Sistemas distribuidos

Grado en Ingeniería Informática

Tema 2.3: Comunicación indirecta

Departamento de Ingeniería Informática
Universidad de Cádiz

Escuela Superior de Ingeniería
Dpto. de Ingeniería Informática

Versión 1.1

Indice

1. Introducción
2. Comunicación en grupo
3. Sistemas publicador-suscriptor
4. Colas de mensaje

Sección 1 | Introducción

Introducción

- La comunicación indirecta se lleva a cabo a través de un intermediario.
- No existe acoplamiento directo entre el emisor y uno o más receptores.
- Algunas técnicas de comunicación indirecta:

  Comunicación en grupo Comunicación mediante una abstracción de grupo en la que el emisor no es consciente de la identidad de los receptores.

  Sistemas publicador-suscriptor Una familia de enfoques que se caracterizan por enviar eventos a múltiples receptores a través de un intermediario.

  Sistemas de cola de mensajes Los mensajes se envían a una cola, extraídos más tarde por los receptores.

Comunicación directa

Recordatorio

- Acoplamiento directo entre un emisor y un receptor.
- Rigidez ante posibles cambios.
- Ejemplo de una interacción cliente-servidor:
  - Gran dificultad para sustituir un servidor por otro con funcionalidades similares.
  - Si el servidor falla, afectará al cliente (que tratará el fallo).

Comunicación indirecta (I)

Definición

Comunicación entre entidades en un sistema distribuido a través de un intermediario sin acoplamiento directo entre emisor y receptores.

R. Needham, M. Wilkes y D. Wheeler

Todos los problemas en informática se pueden resolver mediante otro nivel de indirección.

Desacoplamiento de espacio El emisor no sabe (ni necesita saber) la identidad de los receptores, y viceversa. Por tanto, los participantes (emisores y receptores) pueden ser reemplazados, actualizados, replicados o migrados.

Desacoplamiento de tiempo El emisor y los receptores pueden tener tiempos de vida independientes. No es necesario que emisor y receptores coexistan al mismo tiempo.

Comunicación indirecta (II)

Ventajas

- Muy utilizada en sistemas distribuidos donde el cambio es anticipado.
- Ejemplo: entornos móviles donde los usuarios deben conectarse y desconectarse rápidamente de la red global.
- Estos sistemas deben ser gestionados para ofrecer servicios más fiables.
- Este tipo de comunicación también se utiliza para el envío de eventos en sistemas distribuidos, donde los receptores son desconocidos.
- La infraestructura de Google utiliza la comunicación indirecta.

Comunicación indirecta (III)

Inconvenientes

- Sobrecarga de rendimiento introducida al añadir el nivel de indirección.
- *No existe ningún problema de rendimiento que no pueda resolverse eliminando el nivel de indirección* (J. Gray).
- Estos sistemas son más difíciles de gestionar con precisión.

Comunicación indirecta (IV)

Acoplamiento de espacio y tiempo en sistemas distribuidos

	Tiempo acoplado	Tiempo desacoplado
Espacio acoplado	Comunicación dirigida hacia 1 o varios receptores. Receptores deben existir en ese momento. Ej.: paso de mensajes.	Comunicación dirigida hacia 1 o varios receptores. Emisores y receptores pueden tener tiempos de vida diferentes.
Espacio desacoplado	Emisor no necesita saber la identidad de receptores. Receptores deben existir en ese momento. Ej.: IP *multicast*.	Emisor no necesita saber la identidad de receptores. Emisores y receptores pueden tener tiempos de vida diferentes. Ej.: **comunicación indirecta**.

Comunicación indirecta (V)

Comunicación asíncrona vs desacoplamiento de tiempo

Comunicación asíncrona

- Un emisor envía un mensaje y entonces continúa (sin bloquear).
- No es necesario que el emisor se encuentre con el receptor al mismo tiempo.

Desacoplamiento de tiempo

- Añade una dimensión extra: el emisor y los receptores pueden existir en momentos diferentes.
- Ejemplo: el receptor podría no existir en el instante en que se inicia la comunicación.

Sección 2 | Comunicación en grupo

Introducción

- Ejemplo de paradigma de comunicación indirecta; fundamental en los sistemas distribuidos.
- Proporciona un servicio en el que un mensaje se envía a un grupo y entonces se entrega a todos los miembros de ese grupo.
- El emisor no es consciente de las identidades de los receptores.
- Representa una abstracción de la comunicación *multicast*.
- Áreas de aplicación:
  - Difusión fiable de información a un gran número de clientes.
  - Apoyo para aplicaciones colaborativas, en las que deben enviarse eventos a múltiples usuarios, preservando una vista común de usuario (ej.: juegos multiusuario).
  - Apoyo a un rango de estrategias tolerantes a fallos (actualización consistente de datos replicados o implementación de servidores replicados).
  - Apoyo para la monitorización y gestión de sistemas (ej.: estrategias de balanceo de carga).

El modelo de programación (I)

- Los procesos deben unirse o dejar un grupo.
- Los procesos pueden enviar un mensaje a este grupo, que será propagado a todos los miembros con garantías de fiabilidad y orden.
- La comunicación en grupo implementa una comunicación *multicast*.

  Broadcast Comunicación a todos los procesos del sistema.

  Unicast Comunicación a un único proceso.
- Un proceso utiliza sólo una operación *multicast* para enviar un mensaje a cada grupo de procesos (en Java esta operación es `aGroup.send(aMessage)`):
  - Utilización eficiente del ancho de banda.
  - Se minimiza el tiempo total para entregar el mensaje a todos los destinatarios.
  - Garantía de entrega a todos los destinatarios.
  - Fiabilidad y orden de los mensajes.

El modelo de programación (II)

Grupos de procesos

- Grupos donde las entidades de comunicación son procesos:
  - Los mensajes se entregan a los procesos.
  - Los mensajes son normalmente vectores desestructurados de bytes.
- Ejemplo: *JGroups toolkit*.
- Más utilizados que los grupos de objetos.

