# Software Technology for Internet of Things

## Stream Processing

Aslak Johansen asjo@mmmi.sdu.dk

Apr 1, 2025

# Part 0:
# Introduction

# Anatomy of a (Typical) IoT Cloud

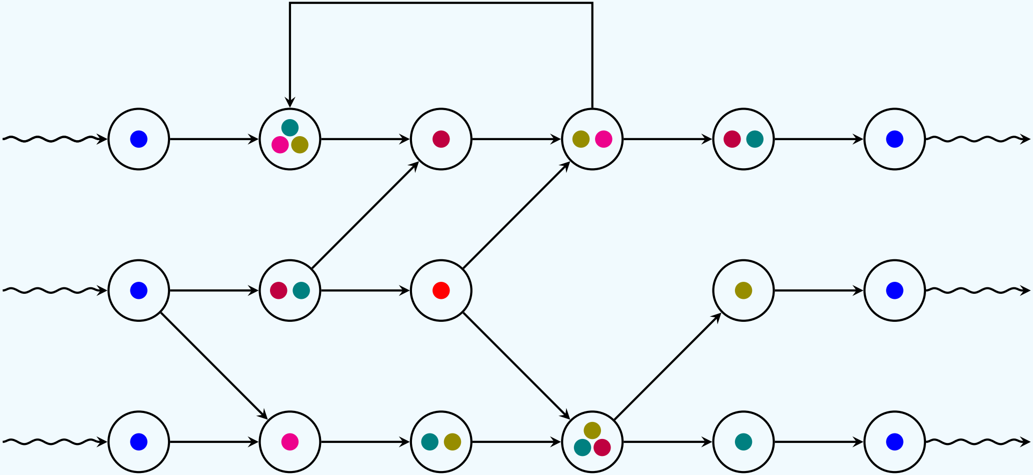
Key properties include:

- Large total number of incoming data streams.
- A large number of customers
  - Customers expect their data to stay private (legal reasons).
  - The processing setup is the same for all customers*.
- Processing have different justifications:
  - Required for promoted functionality.
  - Real-world evaluation of product line.
  - Pattern detection (e.g., for future products).
  - Health monitoring.
- A small number of the processing products are presented to the customer.
- Potentially large number of outgoing data streams.

* at least it is close.

# Processing Graph

These have various names, including: graph processing, stream processing.



# General Processing Case

A stream process is independent piece of logic associated with:

- State.
- Zero or more input streams.
- Zero or more output streams.

A node consumes input(s) in order to produce output(s).

Specific cases:

- ***Source:*** A node only producing output stream(s).
- ***Sink:*** A node only consuming input stream(s).

# Specific Processing Properties

- ***Flow Control*** What is the right policy when data is consumed at a lower rate than it is produced?
  - Buffering or dropping?
- ***Consumability*** Does processing consume the data or should it be retained?
- ***Trigger Condition*** If a node consumes more than one input, when does it fire?
  - Synchronize inputs using a barrier or use latest values?
  - What happens if a value is dropped?
- ***Trigger Arbitrarity*** Can a node fire without inputs?
  - E.g., to implement timeouts.
- ***Batch Size*** Can we accept the latency penalty of operating in batches, and how big should we make them?
- ***Fault Tolerance*** What is the right policy when a node dies?

# Concerns and Remedies

There are a few general problems when dealing with lots of concurrent pieces of logic:

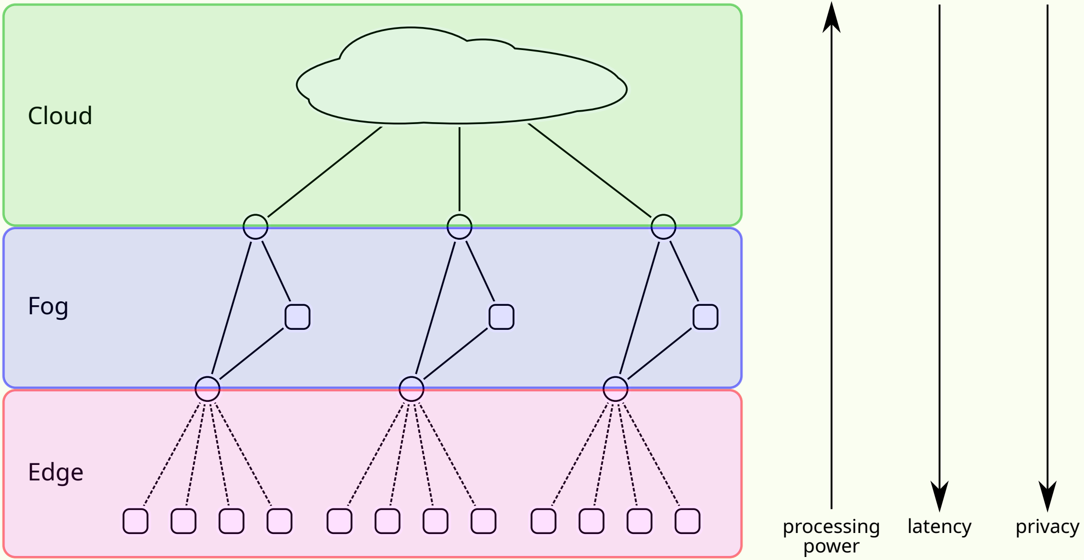
- Context switching
  - Switching – on OS level – from the context of one process to the context of another is one of the most expensive operations.
  - Preempting a process thread with another is still an expensive operation.
  - Multiplexing *can* be *relatively* inexpensive.
- Side-Effects
  - Code with side-effects generate cache misses.
  - Without side-effects, concurrency is trivial$^\dagger$.
- Network
  - In a share-nothing architecture, an update is satisfied by a(ny) single node.

*"Going to disk is 25 million times slower than hitting a general purpose register. Design accordingly."*

