IV. Message Queuing Systems

Application Domain: Bridging heterogenous applications

$\Longrightarrow$ suitable for EAI as well as B2Bi

$\Longrightarrow$ low-level use for the Internet of Things

◁ Applications run in different geographic locations

◁ Application environments evolved over decades

◁ Global standardization (often) not feasible

$\Longrightarrow$ Bottom-Up **Integration** of isolated applications towards interacting IT 'landscapes'

$\Longrightarrow$ Evolving and always changing IoT environments

Requirements: Explicit message-passing interaction

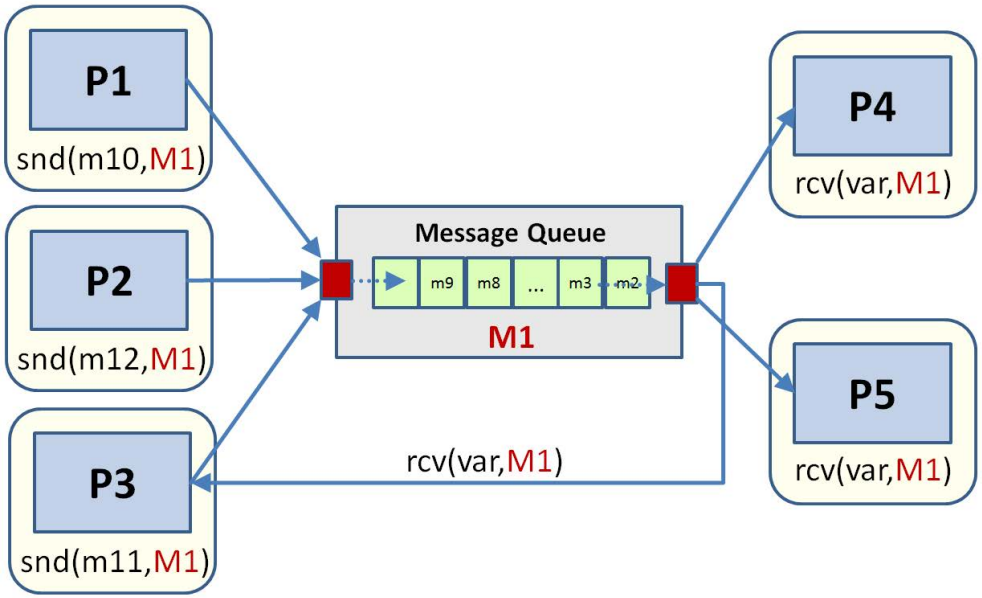
◁ Arbitrary complex messages: size and structure for 'Clients'

◁ Interaction among separately developed applications

$\Longrightarrow$ **Wrapper** for message alignment needed

◁ Tight synchronization tends to be too error-prone

Messaging = Message Passing plus Queuing



Basic Principles: Support for **Asynchronous Msgs** and **Queues**

* interfaces to drop/pick-up messages
* logical naming schemes for multi-cast interaction
* reliable message transfer based on persistent Queues

Decoupling Effects $\Longrightarrow$ convenient and secure message passing

- **Asynchronous:** decoupling w.r.t. time
- **Multi-Cast**: decoupling w.r.t. concrete 'sender' or 'receiver'

IV.1 Basic Characteristics of the Messaging Model

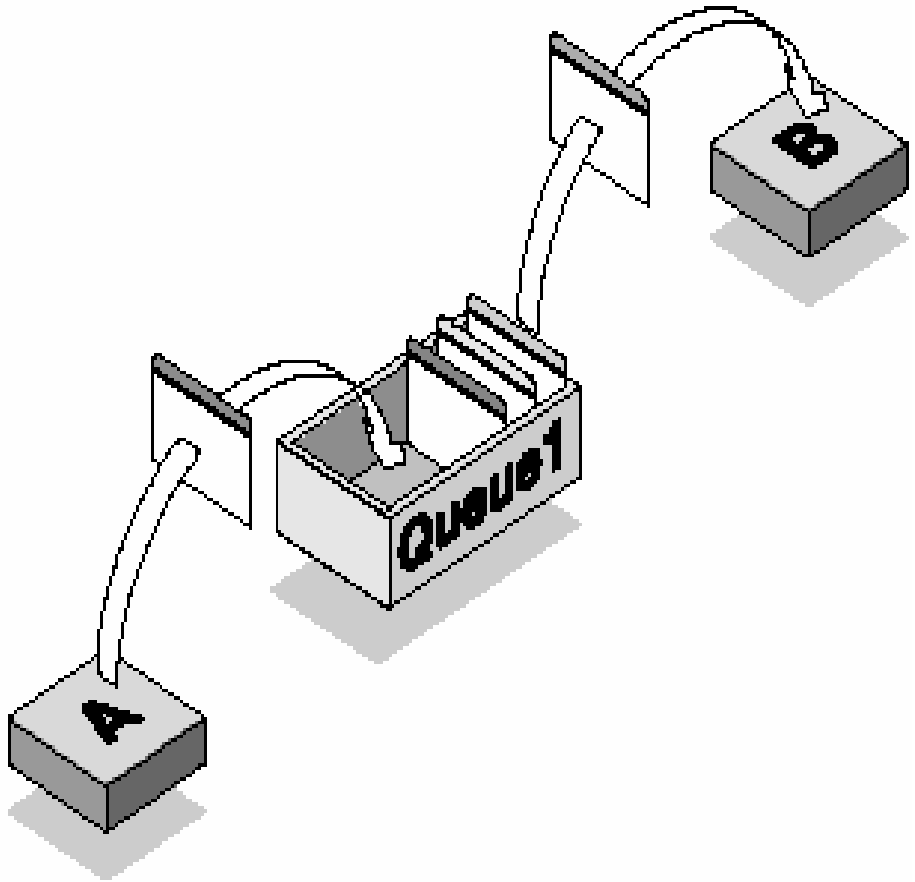
- **Queue Adresses** instead of naming communication partners
  $\Longrightarrow$ Abstraction decouples one-way interaction IV-??

◁ Queues are typically **unidirectional** channels
  $\Longrightarrow$ **Two-way interaction** requires two distinct Queues IV-??

- A wide range of unidirectional interaction 'styles' is easily supported:
  - Multi-destination queues: '$get\ consumes$' IV-??
  - Selective Routing: disjoint vs. shared Queue-Bindings IV-??
  - Multi-Source–Multi-Destination interaction IV-??
    $\Longrightarrow$ **unidirectional m-n channel**
  - Publish/Subscribe–Model: variants depend on $read$ vs. $consume$ 'logic of 'get' operation. IV-??
    $\Longrightarrow$ **unidirectional m-n-Multicast**

