

Comunicaciones entre procesos distribuidos

Unidad 4



Introducción

- Procesos necesitan comunicarse
- Comunicación: base de sistemas distribuidos
- Alternativa 1: memoria compartida
 - No puede existir comúnmente; requiere middleware que implemente abstracción
 - No sirve para sincronización
- Alternativa 2: paso de mensajes
 - Forma más común
 - Facilita abstracciones útiles para programadores

Applications, services

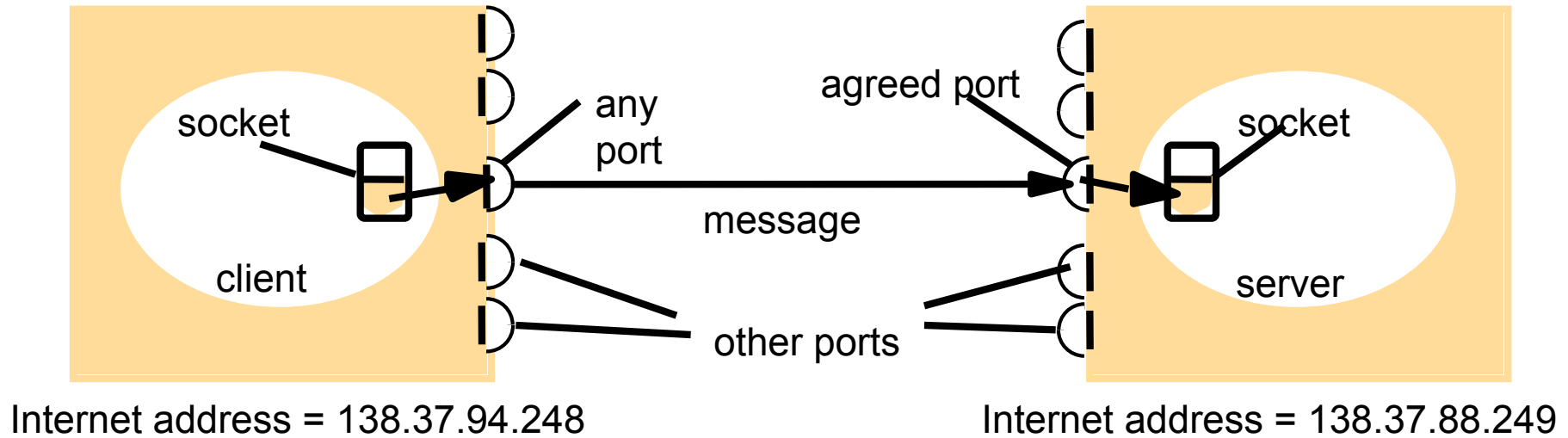
Remote invocation, indirect communication

Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks

UDP and TCP

Middleware
layers

Sockets



Características deseables del canal

- **Confiabilidad**
 - Se garantiza entrega
- **Integridad**
 - Mensajes no deben ser alterados, corrompidos ni duplicados (a nivel de aplicación, no de red)
- **Orden**
 - Ciertas aplicaciones requieren que los mensajes sean entregados en "orden de envío"

UDP

- Envíos sin bloqueo, recepciones con bloqueo (se pueden usar timeouts)
- Recepción no ligada a un origen en particular (receive from any)
- Modelo de fallas
 - Integridad: checksums
 - Fallas de omisión: paquetes pueden perderse o descartarse (omisión en el canal)
 - Ordenamiento: mensajes pueden ser entregados fuera de “orden de envío”

UDP (cont.)

- UDP no es confiable, por lo que es más ligero que TCP
 - No necesita almacenar información en origen ni destino
 - No incurre en transmisión extra de mensajes
 - El origen no debe esperar (los acknowledgements)
- Ejemplo de uso: DNS

TCP

- Abstracción: Corriente (stream) de bytes
- Origen-destino establecen conexión
- Comunicación en conexión no necesita indicar direcciones ni puertos
- Establecimiento de conexión es innecesario y excesivo para simples pedido-respuesta de cliente/servidor
- Procesos acuerdan tipo de información a transmitir
- Recepciones con bloqueo (si cola está vacía)
- Envíos con bloqueo (si cola de destino está llena)

TCP (cont.)

■ No es 100% confiable

- No garantiza entrega de mensajes (en caso de congestión extrema, se rompe la conexión)

■ Modelo de fallas

- Integridad: checksums y paquetes con no. de secuencia (rechaza paquetes fuera de orden)
- Validez: timeouts y retransmisión para detectar y recuperarse de paquetes perdidos

■ Ejemplos

- HTTP, SSH, FTP, Telnet, SMTP

Datos y marshalling

■ Representación de datos varía entre plataformas ¿Cómo enviarlos?

- Usando un estándar
- Usar formato original + descripción del formato

■ Ejemplos

- CORBA: Common data representation
- Java: Serialización de objetos
- Protocol Buffers (Google)
- Apache Thrift (Facebook)
- Apache Avro (Hadoop)
- XML
- JSON

¿Quién se encarga del marshalling?

■ ¡Middleware!

- Evitar re-inventar la rueda
- Evitar introducir errores
- Mayor eficiencia

■ Ejemplos:

- CORBA, JAVA, Protocol Buffers, Thrift, Avro → formato binario
- HTTP, XML, JSON → Texto; ocupa más espacio

Más formatos en: https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

■ Otros usos (a más de RMI y RPC):

- transmisión de mensajes y almacenamiento en archivos

Ejemplo: XML

```
<person id="123456789">  
  <name>Smith</name>  
  <place>London</place>  
  <year>1984</year>  
  <!-- a comment -->  
</person >
```

Ejemplo: Protocol Buffers

- Mecanismo diseñado por Google para serializar datos estructurados
- Neutral a la plataforma y al lenguaje; extensibles
- Ventajas sobre XML:
 - Más simple
 - Más pequeño
 - Más rápido
 - Menos ambigüo
 - Más fácil de usar programáticamente

```
message Person {  
  required string name = 1;  
  required int32 id = 2;  
  optional string email = 3;  
}
```

```
Person john = Person.newBuilder()  
    .setId(1234)  
    .setName("John Doe")  
    .setEmail("jdoe@example.com")  
    .build();  
output = new FileOutputStream(args[0]);  
john.writeTo(output);
```



Marshalling en texto vs. binario

Serialized Contact List Size A comparison

JSON



100-120k

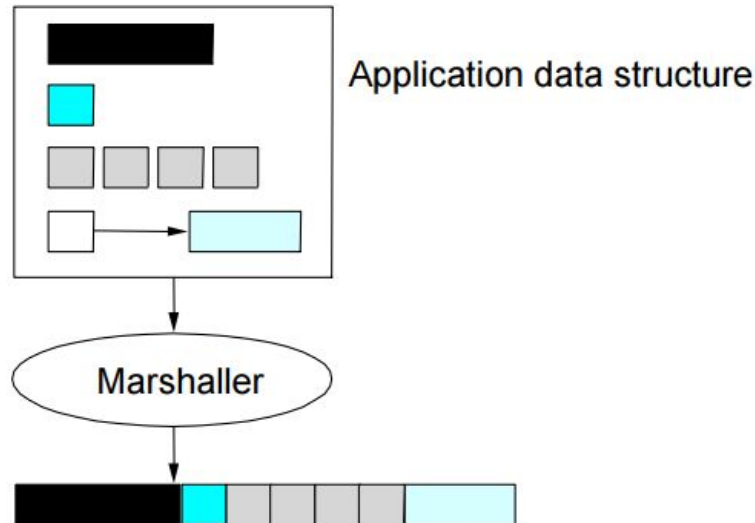
Thrift



40-50k

¿Qué exactamente se hace durante el marshalling?

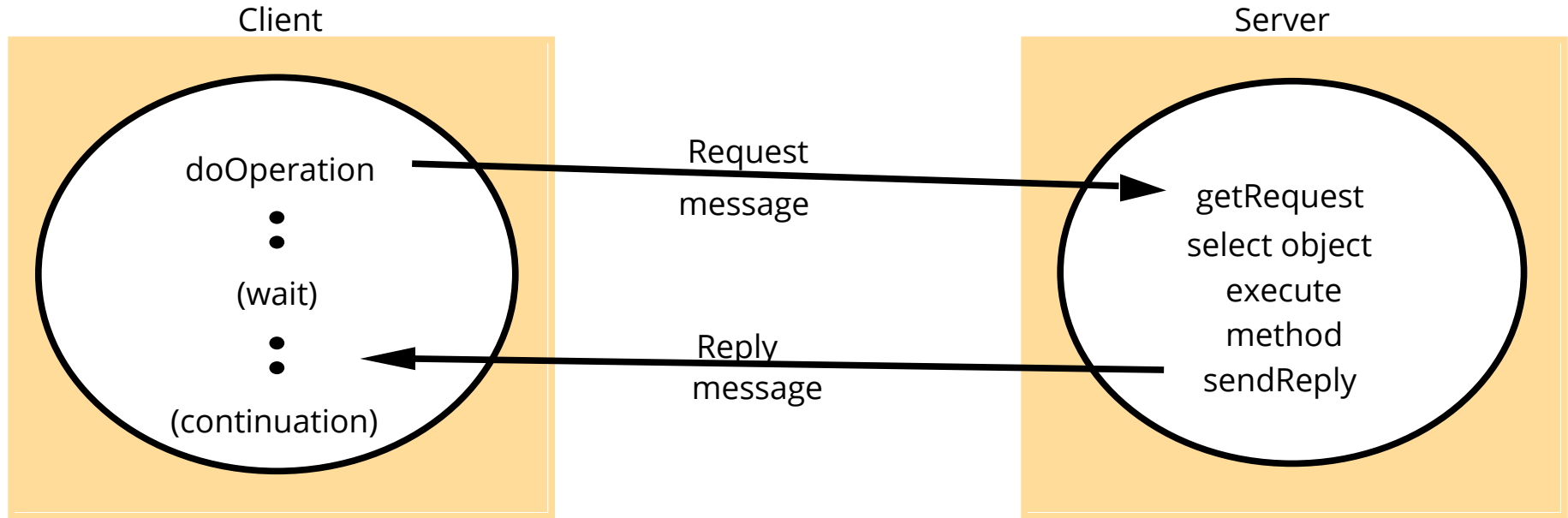
- Tipos de datos base (ints, floats) → convertir
- Tipos planos (estructuras, arreglos) → empacar
- Tipos complejos (punteros, estr. anidadas) → linearizar



Comunicación cliente/servidor

- Protocolo pedido-respuesta (request-reply)
 - Síncrono (cliente espera respuesta del servidor)
 - Confiable (respuesta cliente ayuda al servidor)
- Implementada típicamente sobre UDP
 - Evita sobrecarga (overhead) de TCP
 - ACKs son redundantes (respuestas son suficientes)
 - Establecer conexiones no es necesario
 - Control de flujo es usualmente no necesario (mayoría de invocaciones pasan pocos datos)

Protocolo pedido-respuesta



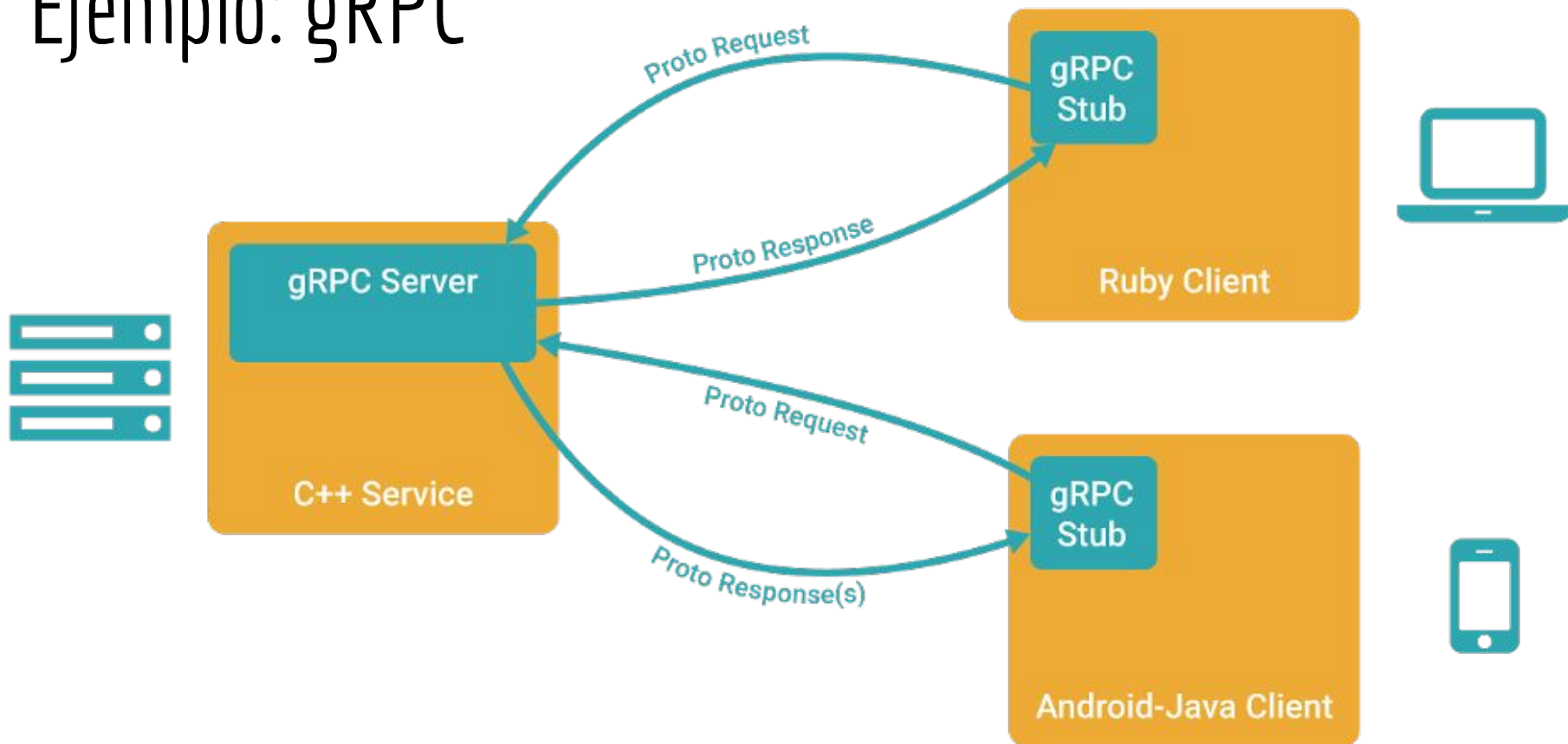
Protocolo pedido-respuesta (cont.)

