MOM

Message Oriented Middleware

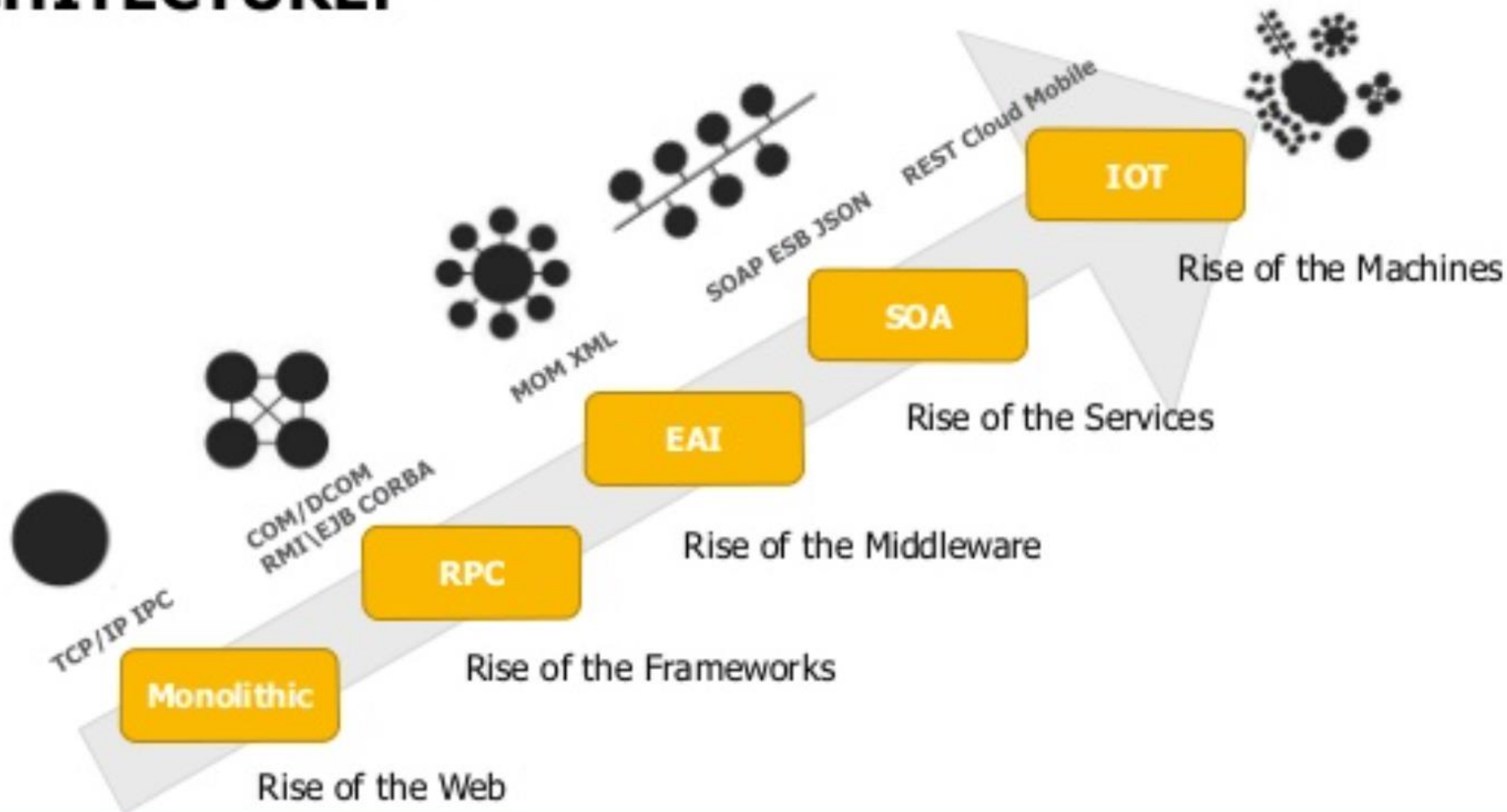
Agenda

1. TIPOS DE MIDDLEWARE
2. RPC - MOM
3. Características MOM
4. Mensajes
5. Brokers
6. Modelos de Mensajeria
7. JMS
8. Patrones de Diseño de Mensajería

TIPOS DE MIDDLEWARE

- **DATABASE MIDDLEWARE**
  - Permite comunicaciones entre uno o más base de datos locales o remotas
- **APPLICATION SERVER MIDDLEWARE**
  - Usado para sistemas entre browsers y legacy systems
- **MESSAGING MIDDLEWARE**
  - Transactional X open XA
  - Preocedural XML RPC
  - Object Oriented CORBA, RMI
- **MESSAGE ORIENTED MIDDLEARE: MOM**
  - Message queues-Brokers-BUS ESB
- **TRANSACTION PROCESSING MIDDLEWARE**

ARCHITECTURE:

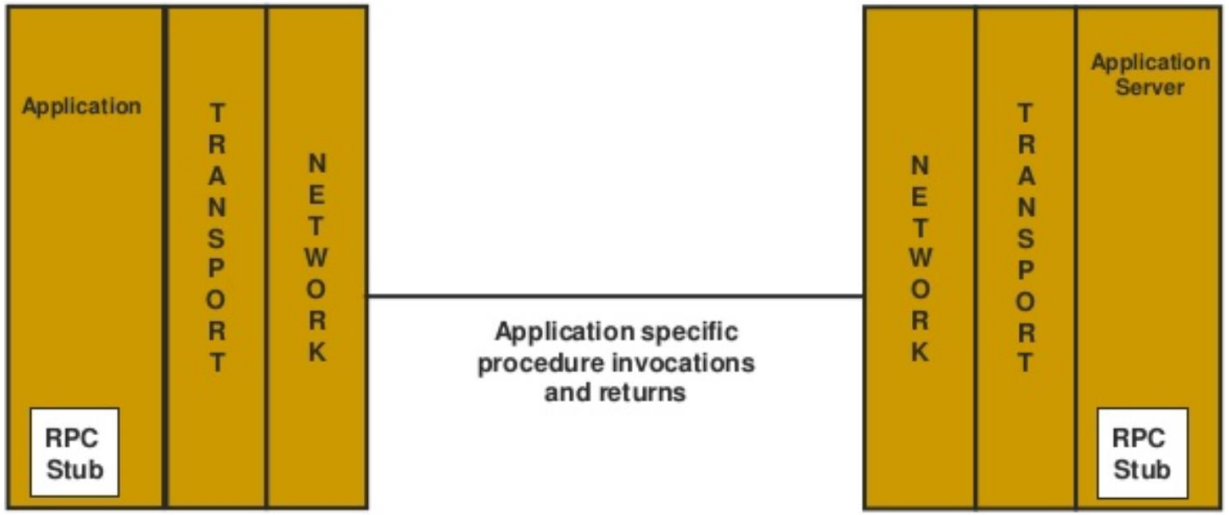


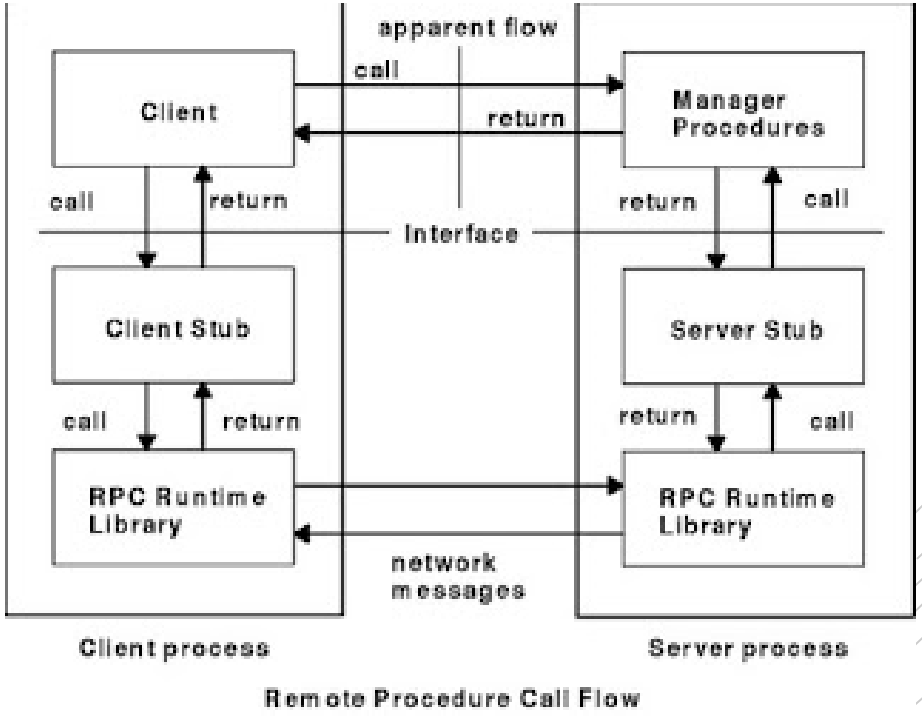
1.RPC

Remote Procedure Call

Bruce Jay Nelson 1981

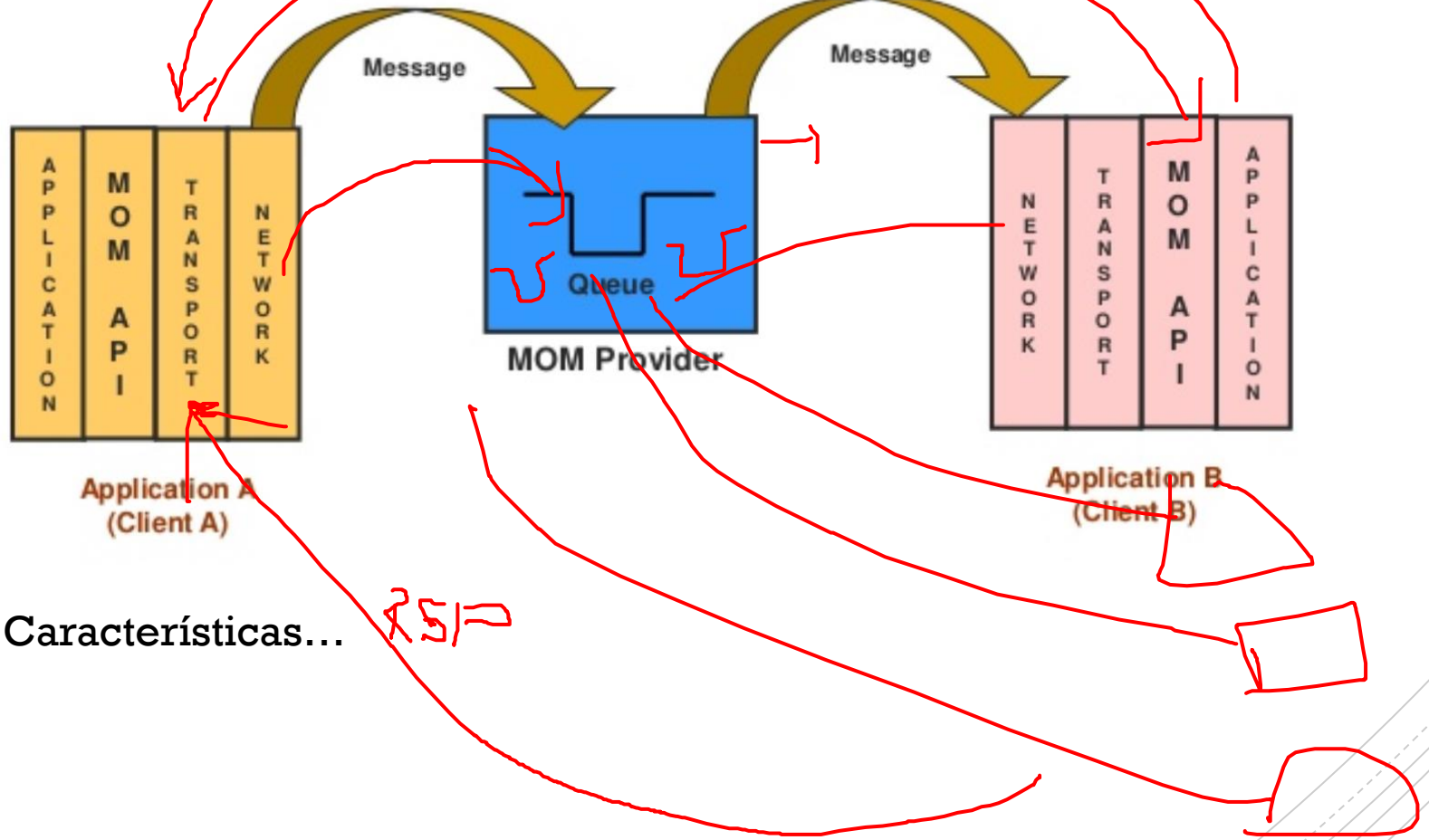
Síncrono, call method





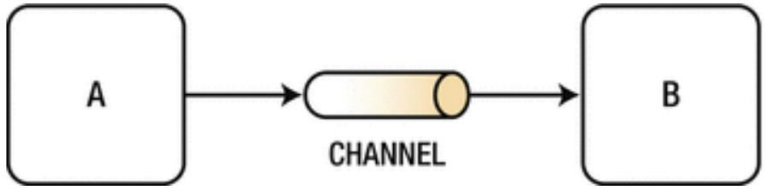
Remote Procedure Call Flow

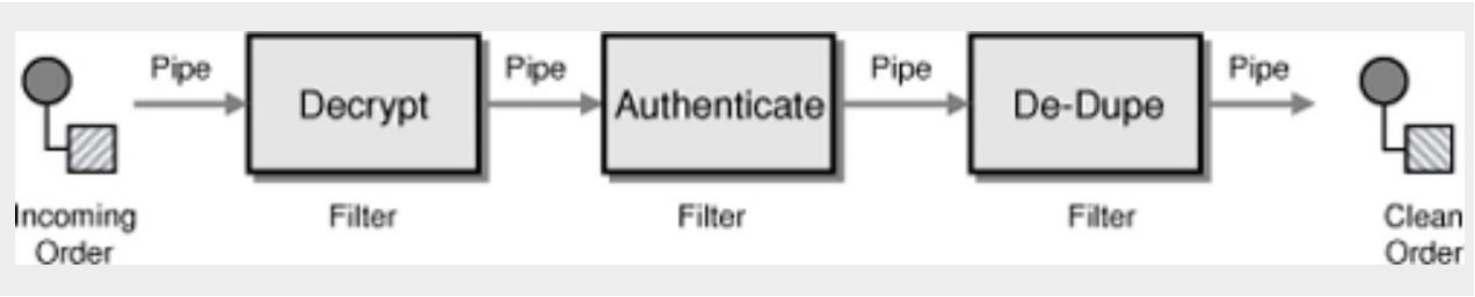
MOM



Características…

Elementos





- Canales (channel): tubería que conecta emisor A con receptor B
- Mensajes (messages): Paquete atómico de datos que se puede transmitir por el canal, paquete que tiene una estructura que contiene los datos.
- Tubos y Filtros: secuencia de pasos de procesamiento independientes más pequeños (filtros) que están conectados por canales (tubos).
- Enrutamiento: Pasa por varios canales para llegar a su destino final, la ruta que debe seguir un mensaje puede ser tan compleja que el remitente original no sabe qué canal llevará el mensaje al receptor final. En cambio, el remitente original envía el mensaje a un *Enrutador de mensajes que es en sí un filtro.*