Uni-directional One-to-One using a single Queue

Fig.: IBM TR GC33-0805 1995

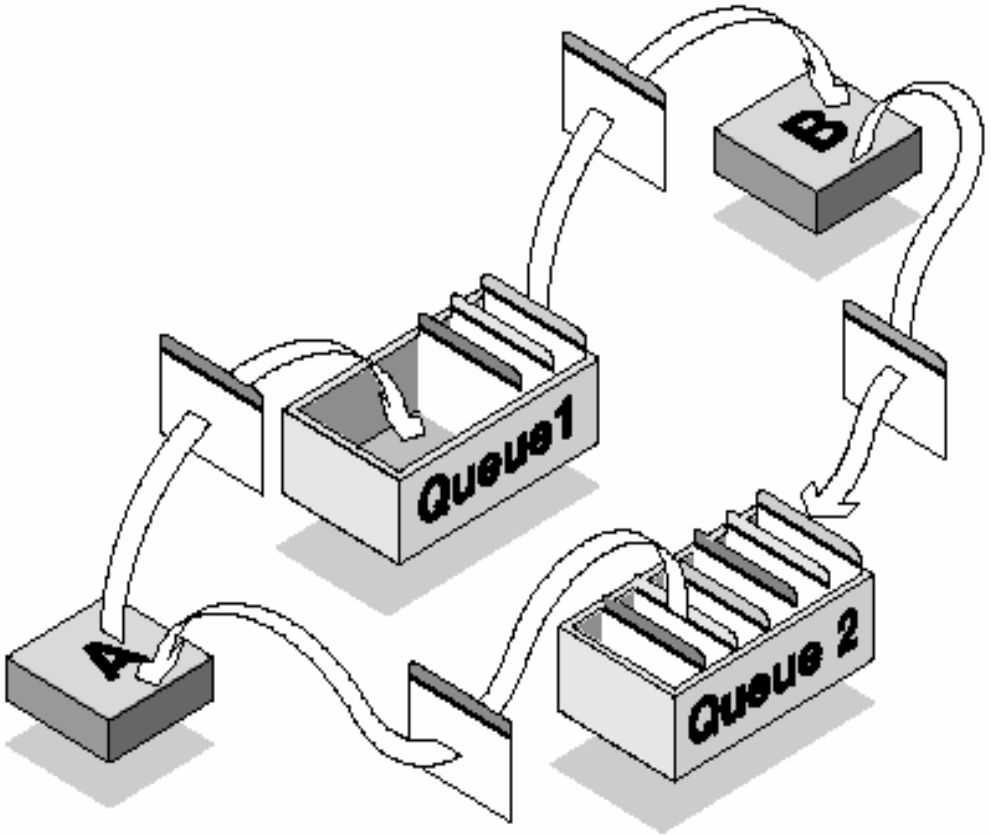


Abstraction:

1. A hands Msg to local `Queue Interface` Queue1 is used as the 'Address'
2. `Queue` stores and transfers Msg
3. B extracts Msg from local `Queue Interface`

Bi-directional Point-to-Point requires two Queues

Fig.: IBM TR GC33-0805 1995



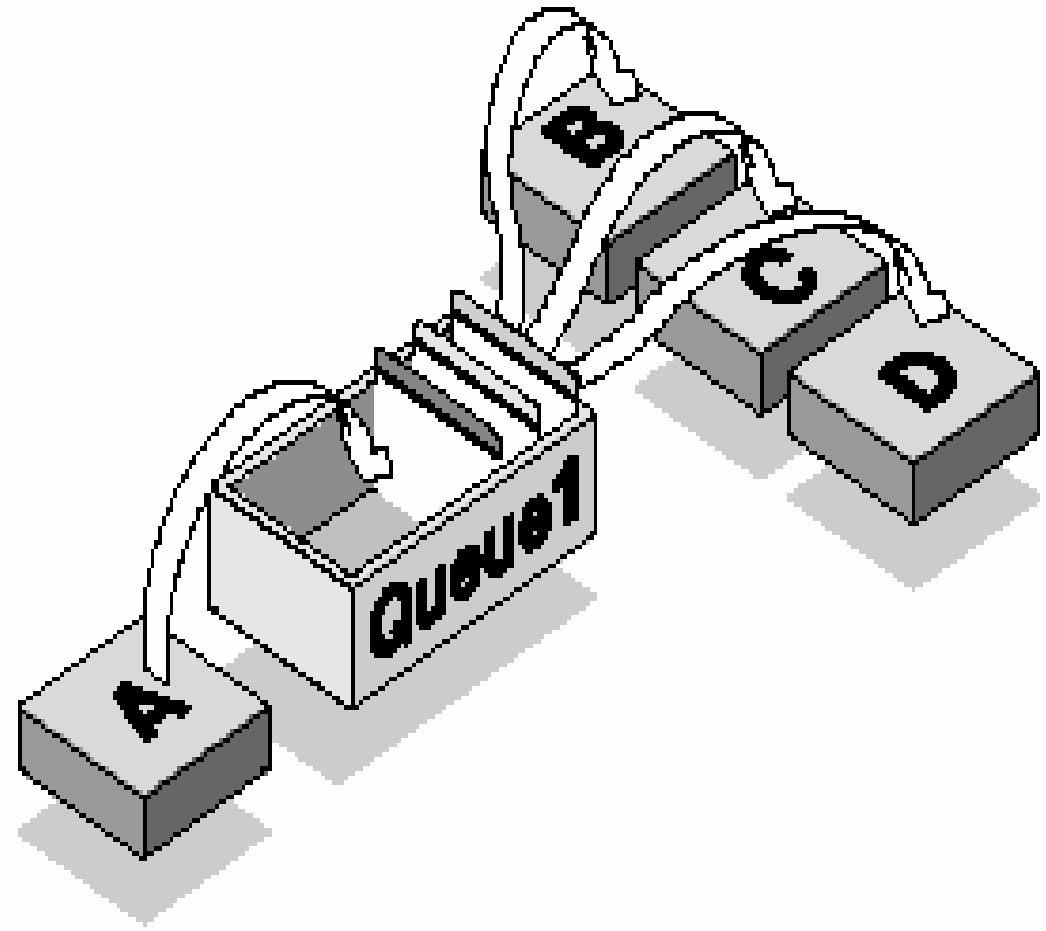
A sends Msg1 via `Queue1`;
B replies with Msg2 using `Queue2`;
Decoupling through asynchronous reaction

Applications:

- Inquiry/Result
- Order with confirmation
- RPC: Call and Reply

Uni-directional One-to-Many using a single Queue

Fig.: IBM TR GC33-0805 1995



Abstraction:

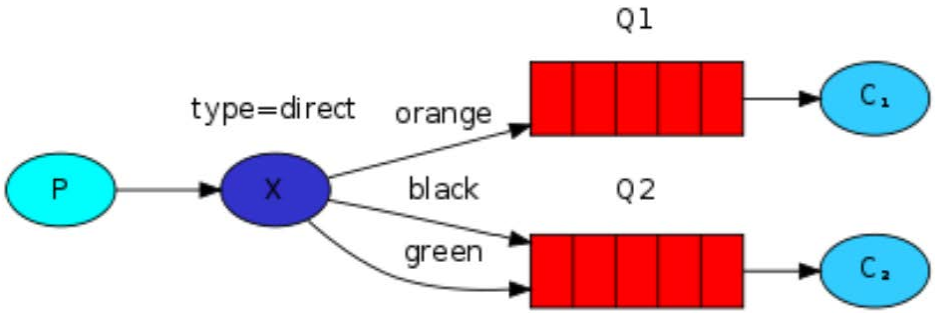
1. A hands Msgs to local `Queue Interface`
2. `Queue` stores and transfers Msgs
3. B, C, D **consume** Msgs from local `Queue Interface`
4. Each Msg is processed by **one** Receiver