- Modelo de fallas (para impl. sobre UDP)
 - Fallas de omisión
 - Ordenamiento no es garantizado
 - Fallos en los procesos
 - Asume modelo “crash”: proceso falla y permanece en ese estado; no se consideran fallas bizantinas
- Luego de un timeout se puede:
 - Retornar con error al cliente
 - Tratar nuevamente
 - Servidor descarta pedido repetido y re-envía respuesta

Pedido-respuesta sobre TCP

- Diferencias con implementación sobre UDP, dadas por las características de TCP
 - Confiable
 - Mensajes no son duplicados (servidor no necesita mantener historial)
 - Cliente no necesita preocuparse de re-transmitir pedidos
- Reduce complejidad en protocolo p-r
- Ejemplo: Java RMI, gRPC

Ejemplo: gRPC



Variates de RR

<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

Ejemplo 2: HTTP

Request:

<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

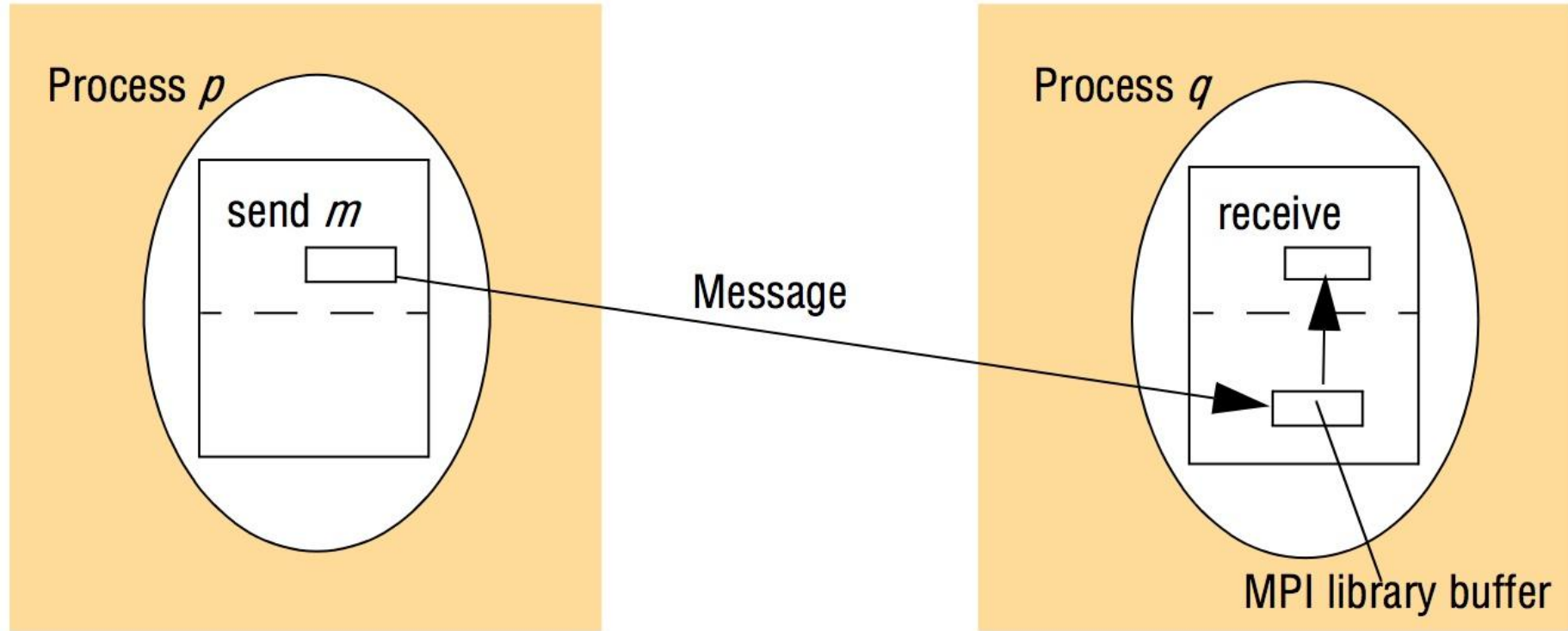
Reply:

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

Message Passing Interface (MPI)

- Sistema de paso de mensajes
- Protocolo de comunicación para programar computadoras paralelas
- Estándar *de facto* en HPC
- Define sintaxis y semánticas de rutinas de librería core
 - Existen implementaciones open source y comerciales
- Portable y escalable
- Lenguajes: C, C++ y Fortran

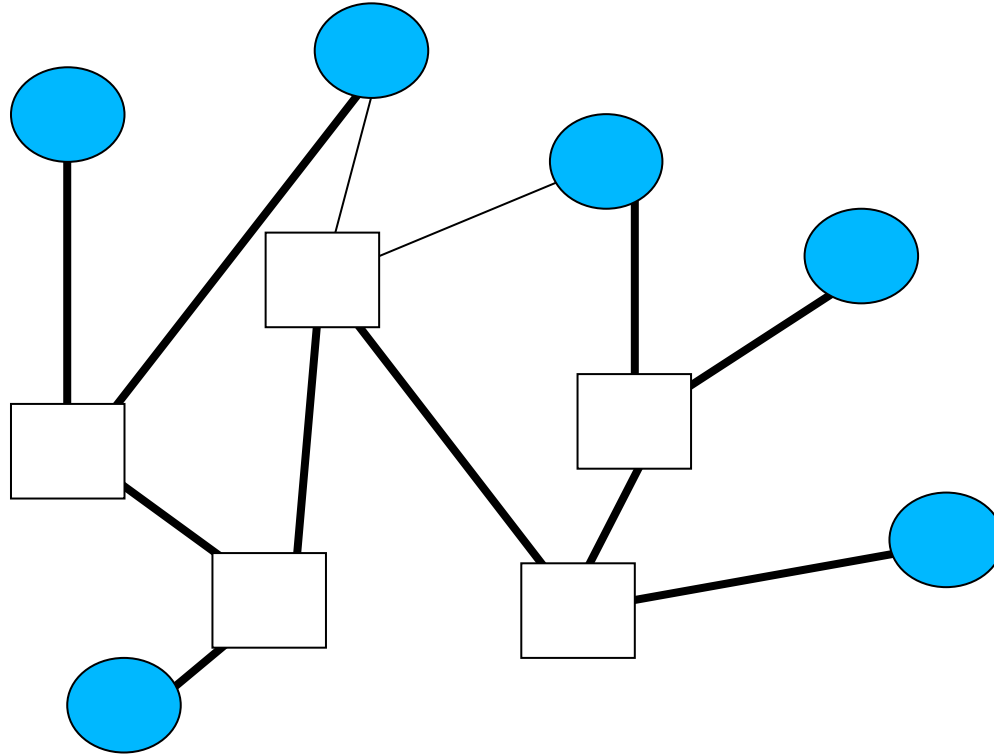
MPI (cont.)



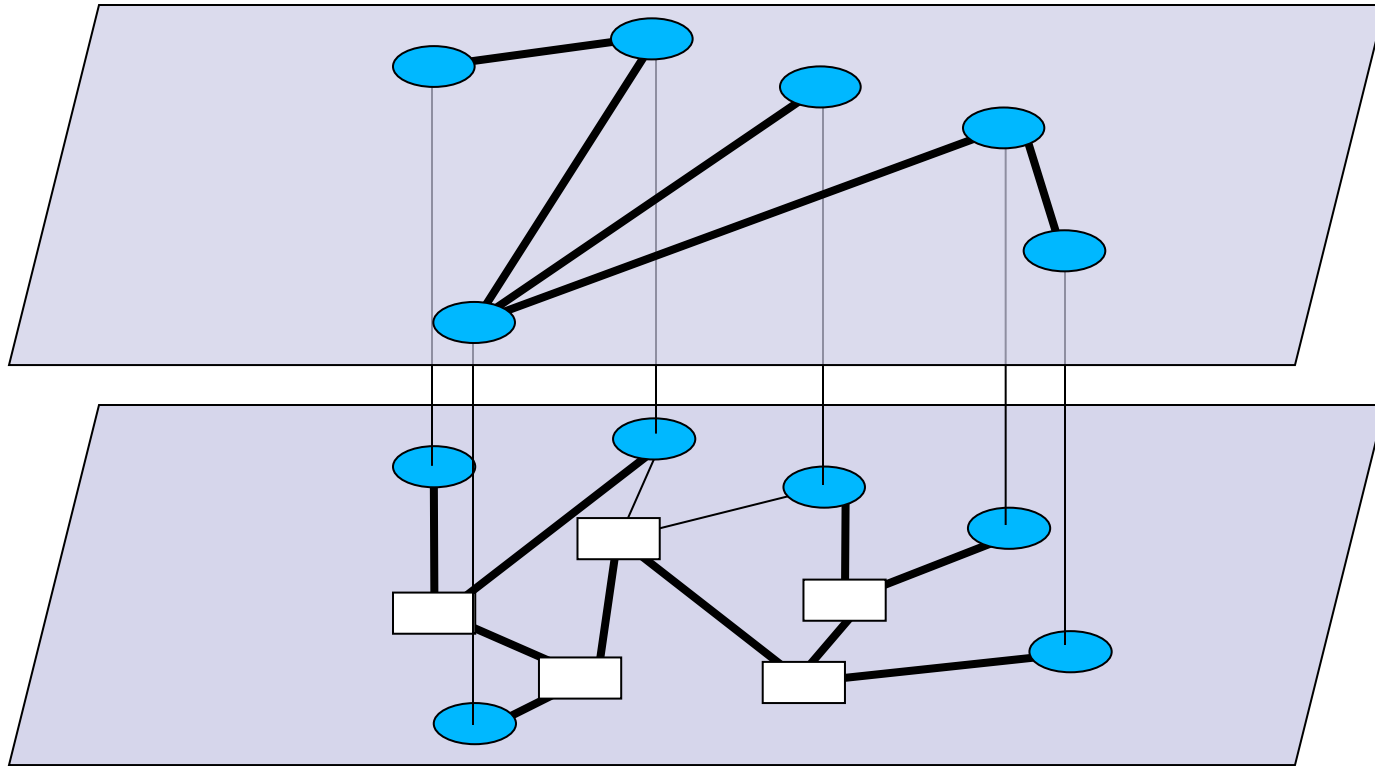
Comunicación de grupos

- Intercambio entre dos puntos no es ideal para comunicación de grupos
- Alternativa: multicast
 - Un mensaje es dirigido a todo un grupo
 - Manejo de membresía debe ser transparente para quien envía
- Puede ser confiable, ordenado, ambos o ninguno
- Implementaciones
 - IP Multicast
 - Multicast en capa de aplicación (overlay network)

IP Multicast



Multicast en capa de aplicación



Tipos de overlay networks (1/2)

<i>Motivation</i>	<i>Type</i>	<i>Description</i>
<i>Tailored for application needs</i>	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

Tipos de overlay networks (2/2)

*Tailored for
network style*

Wireless ad hoc
networks

Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.

Disruption-tolerant
networks

Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.

*Offering additional
features*

Multicast

One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [[mbone](#)].

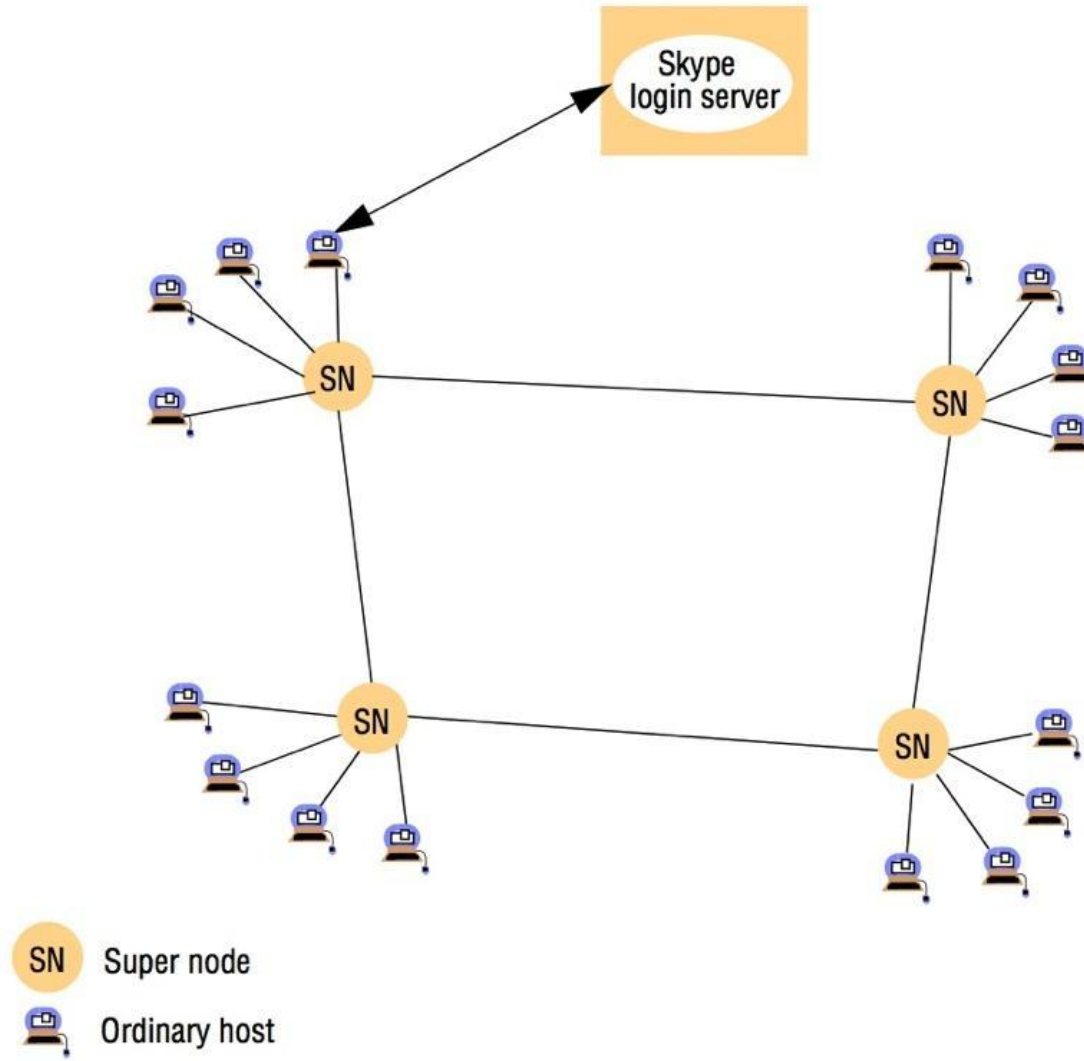
Resilience

Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [[nms.csail.mit.edu](#)].