T

- **Transformación:** varias aplicaciones pueden no estar de acuerdo con el formato de los mismos datos conceptuales; el remitente formatea el mensaje de una manera, pero el receptor espera que se formatee de otra manera. Para conciliar esto, el mensaje debe pasar por un filtro intermedio, un *traductor de mensajes*, que convierte el mensaje de un formato a otro.

2. MOM Message Oriented Middleware

- **Middleware Orientado a Mensajes** :Infraestructura de software y/o hardware que admite el envío y recepción de mensajes entre sistemas distribuidos .
- MOM permite que los módulos de aplicaciones se distribuyan en plataformas heterogéneas y reduce la complejidad del desarrollo de aplicaciones que abarcan múltiples sistemas operativos y protocolos de red .
- El middleware crea una capa de comunicación distribuida que aísla al desarrollador de la aplicación de los detalles de los diversos sistemas operativos e interfaces de red.
- Las API que se extienden a través de diversas plataformas y redes suelen ser proporcionadas por MOM.

Asincronía

Los mensajes puede ser procesados de forma asíncrona de forma que si un mensaje desencadena un proceso **largo en tiempo** el emisor del mensaje no tiene que esperar a que el proceso termine, el productor puede enviar el mensaje y liberarse.

Además, **ni el productor ni el receptor tienen que estar disponibles al mismo tiempo** para comunicarse. De hecho, el productor no tiene porqué saber nada del receptor, y viceversa. Ambos sólo deben saber el formato del mensaje y cual es el destino del mensaje.

3. Ventajas de MOM / Características

1. Asíncrono, bajo acoplamiento. Permite que los componentes publiquen mensajes para otros componentes
2. Interoperable
3. Persistencia, seguro
4. Asegura el envío del mensaje
5. Reduce la complejidad
6. Portable, Flexible
7. Soporta:
   - Tolerancia a fallos.
   - Escalabilidad.

- Envío de Correos
- Chats
- Alertas y Notificaciones
- Auditorias
- Ordenes de compra en web sites
- Apps altamente concurrentes
- Apps con alta carga de usuarios
- Pago de impuestos
- Votaciones
- IOT

Fault tolerance, High Availability

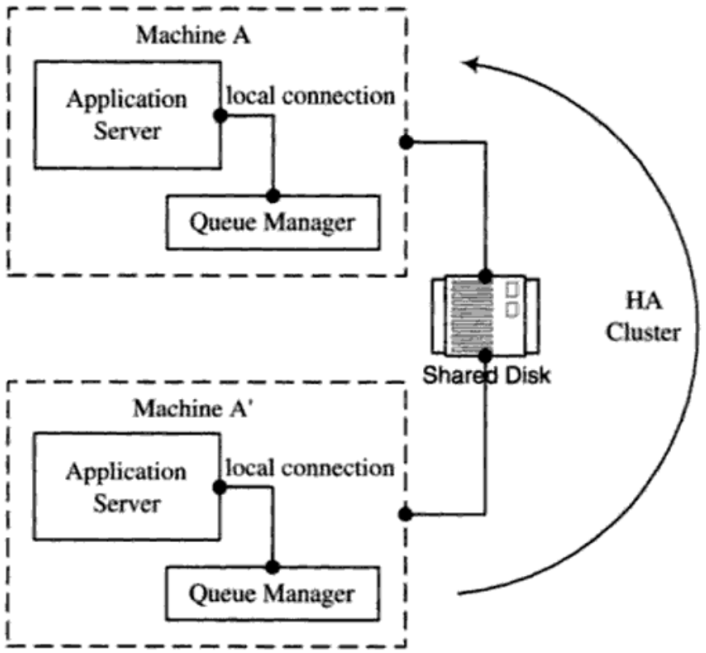


Figure 8-3 High-Availability Topology: Application Server and Local Queue Manager