El modelo de programación (III)

Grupos de objetos

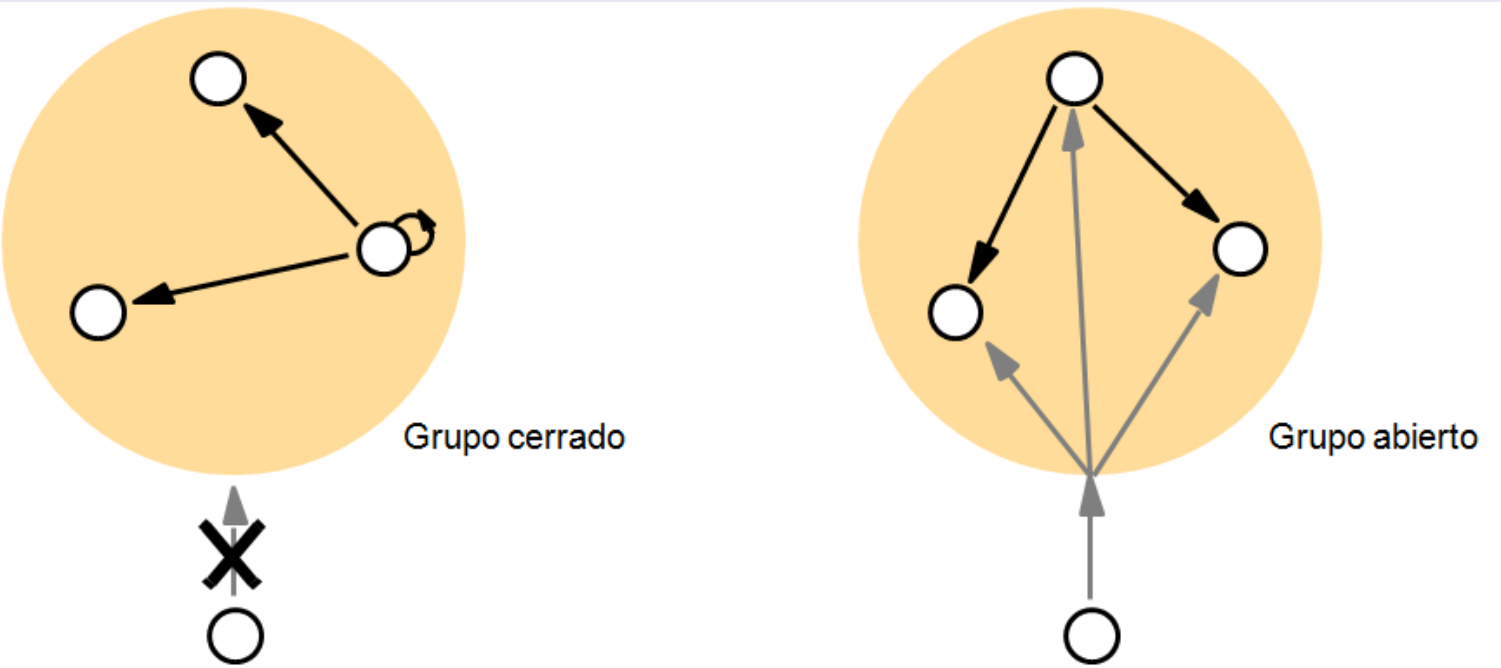
- Ofrecen un enfoque de alto nivel para la computación de grupos.
- Un grupo de objetos es una colección de objetos que procesan el mismo tipo de invocaciones concurrentemente.
- Los objetos del cliente no necesitan ser conscientes de la replicación.
- Invocan operaciones en un objeto local y único que actúa como un proxy para el grupo.
- El proxy usa un sistema de comunicación de grupo para enviar las invocaciones a los miembros de los grupos de objetos.
- Ejemplo: *Electra*.

El modelo de programación (IV)

Grupos cerrados y abiertos

Cerrado Únicamente los miembros del grupo pueden realizar *multicast* dentro del grupo. Un proceso se entrega a sí mismo cualquier mensaje que disemine al grupo.

Abierto Procesos externos al grupo pueden realizar *multicast*.



El modelo de programación (V)

Grupos solapados y no solapados

Solapados Las entidades (procesos u objetos) podrían ser miembros de distintos grupos.

No solapados Los miembros no se solapan, cada proceso pertenece a un único grupo.

Sistemas síncronos y asíncronos

Se deben considerar ambos entornos.

Aspectos de implementación (I)

Fiabilidad en las comunicaciones

El *multicast* fiable tiene 3 propiedades:

Integridad El mensaje que se recibe es el mismo que el que se envió y se entrega correctamente una única vez.

Validez El mensaje que se envía será finalmente entregado.

Acuerdo Si el mensaje se entrega a un proceso, entonces será entregado a todos los procesos del grupo.

Aspectos de implementación (II)

Orden relativo de los mensajes entregados a múltiples destinos

Los servicios de comunicación en grupo ofrecen *multicast* ordenado, con algunas de estas propiedades de ordenación:

FIFO Si un proceso envía un mensaje antes que otro, se entregará en este orden a todos los procesos del grupo (punto de vista del proceso).

Causal Si un mensaje ocurre antes que otro mensaje (relación causal) también será así en la entrega de los mensajes asociados a todos los procesos (punto de vista de los mensajes).

Total Si un mensaje se entrega antes que otro en un proceso, entonces se utilizará el mismo orden en todos los procesos.