Security

Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Ejemplo:





Abstracciones en capas superiores



Applications, services

Remote invocation, indirect communication

Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks

UDP and TCP

Middleware
layers

Paradigmas de comunicación distribuida

- 1º (nivel de abstracción):
Paso de mensajes (sockets TCP o UDP)
- 2º:
Protocolo pedido-respuesta o R o RRA, sobre sockets TCP o UDP
- 3º:
Invocaciones distribuidas: RPC/RMI, REST/SOAP
- 4º:
Tuple spaces
Message queuing
- 5º:
Publish/subscribe

Table 1. Decoupling Abilities of Interaction Paradigms

Abstraction	Space decoupling	Time decoupling	Synchronization decoupling
Message passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (observer pattern)	No	No	Yes
Tuple spaces	Yes	Yes	Producer-side
Message queuing (Pull)	Yes	Yes	Producer-side
Publish/subscribe	Yes	Yes	Yes

Introducción

- Esta sección estudia modelos de programación para aplicaciones distribuidas
- Modelos tradicionales han sido adaptados:
 - Llamadas a procedimientos → RPC
 - Programación Orientada a Objetos → RMI
 - Programación basada en eventos → Eventos distribuidos
 - Notificaciones
 - Publish-subscribe
 - Comunicación productor/consumidor → Message queues

Elemento Clave: Middleware

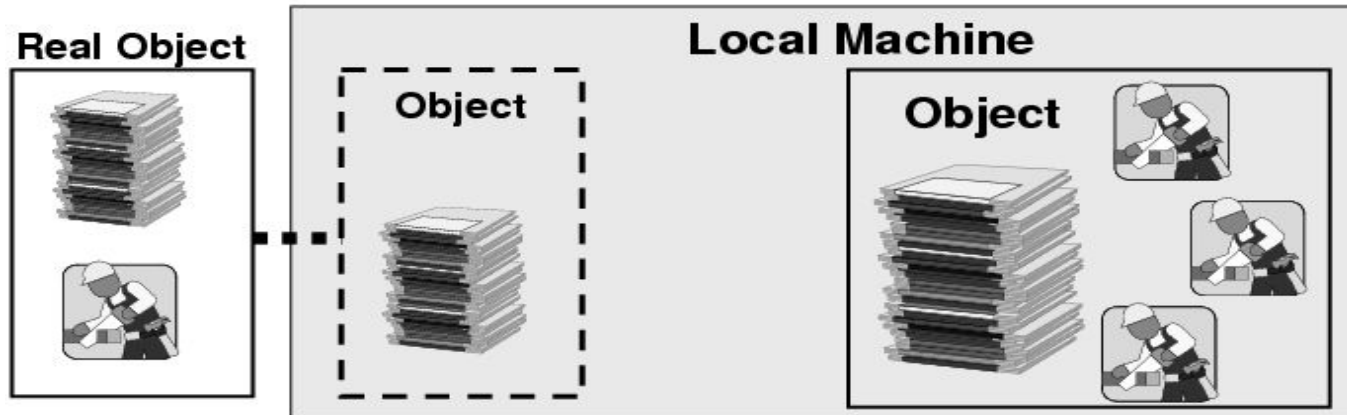
■ Transparencia de

- Ubicación: Local o remoto
- Protocolos de comunicación: TCP o UDP
- Hardware (Representación de datos)
- Sistema operativo
- Lenguaje de programación: Ej. CORBA
 - IDL: Lenguaje de Definición de Interfaces

Invocaciones Remotas (RPC)

Invocaciones remotas

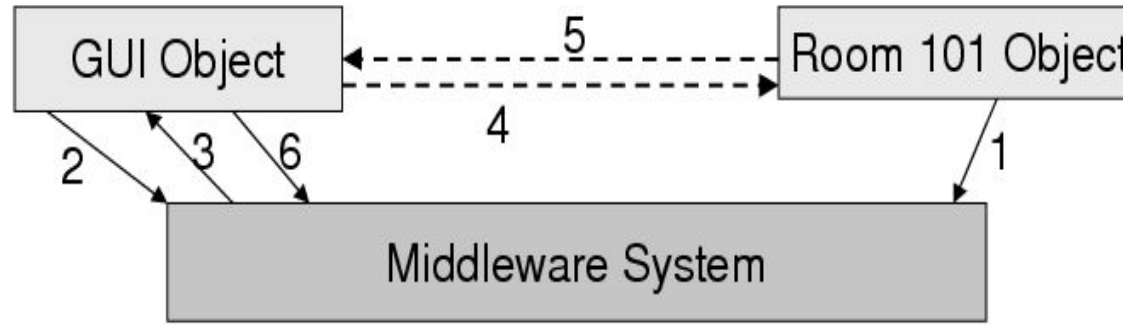
- Son un mecanismo de comunicación entre procesos distribuidos
- Buscan que las invocaciones a procedimientos o métodos remotos parezcan como invocaciones locales



Metas de RPC

- Facilidad de programación
- Esconder la complejidad (retransmisiones, tolerancia a fallos, marshalling, etc.)
- Automatizar tarea de implementar cómputos distribuidos
- Modelo de programación familiar para los programadores

Ejemplo: Sistema de Reservación de Habitaciones



1. Registrar control de habitación 101
2. ¿Quién controla la habitación 101?
3. El objeto 5a62b9D
4. Objeto 5a62b9D, reservar habitación 101 de 3pm-5pm para miércoles 15 de nov de 2000
5. Si valor de retorno es TRUE, reservación culminada con éxito
6. "Liberar" objeto 5a62b9D

Servicios Necesarios

- El middleware puede otorgar servicios útiles:
 - Naming
 - ¿Dónde puedo encontrar un objeto llamado X?
 - Detección de fallas
 - ¿Qué objeto o servicio ha *muerto*?
 - Comercialización (trading)
 - ¿Dónde puedo encontrar un objeto que haga Y?

Algunas implementaciones de invocaciones remotas

- CORBA – Independiente del lenguaje
- DCOM – Microsoft
- .Net Remoting - .Net
- Java RMI – Java
 - Permite paso de “comportamiento” (código de la clase puede ser pasado por valor si cliente no lo tiene)
- SOAP (Simple Object Access Protocol)
 - Usa HTTP (ya es pedido-respuesta)
 - Representación de datos: XML
- Otros: Apache's Thrift, Avro y Etch, gRPC, ZeroC's ICE

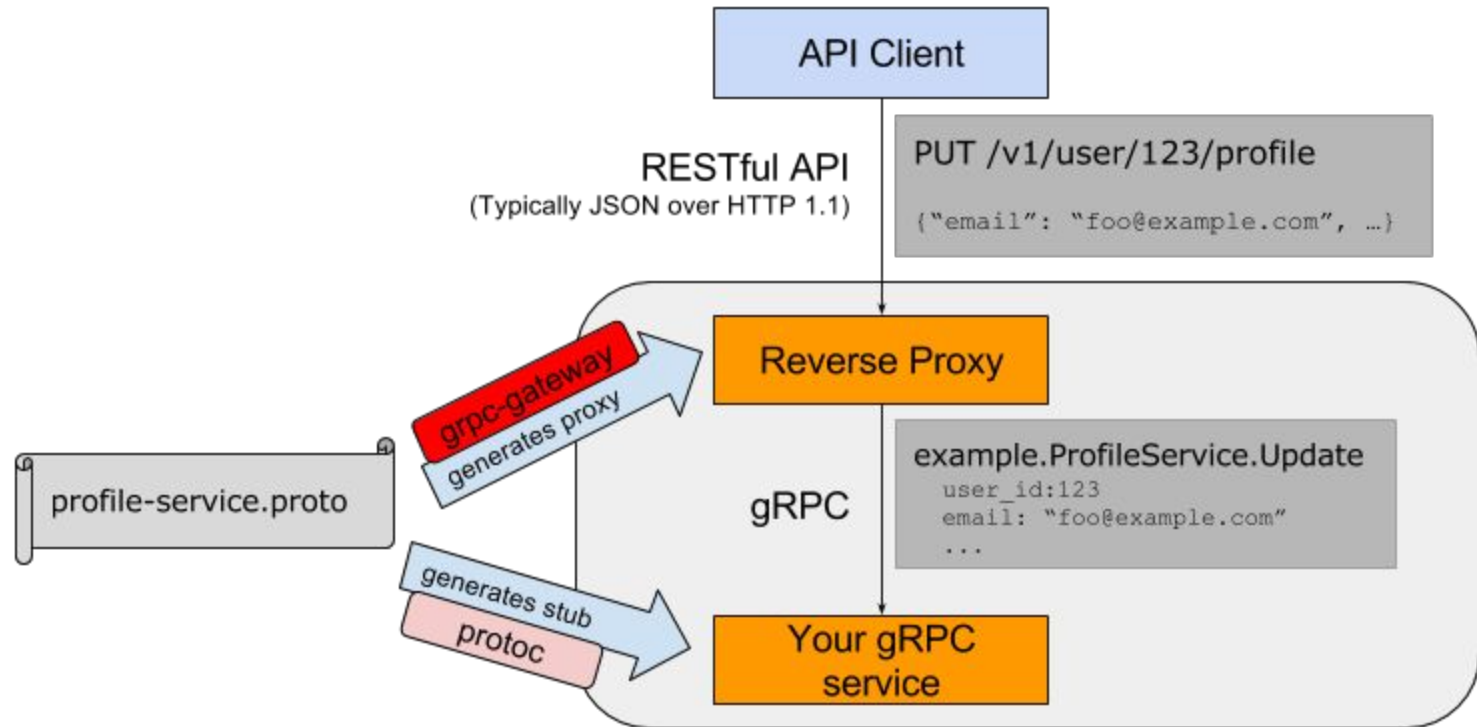
Evolución histórica

- 1970s: Primeras ideas, papers, RFCs
- 1980s: Sun RPC
- 1990s: CORBA, Java RMI, DCOM, .Net Remoting
- 2000s: SOAP
 - REST+JSON puede ser usado para implementar funcionalidad similar
- 2010s (algunos empezaron antes): Apache Thrift (Facebook), Apache Avro (Hadoop), gRPC (Google)

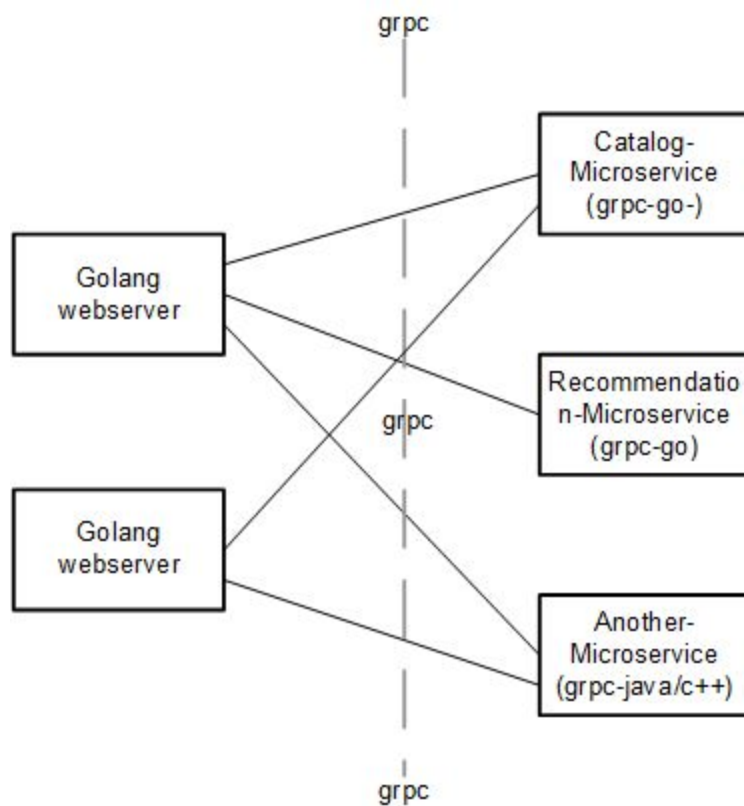
Recomendación actual:  ¡Cambiará en los próximos años! S.D. en constante evolución

- Usar servicios sobre HTTP (REST+JSON) para servicios externos
- Usar gRPC (Protocol Buffers) o Thrift para servicios internos

