02 - Arquitecturas de Sistemas Distribuidos

IIC2523 - Sistemas Distribuidos

Cristian Ruz – cruz@ing.puc.cl

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Semestre 2-2020

Contenidos

1. Tipos de arquitecturas
   - Arquitecturas por capas (*layered*)
   - Arquitecturas basadas en objetos/servicios
   - Arquitecturas basada en recursos
   - Arquitecturas basadas en publish/subscribe
2. Organización de *middleware*
3. Arquitecturas de sistemas
   - Sistemas centralizados
   - Sistemas descentralizados: P2P
   - Arquitecturas híbridas

Arquitecturas para sistemas distribuidos

Un sistema distribuido es *software* complejo.
Con elementos repartidos ...
¿Cómo organizarlos?

Arquitectura de un sistema distribuido

- Cómo proveer el *middleware*
- Con algunos grados de transparencia
- Mediante **componentes**, interfaces, agentes, etc
- Mediante roles
- A través una organización lógica: arquitectura de *software*

Contenidos

Arquitectura de *software*

¿Qué define una arquitectura?

- **Componentes** "reemplazables", con **interfaces** definidas
- La manera en que estos componentes se conectan
- Qué datos intercambian estos componentes
- Cómo los componentes y sus conectores forman un sistema completo

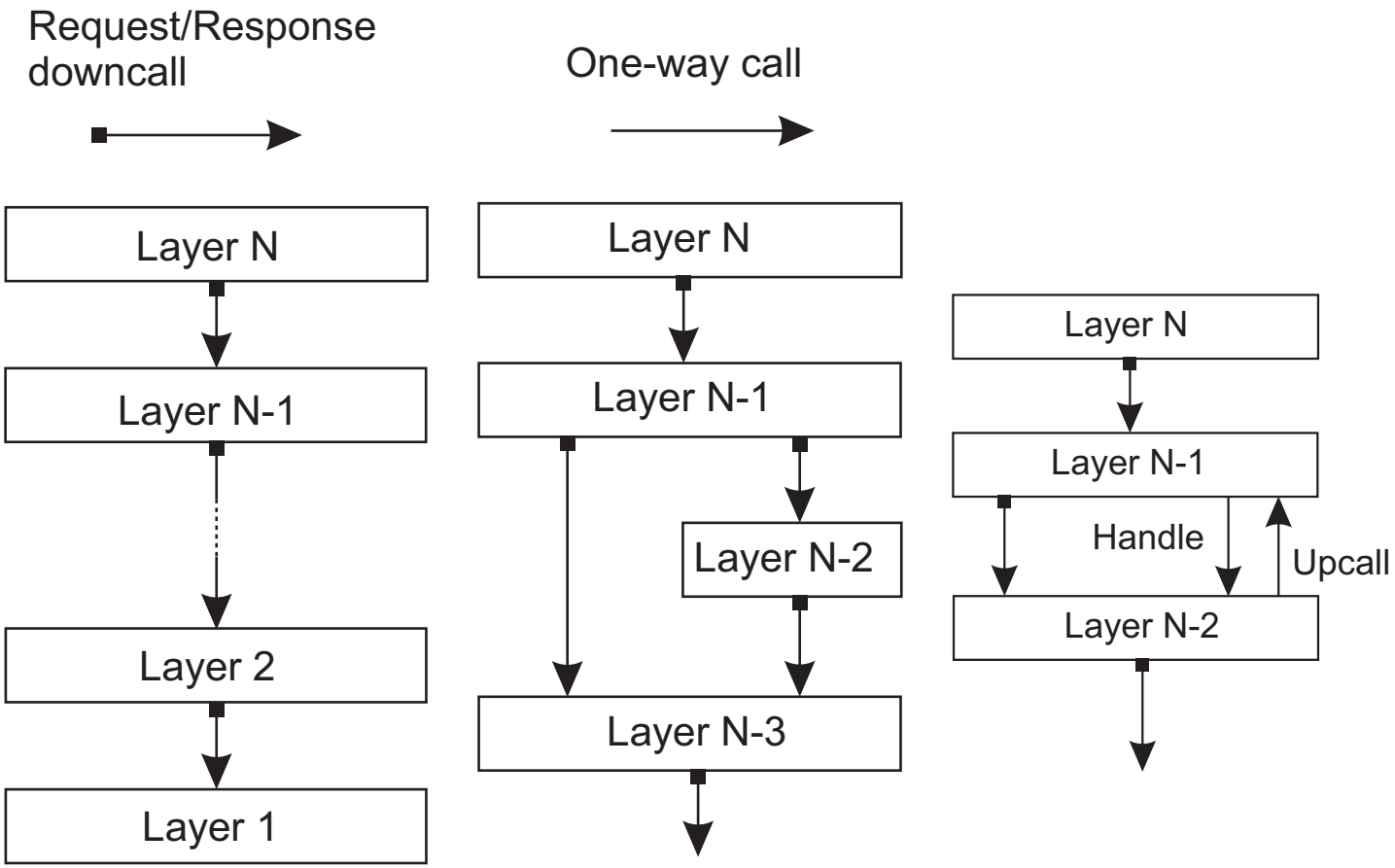
¿Componente?