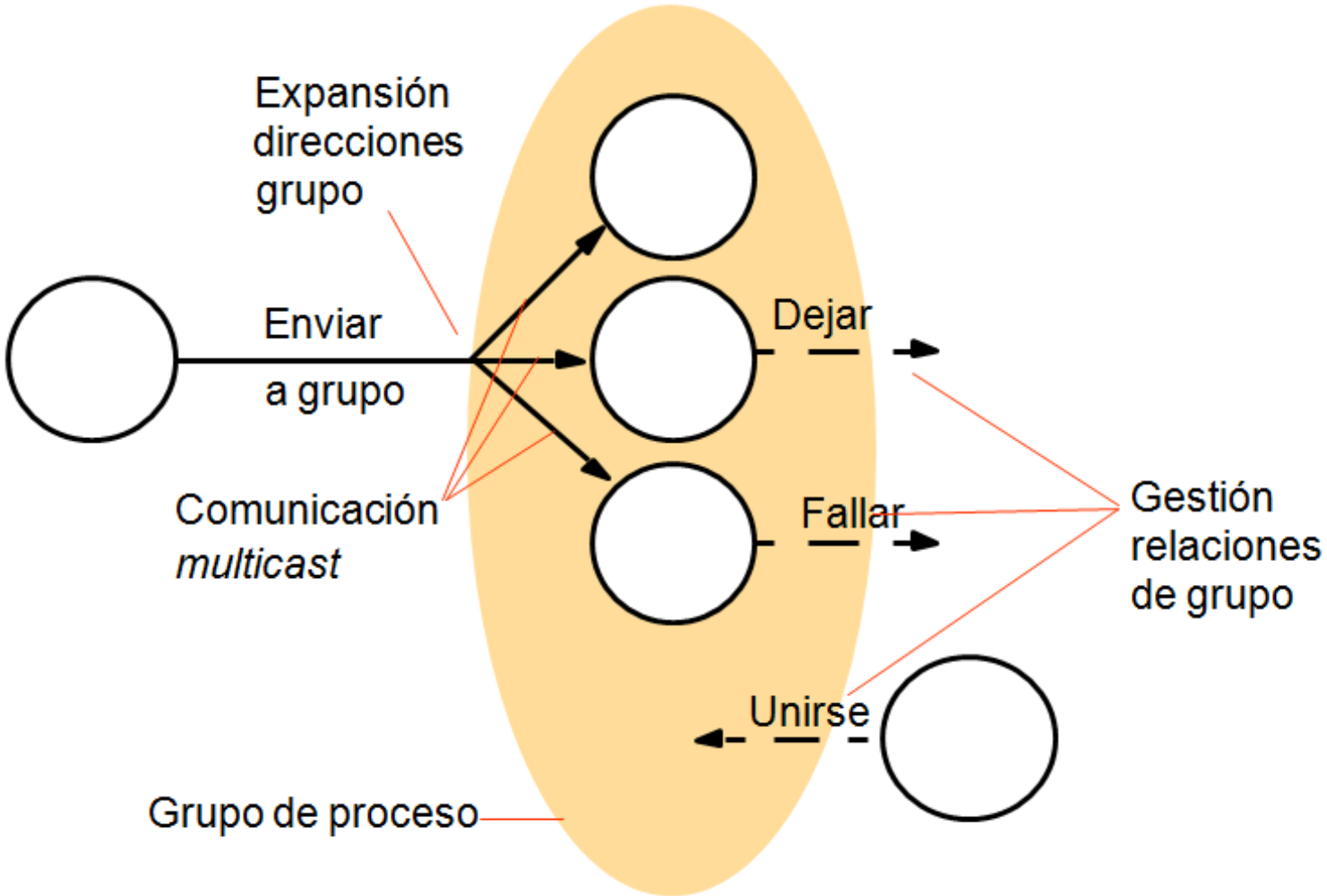
Aspectos de implementación (III)

Gestión de las relaciones de grupo

- Las entidades que participan en la comunicación pueden **unirse** o **dejar** el grupo, e incluso **fallar**.
- Un servicio de relación de grupo realiza 4 tareas:
  - **Ofrecer una interfaz para los cambios en las relaciones de grupo**: El servicio ofrece operaciones para crear y destruir grupos de procesos y añadir o eliminar un proceso de un grupo. Un proceso puede pertenecer a varios grupos (solapamiento de grupo).
  - **Detección de fallo**: El servicio monitoriza colisiones y miembros inalcanzables debido a fallos de comunicación. El detector marcará los procesos como *Suspected* o *Unsuspected*.
  - **Notificación de los cambios a los miembros del grupo**: El servicio notifica a los miembros cuando se añade un proceso o se excluye.
  - **Realizar una expansión de las direcciones de grupo**: Cuando un proceso disemina un mensaje proporciona el identificador de grupo (no una lista de procesos del grupo). El servicio obtendrá del identificador las relaciones de grupo para realizar la entrega.

Aspectos de implementación (IV)

Rol de gestión de las relaciones de grupo



Sección 3 | Sistemas publicador-suscriptor

Introducción (I)

Sistemas **publicador-suscriptor** (o **distribuidos basados en eventos**)

- Sistema donde los **publicadores** publican eventos estructurados a un servicio de eventos.
- Y los **suscriptores** expresan interés por ciertos eventos, a través de **suscripciones** que pueden ser patrones sobre estos eventos estructurados.
- Ejemplo: un suscriptor podría estar interesado en todos los eventos relacionados con noticias de la ciudad de Cádiz.
- El objetivo principal de estos sistemas es relacionar las suscripciones con los eventos publicados y asegurar la entrega correcta de las **notificaciones de eventos**.
- Un evento será entregado a algunos suscriptores (un paradigma de comunicaciones *one-to-many*).

Introducción (II)

Aplicaciones de sistemas publicador-suscriptor

Gran variedad de dominios de aplicación, especialmente relacionados con diseminación de eventos a gran escala:

- Sistemas de información financieros.
- Flujos de datos en tiempo real: RSS *feeds*, plataformas IoT (Cosm)...
- Apoyo al trabajo colaborativo: los participantes necesitan estar informados de eventos de interés compartido.
- Apoyo a la computación ubicua, incluyendo la gestión de eventos que provienen de infraestructuras ubicuas (eventos de localización...).
- Aplicaciones de monitorización, incluyendo monitorización de redes en Internet.
- Componente clave de la infraestructura de Google (diseminación de eventos sobre anuncios, *ad clicks*, a las partes interesadas).

Introducción (III)

Ejemplo de sistema publicador-suscriptor (I)

- Sistema cuya tarea consiste en permitir que los comerciantes usen los ordenadores para ver los precios actualizados de los *stocks* en los que están interesados.
- El precio del mercado para un *stock* concreto se representa con un objeto.
- La información llega a este sistema desde distintas fuentes externas en forma de actualizaciones a algunos o todos los objetos que representan los *stocks*.
- Esta información es recogida por procesos (llamados **proveedores de información**).
- Los comerciantes sólo están interesados en unos determinados *stocks*.