`Work Queues` $\Longrightarrow$ **Competing Consumer Pattern**

Selective Routing with Filtering Attributes

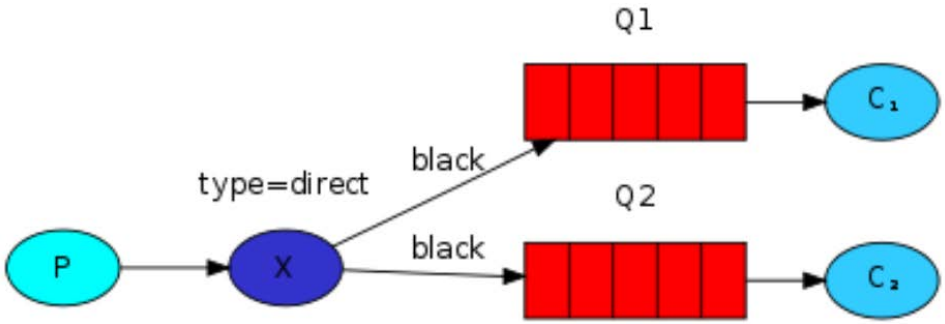
Fig.: www.rabbitmq.com/getstarted.html

Attribute(s)/Bindings: between 'Exchange' and 'Queues'



disjoint $\Longrightarrow$

Msgs go to specific queues



shared $\Longrightarrow$

Msgs go to multiple queues

Remark: *More on* `RabbitMQ` *in section IV.4*

Multiple One-to-One using a single Queue

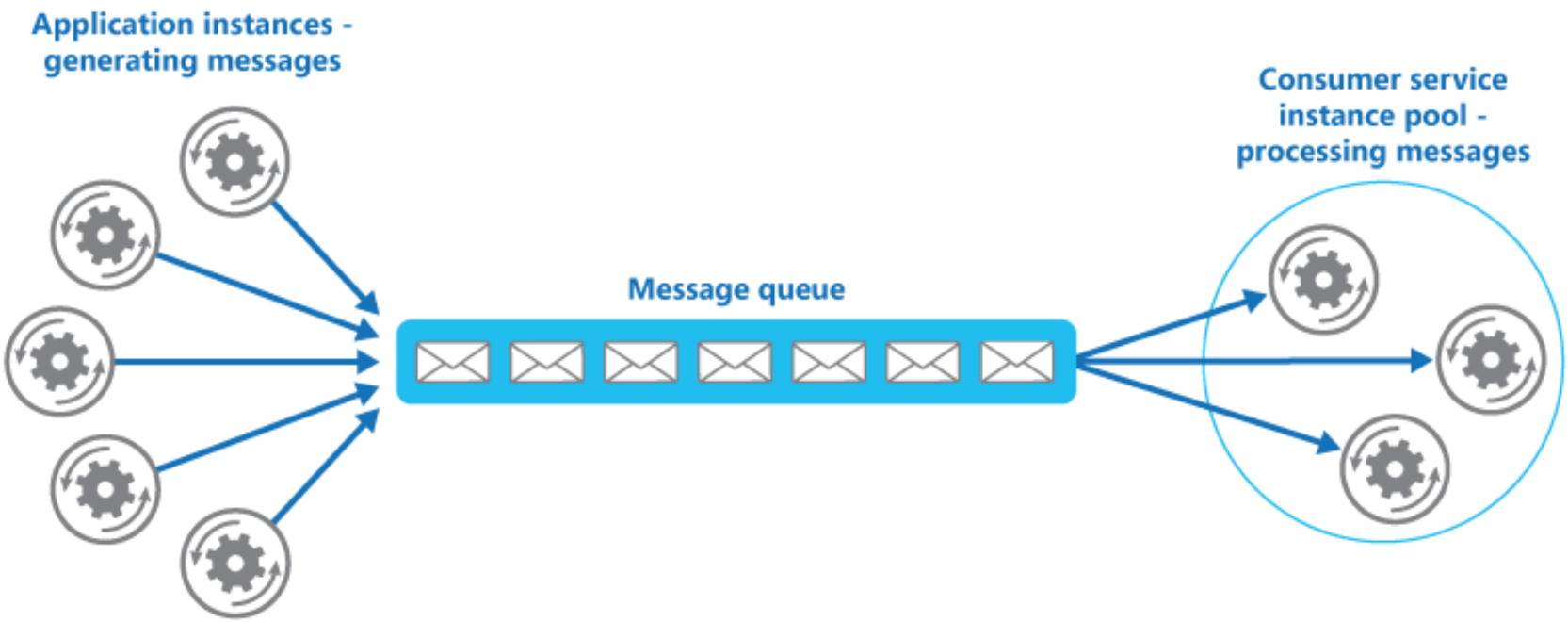


Fig.: docs.microsoft.com/en-us/azure/architecture/patterns/

- Msgs are extended to hold specific receiver addresses or logical attributes allowing inquiries by potential receivers
- All Sender/Receiver use common queue endpoint
- Lots of processes use the same queue $\Longrightarrow$ reduced overhead
- Each Msg goes to a single Receiver

Application: load balancing, work-list processing (Workflow)

Publish-Subscribe Organization using 'Topic's

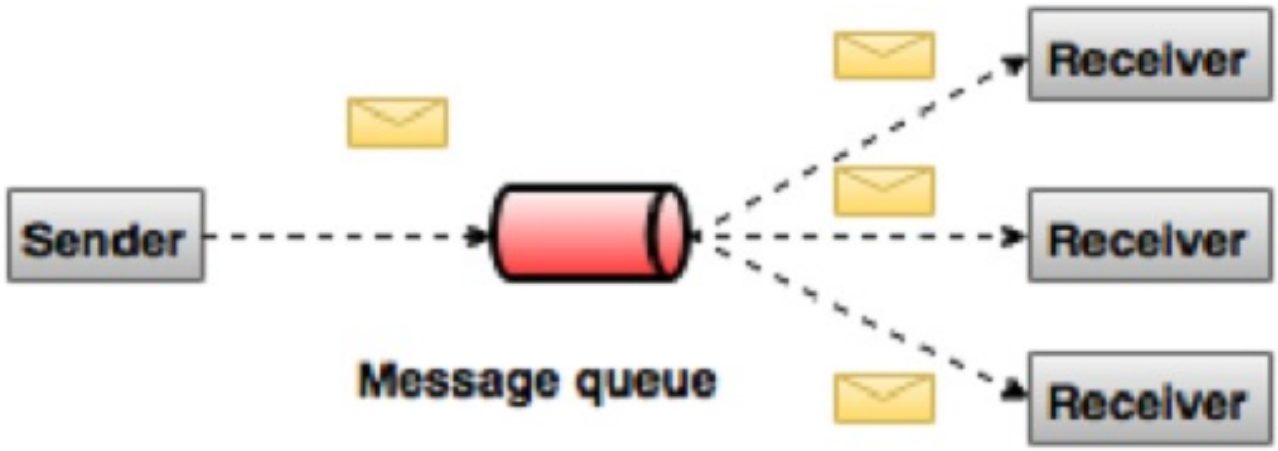


Fig.: www.tutorialspoint.com/apache_kafka/apache_kafka_introduction.htm

- Abstraction: tagging msgs with/subscribing to specific **topic**s
- Enhanced Decoupling w.r.t.
  - ∗ Logic: no direct Sender/Receiver-Roles required
  - ∗ Time: *durable subscription* msgs stored for offline subscribers
    expiration date limits costs for storing msgs indefinitely