Unidad modular con interfaces bien definidas

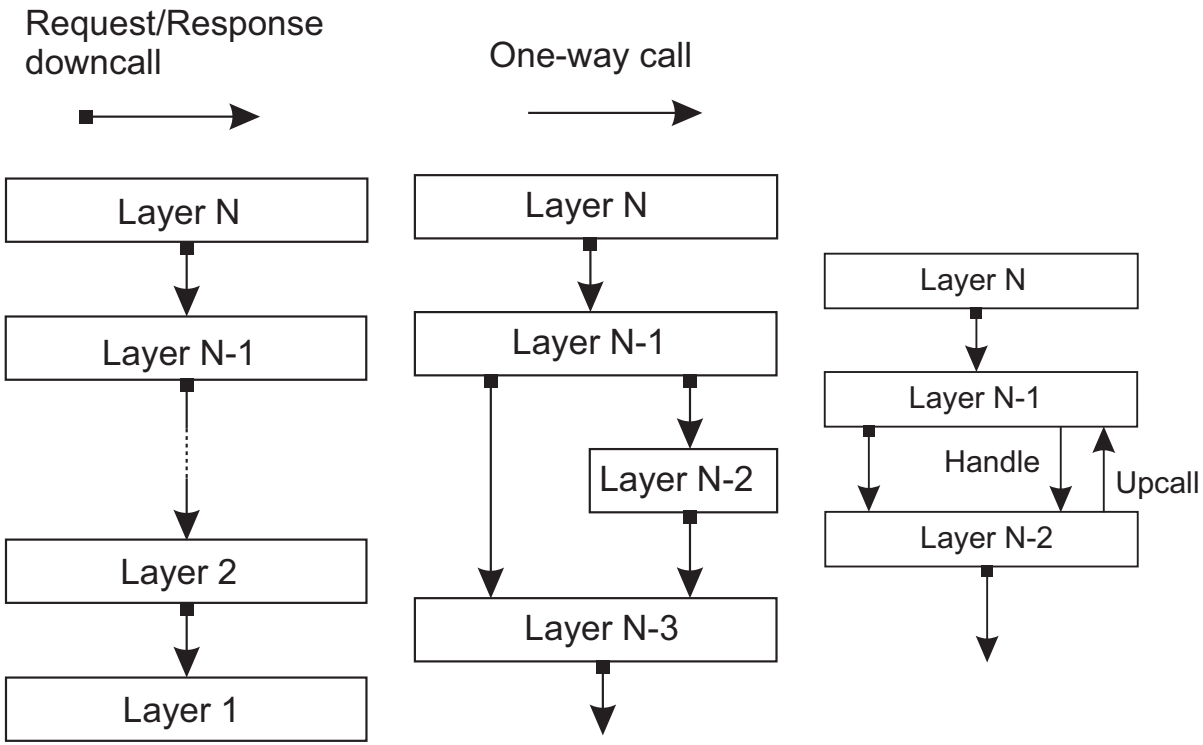
¿Conector?

Mecanismo para proveer comunicación, coordinación o cooperación entre componentes. Ej: RPC, MOM, *streaming*

Arquitectura por capas (*layered*)