Introducción (IV)

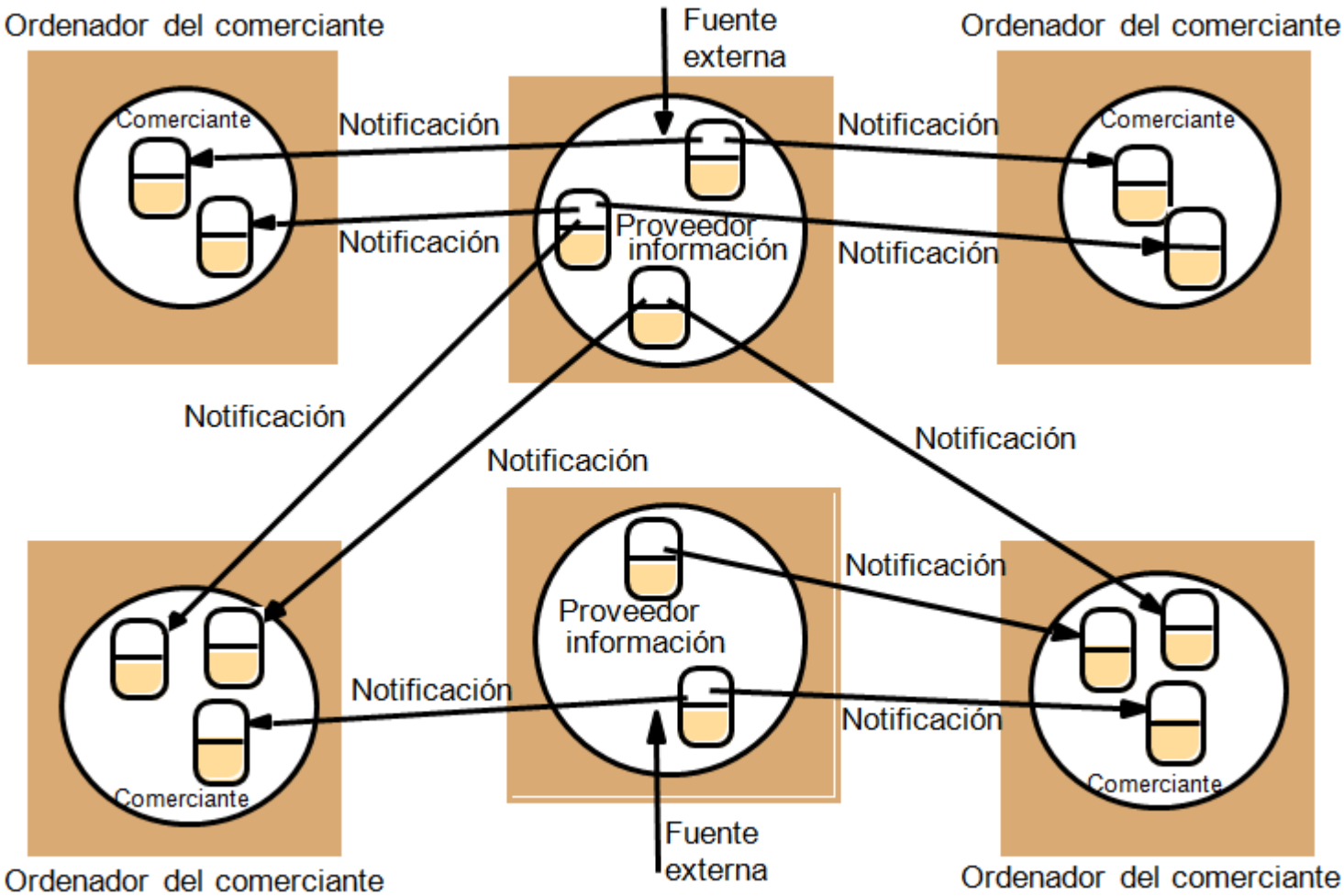
Ejemplo de sistema publicador-suscriptor (II)

Este sistema puede implementarse con procesos con 2 tareas diferentes:

- Tarea 1:
  - Un proceso proveedor de información continuamente recibe nueva información desde una fuente externa.
  - Cada una de las actualizaciones es vista como un **evento**.
  - Este proveedor publica cada evento al sistema publicador-suscriptor para entregarlo a todos los comerciantes que están interesados.
  - Habrá un proceso proveedor de información distinto para cada fuente externa.
- Tarea 2:
  - Un proceso comerciante crea una suscripción.
  - Cada suscripción expresa un interés sobre eventos relacionados con un *stock* dado por un proveedor de información.
  - El proceso recibirá toda la información enviada en forma de notificaciones y la mostrará al usuario.

Introducción (V)

Ejemplo de comunicación de notificaciones en un sistema publicador-suscriptor



Introducción (VI)

Características de los sistemas publicador-suscriptor

- **Heterogeneidad**:
  - Cuando las notificaciones de eventos son utilizadas como medio de comunicación, los componentes en un sistema distribuido que no fueron diseñados para interoperar pueden trabajar juntos.
  - Se requiere que los objetos generadores de eventos publiquen los tipos de eventos que ofrecen, y que los otros objetos se suscriban a **patrones de eventos** y ofrezcan una interfaz para recibir y tratar las notificaciones resultantes.
- **Asincronía**:
  - Las notificaciones se envían asíncronamente por publicadores generadores de eventos a todos los suscriptores que están interesados en éstas.
  - Se evita que los publicadores necesiten sincronizarse con los suscriptores (publicadores y suscriptores desacoplados).