Application: Order goes to processing, logging and accounting

IV.2 Message Queuing 'Products'

- **Messaging – been around for decades and is still alive:**
  $\Longrightarrow$ Starting Point `IBM MQSeries` in 1993
  - `IBM MQ V.9.1` cloud-based product suite today
  - `Microsoft MSMQ Series`: almost the same ...
- Support from all **Integration** and **Cloud Providers:**
  - Important part of ESB-, SOA- or Cloud-Suites
    * `MS Azure Service Bus` or `Storage Queues`
    * `Amazon AWS Simple Queue Service` (AWS SQS)
    * Open Source Products: `Apache ActiveMQ`, `RabbitMQ`, ...
  - Used to connect all kinds of modern infrastructure
    * Microservice architectures, Serverless (AWS Lambda), ...
    * Streaming applications (e.g. `Apache Kafka`); ...
- **New application areas**: Embedded systems and the IoT

www.ibm.com/support/knowledge center/en/SSFKSJ_com.ibm.mq.pro.doc/q001010_.htm

Modern Offerings and Standards

- Java: **J**ava **M**essage **S**ervice JMS 2.0a Spec. (03/2015)
  - **Interface Architecture**: Java $\longleftrightarrow$ Messaging started in 1998; Jakarta Messaging 3.0 in Jakarta EE 9 — jakarta.ee
  - Open Source implementations and commercial version(s)
    - Amazon SQS, JBoss messaging, ...
    - Apache ActiveMQ, Rabbit MQ Client/Plugin, ...
- **Emerging 'Standards'**:
  - Advanced Message Queuing Protocol (**AMQP**) — OASIS ISO/IEC 19464
    alternative to http $\Longrightarrow$ general interaction (Vers.1.0, 2014)
  - Message Queuing Telemetry Transport (**MQTT**) — OASIS
    based on TCP/IP $\Longrightarrow$ light-weighted for IoT (Vers.5.0, 2019)
  - Streaming Text Oriented Messaging Protocol (STOMP)
    based on http $\Longrightarrow$ Text-based interaction (Vers.1.2; 2012)

IV.3 Java Message Service API – Overview

from: Jakarta EE Tutorial, chap. 48 eclipse-ee4j.github.io/jakartaee-tutorial/2021

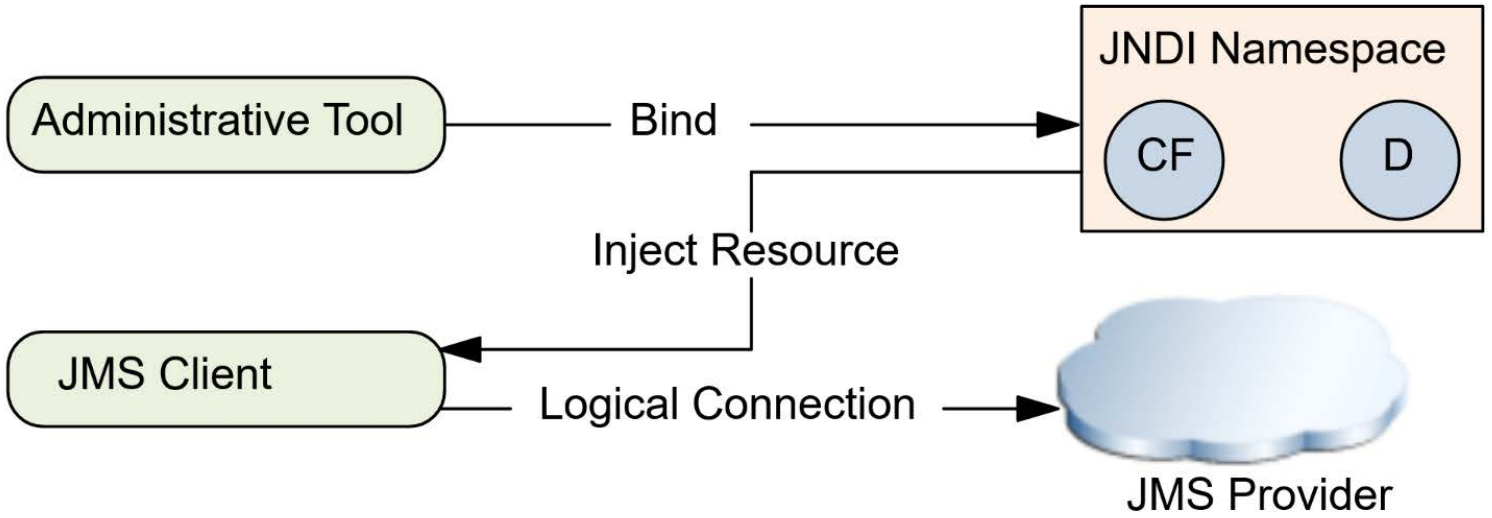


Fig.48-2

Figure 48-2 Jakarta Messaging Architecture

1. JMS Provider: Platform, Control and Administration
2. JMS Clients: Use platform to produce/consume messages
   alternative: native clients that adhere to the API rules
3. Messages using predefined `Java` types and formats
4. Administered Obj.: `destination`(D)/`connection` factories(CF)
   Interfaces for creating and managing connections

lookup vs. injection DSG-DSAM

IV-??

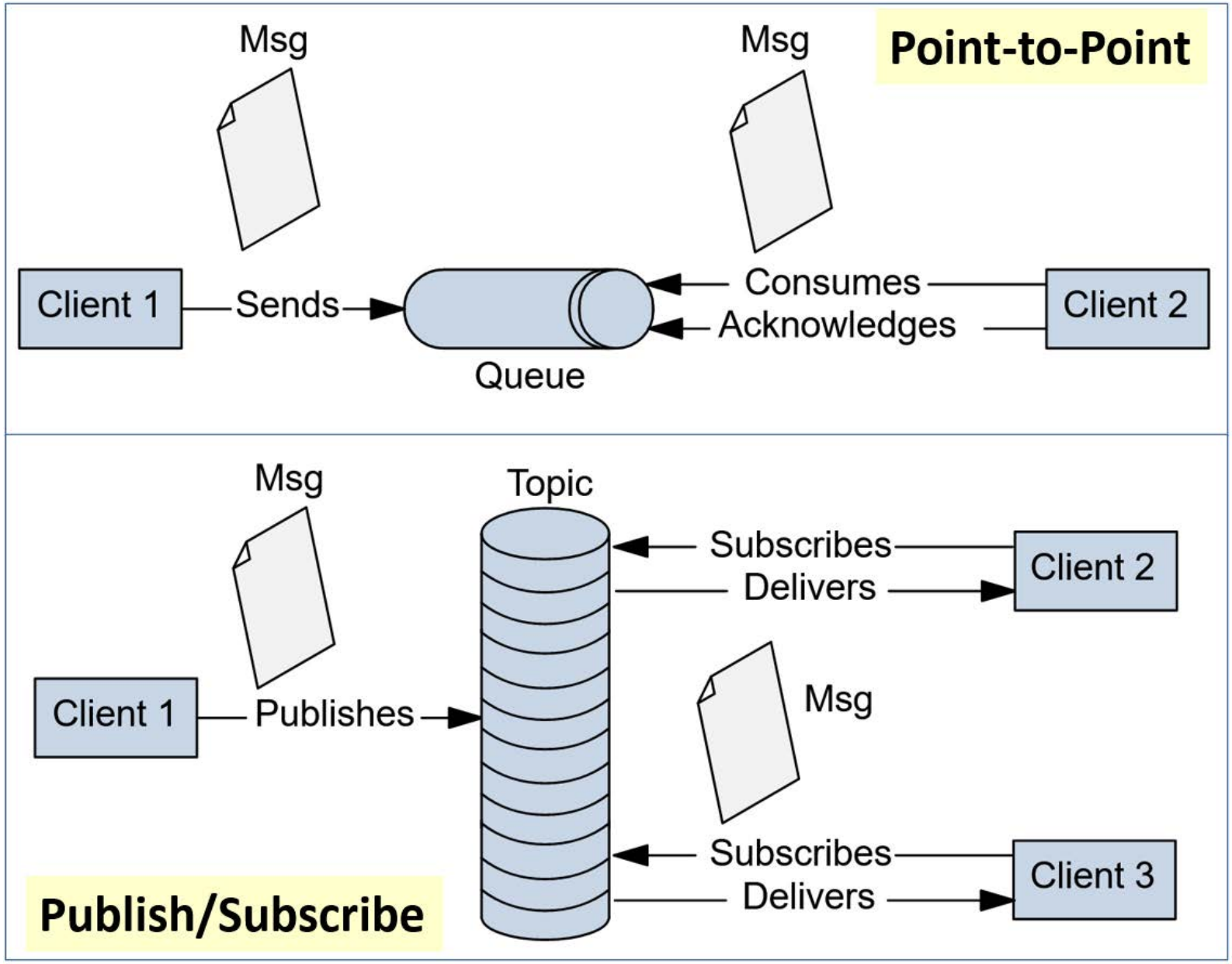
IV-??