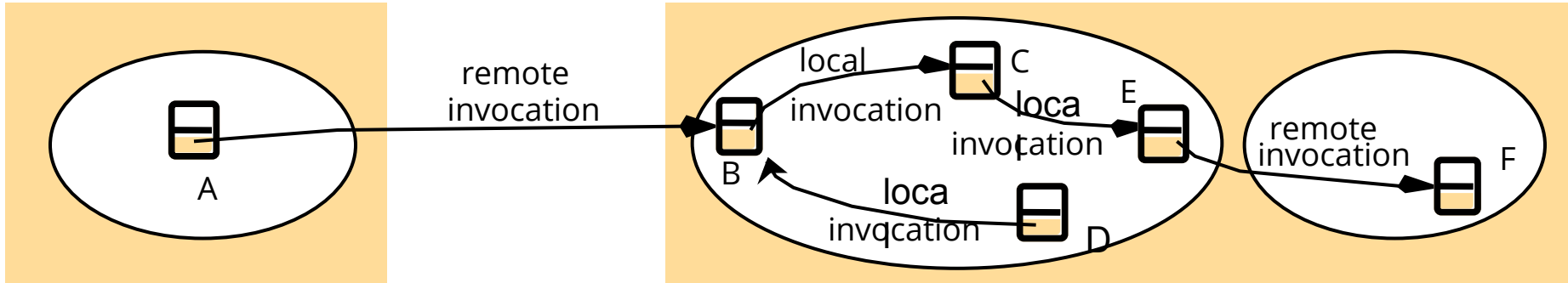
Ejemplo actual



Otro ejemplo actual

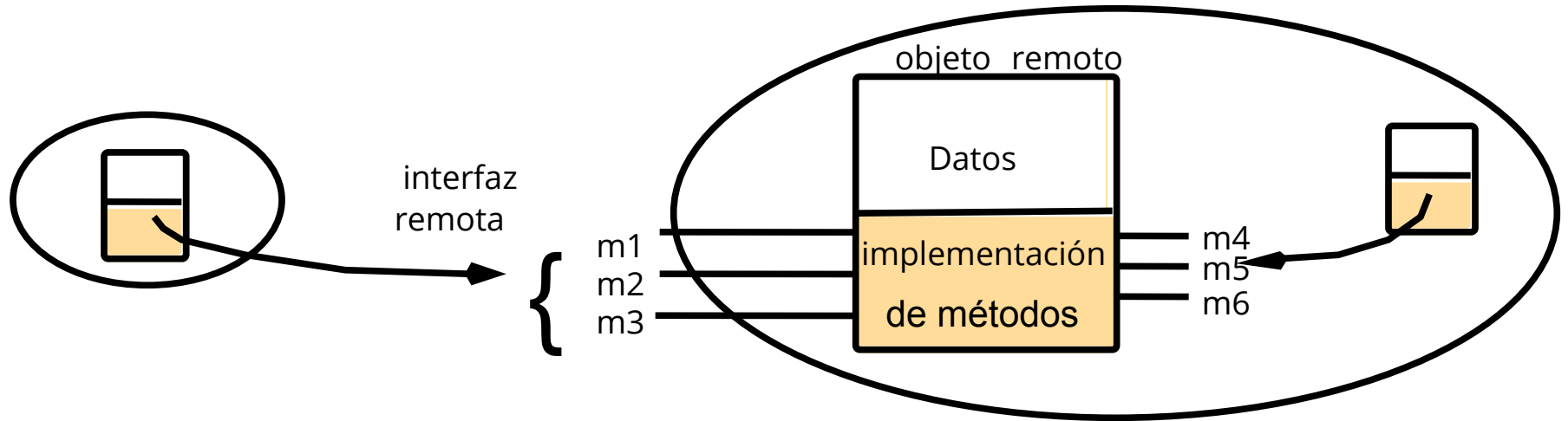


Modelo de Objetos Distribuidos



- Cada proceso contiene objetos, algunos de los cuales pueden recibir invocaciones remotas
- Objetos que pueden recibir invocaciones remotas se denominan *objetos remotos*
- Objetos necesitan saber la *referencia de objeto remoto* a un objeto en otro proceso para poder invocar sus métodos
- *Interfaz remota* especifica qué métodos pueden ser invocados remotamente

Objeto y su Interfaz Remota



Aspectos de Diseño

■ Semánticas de invocación

- Invocaciones locales se ejecutan “exáctamente una vez”
- Semánticas de invocación en sistemas distribuidos difieren según implementación de protocolo pedido-respuesta

■ Transparencia

- ¿Cuánta transparencia debe haber con respecto a invocaciones remotas?
 - Consenso: sintaxis debe ser igual a objetos locales, pero interfaces deben marcar diferencias
 - Ej.: métodos que generan excepciones remotas

Semánticas de Invocación

- Determinadas por implementación de *doOperation*
 - Tratar de re-transmitir mensaje: hasta que llegue respuesta o se asume servidor ha fallado
 - Filtrado de duplicados: si se usan retrans-misiones, filtrar (o no) pedidos duplicados
 - Retransmisión de resultados: se puede mantener historial de resultados para retransmitir resultados sin re-ejecutar operaciones

Semánticas de Invocación Remota

Medidas de Tolerancia a Fallos

Semánticas de invocación

*Retransmitir mensaje
de pedido*

*Filtrado de
duplicados*

*Re-ejecutar procedimiento
o re-transmitir respuesta*

No

N/A

N/A

Tal-vez

Si

No

Re-ejecutar proc.

Al-menos-una-vez

Si

Si

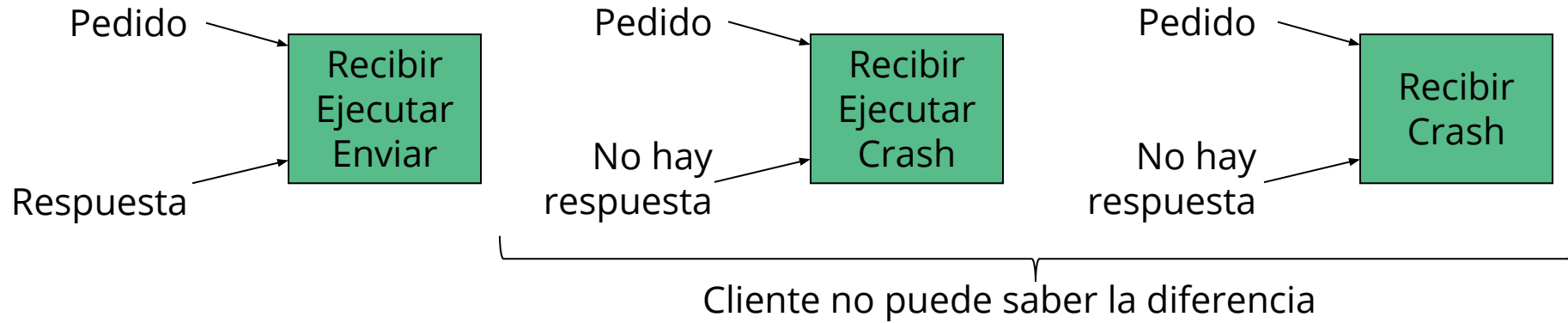
Re-transmitir respuesta

A-lo-mucho-una-vez

Semánticas de Invocación Remota: Modelo de fallas

- Fallas crash (ej.: el servidor remoto falla) afectan a todas
- *Tal-vez* – si no hay respuesta, cliente no sabe si método fue ejecutado o no
 - Fallas de omisión si el pedido o respuesta se pierden
 - Usar en sistemas en que fallas en invocaciones son aceptables
- *Al-menos-una-vez* – el cliente recibe la respuesta (y el método se ejecutó al menos una vez), o una excepción (sin resultado)
 - Fallas arbitrarias. Si el pedido es retransmitido, objeto remoto puede ejecutar método más de una vez, pudiendo causar resultados incorrectos
 - Si se usan operaciones idempotentes, fallas arbitrarias no ocurren
- *A-lo-mucho-una-vez* – el cliente recibe la respuesta (y el método fue ejecutado exactamente una vez), o una excepción (en lugar de la respuesta, en cuyo caso el método se ejecutó una vez o ninguna)

Si el servidor se cuelga (crash)...

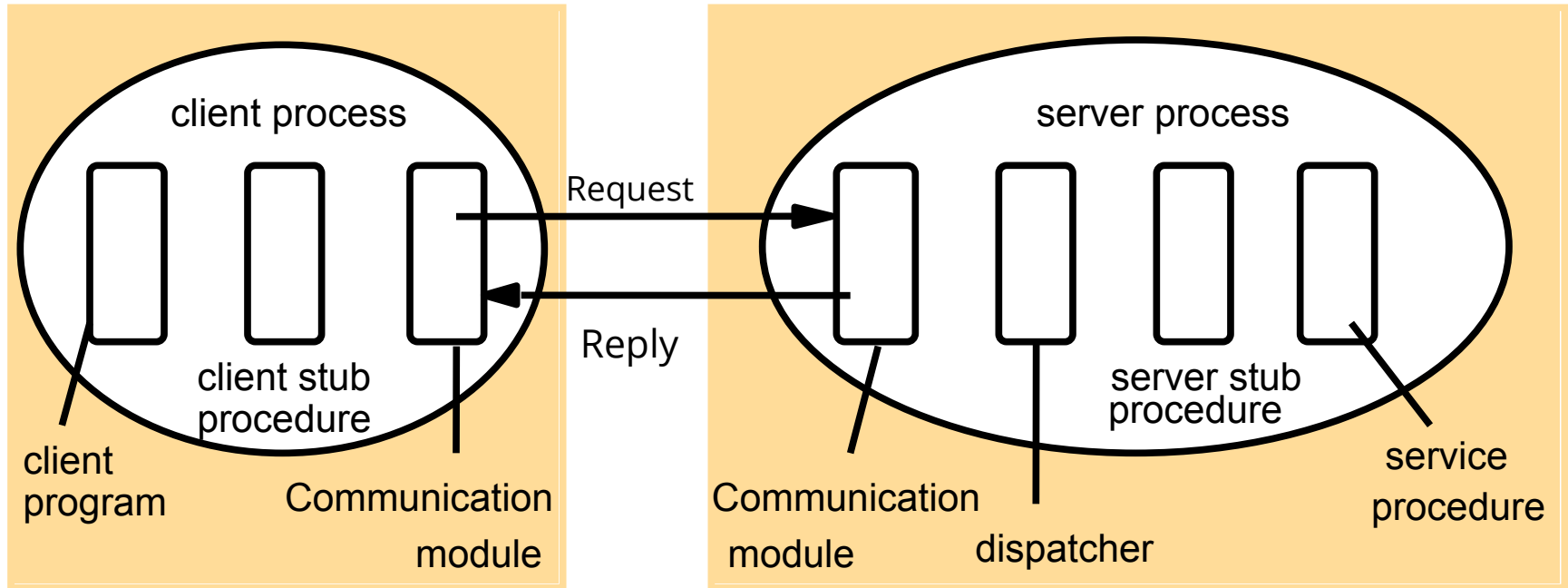


- Al-menos-una-vez: se sigue tratando hasta que se recibe respuesta del servidor
- A-lo-mucho-una-vez: no se vuelve a tratar
- Exactamente-una-vez: imposible en sistemas distribuidos

Semánticas de Invocación Remota

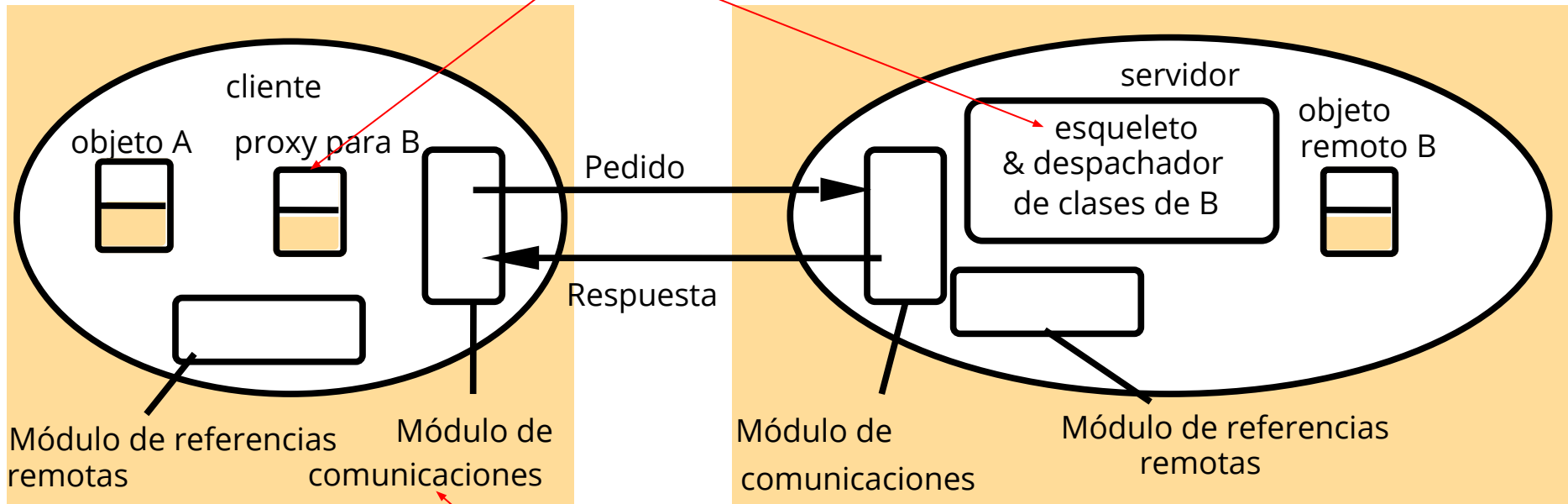
- Diferentes implementaciones → diferentes semánticas de invocación
- Java RMI → *a-lo-mucho-una-vez*
- CORBA → *a-lo-mucho-una-vez*
 - Se puede usar *tal-vez*, bajo pedido, para métodos que no retornan resultados
- gRPC → *a-lo-mucho-una-vez*
 - Ref: <https://github.com/grpc/grpc-go/issues/238>
- **Excepción:** Sun RPC → *al-menos-una-vez*

Implementación de RPC



Implementación de RMI

Marshalling y unmarshalling



Implementa RR; da las
semánticas de invocación

Programación con (Java) RMI

- ¿Cómo se puede instanciar objetos remotos?
- Enlazador (binder) → mantiene una tabla con mapeos entre nombres textuales y referencias a objetos remotos (ej.: RMI registry en puerto 1099)
- Middleware crea un nuevo hilo para cada invocación remota
- Objetos remotos deben ser recolectados y memoria liberada cuando no hay referencias a ellos
 - Solución: Conteo de referencias
 - Nuevo problema: Hacer que recolección de basura sea tolerante a fallas remotas
 - ¿Solución?
- Objetos que se pasan/retornan deben ser *Remote* o *Serializable*
 - Argumentos de métodos son parámetros de entrada
 - Retorno de método: parámetro de salida
- Implementaciones (ej.: Java) puede pasar código (.class) de ser necesario
 - ¿Problema? ¿Solución?