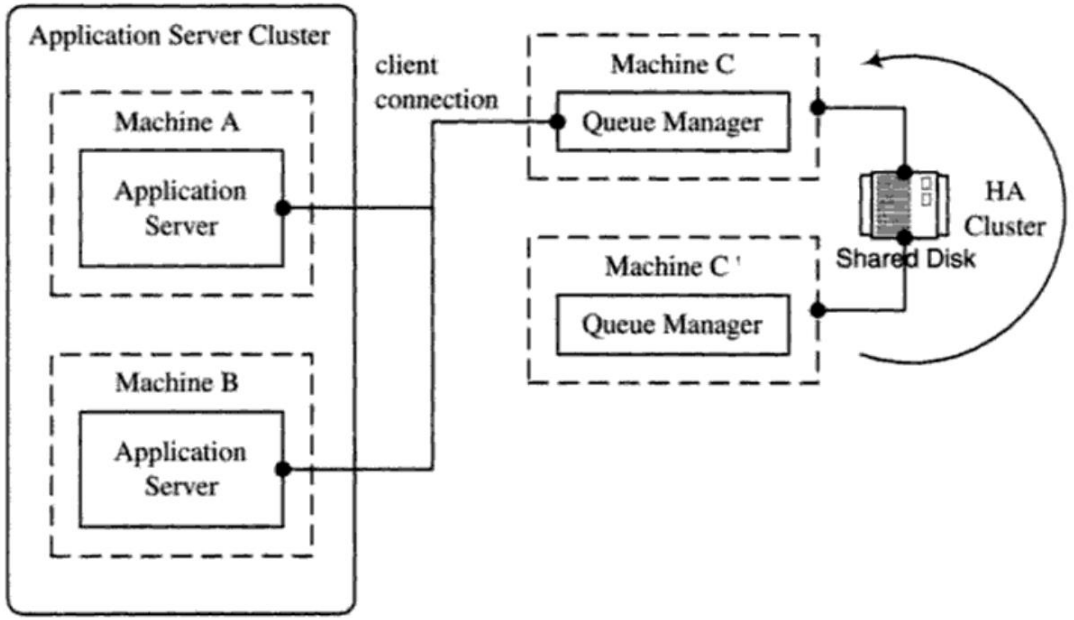
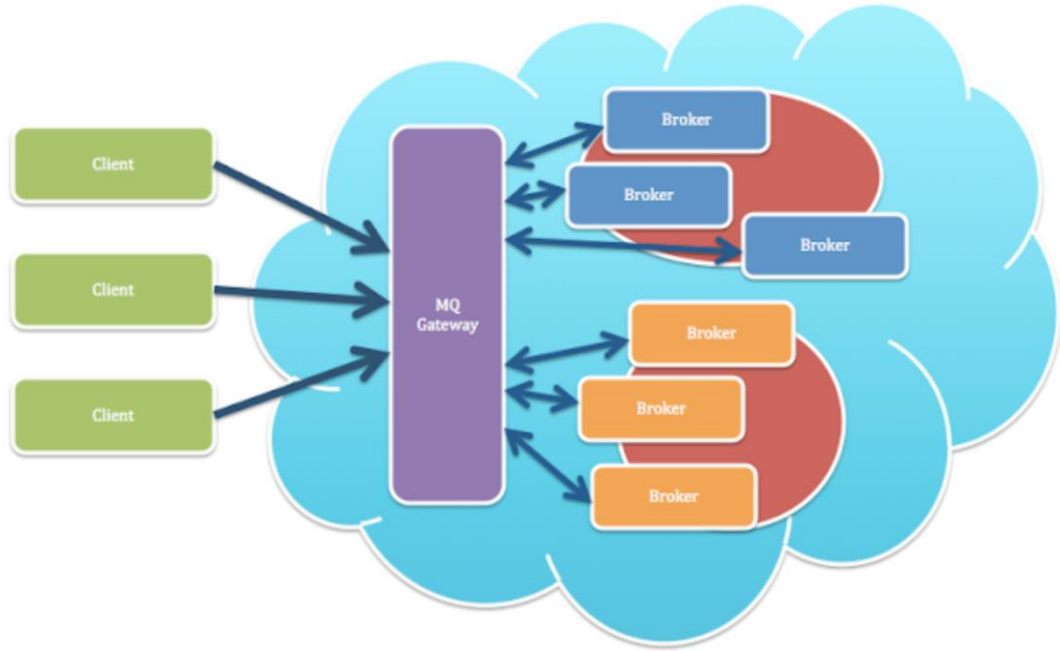
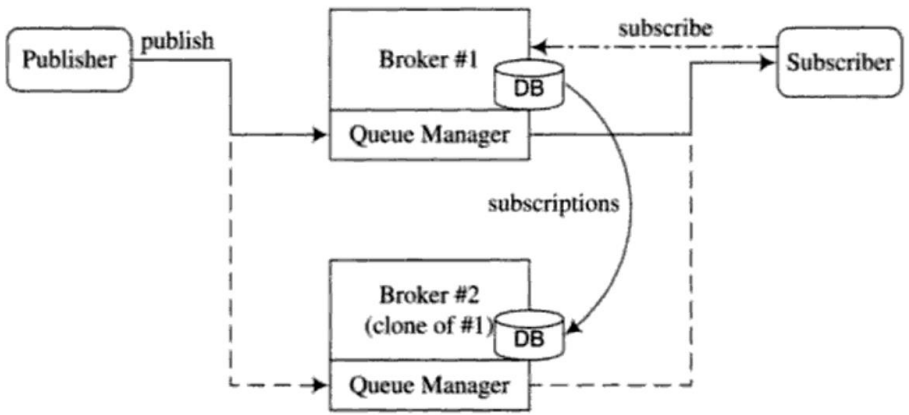


Figure 8-4 High-Availability Topology: Application Server and Remote Queue Manager





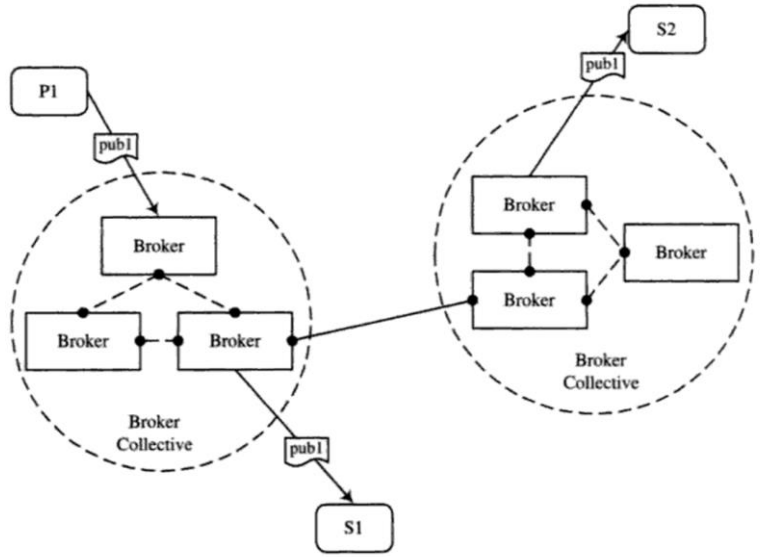


Figure 8-8 Broker Clone

5. Broker / Agente de Mensajería

Intermediario que traduce un mensaje del protocolo de mensajería formal del remitente al protocolo de mensajería formal del receptor. Los intermediarios de mensajes son elementos de las redes de telecomunicaciones o computadoras donde las aplicaciones de software se comunican intercambiando mensajes formalmente definidos.

El **Broker** puede considerarse como una instancia completa del proveedor de mensajes. Un broker contiene destinos físicos (colas y temas) y es responsable de encaminar los mensajes enviados de los productores al destino apropiado y entregar mensajes de los destinos a los consumidores registrados.

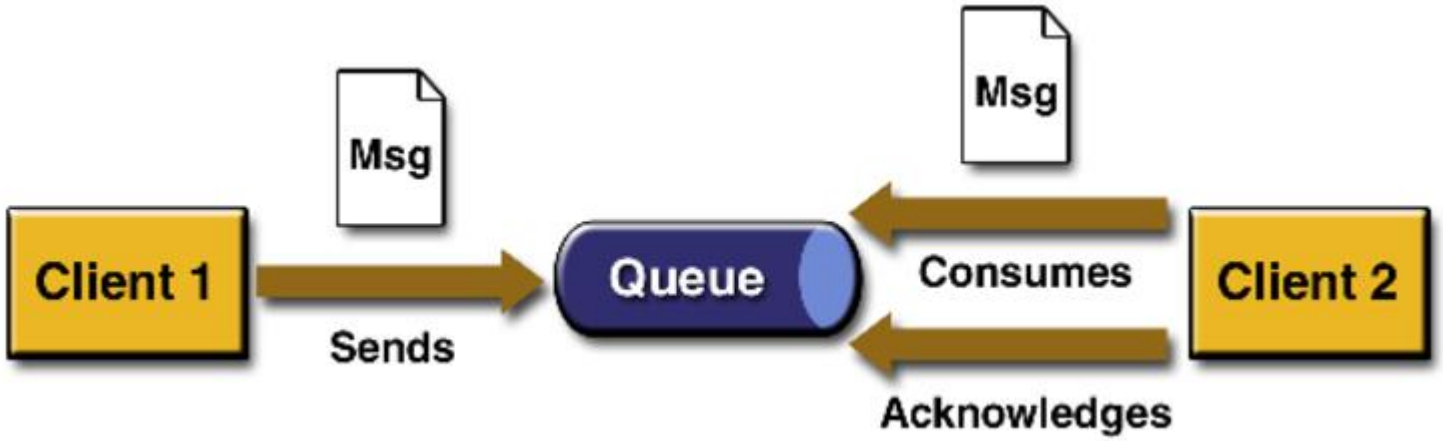
El broker soporta múltiples mecanismos de conexión entre los componentes de la aplicación y el broker, y también se puede configurar con diferentes características de rendimiento y seguridad.