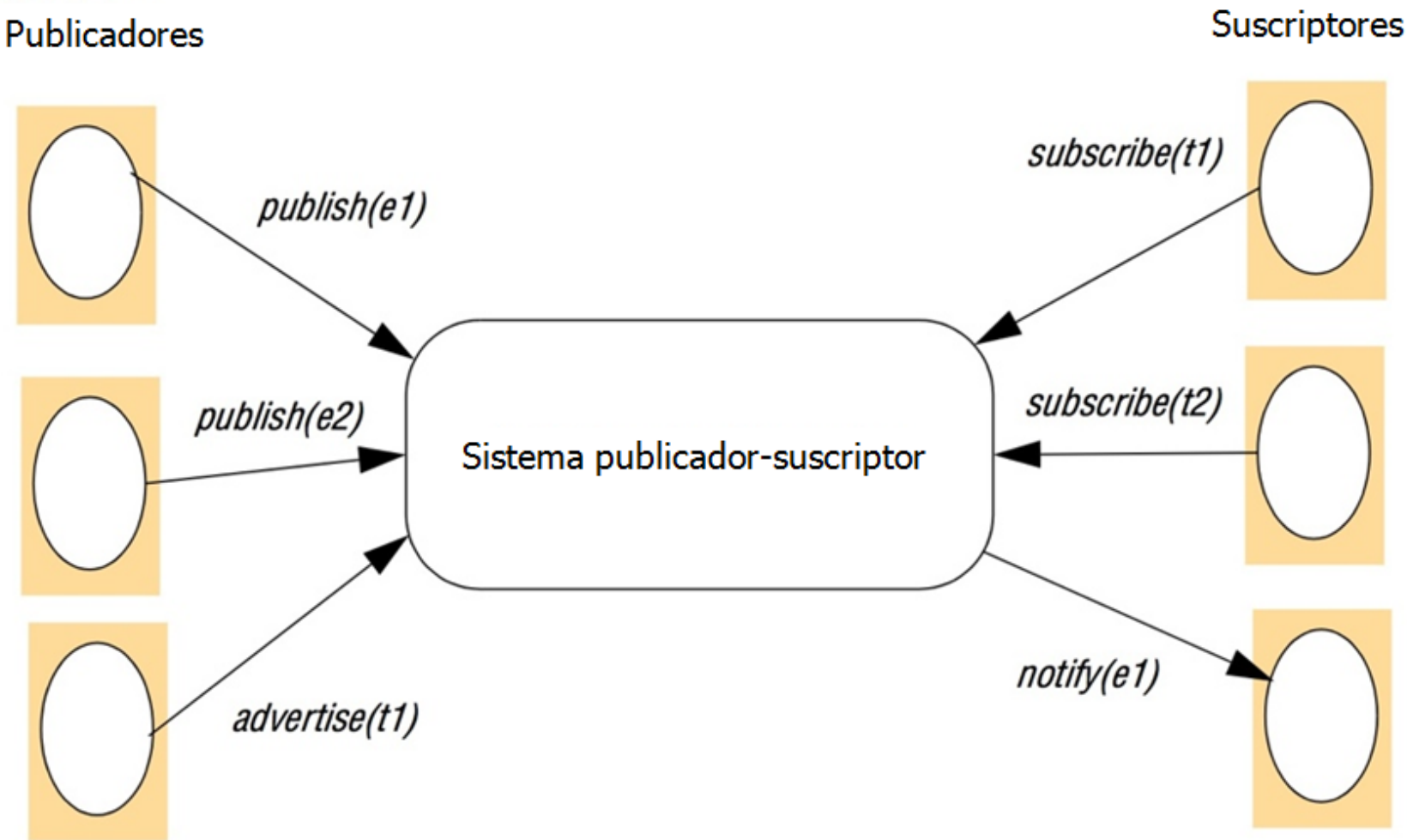
El modelo de programación (I)

Se basa en un conjunto de operaciones:

- Los publicadores diseminan un evento e a través de una operación `publish(e)`.
- Los suscriptores expresan interés en un conjunto de eventos a través de suscripciones: operación `subscribe(f)`, `f` es un filtro (patrón definido sobre el conjunto de todos los eventos posibles).
- La expresividad de los filtros viene determinada por el modelo de suscripción.
- Los suscriptores pueden revocar más tarde su interés con la operación `unsubscribe(f)`.
- Los eventos que llegan a un suscriptor son entregados usando la operación `notify(e)`.
- Los publicadores opcionalmente declararán los estilos de eventos que generarán a través de anuncios, con la operación `advertise(f)`. Para revocarlos: `unadvertise(f)`.
- Los anuncios son definidos como tipos de eventos de interés.

El modelo de programación (III)

Paradigma publicador-suscriptor



El modelo de programación (II)

La expresividad de los filtros viene determinada por el modelo de suscripción, a partir de distintos esquemas basados en:

- **Canal** (*channel-based*): Los publicadores publican eventos a canales concretos y los suscriptores entonces se suscriben a uno de esos canales para recibir todos los eventos que le lleguen (único esquema que define un canal físico).
- **Tema** (*topic-based* o *subject-based*): Uno de los atributos de las notificaciones especifica el tema. Los suscriptores especificarán el tema en el que están interesados.
- **Contenido** (*content-based*): Generalización del basado en temas. Una consulta definida mediante restricciones sobre los valores de los atributos de eventos.
- **Tipos** (*type-based*): Cada objeto es de un tipo específico. Las suscripciones se definen mediante tipos de eventos. También se permite realizar consultas sobre atributos y métodos de objetos. Puede ser integrado con lenguajes de programación.

El modelo de programación (III)

Otros enfoques actuales y de investigación

Añadir la expresividad del contexto

- El contexto puede ser definido como circunstancias físicas que son relevantes en el comportamiento del sistema.
- Un ejemplo de contexto es la localización.
- Muy utilizado en la computación ubicua y móvil.

El modelo de programación (IV)

Otros enfoques actuales y de investigación

Procesamiento de eventos complejos o *Complex Event Processing* (CEP)

- En algunos sistemas (por ejemplo, la bolsa) no es suficiente para las suscripciones expresar consultas sobre eventos individuales.
- CEP permite procesar, analizar y correlacionar grandes cantidades de eventos.
- Para detectar y responder en tiempo real a situaciones críticas o relevantes del negocio.
- Se utilizan unos patrones de eventos que inferirán nuevos eventos más complejos y con un mayor significado semántico.
- Requisitos: motor CEP (p.e. Esper) y lenguaje específico (p.e. EPL).

El modelo de programación (V)

Otros enfoques actuales y de investigación

Detección de un caso sospechoso de gripe aviar utilizando CEP

Patrón de evento complejo

Evento complejo

Aspectos de implementación (I)

Implementaciones centralizadas

- Un único nodo con un servidor que funciona como un agente (*broker*) de eventos. Los publicadores envían eventos (opcionalmente también anuncios) a este agente y los suscriptores envían suscripciones al agente para recibir notificaciones.
- La interacción con el agente es mediante una serie de mensajes punto-a-punto, implementados mediante paso de mensajes o invocación remota.
- Poco escalable: el agente puede fallar y además se convierte en un cuello de botella.