Transparencia

- Sintaxis igual a invocación local
`referenciaRemota.metodo()`
- Pero no todo es transparente:
 - Objeto local sabe que un objeto es remoto y debe manejar *RemoteException*
 - Quien implementa un objeto que va a ser usado remotamente debe extender la interfaz *Remote*

Ejemplo: Pizarrón distribuido

- Usuarios pueden compartir una vista común de un área de dibujo donde pueden añadir objetos gráficos
- Servidor mantiene el estado actual del dibujo
- Clientes informan a servidor sobre cada figura añadida al dibujo
- Clientes puede preguntar a servidor qué otras figuras han sido añadidas
- Cada figura tiene un número de versión

Interfaces Remotas

```
public interface Shape extends Remote {  
    int getVersion() throws RemoteException;  
    GraphicalObject getAllState()  
        throws RemoteException;  
}
```

```
public interface ShapeList extends Remote {  
    Shape newShape(GraphicalObject g)  
        throws RemoteException;  
    Vector allShapes() throws RemoteException;  
    int getVersion() throws RemoteException;  
}
```

Métodos de la Naming (RMI Registry)

void rebind (String name, Remote obj)

Usado por un servidor para registrar el identificador a un objeto remoto en base a su nombre.

void bind (String name, Remote obj)

Alternativamente, utilizado por servidor para registrar un objeto remoto, pero si su nombre ya se encuentra asociado a una referencia remota, arroja una excepción.

void unbind (String name, Remote obj)

Elimina un enlace nombre-referencia.

Remote lookup (String name)

Usado por clientes para buscar un objeto remoto basado en su nombre.

Retorna una referencia a un objeto remoto.

String [] list()

Retorna un arreglo de Strings con todos los nombres de objetos enlazados en el registro.

Programa Servidor

```
import java.rmi.*;

public class ShapeListServer
{
    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();
            Naming.rebind("Shape List", aShapeList );
            System.out.println("ShapeList server ready");
        } catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());
        }
    }
}
```

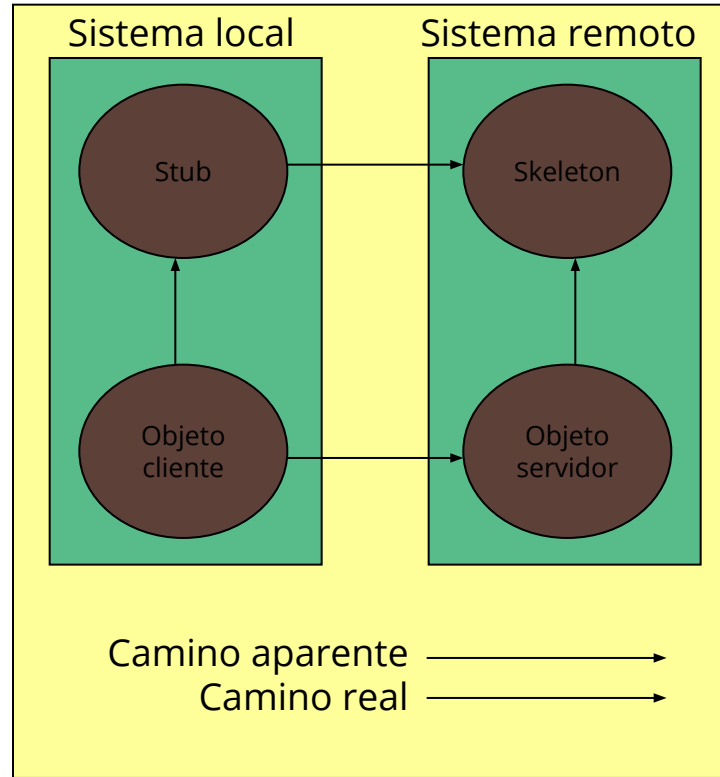
```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant
    extends UnicastRemoteObject
    implements ShapeList
{
    private Vector theList;
    private int version;
    public ShapeListServant() throws RemoteException {...}
    public Shape newShape(GraphicalObject g)           // Método fábrica
        throws RemoteException
    {
        version++;
        Shape s = new ShapeServant(g, version);
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() throws RemoteException {...}
    public int getVersion() throws RemoteException { ... }
}
```