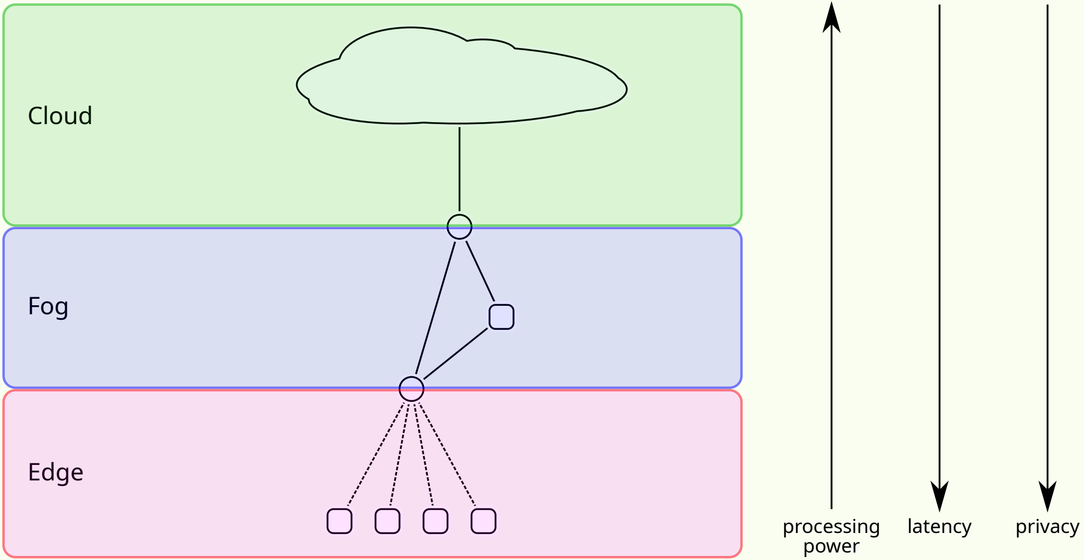
– Robert Love, GUADEC 2005

# Part 1:
# System Design

# Overview



# Overview



# Part 2:
# Message Queues

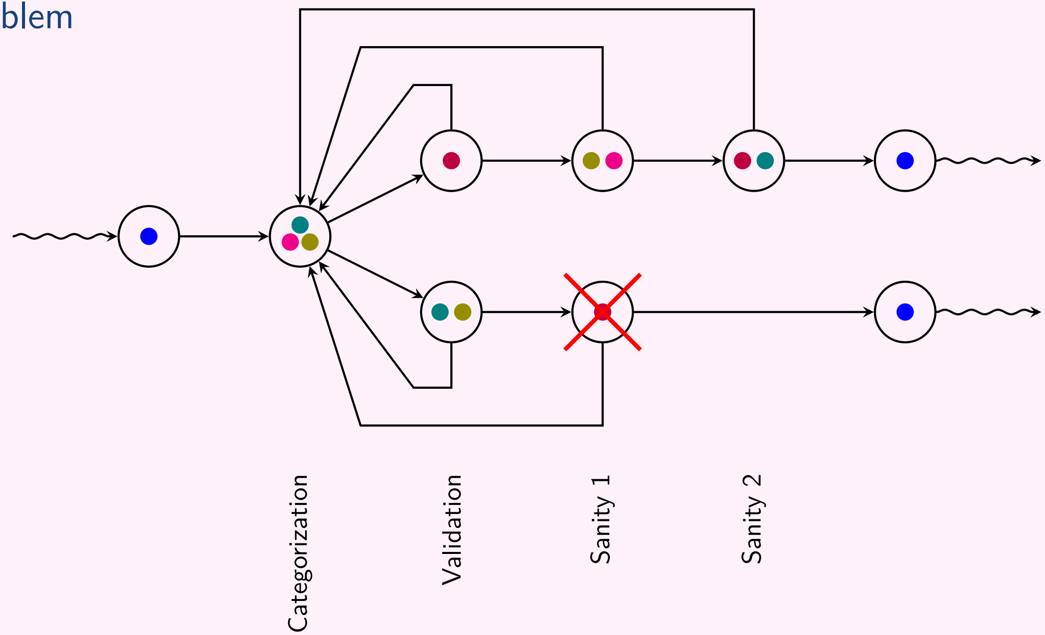
# Case

Lets look at a data ingestion situation:

- Incoming data needs to be stored in a database.
- A few steps are inserted along this path covering
  - Categorization
  - Validation
  - Sanity checking
- A failed step may require another processing attempt (e.g., using the notion of a TTL).
- All incoming data needs to be processed, independently of the availability of the components along the way.

What could be defining for the design?

# Problem



Categorization

Validation

Sanity 1

Sanity 2

# Part 3:
Event Busses

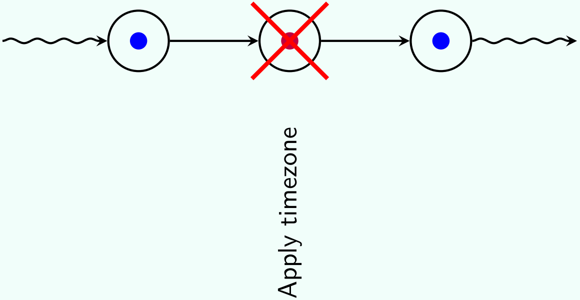
# Case

Lets look at a clock signal:

- A stream of timestamps are being received.
- These needs to be adjusted for timezone.
- Timeliness matters.

What could be defining for the design?

# Problem



# Part 4:
# Publish Subscribe Pattern

# Publish Subscribe

Can someone describe what publish subscribe is?

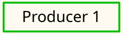
A communication pattern, in which:

- Messages can be published through some name on a *broker*.
- Clients can subscribe to names on a broker.
- Whenever a message is published, all clients subscribing to that name are notified (i.e., forwarded the message).

Often referred to as *pubsub*.