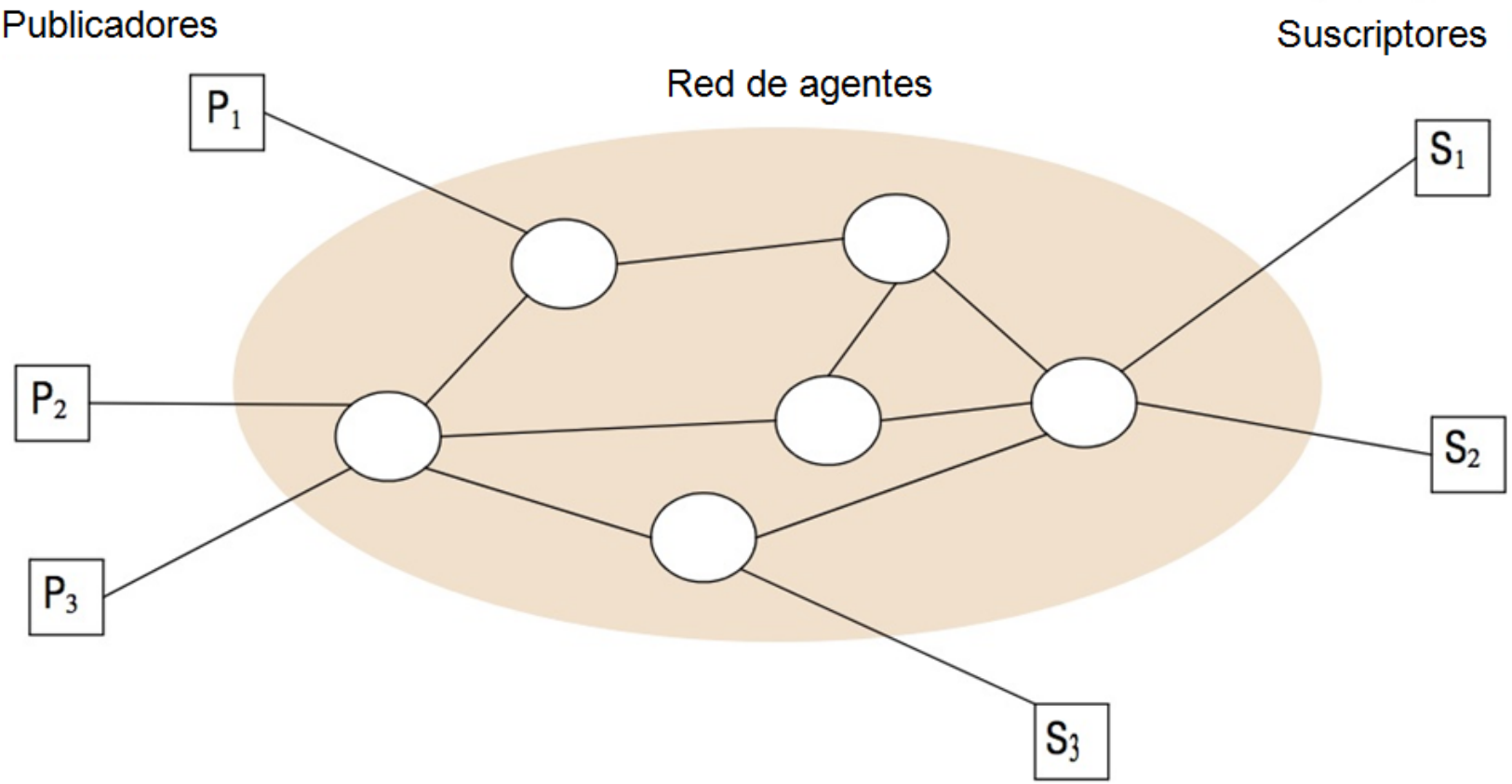
Aspectos de implementación (II)

Implementaciones distribuidas

- El agente centralizado se sustituye por una red de agentes (*network of brokers*).
- Estos agentes cooperan para ofrecer la funcionalidad deseada.
- Sobreviven a fallos de nodos.
- Buen funcionamiento en aplicaciones a escala Internet.

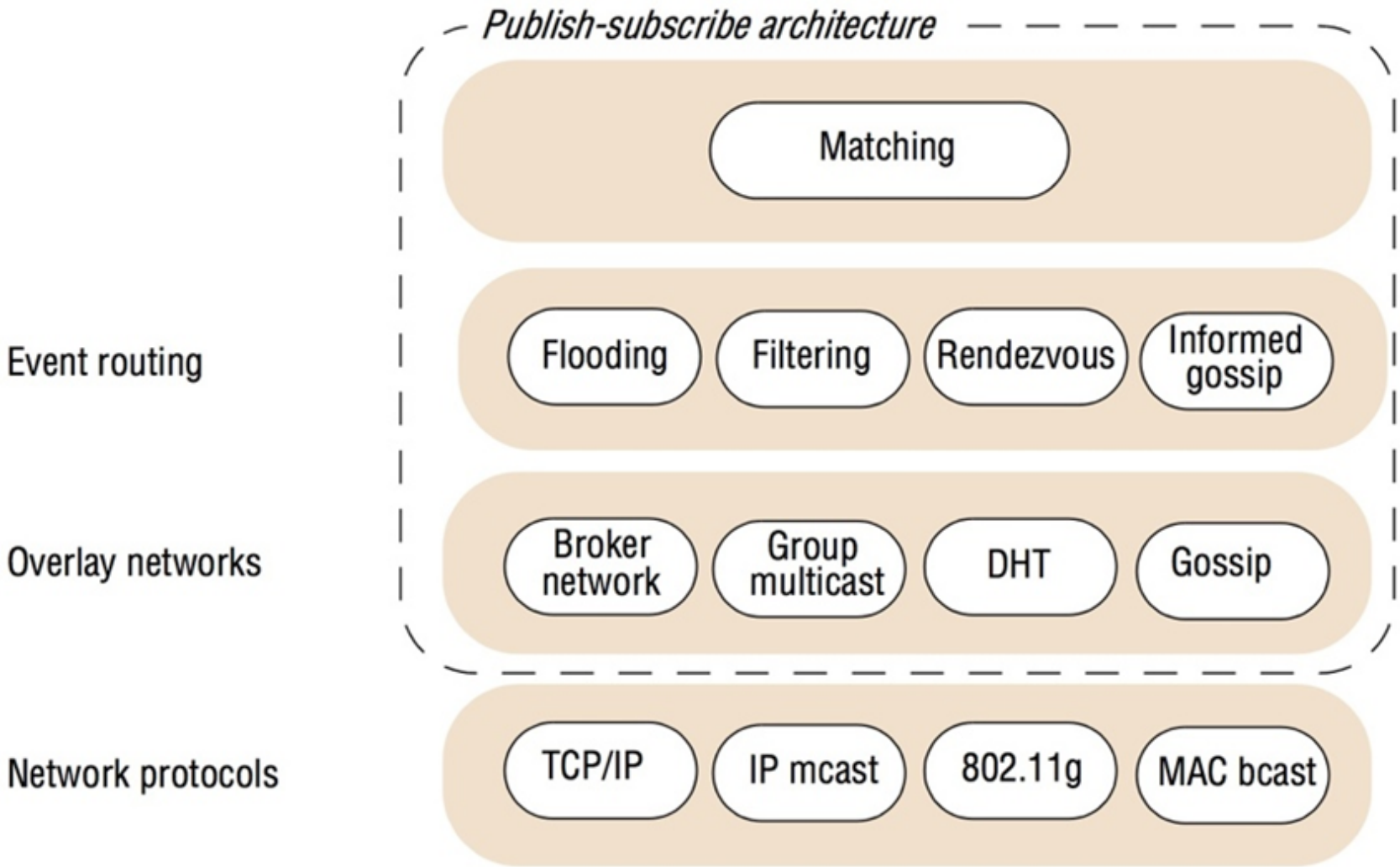
Aspectos de implementación (III)