Java Message Service API: Functionality

- **Styles/Domains:** specific paradigms of messaging usage IV-??
  - Point-to-Point interaction
  - Publish/Subscribe paradigm
    - `durable subscription` supports **de-coupling w.r.t. time** IV-??
    - `unshared/shared subscriptions`: single/multi consumers
- **Interaction: synchronous** via `receive` with/without timeouts
  asynchronous via message listener: `onMessage` method
- **Robustness:** series of messaging ops in transaction context
- **Messages:** header with meta information
  additional: properties used as criteria for queue selection
  $\Longrightarrow$ application-specific queue handling implementable
  $\Longrightarrow$ IDs from foreign MQ systems may be integrated
  Content: 6 different types of `Java` messages IV-??

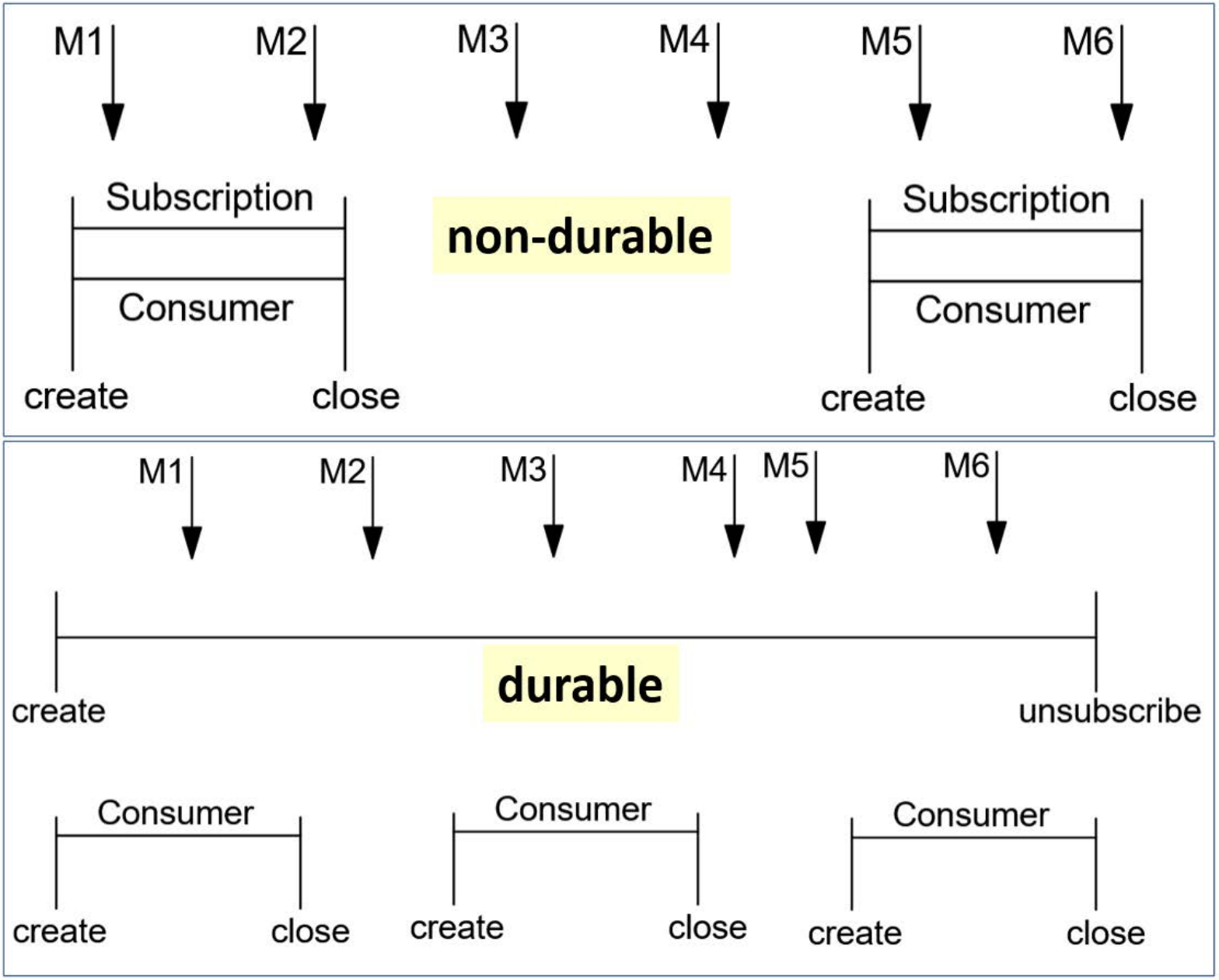
Java Message Service API: Messaging Styles

from: Jakarta EE Tut. Fig.48-3/4



Java Message Service API – 4: Subscription Types

Jakarta EE Tutorial, chap. 48 Fig. 48-6 and 48-7



Java Message Service API: Message Types

Jakarta EE Tutorial, chap. 48 Table 48-2

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an Extensible Markup Language file).
`MapMessage`	A set of name/value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

`struct`

JMS API: Software Architecture

1. **Configuration:** administered objects via `asadmin` (pre-runtime)
   - Create and parameterize factories using `JNDI Namespace`
   - Specify msg source and destination $\approx$ Destination

   resource annotations

2. **Usage:** Create and assemble messages

   (a) Creating a `JMSContext` object provides
   - a connection to a JMS provider and
   - a session as a single-threaded context

   (b) Create `JMSContext` communication objects
   - message producer: used to `send` messages
   - message consumer: synchronous via `receive(timeout)`
     asynchronous via `MessageListener`

   Additional support for message selectors to filter messages and `QueueBrowser` objects to inspect queues.