5.Algunos Brokers de Mensajería

- Apache ActiveMQ
- Financial Fusion Message Broker (Sybase)
- Fuse Message Broker (empresa ActiveMQ)
- HornetQ (Red Hat)
- IBM Integration Bus
- IBM MQ Series
- JBoss Messaging (JBoss)
- Microsoft MMSQ
- NATS (Licencia de código abierto MIT , escrito en Ir)
- Oracle Message Broker (Oracle Corporation)
- RabbitMQ (Licencia pública de Mozilla , escrita en Erlang)
- WSO2 Message Broker

6.Modelos de Mensajería

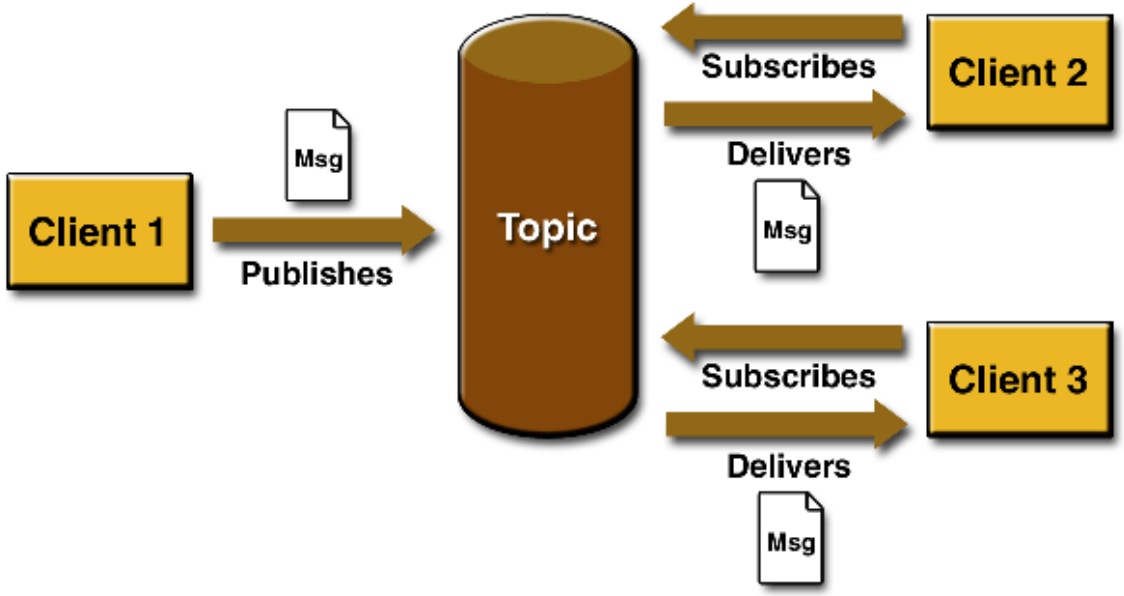
- El modelo point to point (P2P): En este modelo, el destino utilizado para retener mensajes se llama una cola (queue), un cliente coloca un mensaje en una cola y otro cliente recibe el mensaje. Una vez que se reconoce (knowledge) el mensaje, el proveedor de mensajes elimina el mensaje de la cola.



6. Modelos de Mensajería

El modelo publish-subscribe (pub-sub):

- El destino es denominado **Topic**.
- Cuando se usa publicar / subscribir mensajes, un cliente publica un mensaje a un **Topic** y todos los suscriptores a ese **Topic** reciben el mensaje.