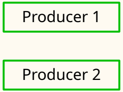
# Publish Subscribe

# Publish Subscribe



Producer 1

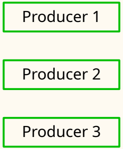
# Publish Subscribe



Producer 1

Producer 2

# Publish Subscribe

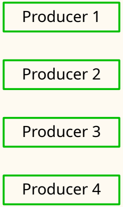


Producer 1

Producer 2

Producer 3

# Publish Subscribe



Producer 1

Producer 2

Producer 3

Producer 4

# Publish Subscribe

Producer 1

Producer 2

Producer 3

Producer 4

Consumer 1

# Publish Subscribe

Producer 1

Producer 2

Producer 3

Producer 4

Consumer 1

Consumer 2

# Publish Subscribe

Producer 1

Producer 2

Producer 3

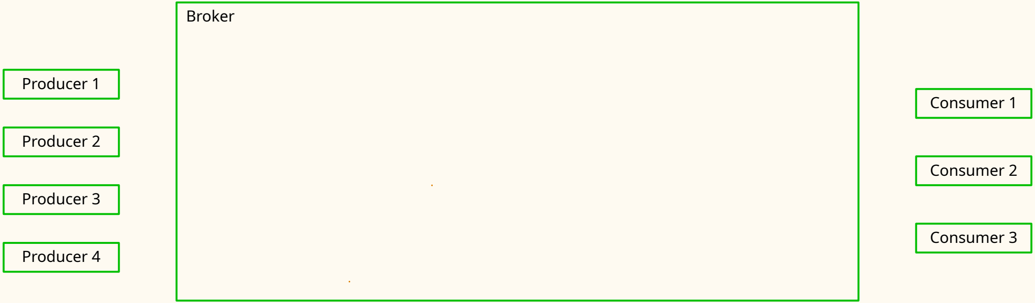
Producer 4

Consumer 1

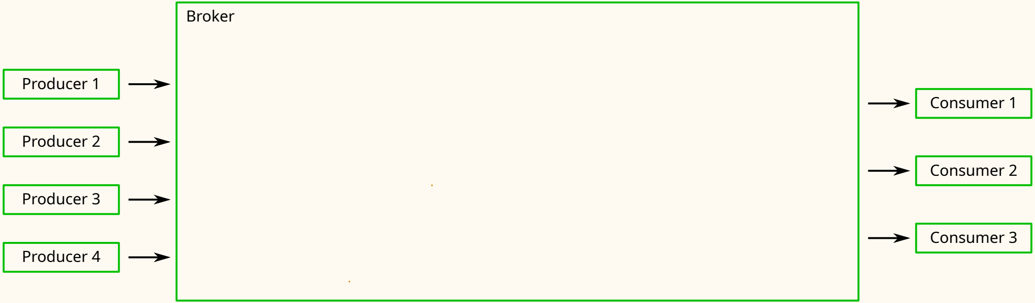
Consumer 2

Consumer 3

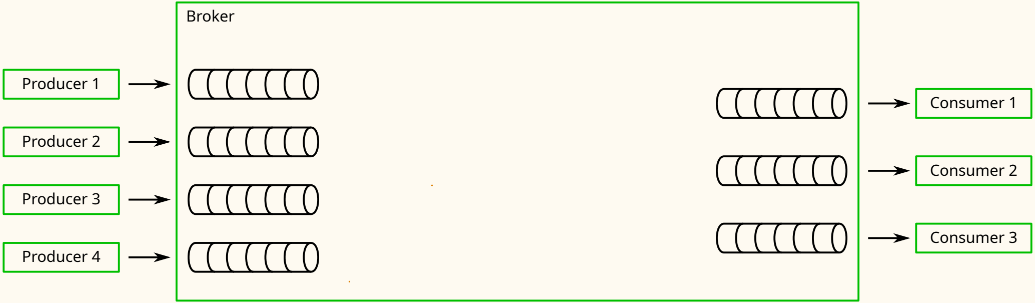
# Publish Subscribe



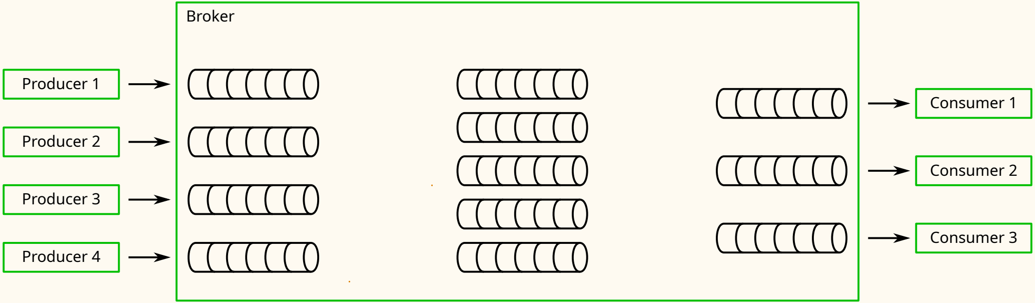
# Publish Subscribe



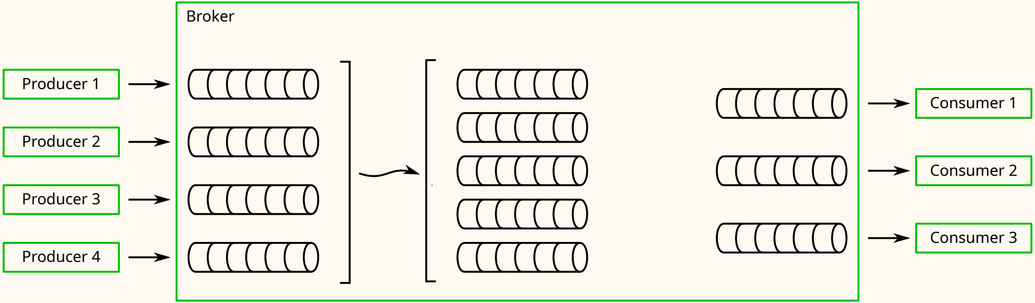
# Publish Subscribe



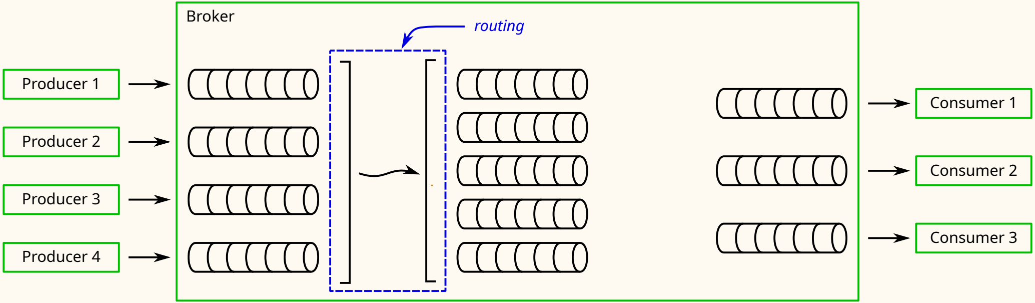
# Publish Subscribe



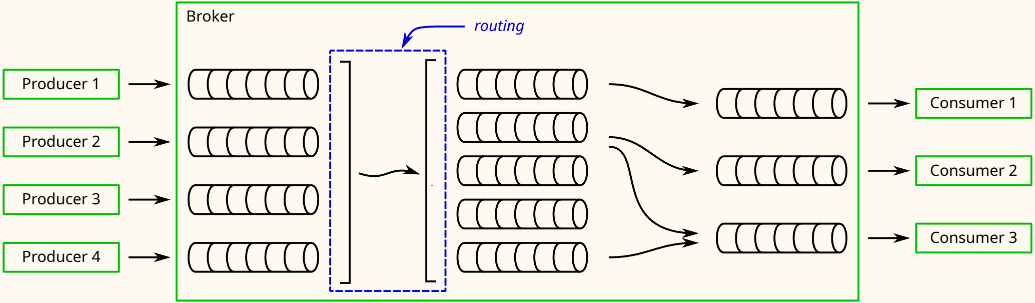
# Publish Subscribe



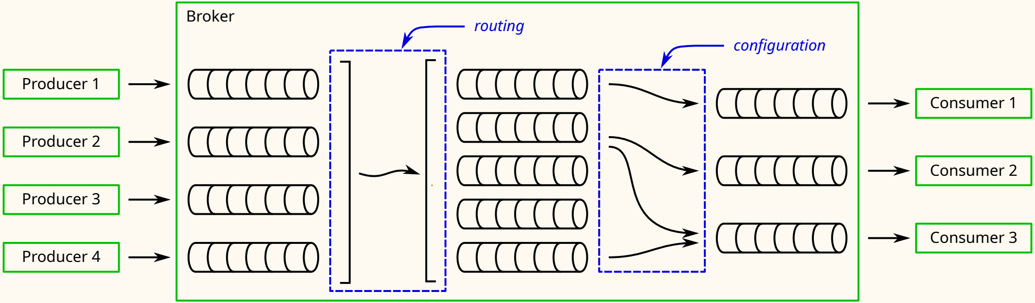
# Publish Subscribe



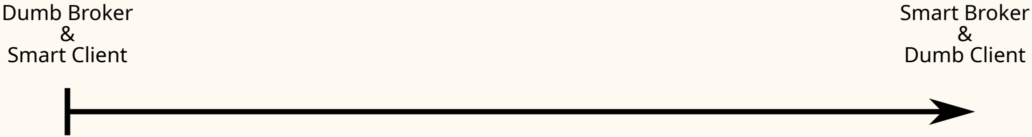