Remark: If interested in more details, please refer to Jakarta EE tutorial.

JMS API: Software Architecture

Jakarta EE Tutorial

ad-minist ered objects

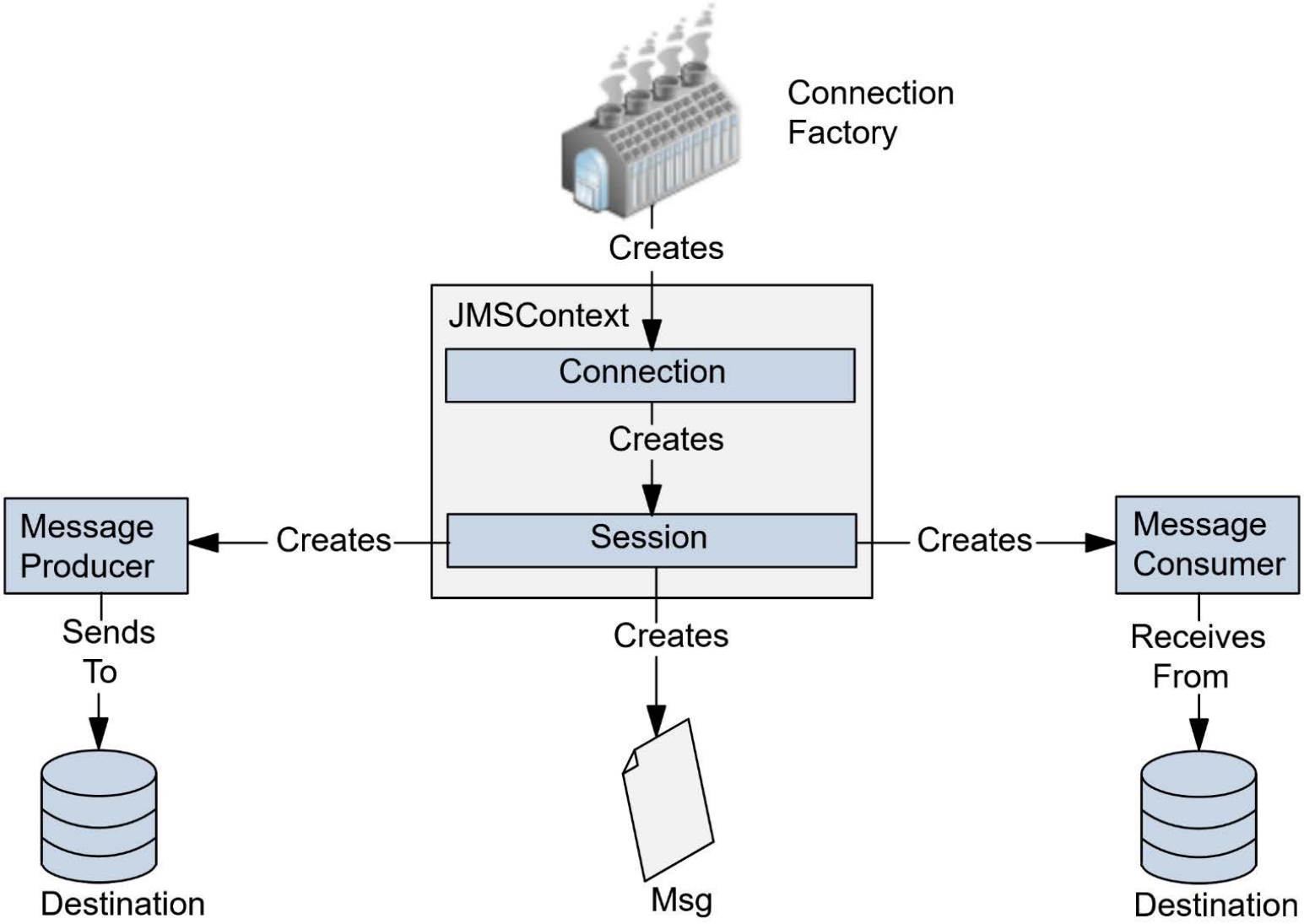


Figure 48-5 Jakarta Messaging Programming Model

ad-minist ered objects

IV.4 Advanced Message Queuing Protocol

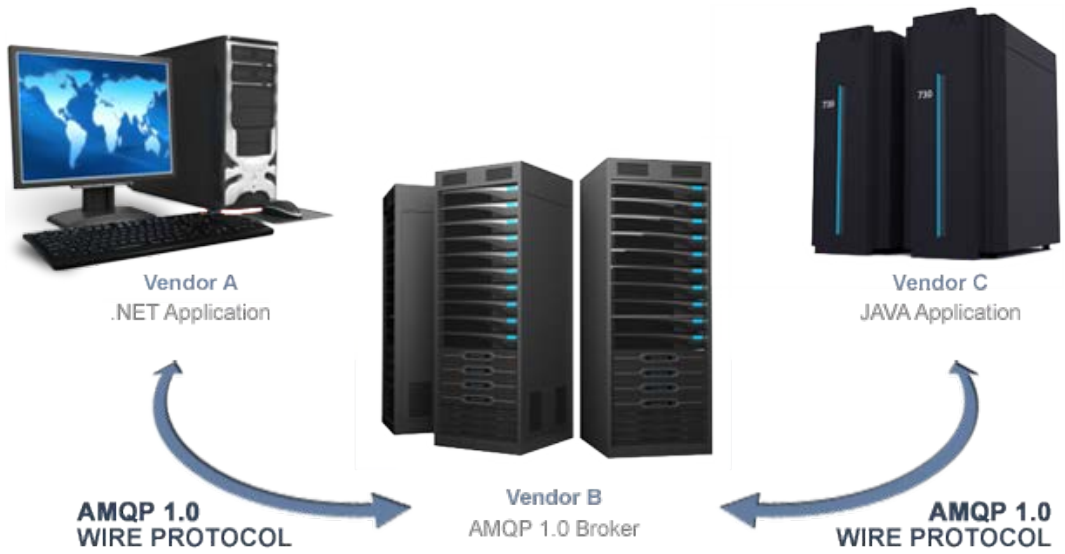


Fig.: www.amqp.org/product/overview

Advanced Message Queuing Protocol (AMQP)

OASIS

- ISO/IEC19464 standard since 2014
- general alternative protocol to http
- based on TCP; SSL/TLS and SASL usable
- used for almost all distributed interaction scenarios from cloud infrastructures to mobile clients
- support from (almost) all important vendors over the last years

