7.4.2 RETAIN flag in a publish packet
7.4.3 Message ordering
7.4.1 Sessions

- Session = A stateful interaction between a client and a server/broker

- Some sessions last only as long as the network connection, others can span multiple consecutive network connections

- When a client connects with CleanSession set to 0, it is requesting that the server maintain its state after disconnection

- When a client has determined that it has no further use for the session, it should connect with CleanSession set to 1 and then disconnect

- A server is permitted to disconnect a client that it determines to be inactive or non-responsive at any time
If the RETAIN flag is set to 1, the server must store the message and its QoS so that it can be delivered to future subscribers.

When a new subscription is established, the last retained msg (if any) is sent.

If the server receives a QoS0/“At most once” message with the RETAIN flag, it discards any message previously retained for that topic.

- It should store the new message as the new retained message for that topic.
- But it may choose to discard it at any time.
  - If this happens there will be no retained message for that topic.
7.4.3 Message ordering

- When a client reconnects with `CleanSession` set to 0 when connecting, both the client and server must re-send any unacknowledged publish packets (where QoS > 0) and PUBREL packets using their IDs.

- A server 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”