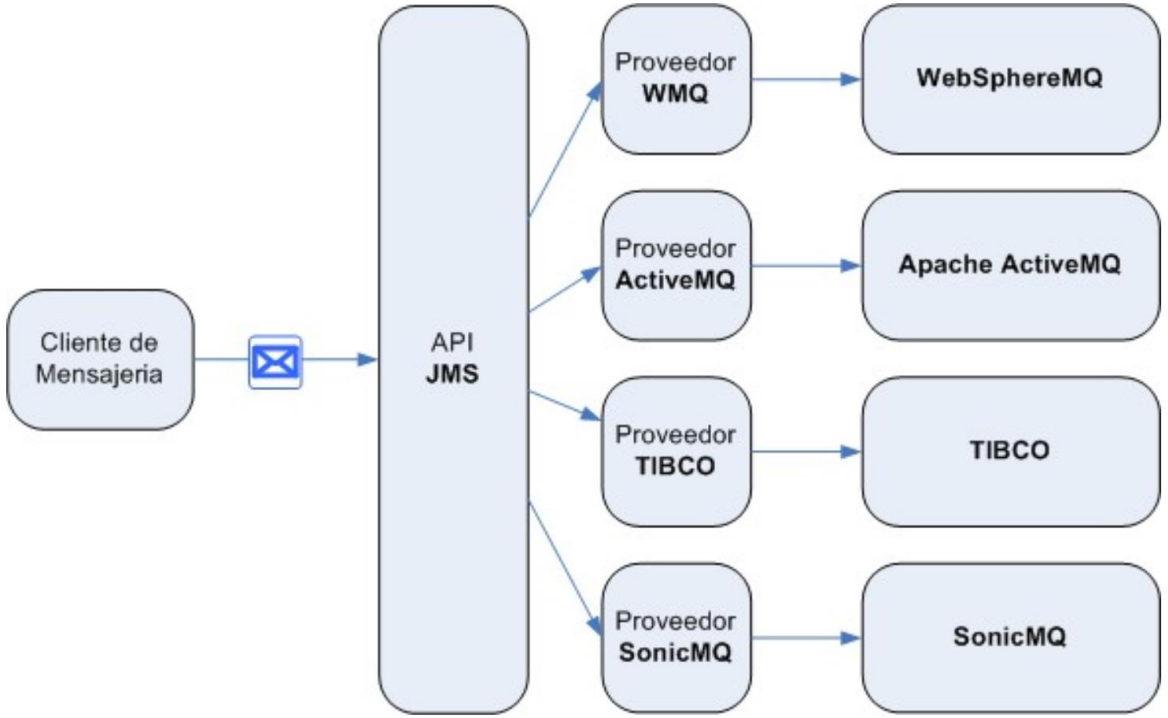


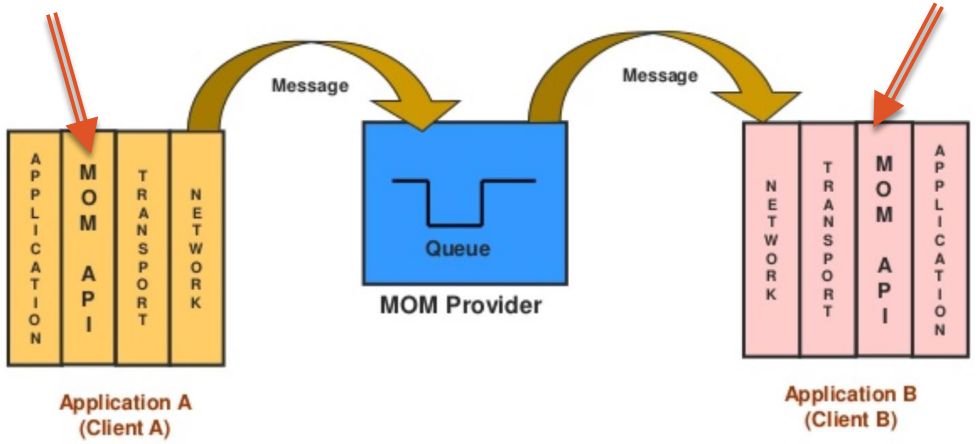
7.JMS
JAVA MESSAGE SERVICE

Que hace JMS

- API Java que permite:
  - Crear
  - Enviar
  - Recibir
  - Leer mensajes

- Parecido a un sistema de noticia ☺





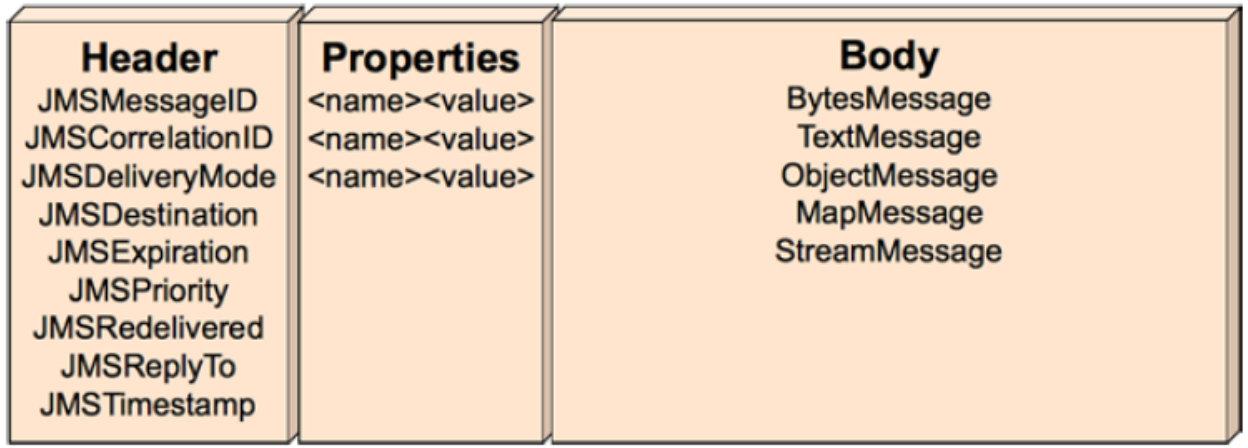
Mensajes

Los mensajes son objetos que encapsulan información y se dividen en tres partes:

Un encabezado: contiene información estándar para identificar y enrutar el mensaje.

Propiedades: son pares nombre-valor que la aplicación puede establecer o leer. Las propiedades también permiten destinos para filtrar mensajes basados en valores de propiedad.

Un cuerpo: contiene el mensaje real y puede tomar varios formatos (texto, bytes, objeto, etc.).



Tipos de mensajes

- **BytesMessage**
  Permite enviar una matriz de bytes como un mensaje. JMSProducer tiene un método send () que toma una matriz de bytes como uno de sus parámetros; Este método crea una instancia de javax. Jms.BytesMessage
- **ObjectMessage**

  Permite enviar cualquier Objeto Java serializable.
- **StreamMessage**

  Permite enviar un array de bytes como un mensaje. Difiere de BytesMessage en que se carga el tipo primitivo como un stream.
- **TextMessage** Permite enviar un java.lang.String como un mensaje el parámetro crea una instancia javax.jms.TextMessage en el camino al ser enviado.

Consumidores

Consumo de mensajes Sincronos

- Un suscriptor o receptor obtiene un mensaje explícitamente del destino usando el método "recibir" el método de "recepción" puede bloquearse hasta que llegue un mensaje o puede expirar el tiempo si un mensaje no llega dentro de un límite de tiempo especificado

Consumo de mensajes Asíncronos

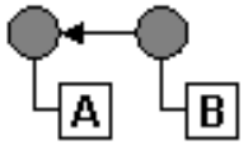
- Un cliente puede registrar un oyente de mensajes (como un oyente de eventos) con un consumidor cada vez que llega un mensaje al destino, el proveedor JMS entrega el mensaje llamando al método Listener "onMessage" que actúa sobre el contenido del mensaje.

8.Patrones de Mensajeria

Patrón: Request / Replier

Components of the Request/Reply Example

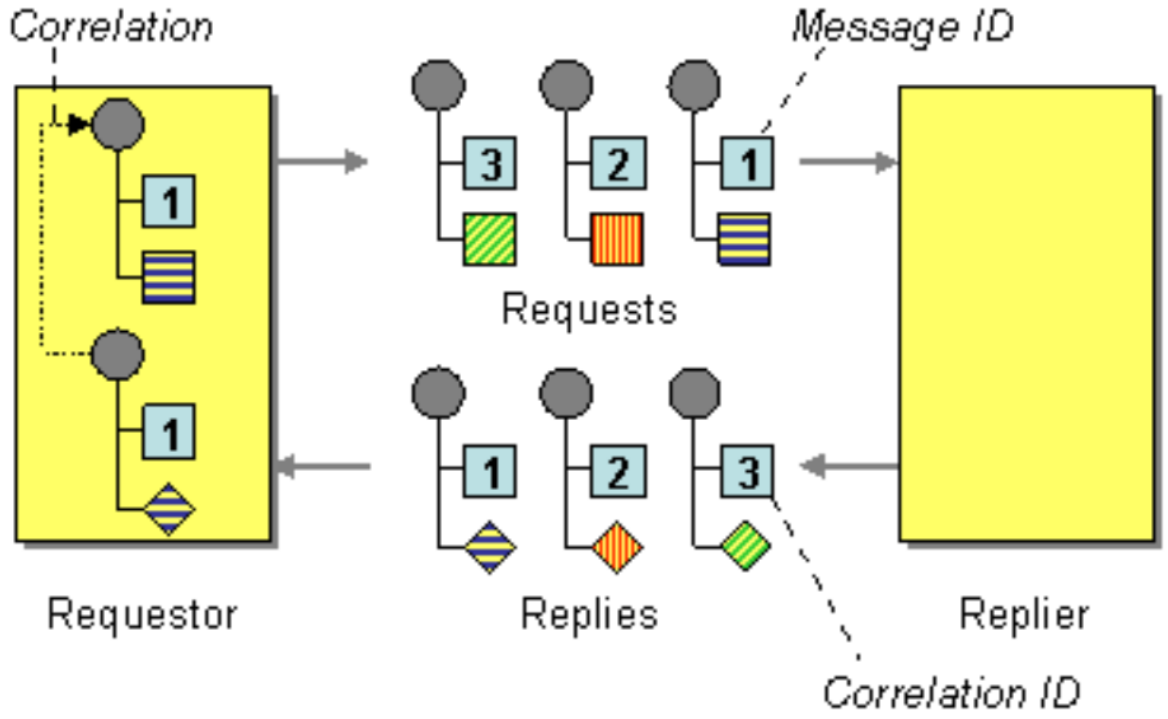
http://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReplyJmsExample.html



Correlation Identifier

My application is using *Messaging* to perform a *Request-Reply* and has received a reply message.

How does a requestor that has received a reply know which request this is the reply for?