Programa Cliente

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager() );
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno/ShapeList");
            Vector sList = aShapeList.allShapes();
        } catch(RemoteException e) {
            System.out.println(e.getMessage());
        } catch(Exception e) {
            System.out.println("Client: " + e.getMessage());
        }
    }
}
```

Flujo de una Llamada Remota



HTTP fallback – si puertos usados por *rmi* están bloqueados, se puede utilizar HTTP tunneling

RPCs modernos: Apache Thrift

Ejemplo “Hello World” en:

<https://thedulinreport.com/2013/02/24/hello-world-using-apache-thrift/>



Comunicación indirecta

Capítulo 6 del libro de Coulouris, 5ta edición





Comunicación indirecta

- Desliga a clientes/servidores en el espacio y en el tiempo
- Algunos ejemplos:
 - Comunicación en grupos ← Ej.: JGroupsToolkit; no lo vamos a estudiar
 - Publish-subscribe
 - Colas de mensajes
 - Memoria compartida ←No lo vamos a estudiar
 - Tuple spaces ←Ej.: JavaSpaces; no lo vamos a estudiar

Table 1. Decoupling Abilities of Interaction Paradigms

Abstraction	Space decoupling	Time decoupling	Synchronization decoupling
Message passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (observer pattern)	No	No	Yes
Tuple spaces	Yes	Yes	Producer-side
Message queuing (Pull)	Yes	Yes	Producer-side
Publish/subscribe	Yes	Yes	Yes



Publish-subscribe

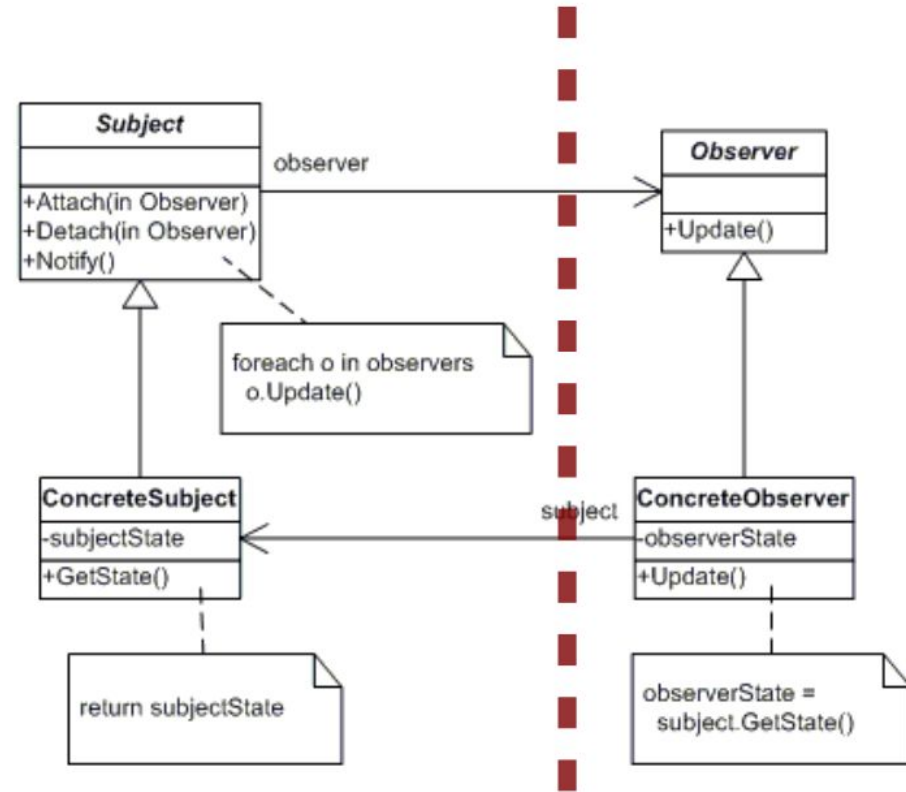
Eventos y Notificaciones Distribuidas

- Notificaciones de eventos son asíncronas
- Dos patrones de diseño:
 - Observer ← No es comunicación indirecta
 - Publish-subscribe ← Este estudiaremos más en detalle
- Paradigma *publicar-suscribir* (*publish-subscribe*)
 - Objetos generan eventos
 - Objetos *publican* tipos de eventos disponibles
 - Objetos se *suscriben* a los tipos de eventos que les interesan (recibir notificaciones)

Observer pattern

- Típicamente implementado con *callbacks*
- Puede ser implementado con RMI
- Algunos RPC/RMI lo implementan explícitamente
- Más información del patrón de diseño en:

<http://circe.di.unipi.it/~gervasi/DSL10/DSDL-07.pdf>



Terminología

- Notificaciones: Mensaje que indica que evento ocurrió
 - Pueden ser almacenadas, enviadas en mensajes, consultadas y aplicados
- Suscribirse a un tipo de evento también es llamado *registrar interés* en el evento
- Otros nombres comunes:
 - Pubsub
 - Message broker

Más info. en:

https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

Ejemplos de Usos de Eventos

- Comunicar que:
 - Una forma se ha añadido a un dibujo
 - Un documento ha sido modificado
 - Una persona ha entrado o salido de un cuarto
 - Computación ubicua (Pervasive/Ubiquitous Computing)
 - Un equipo ha sido reubicado
 - Un libro (con un dispositivo especial) ha cambiado de ubicación

Características de Sistemas Basados en Eventos

■ Heterogeneidad

- Uso de eventos lleva a interoperabilidad de componentes que no fueron diseñados para cooperar entre sí

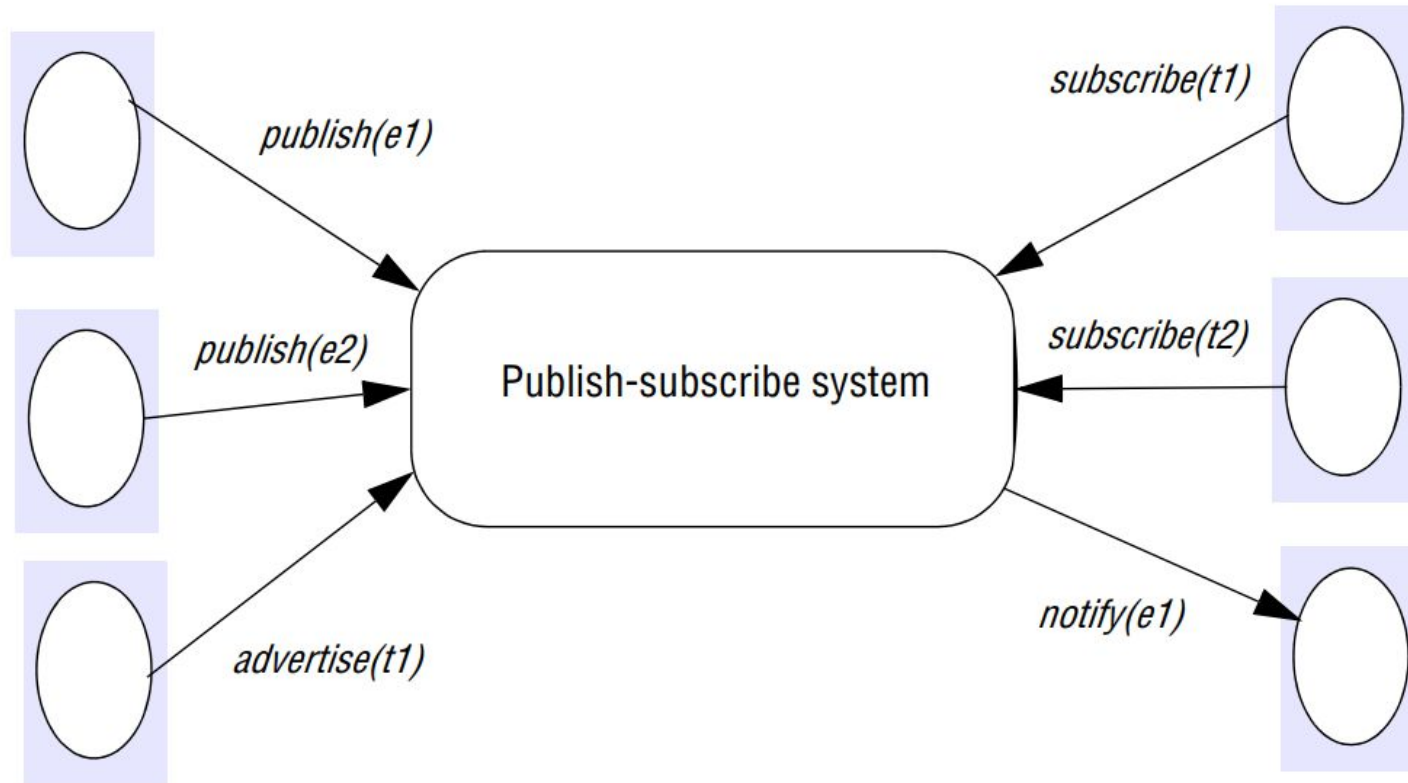
■ Asincronía

- Notificaciones se envían a suscriptores de manera asíncrona → se evita sincronización entre publicadores y suscriptores

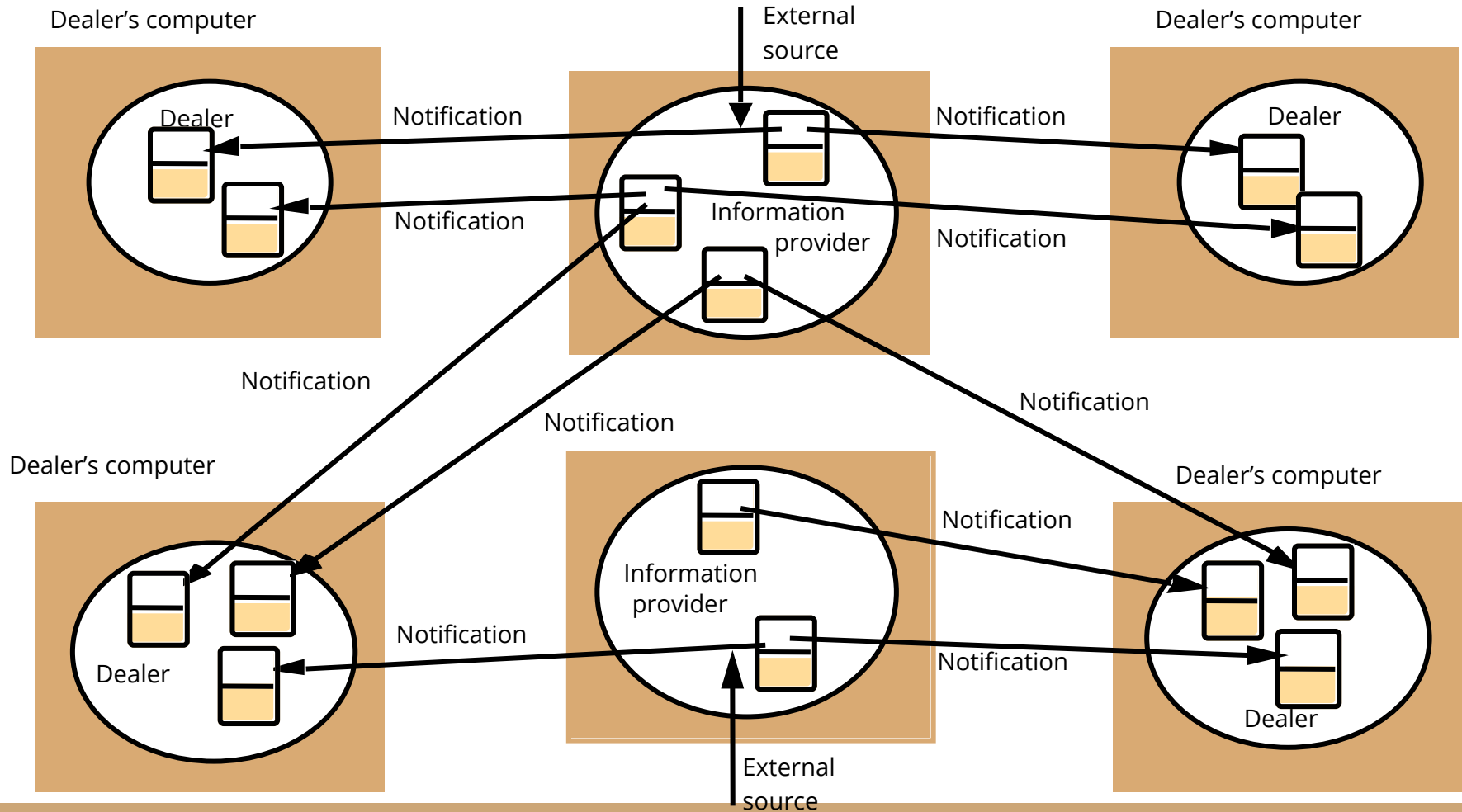
Modelo de comunicaciones

Publishers

Subscribers



Ej.: Sistema de Mercado de Valores



Tipos de Eventos

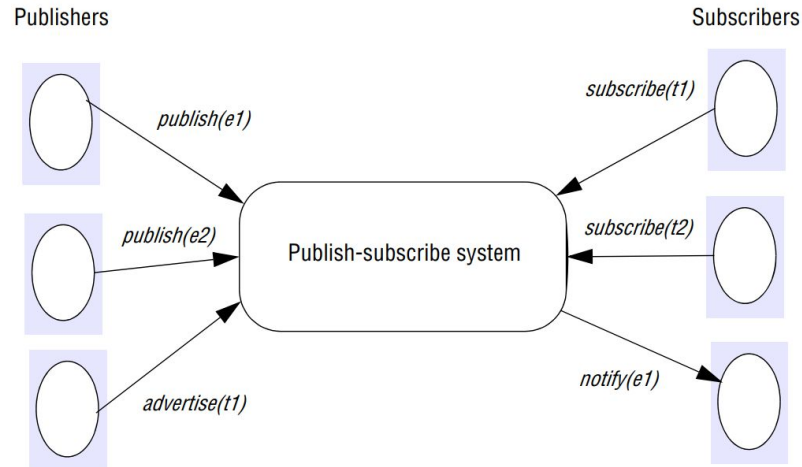
- Eventos tienen atributos que especifican información sobre ese evento. Ej.:
 - Nombre
 - ID de objeto que lo generó
 - Operación
 - Parámetros
 - Estampa de tiempo
- En ciertos sistemas es posible suscribirse a tipos de eventos, pero siempre y cuando cumplan cierto criterio (especificado en atributos)

Semánticas de Entrega

- Garantías de entrega dependen de implementación y de requerimientos de aplicación
- Implementaciones con IP multicast son de *mejor esfuerzo* y no garantizan entrega
 - No apropiado para ciertas aplicaciones
- Requerimientos pueden ser de *tiempo real*
- Alternativas:
 - Servicios centralizados
 - Servidores múltiples
 - Multicast en capa de aplicación

Tipos de implementación del middleware pubsub

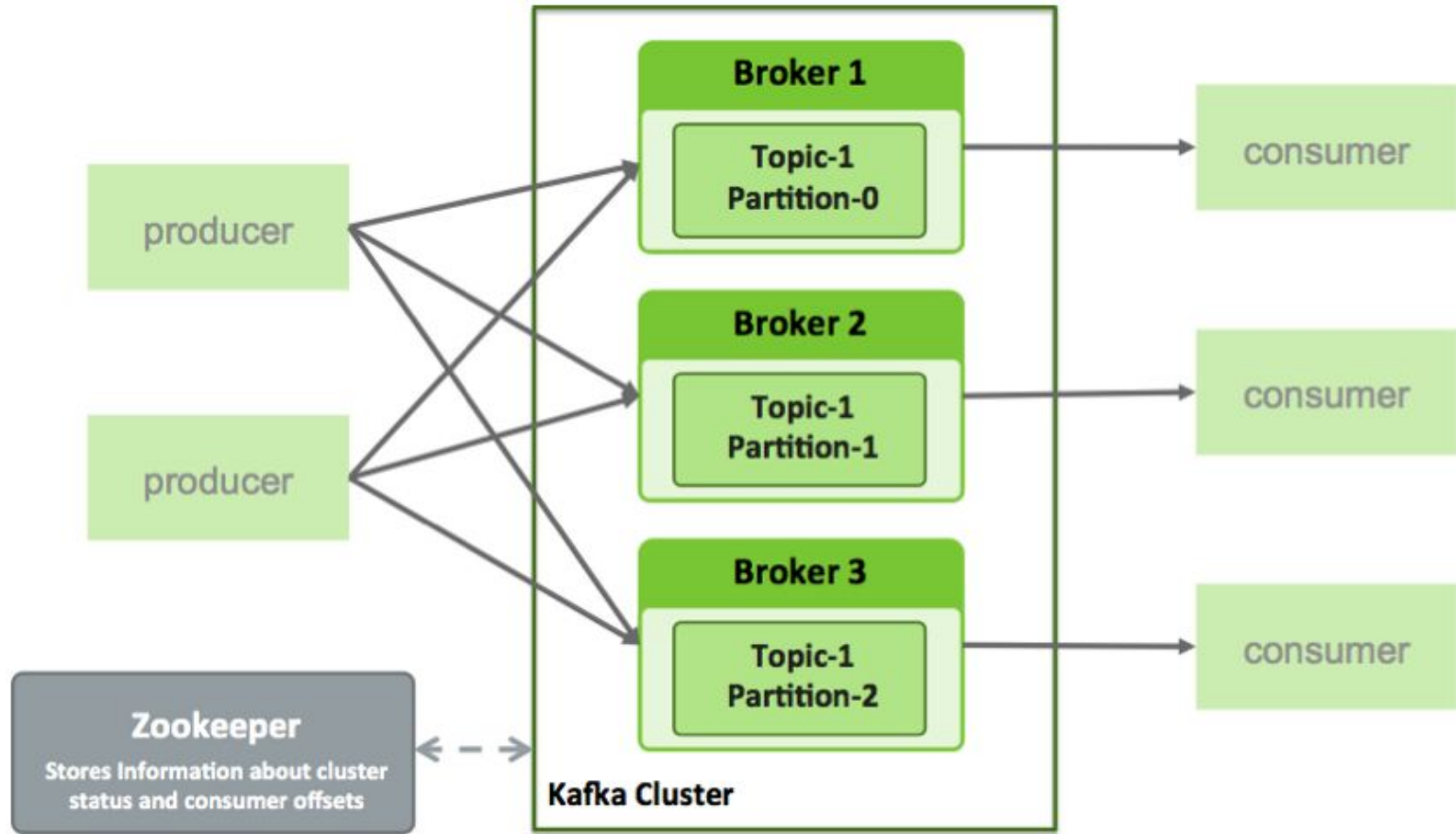
- Servicios centralizados
- Servidores múltiples
- Multicast en capa de aplicación



Implementaciones publish/subscribe


- Redis Pub/Sub → <http://redis.io/topics/pubsub>
- ZeroMQ → <http://zeromq.org/>
 - En realidad, es un middleware que implementa Observer pattern
 - No hay decoupling entre subscriptores y divulgadores (?)
- RabbitMQ → <http://www.rabbitmq.com/>
- Apache ActiveMQ → <http://activemq.apache.org/>
- Apache Apollo → <https://activemq.apache.org/apollo/> (ActiveMQ next gen)
- Internet-wide pubsub:
 - RSS, Atom, PubSubHubbub, Twitter!
 - Google's Cloud PUB/SUB → <https://cloud.google.com/pubsub/>
 - Amazon's Simple Notification Service (SNS) → <https://aws.amazon.com/sns/>
 - Apache Kafka → Pub/sub masivamente escalable; originado en LinkedIn; usado por Netflix, Yahoo, Paypal, Spotify, Uber,