Implementaciones distribuidas: Red de agentes



Aspectos de implementación (IV)

Arquitectura de sistemas publicador-suscriptor



Aspectos de implementación (V)

Arquitectura de sistemas publicador-suscriptor (I)

Algunas de las implementaciones de encaminamiento de eventos basado en contenido (*content-based routing*):

Flooding Enviar una notificación de eventos a todos los nodos de la red. Puede implementarse utilizando un *broadcast* o *multicast*.

Filtering Encaminamiento basado en filtrado. Los agentes envían notificaciones a través de la red donde haya un camino hacia un suscriptor válido. Cada nodo debe mantener una lista de todos los vecinos conectados, una lista de suscripción con todos los suscriptores conectados por ese nodo y una tabla de encaminamiento.

Aspectos de implementación (VI)

Arquitectura de sistemas publicador-suscriptor (II)

Advertisements *Filtering* puede generar una gran cantidad de tráfico debido a la propagación de suscripciones. *Advertisements* lo reduce propagando los anuncios hacia los suscriptores de forma simétrica a la propagación de suscripciones.

Rendezvous Se definen unos nodos *rendezvous* que son nodos agentes responsables de un subconjunto del espacio de eventos. El espacio de eventos puede ser relacionado (mapeado) con una tabla de dispersión distribuida o *Distributed Hash Table* (DHT) (se distribuye sobre un conjunto de nodos en una red *peer-to-peer*).

Informed gossip Se operan con nodos de red intercambiando periódicamente y de forma probabilística eventos o datos con los nodos vecinos.

Sección 4 | Colas de mensaje

Introducción (I)

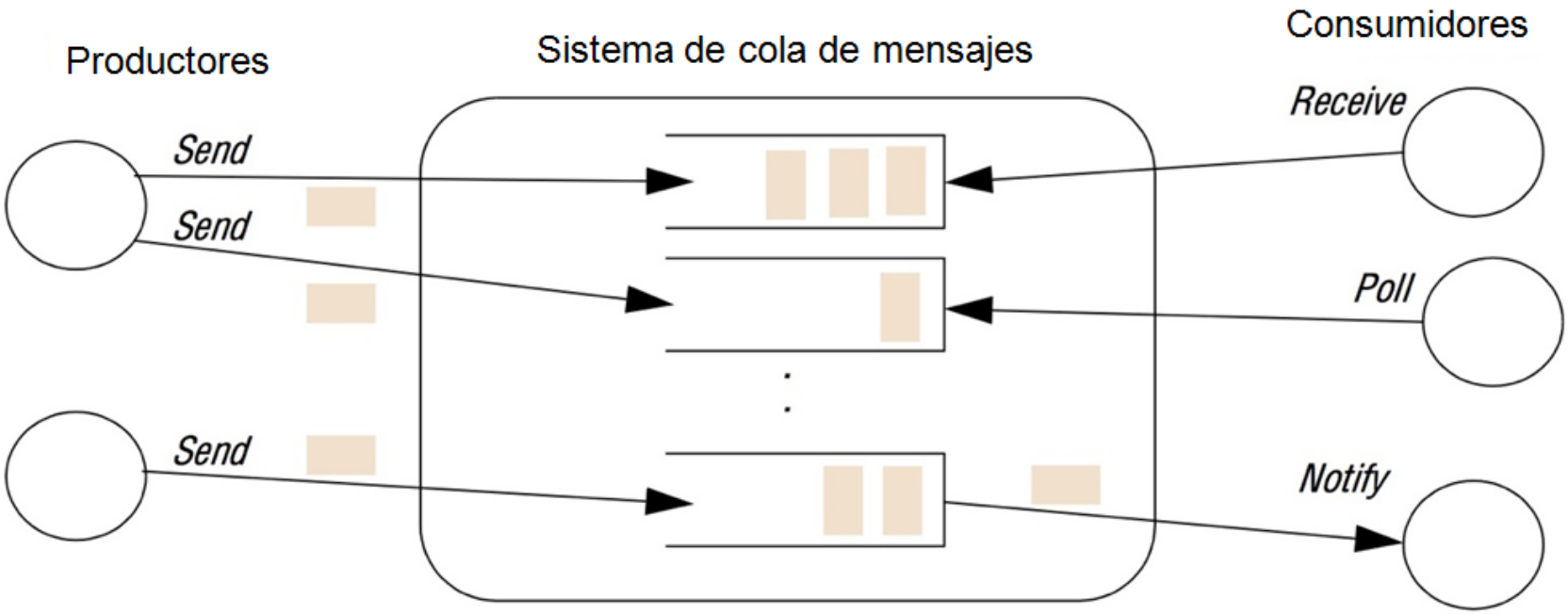
- Las colas de mensajes son una categoría importante de los sistemas de comunicación indirecta.
- Los grupos y sistemas publicación-suscripción proporcionan un estilo de comunicación uno-a-muchos.
- Las colas de mensajes proporcionan un servicio punto-a-punto (desacoplamiento de espacio y tiempo).
- El emisor envía el mensaje a una cola y será recogido por un único proceso.
- Las colas también son vistas como *Message-Oriented Middleware* (MOM).
- Ejemplo: *Java Messaging Service* (JMS), soporta cola de mensajes y publicación-suscripción.