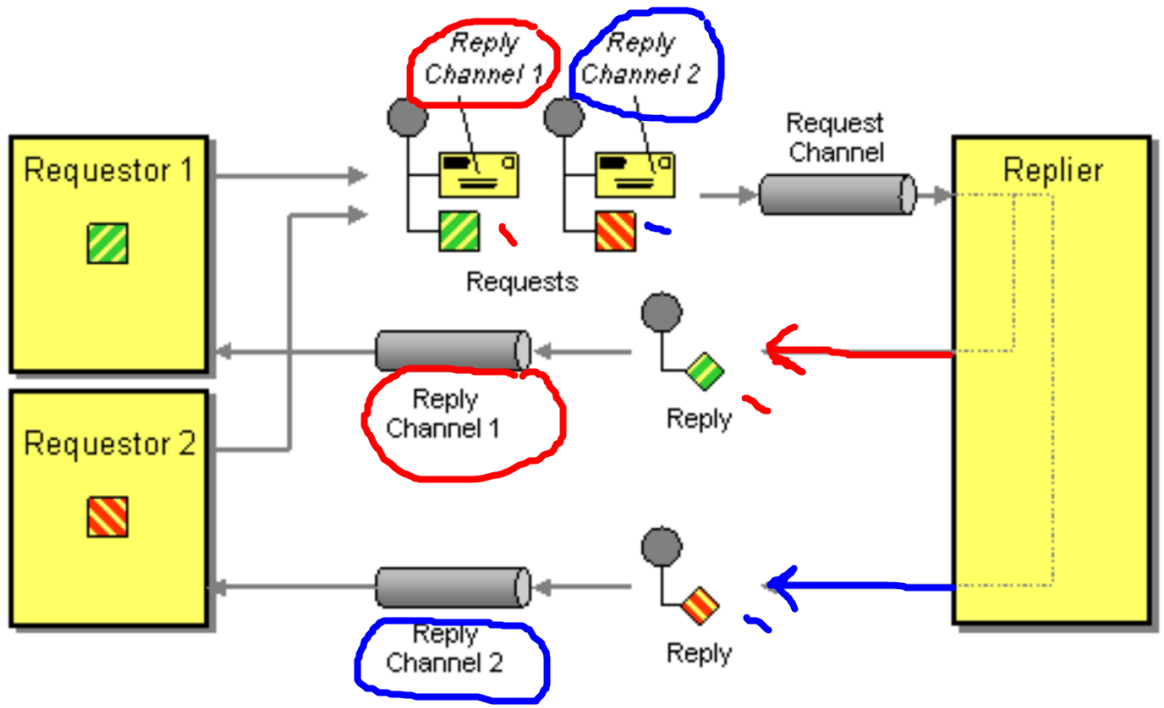


http://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.ht

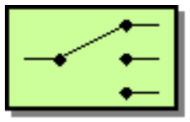
Return Address

My application is using *Messaging* to perform a *Request-Reply*.

How does a replier know where to send the reply?



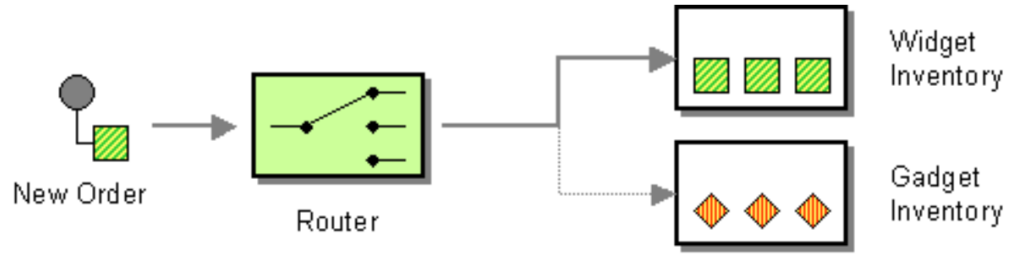
The request message should contain a *Return Address* that indicates where to send the reply message.



Content-Based Router

Assume that we are building an order processing system. When an incoming order is received, we first validate performed by the inventory system. This sequence of processing steps is a perfect candidate for the *Pipes and* and route the incoming messages through both filters. However, in many enterprise integration scenarios more

How do we handle a situation where the implementation of a single logical function (e.g., inventory checl



Use a *Content-Based Router* to route each message to the correct recipient based on message content.

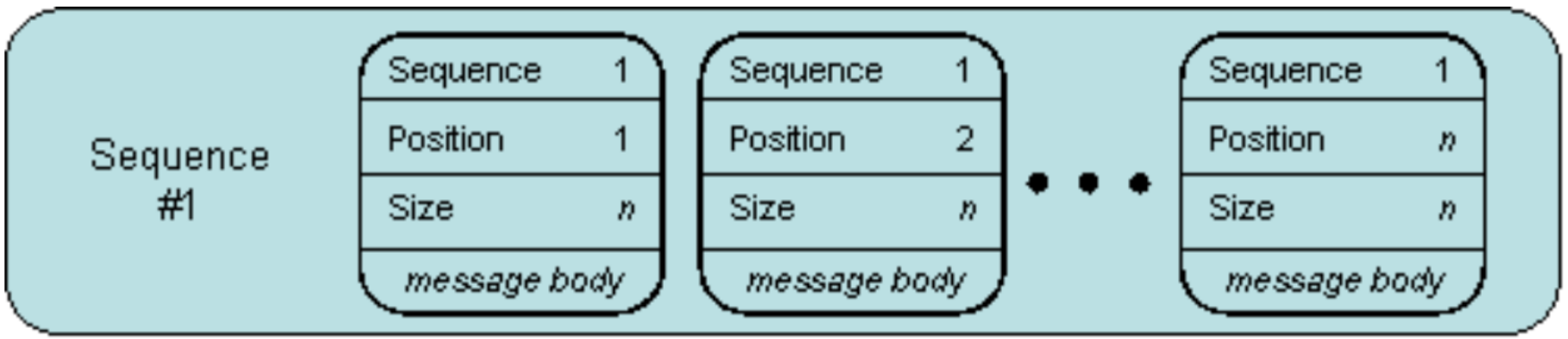
```
RouteBuilder Builder = nuevo RouteBuilder () {
  public void configure () {
    errorHandler (deadLetterChannel ("simulacro: error"));

    desde ("directo: en")
      .elección()
        .when (encabezado ("tipo"). isEqualTo ("widget"))
          .to ("directo: widget")
        .when (encabezado ("tipo"). isEqualTo ("gadget"))
          .to ("directo: gadget")
        .de otra manera()
          .to ("directo: otro");
  }
};
```

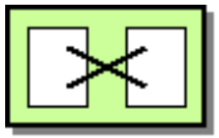
Message Sequence

My application needs to send a huge amount of data to another process, more than

How can messaging transmit an arbitrarily large amount of data?