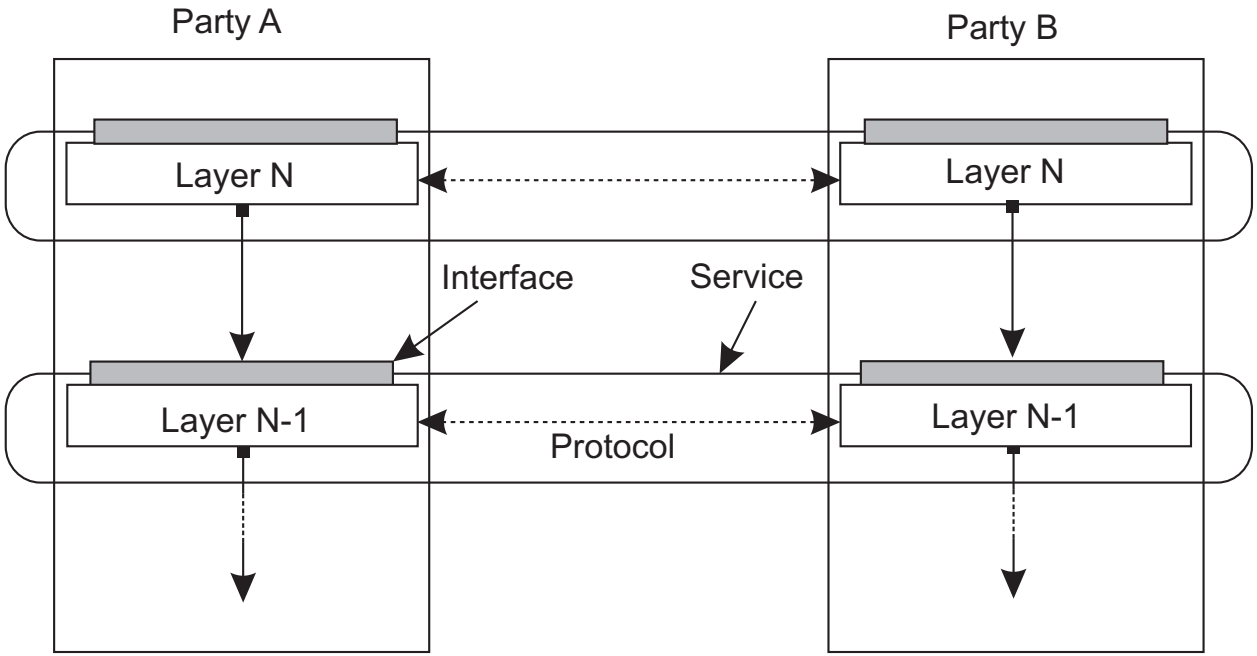


Arquitectura por capas (*layered*)



- (a) Arquitectura clásica de capas. Ej: redes
- (b) Arquitectura *mixed*. Ej: sistemas operativos
- (c) Arquitectura con llamadas inversas. Ej: señales, callbacks

Arquitectura por capas (*layered*)



Necesitamos definir:

- Interfaces
- Servicios
- Protocolos

Aplicaciones en capas (*layered*)

Server

```
1 from socket  import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept()   # returns new socket and addr. client
4 while True:                 # forever
5   data = conn.recv(1024)    # receive data from client
6   if not data: break        # stop if client stopped
7   conn.send(str(data)+"*")  # return sent data plus an "*"
8 conn.close()                # close the connection
```

Client

```
1 from socket  import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world')  # send some data
5 data = s.recv(1024)     # receive the response
6 print data               # print the result
7 s.close()               # close the connection
```