Ejemplo: Apache Kafka




Ejemplo: Apache Kafka Quickstart

<https://kafka.apache.org/quickstart>



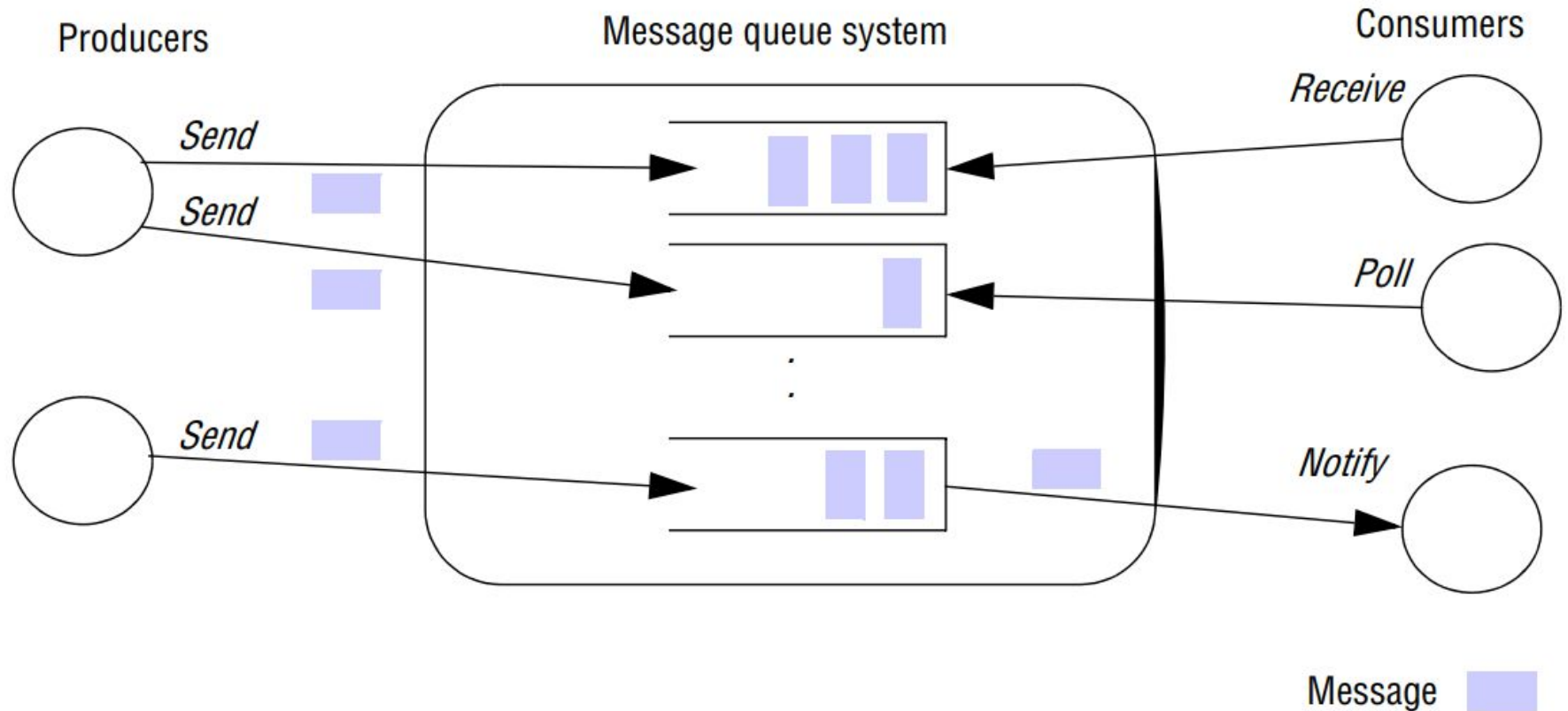
Message Queueing



Colas de mensajes (distribuidas)

- Publish/subscribe → uno-a-muchos
- Colas de mensajes → punto-a-punto
 - Un mensaje es recibido por un solo consumidor
- Tres tipos de “receives”
 - Con bloqueo
 - Sin bloqueo (poll)
 - Notificación
- La cola es típicamente FIFO
 - Muchas implementaciones adicionalmente soportan prioridades
- A veces se permite que consumidor seleccione si recibir un mensaje o no, en base a sus propiedades

Modelo de comunicaciones



Implementaciones de MQ

- ZeroMQ → <http://zeromq.org/>
- RabbitMQ → <http://www.rabbitmq.com/>
- Apache ActiveMQ → <http://activemq.apache.org/>
- Apache Apollo → <https://activemq.apache.org/apollo/>
 - ActiveMQ next gen
- Apache Qpid → <https://qpid.apache.org/>
- En la nube:
 - Amazon Simple Queue Service (SQS) → <http://aws.amazon.com/sqs/>
 - Windows Azure Service Bus → <https://azure.microsoft.com/en-us/services/service-bus/>
 - Google's Cloud PUB/SUB → <https://cloud.google.com/pubsub/>

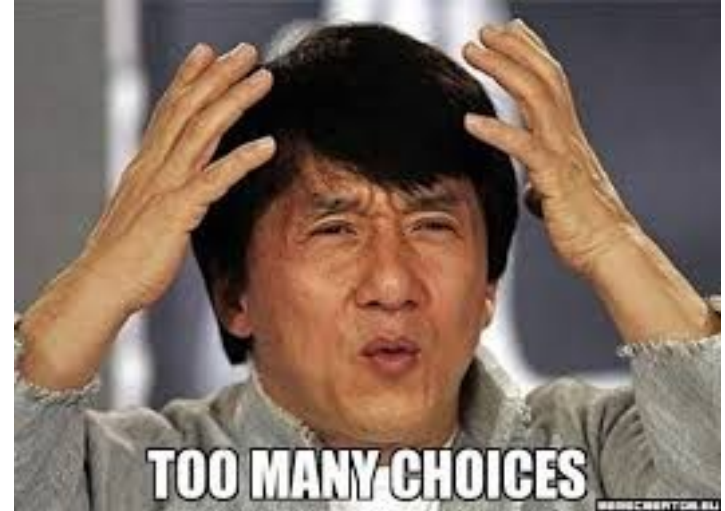
Google sugiere que si se desea una cola de mensajes, se debe usar el cloud pub/sub y enviar/recibir mensajes usando un tópico único y suscripción uno-a-uno (en lugar de uno a muchos)

Message Oriented Middlewares (MOMs)

- Tanto pub/sub como las colas de mensajes son MOMs
- Estándares MOM:
 - Advanced Message Queuing Protocol (AMQP) → Para los dos paradigmas (y otros más, como pedido-respuesta)
 - SwiftMQ, Apache's Qpid, ActiveMQ, Apollo
 - Data Distribution Service (DDS) → Pub/sub
 - OpenDDS → <http://opendds.org/>
 - eXtensible Messaging and Presence Protocol (XMPP) → Pub/sub
 - Originalmente: Jabber
 - Estándar en sistemas de mensajería: WhatsApp, Gtalk, PlayStation chats, etc.
 - MQTT → Pub/sub ultra ligero, para IoT
 - Apache ActiveMQ y Apollo
 - Java Messaging Service (JMS) → prod/cons (MQ) y pub/sub
 - Amazon SQS, Apache ActiveMQ y Qpid, RabbitMQ, SwiftMQ
 - STOMP → Streaming Text
 - ActiveMQ, Apollo, HornetQ, OpenMQ, RabbitMQ

¡Demasiadas opciones! ¿Cuál escoger?

- Apache ActiveMQ o RabbitMQ
 - Productos maduros
 - Muchas opciones
- Apache Apollo
 - Escalable y rápido
- Apache Kafka
 - Ultra-escalable
 - Por eso lo usan muchas compañías de servicios de Internet
- Opciones de Amazon/Google/Azure en una nube pública
 - PRO: No hay que instalar/mantener/configurar el middleware
 - CON: Difícil migrar de una nube a otra

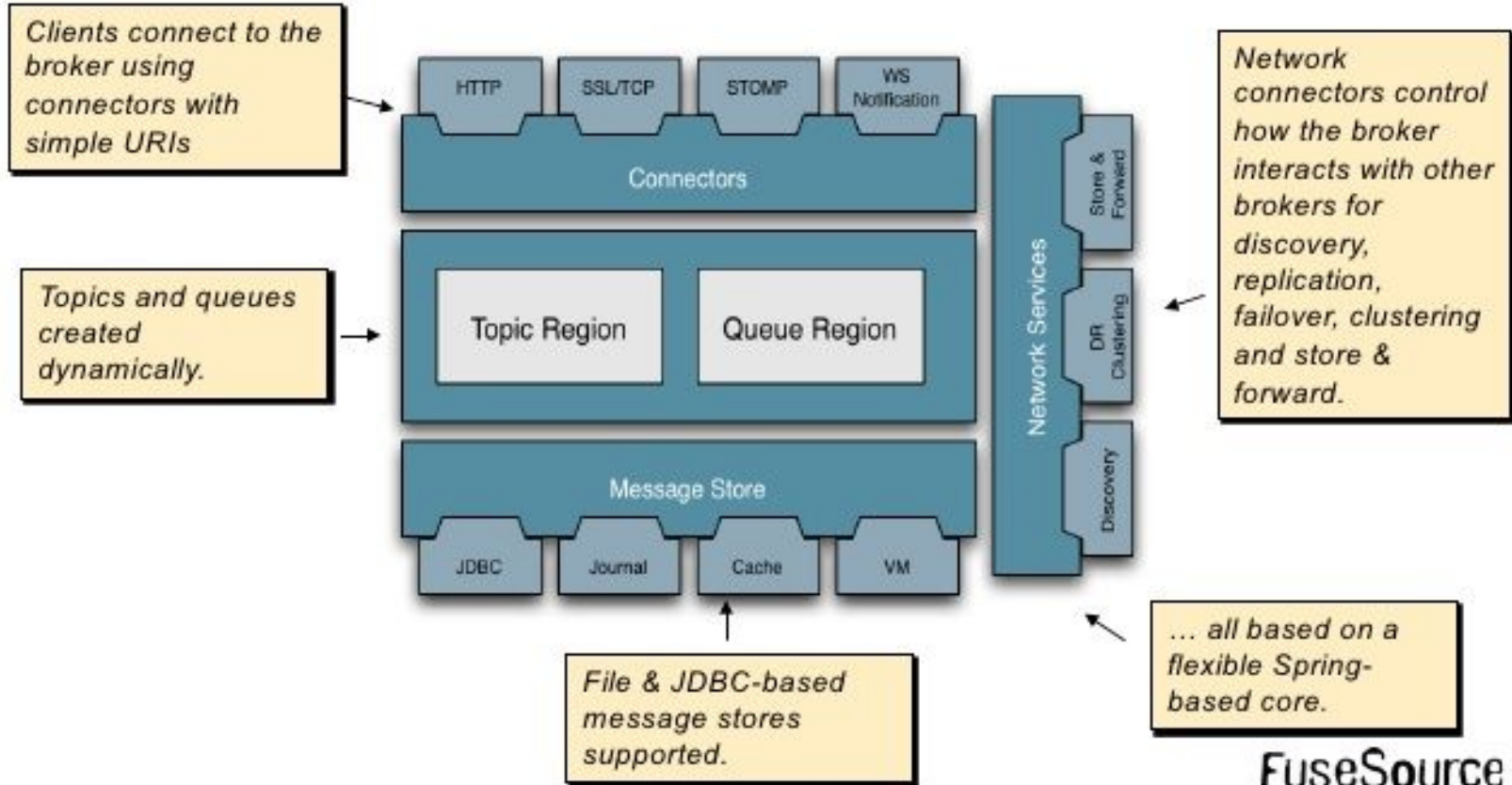


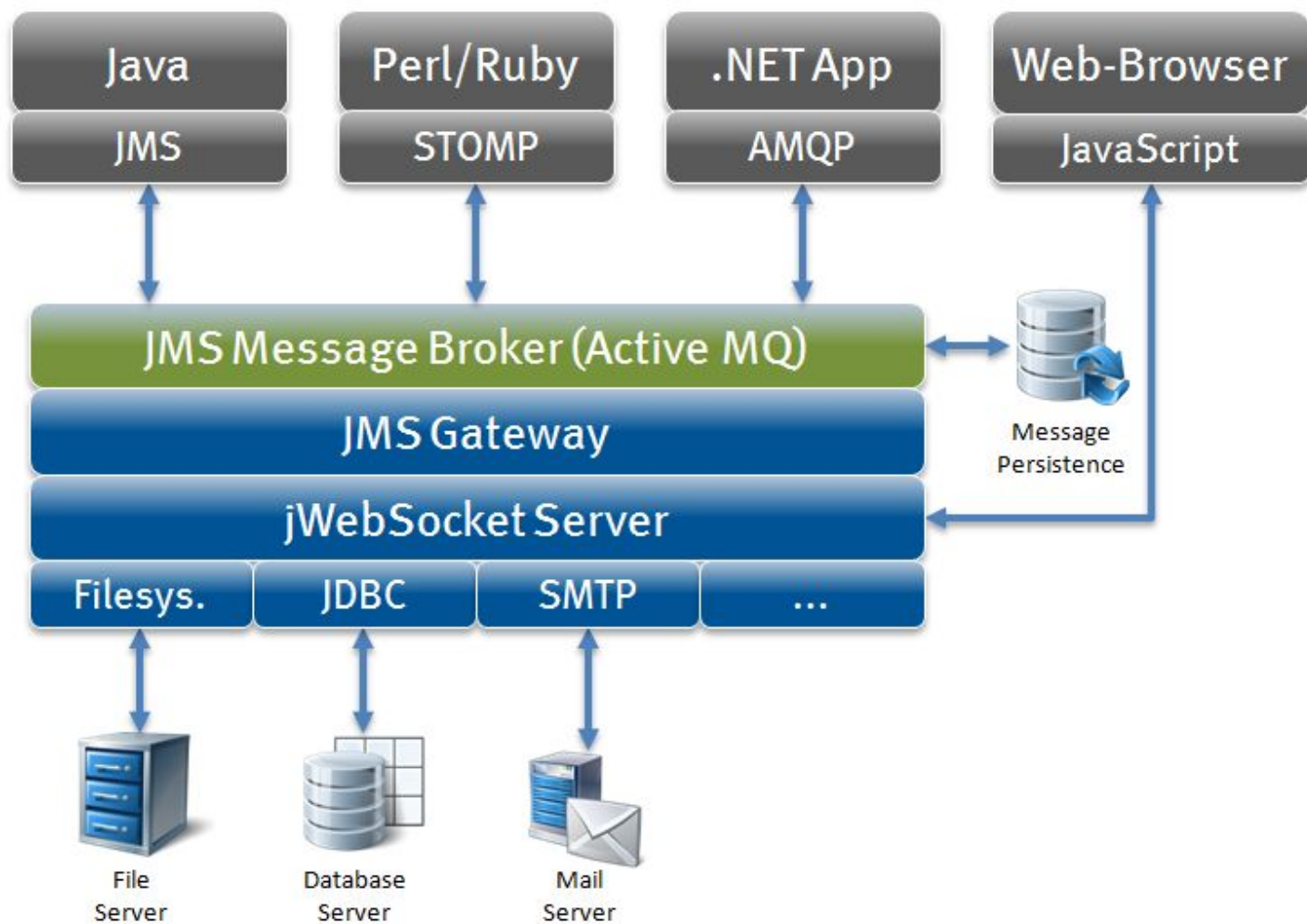
Enlaces a otras opciones en: <http://queues.io/>

Algunas comparaciones en:

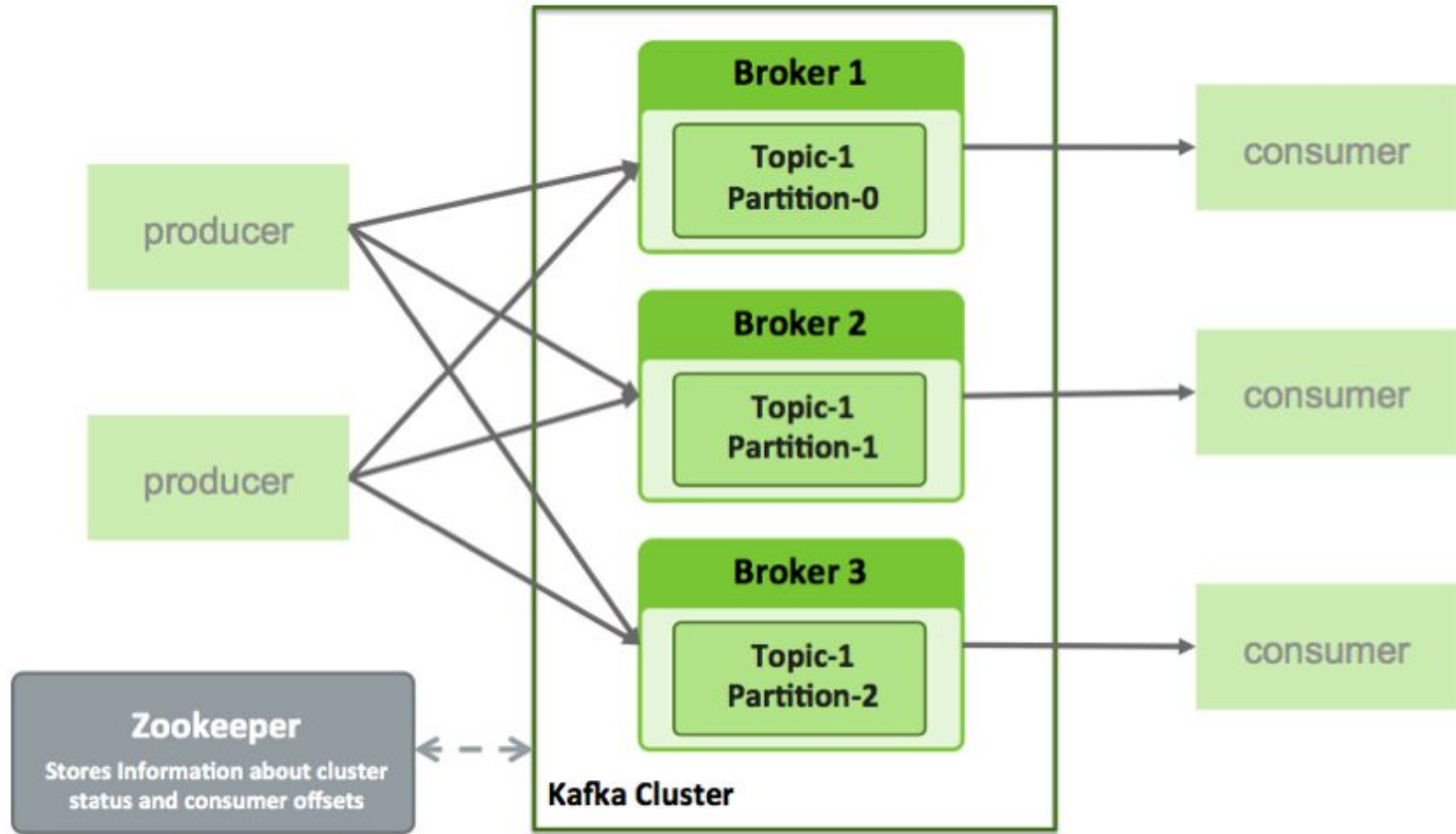
<http://www.predic8.com/activemq-hornetq-rabbitmq-apollo-qpid-comparison.htm>

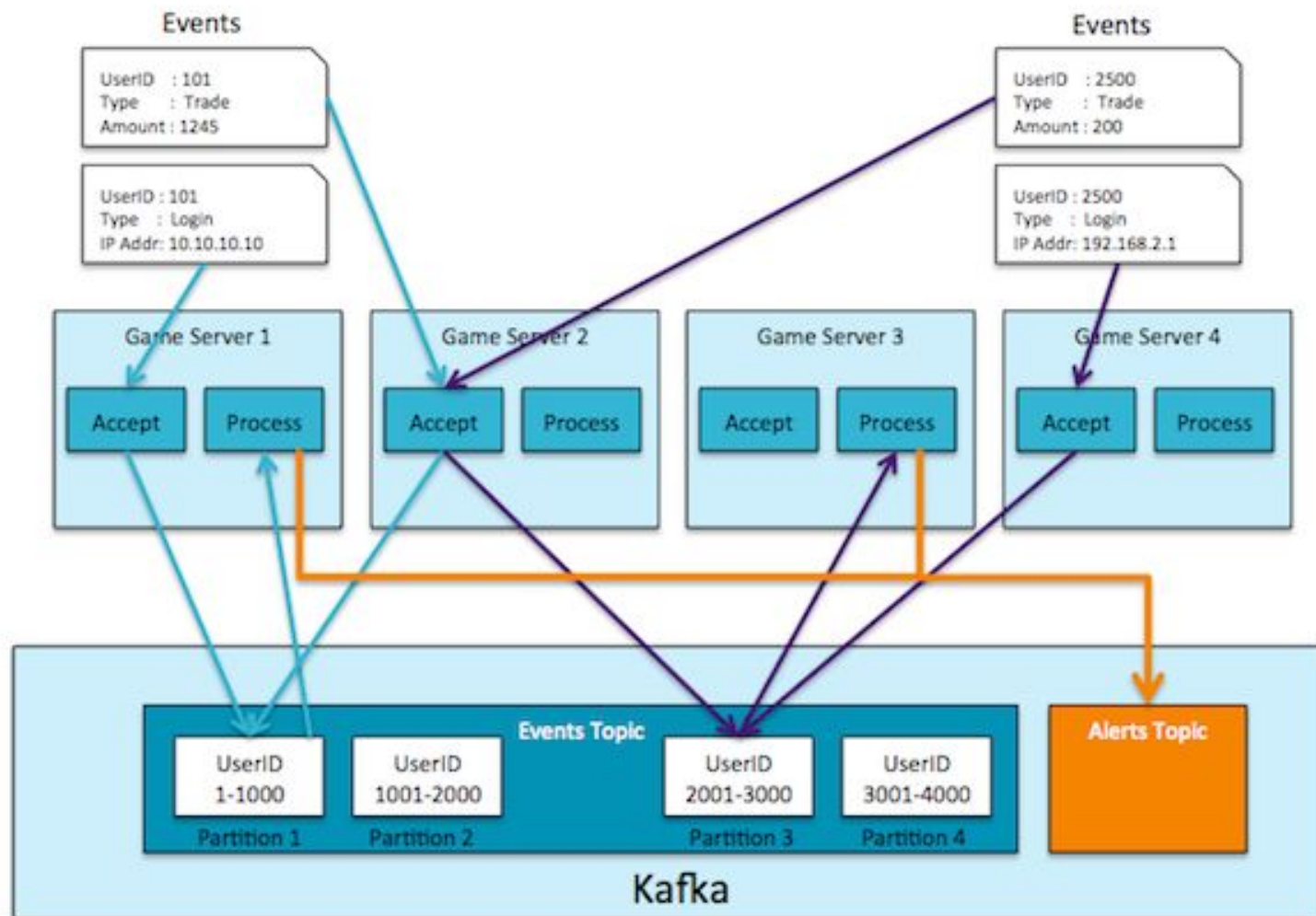
Ejemplo: Apache ActiveMQ





Ejemplo: Apache Kafka





Resumen

Figure 6.27 Summary of indirect communication styles

	<i>Groups</i>	<i>Publish-subscribe systems</i>	<i>Message queues</i>	<i>DSM</i>	<i>Tuple spaces</i>
<i>Space-uncoupled</i>	Yes	Yes	Yes	Yes	Yes
<i>Time-uncoupled</i>	Possible	Possible	Yes	Yes	Yes
<i>Style of service</i>	Communication-based	Communication-based	Communication-based	State-based	State-based
<i>Communication pattern</i>	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
<i>Main intent</i>	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
<i>Scalability</i>	Limited	Possible	Possible	Limited	Limited
<i>Associative</i>	No	Content-based publish-subscribe only	No	No	Yes



Hasta aquí deben estudiar para el examen





Información adicional de respaldo (backup slides)



¿Por qué distribuir objetos?

- Modelo abstracto
- Tolerancia a fallos
- Escalabilidad
- Disponibilidad
- Rendimiento

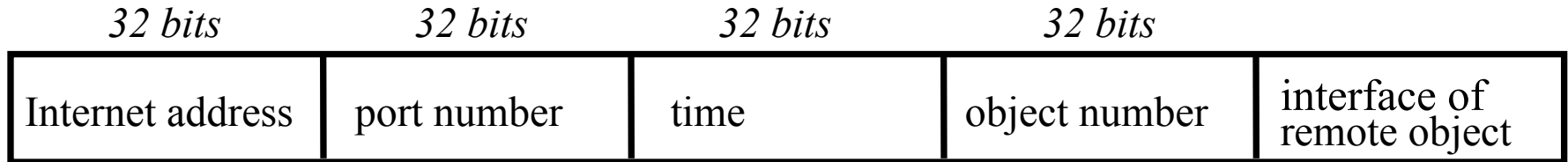
Problemas con Objetos Distribuidos

- Modelo abstracto
- Rendimiento
- Falla parcial
- Sincronización

Invocación de objetos remotos

- Se necesita una referencia al objeto
 - Debe ser única, aún con el paso del tiempo
 - No deben ser re-utilizadas

Ejemplo:



Importancia de interfaces en RMI/RPC

- IDL → Lenguaje de definición de interfaces
- Diseños modulares requieren interfaces
- Interfaces definen posibles interacciones entre módulos
- Encapsulamiento
- Uso de interfaces permite cambiar implementaciones sin cambiar interacciones
- Interfaces en sistemas distribuidos difieren de tradicionales:
 - Acceso a variables no puede ser directo
 - Paso de punteros no es posible (paso de referencias OK)

Módulo de Referencias Remotas

- Realiza conversión entre referencias locales y remotas
- Usa una tabla de objetos remotos
 - Mantiene correspondencia entre referencias a objetos locales y referencias a objetos remotos
- Llamado cuando se recibe una llamada remota, para saber a qué referencia local la referencia remota se refiere

Componentes del Software RMI

■ Proxy:

- Hace RMI transparente al cliente
- Implementa interfaz remota
- Marshalling y unmarshalling
- Recibe pedidos y los reenvía a proceso remoto

■ Despachador:

- Recibe pedidos de módulo de comunicaciones
- Invoca método en esqueleto

■ Esqueleto:

- Implementa métodos en interfaz remota
- Unmarshalling y marshalling
- Invoca método en objeto remoto.

Implementación: Otros Detalles

- Clases proxies, despachadoras y esqueletos: generadas por compilador de interfaces
- Programa servidor
 - Despachadores, esqueletos e implementaciones de clases de objetos remotos
 - Sección de inicialización: crea e inicializa al menos un objeto remoto (generalmente usado para acceder otros objetos remotos)
 - Registra uno o más obj. remotos con *enlazador*

Implementación: Otros Detalles

■ Programa cliente

- Clases proxies de objetos remotos a invocar
- Usa enlazador para buscar referencias a objetos remotos
- Puede crear objetos remotos usando métodos tipo *fábrica* (no hay constructores remotos)

■ Enlazador (binder): servicio que mantiene una tabla con mapeos entre nombres textuales y referencias a objetos remotos

- CORBA: Naming service
- Java: RMI registry

Implementación: Otros Detalles

- Hilos del servidor (threads)
 - Nuevo hilo para c/ invocación remota u objeto
 - Evitan retrasos en invocaciones concurrentes
 - Tener en cuenta efectos de concurrencia
- Para Δ eficiencia, procesos y obj. remotos pueden activarse cuando van a ser usados
 - Obj. pasivo: no en memoria. Se activa con implementación de métodos y estado (marshalled)
 - CORBA: *activador* \rightarrow *repositorio de implementaciones*

Implementación: Otros Detalles

- Almacenamiento persistente de objetos
 - Objeto persistente: garantizado que existe entre activaciones de procesos
 - Bodega de objetos persistentes: almacena los objetos en disco
 - CORBA: servicio de objetos persistentes
 - Java: Java persistente
 - Dependiendo de implementación, pueden activarse en cliente (usando caché y protocolo de consistencia)
 - Optimización: almacenar únicamente objetos que han cambiado su estado

Ubicación de Objetos

32 <i>bits</i>	32 <i>bits</i>	32 <i>bits</i>	32 <i>bits</i>	
Dirección IP	Puerto	Tiempo	Número de Obj.	Interfaz objeto remoto

- Referencia a objetos remotos debe ser única en un SD. No debe ser reutilizada si objeto ha sido eliminado.
- IP + Puerto permiten localizar objeto a menos que haya migrado o sido re-activado en un nuevo proceso
- Cuarto campo identifica a objeto dentro del proceso
- Interfaz: indica a receptor qué métodos tiene
- Pero, hay objetos que pueden cambiar de ubicación
 - Solución: usar un servicio de localización
 - Base de datos con posible ubicación de objetos

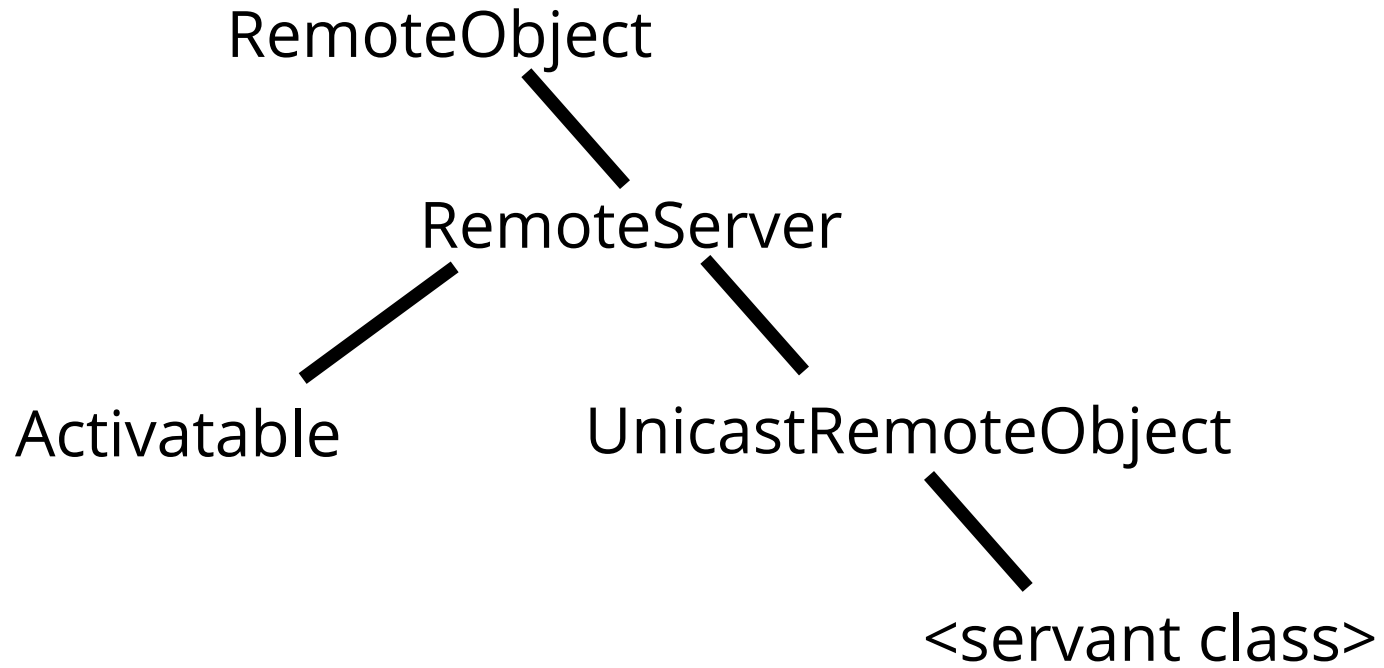
Recolección de Basura

- Objetos deben existir si referencias locales o **remotas** existen
- Objetos deben ser recolectados y memoria liberada al momento que no hay referencias a ellos
- Solución: Usar conteo de referencias
- Proxies se crean para usar referencias remotas
- Servidor debe ser informado cuando un nuevo proxy es creado y cuando uno termina

Recolector de Basura Distribuido

- Coopera con recolector de basura local
- Java
 - Tolera fallas de procesos clientes usando *préstamos*
 - Servidores *prestan* objetos (referencias) a clientes por un tiempo determinado. Clientes renuevan préstamos antes de que el tiempo expire.

Clases que dan Soporte a RMI



Java RMI: Paso de Parámetros y Resultados

- Serializando objetos:
 1. Objetos *Remote*: se reemplaza por su referencia remota, la cual contiene el nombre de su clase
 2. Otros objetos: Java los serializa e incluye una dirección para ubicar su clase (un URL), para que pueda ser bajada por el receptor de ser necesario

Java RMI: Bajando Clases

- Si el destino no contiene la clase del objeto que recibe, se baja el código de la clase automáticamente
- De igual manera, si quien recibe una referencia remota no tiene la clase proxy, se baja el código automáticamente

Ejemplo

- Supongamos que *GraphicalObject* no soporta texto
- Cliente puede crear una subclase que soporte texto y mandarla como parámetro de *newShape*
- Cuando otros clientes se enteren de la existencia de esa nueva figura, se bajarán el código de la clase automáticamente

Java RMI: Seguridades

- RMI Security.policy

```
grant{  
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";  
}
```

- Cargador de clases usado por RMI no se baja ninguna clase de localidades remotas a menos que haya un *SecurityManager* instalado

```
System.setProperty("java.security.policy", "RMI Security.policy");
```

- *UnicastRemoteObject*: para objetos remotos que viven mientras vive su proceso padre
- *newShape*: método fábrica, crea objetos remotos
- *RMISeurityManager*: manejador de seguridades por defecto
- Si no se proporciona un manejador de \seguridades, proxies y otras clases solo pueden ser cargados del *CLASSPATH* local (no bajadas de otros procesos)