El modelo de programación (I)

- Comunicación en sistemas distribuidos a través de colas.
- Los procesos productores envían mensajes a una cola específica y otros procesos (consumidores) podrán recibir los mensajes de esta cola.
- Los estilos de recepción pueden ser:
  - **Recepción bloqueante (*blocking receive*)**: se bloqueará hasta que esté disponible un mensaje apropiado.
  - **Recepción no bloqueante (*non-blocking receive*)**: una operación *polling* comprobará los estados de la cola y devolverá un mensaje si está disponible. En caso contrario, indicará que no lo está.
  - **Operación de notificación (*notify*)**: emitirá una notificación de evento cuando un mensaje esté disponible en la cola asociada.

El modelo de programación (II)

Paradigma de cola de mensajes



El modelo de programación (III)

- Varios procesos pueden enviar mensajes a la misma cola.
- Varios receptores pueden obtener mensajes de una misma cola.
- La política para gestionar la cola suele ser FIFO, aunque también se puede utilizar la prioridad (entregando primero los mensajes de mayor prioridad).
- Los procesos consumidores pueden también seleccionar los mensajes de la cola según las propiedades del mensaje.
- Un mensaje contiene:
  - Destino: identificador único que indica la cola destino.
  - *Metadata* asociado con el mensaje.
  - Otros campos, como la prioridad del mensaje y el modo de entrega.
  - El cuerpo del mensaje, es opaco.

El modelo de programación (IV)

- Los mensajes son persistentes (las colas almacenan los mensajes indefinidamente hasta que sean consumidos y escriben los mensajes en disco para asegurar la entrega).
- Comunicación fiable: cualquier mensaje enviado es recibido (validez) e idéntico al enviado, y se entrega una única vez (integridad).
- Las colas de mensaje garantizan la entrega de los mensajes, pero no el tiempo de la entrega.
- Estos sistemas pueden ofrecer otras funcionalidades:
  - Envío y recepción de mensajes en forma de transacción.
  - Transformación de mensajes.
  - Mecanismos de seguridad: *Secure Sockets Layer* (SSL), soporte para autenticación y control de acceso. Ejemplo: WebSphere MQ de IBM.

Aspectos de implementación (I)

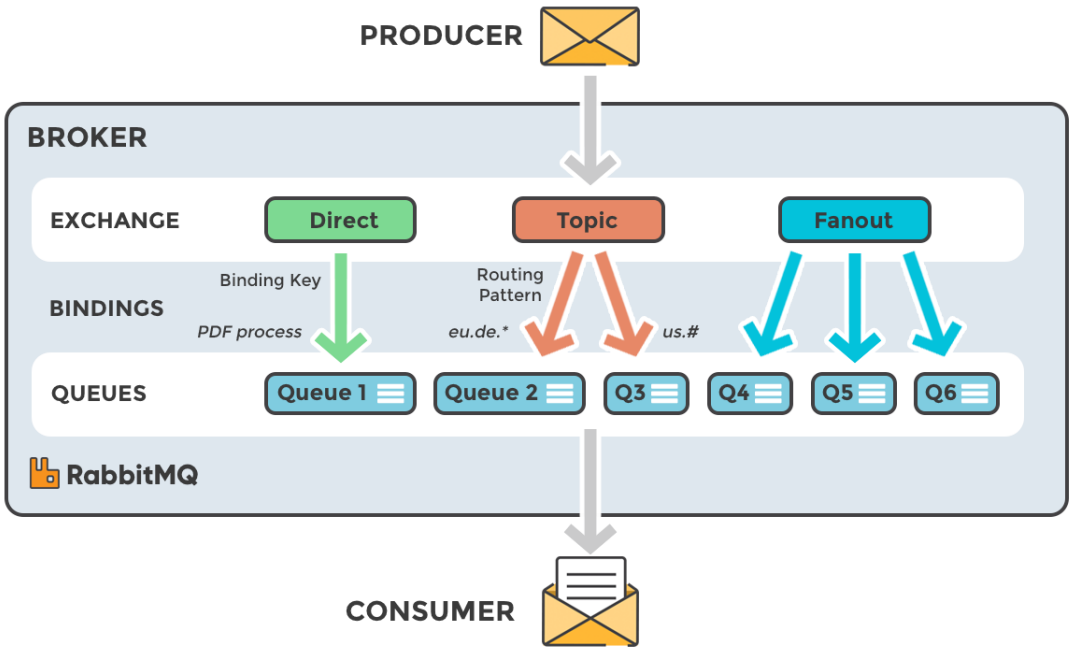
RabbitMQ

- Un software de código abierto para la gestión de mensajes.
- Implementa, desde sus inicios, el estándar *Advanced Message Queuing Protocol* (AMQP).
- Ampliado para dar soporte a los protocolos *Streaming Text Oriented Messaging Protocol* (STOMP) y *MQ Telemetry Transport* (MQTT).
- Implementa una gran variedad de clientes: Python, Java, .NET, Ruby, PHP, etc.
- RabbitMQ es el gestor de mensajes (message broker).

Aspectos de implementación (II)

Pika

- Es una biblioteca de Python que nos permite interactuar con RabbitMQ.
- En definitva, es un cliente AMQP 0-9-1 puro de Python.



Aspectos de implementación (III)

Envío

```
import pika

connection = pika.BlockingConnection(pika.
    ↪ ConnectionParameters(host="localhost"))
channel = connection.channel()
channel.queue_declare(queue="hello")
channel.basic_publish(exchange="", routing_key="hello",
    ↪ body="Hello␣World!")
print("␣[x]␣Sent␣'Hello␣World!'")
connection.close()
```

Aspectos de implementación (IV)

Recepción

```
import pika

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

connection = pika.BlockingConnection(pika.
    ↪ ConnectionParameters(host="localhost"))
channel = connection.channel()
channel.queue_declare(queue="hello")
print(" [*] Waiting for messages. To exit press Ctrl+C")
channel.basic_consume(queue="hello", on_message_callback
    ↪ =callback)
channel.start_consuming()
```

Bibliografía

Coulouris, G.; Dollimore, J.; Kindberg, T.
Distributed Systems: Concepts and Design (5ª ed.)
Addison-Wesley, 2012.