# Publish Subscribe



# Publish Subscribe



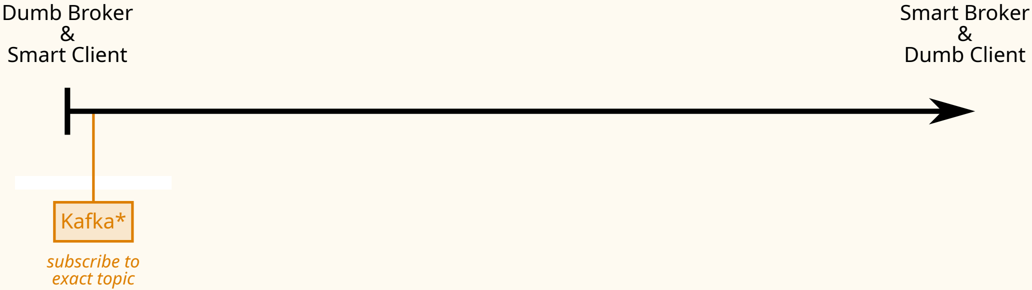
# **Publish Subscribe** Smart and Dumb Components



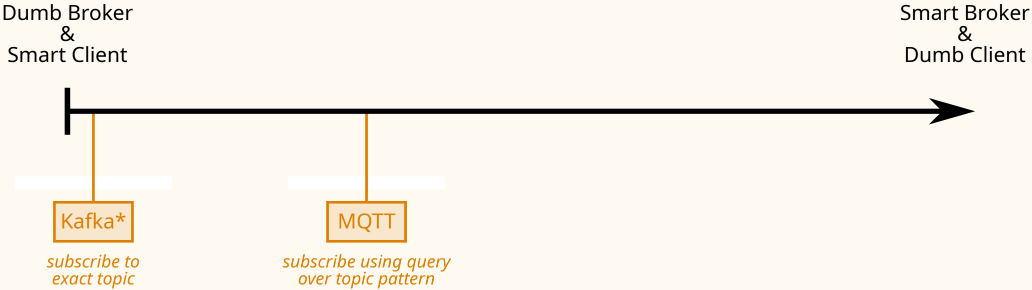
Dumb Broker
&
Smart Client

Smart Broker
&
Dumb Client

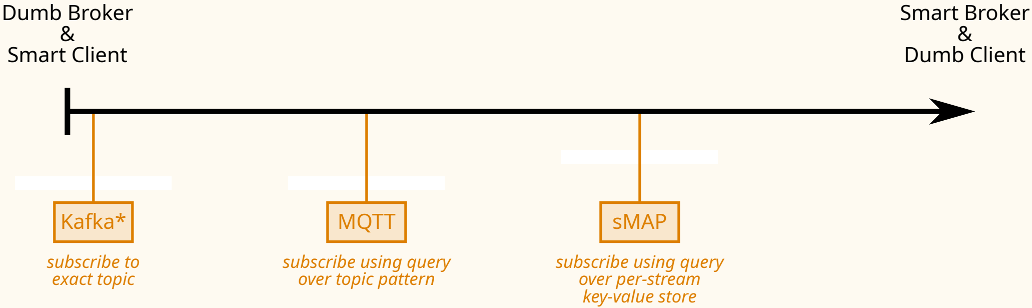
# **Publish Subscribe** Smart and Dumb Components



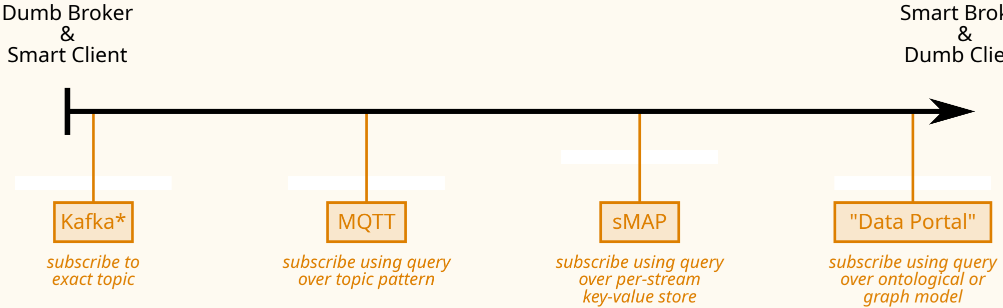
# **Publish Subscribe** Smart and Dumb Components


Dumb Broker
&
Smart Client
Smart Broker
&
Dumb Client
Kafka*
*subscribe to exact topic*
MQTT
*subscribe using query over topic pattern*