AMQP

Basic Model

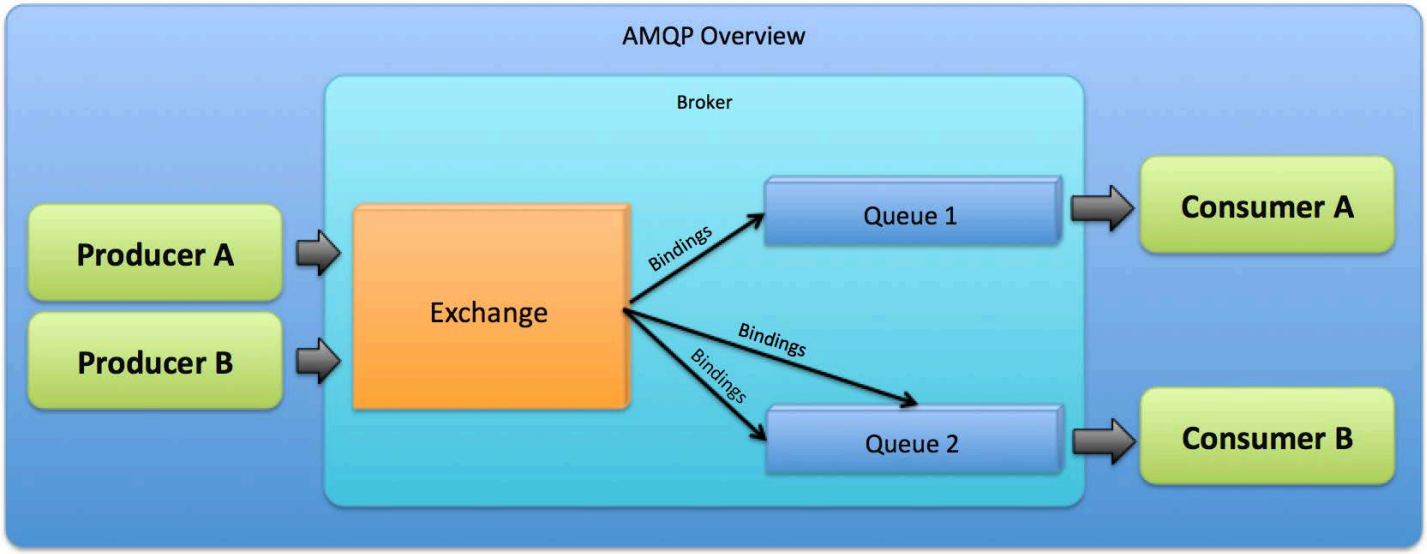


Fig.: alex volov .com/ 2016/06/ amqp/

- `Message Broker` as the central instance to implement Queues
- Producer (P) and Consumer (C) act as 'Clients' of the Broker
- `Exchange`: connection between Producer and Queues
  - `Name`: used for discovering and to setup connections
  - `Type`: interaction 'style': `direct`, `fanout`, `header`, `topic`
  - `Bindings`: Keys as basis für routing messages 'to' queues
- `Queues` store the messages
  - have to be attached to an `Exchange` after creation
  - can be used as a pull or a push medium

c.f. pg. IV-21

AMQP Interaction Styles and Implementations

Realizing different interaction styles:

- ▷ organized by `Exchange` component in Broker
- ▷ based on Strings as `routing keys` in messages
- ▷ based on Strings as `binding keys` when attaching queues

Four different methods to route messages:

- `direct`: matches msg routing keys with queue binding keys
- `fanout`: msg goes to all attached queues ignoring routing keys
- `topic`: implements publish/subscribe pattern
- `header`: uses msg attributes (data types) instead of routing keys

- ▶ Many Implementations on offer today:
  - ∗ `Apache Qpid/ActiveMQ/Artemis`
  - ∗ `MS Azure Service Bus`, `SwiftMQ`, ..., `RabbitMQ`
- ▶ Additionally: Bridges to other protocols, e.g. `MQTT`

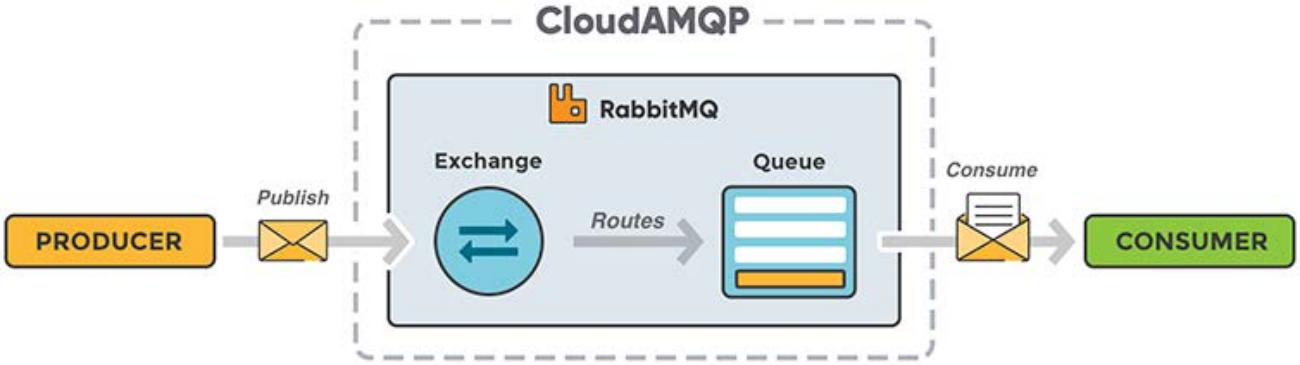
IV.5 AMQP Implementation: RabbitMQ

- Open Source Message Broker Software (based on `Erlang`)
- Pivotal Software (VMWare, SpringSource, General Electric)

- Multiple **Standards** supported:
  - AMQP implementation (widely used)
  - MQTT implementation (IoT)
  - STOMP integration
  - JMS client plugins etc.
  - HTTP, WebSockets
- **Multi-platform/language support:**
  - `Spring`, `.NET`, `JVMs`, `...`
  - `Java`, `Java Script/Node Ruby`, `Python`, `PHP ... Haskell`

- **Light-weigthed** and not too hard to host
- `CloudAMQP`: Cloud hosting at `AWS Marketplace` and `heroku`