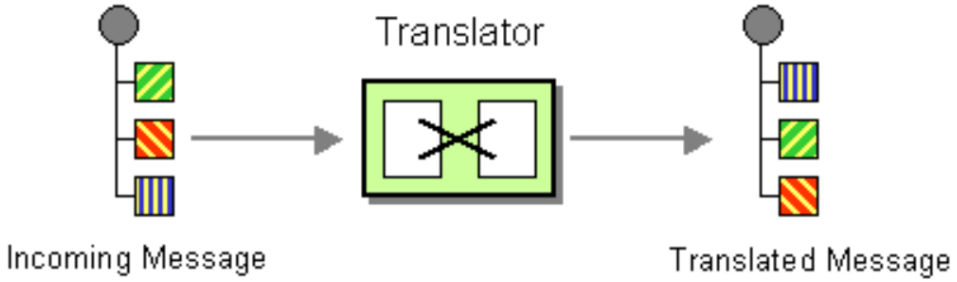


Whenever a large set of data may need to be broken into message-size chunks

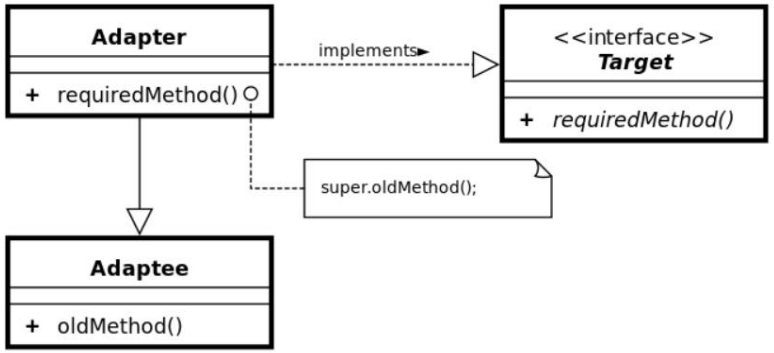


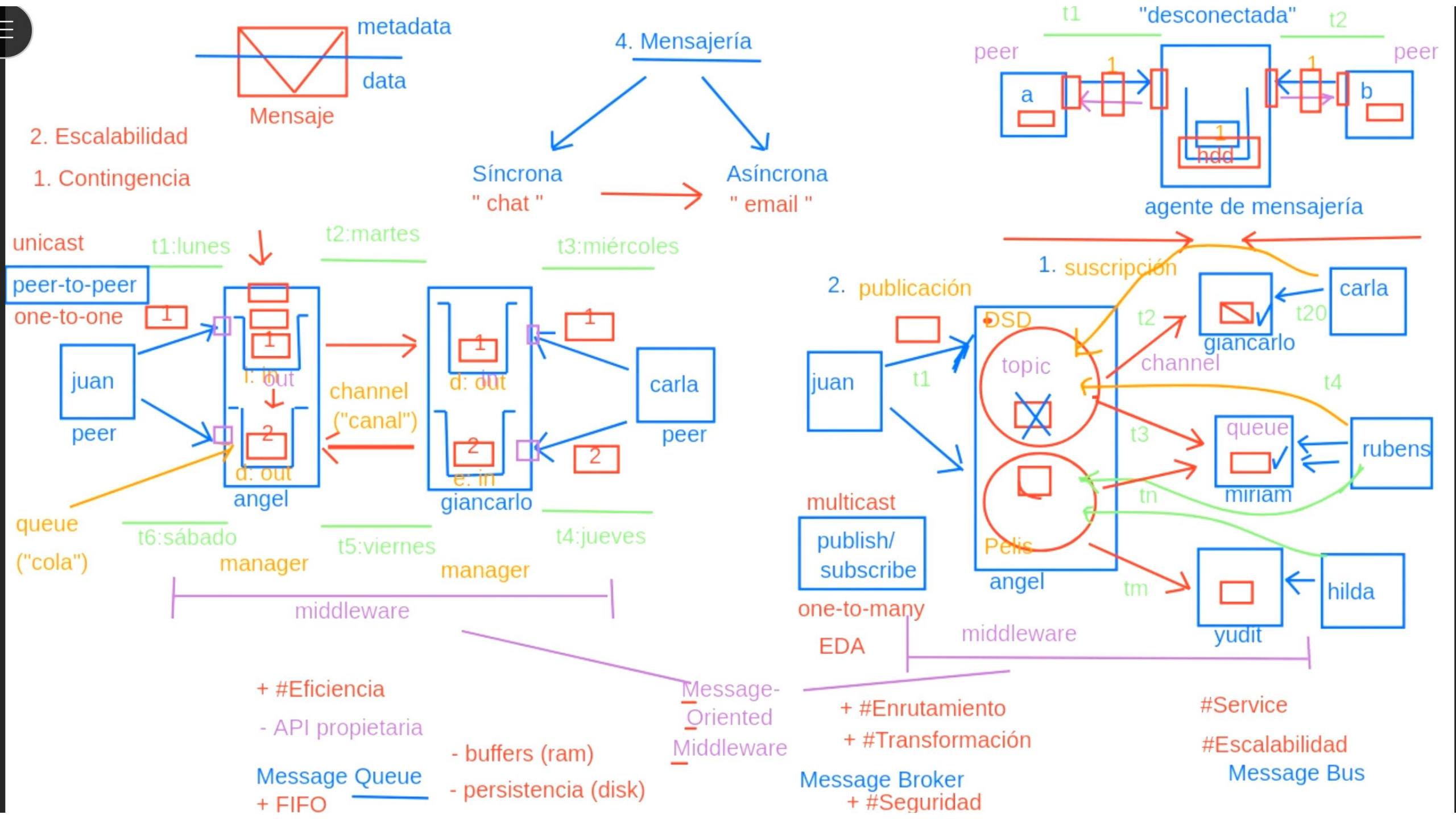
Message Translator

How can systems using different data formats communicate with each other using messaging?



Use a special filter, a *Message Translator*, between other filters or applications to translate one data format into another.




metadata
data
Mensaje
4. Mensajería
Síncrona
" chat "
Asíncrona
" email "
2. Escalabilidad
1. Contingencia
t1
"desconectada"
t2
peer
peer
a
b
1
hdd
agente de mensajería
unicast
t1:lunes
t2:martes
t3:miércoles
peer-to-peer
one-to-one
juan
peer
f: out
d: out
angel
channel
("canal")
d: out
e: in
giancarlo
carla
peer
queue
("cola")
t6:sábado
manager
t5:viernes
manager
t4:jueves
middleware
1. suscripción
2. publicación
juan
t1
DSD
topic
Pelis
angel
t2
t20
giancarlo
carla
channel
t4
t3
queue
miriam
rubens
tn
tm
yudit
hilda
multicast
publish/
subscribe
one-to-many
EDA
middleware
+ #Eficiencia
- API propietaria
Message Queue
+ FIFO
- buffers (ram)
- persistencia (disk)
Message-
Oriented
Middleware
+ #Enrutamiento
+ #Transformación
Message Broker
+ #Seguridad
#Service
#Escalabilidad
Message Bus

Connection Pooling JMS

- Sin el connection pooling, JMSTemplate, de forma predeterminada, crea una nueva conexión, sesión, producido para cada mensaje enviado y luego los cierra nuevamente. Esto da como resultado un largo tiempo de solución y se vuelve a conectar con cada mensaje JMS enviado.
- Connection pooling agrupa los recursos JMS para que funcionen de manera eficiente con JMSTemplate de Spring, de modo que reutilice las conexiones. Hay varias agrupaciones de conexiones disponibles para su uso en Spring Framework, incluidas SingleConnectionFactory , CachingConnectionPooling y la conexión específica del proveedor Pooling ActiveMQ's PooledConnectionFactory.

Properties de Active MQ - JMS

```
# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url= # URL of the ActiveMQ broker. Auto-generated by default. For instance `tcp://localhost:61616`
spring.activemq.in-memory=true # Specify if the default broker URL should be in memory. Ignored if an explicit broker has been specified.
spring.activemq.password= # Login password of the broker.
spring.activemq.user= # Login user of the broker.
spring.activemq.packages.trust-all=false # Trust all packages.
spring.activemq.packages.trusted= # Comma-separated list of specific packages to trust (when not trusting all packages).
spring.activemq.pool.configuration.*= # See PooledConnectionFactory.
spring.activemq.pool.enabled=false # Whether a PooledConnectionFactory should be created instead of a regular ConnectionFactory.
spring.activemq.pool.expiry-timeout=0 # Connection expiration timeout in milliseconds.
spring.activemq.pool.idle-timeout=30000 # Connection idle timeout in milliseconds.
spring.activemq.pool.max-connections=1 # Maximum number of pooled connections.
```