# **Publish Subscribe** Smart and Dumb Components


Dumb Broker
&
Smart Client
Smart Broker
&
Dumb Client
Kafka*
subscribe to exact topic
MQTT
subscribe using query over topic pattern
sMAP
subscribe using query over per-stream key-value store

# **Publish Subscribe** Smart and Dumb Components



Dumb Broker
&
Smart Client

Smart Broker
&
Dumb Client

Kafka*

*subscribe to exact topic*

MQTT

*subscribe using query over topic pattern*

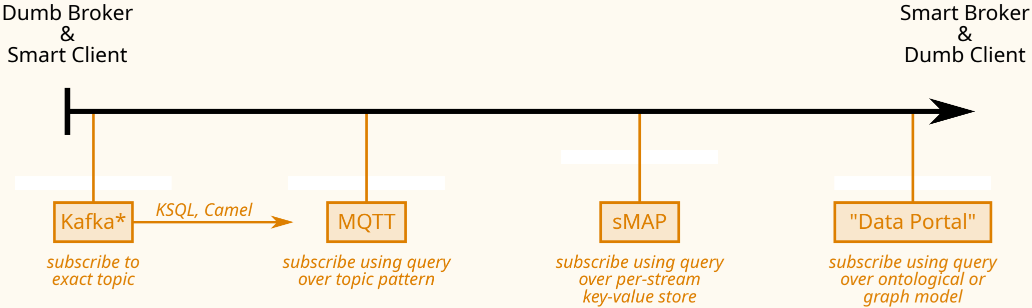
sMAP

*subscribe using query over per-stream key-value store*

"Data Portal"

*subscribe using query over ontological or graph model*

# **Publish Subscribe** Smart and Dumb Components


Dumb Broker
&
Smart Client
Smart Broker
&
Dumb Client
Kafka*
KSQL, Camel
MQTT
sMAP
"Data Portal"
subscribe to exact topic
subscribe using query over topic pattern
subscribe using query over per-stream key-value store
subscribe using query over ontological or graph model

# Message Queuing Telemetry Transport (MQTT)

An example of a subsub protocol that is frequently used in the IoT space.

Properties:

- Subscriptions through topic patterns.
- Identifier patterns may be expressed using two kinds of wildcards:
  - Topics are matched against patterns.
  - Topics are structured strings and use "/" as separator.
  - The "+" wildcard matches a single level.
  - The "#" wildcard may match multiple levels.
- Quality of service (for each of the two transmissions): at-most-once, at-least-once and exactly-once.
- Last will and testament: On-disconnect events.
- Persistent sessions: Per-client offline buffering.
- Very simple protocol!
  - Broker and framework choice.
  - Innovation.

# Part 5:
# Distributed Concerns

# Dealing with System Failures

Rule: *"As the number of machines in a system goes towards infinity the time to one of them breaking goes to zero"*.

Accordingly, as we grow our system, the need for dealing with failures increases.

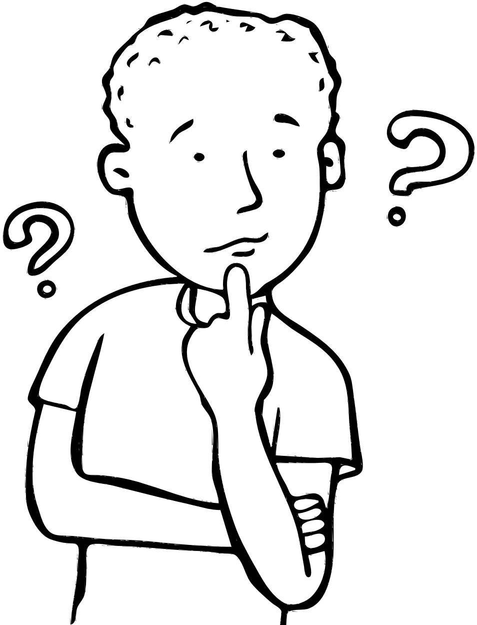
Several support systems have been designed to deal with this **on a service level**.

They generally:

- Can be told to run a service on *n* machines.
- Makes sure to restart a service when it goes down.
- Hosts distributed filesystem(s) on the same hardware as the services.
- Allows for geographical redundancy.

For heterogeneous and low-granularity deployments, Kubernetes and Ceph comes into play, with Docker Swarm being another option. For homogeneous and high-granularity deployment BEAM is king. Often these are combined.

# Questions?



https://openclipart.org/detail/238687/boy-thinking-of-question