RabbitMQ - a short Overview



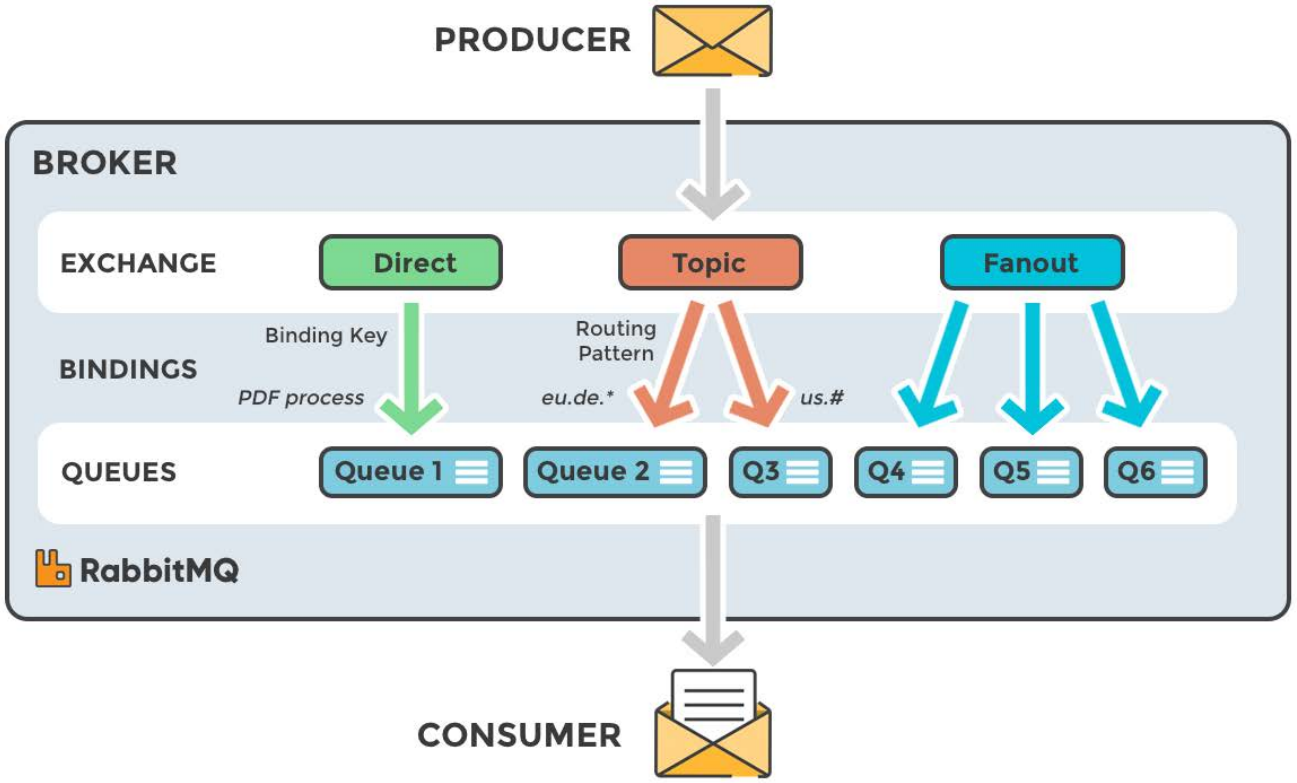
Remark: Tutorial material online: `www.cloudamqp.com/docs/index.html`

Interface to Programming: P and C connect to the Broker

simplified

- Client Producer:
  1. setting up a `connection` which provides a `channel` that allows to
  2. `declare` (an `exchange` that binds to) a `queue`.
  3. publishing uses an `exchange`, a `routing key` and the payload
  4. close the connection at the end
- Client Consumer:
  1. setting up a `connection` which provides a `channel` that allows to
  2. `declare` (an `exchange` that binds to) a `queue` (idempotent)
  3. receiving requires defining a `callback function` to handle the message
  4. `consume` declares a `queue name` and the `on_message_callback`
  5. consuming is done in an 'endless' waiting loop `start_consuming`

RabbitMQ - Different Interaction Styles



Supported Interaction Paradigms:

* direct 1-1 or 1-n with named queue and anonymous exchange
* Competing Consumer Pattern
* publish/subscribe with single or multiple topics
* routing via Strings as routing/binding keys
* simulating remote procedure calls using Request/Reply queues

Summary: Importance of Message Queuing

- **Integration** of already existing, heterogenous applications, esp.
  - Widespread solution for **inter middleware integration**
  - Integration of **Server-side components**, e.g., EJBs
  - Combination of **Online and Batch** processing
  - Implementation of reliability and load-balancing for msg passing

  c.f. DSG-DSAM-M

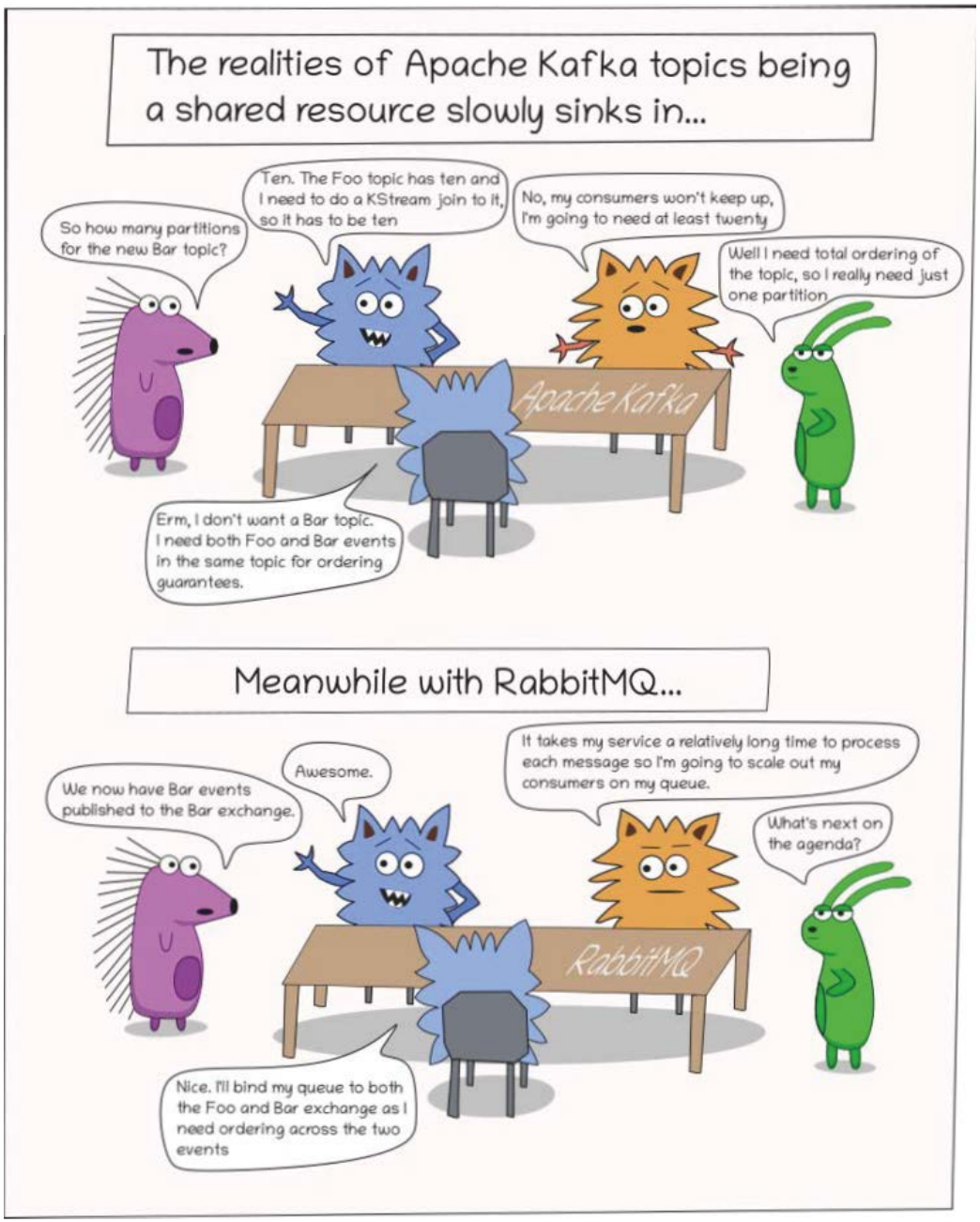
- Suitable for all kinds of **loosely-coupled systems**
  - (highly) asynchronous
  - often not at the same time active

  Applications in big corporations or between different enterprises

- Basic paradigm that is (almost) universally applicable, **but:**
  - low-level message view not always adequate
  - configuration/administration of messaging systems rather tricky

  $\Longrightarrow$ **too cumbersome for tightly-coupled systems**

from: jack-van lightly.com/sketches



End of chapter IV