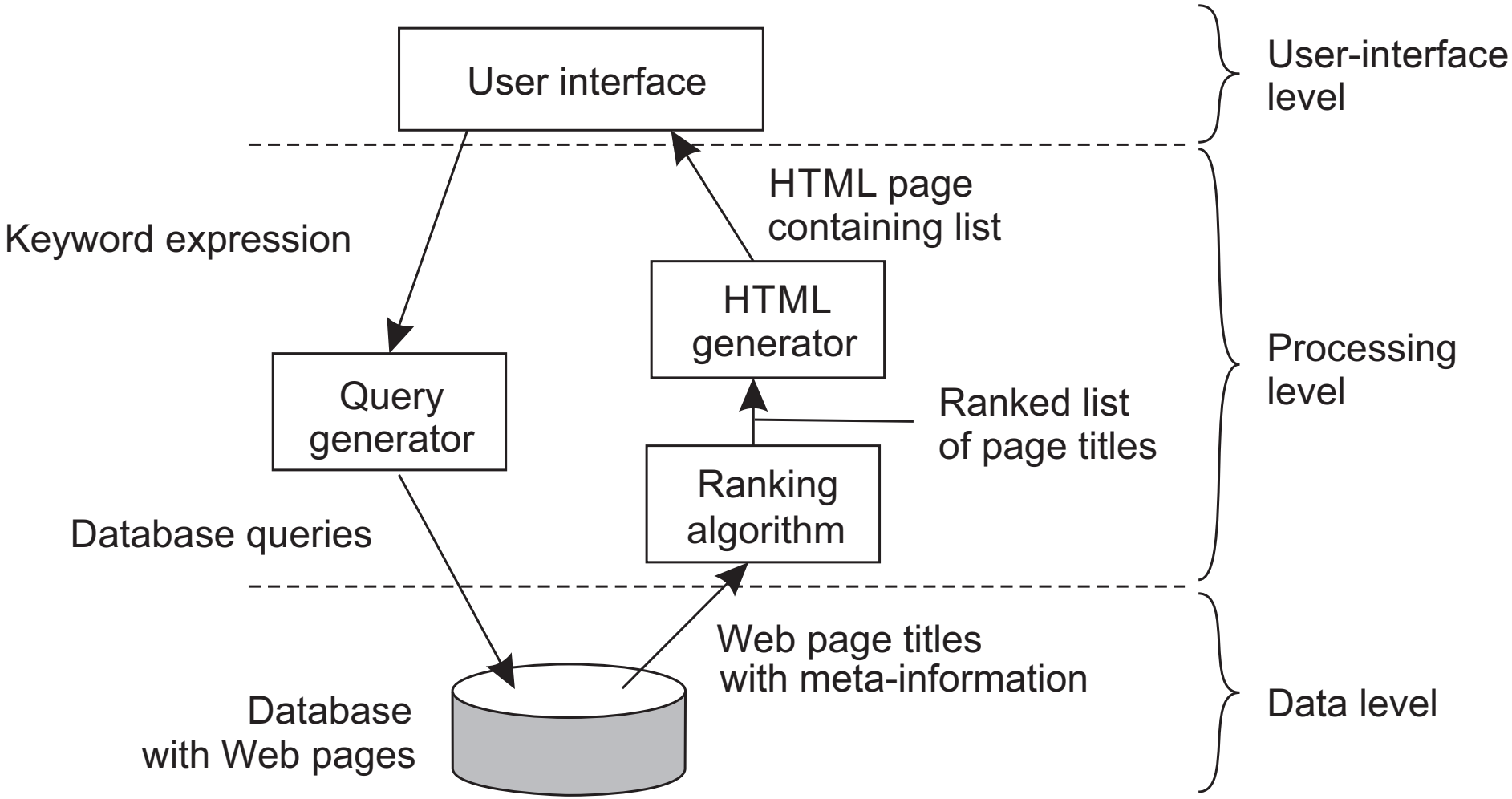
Aplicaciones en capas (*layered*)

Una arquitectura tradicional

Arquitectura de tres capas

- **Capa Application-interface**. Componentes que interactúan con el usuario o con aplicaciones externas.
- **Capa de procesamiento**. Funcionamiento (lógica) de la aplicación.
- **Capa de datos**. Datos que la aplicación debe manipular.

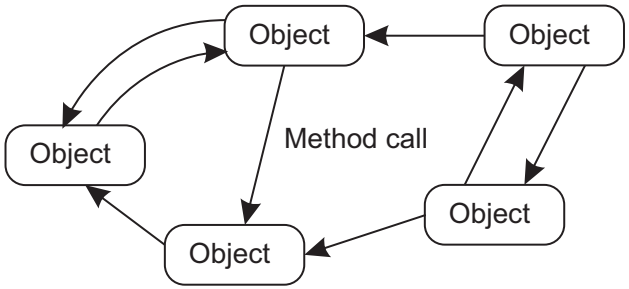
Aplicaciones en capas (*layered*)

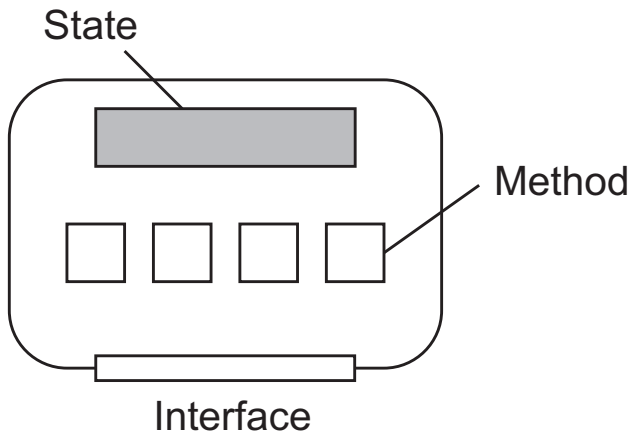


Arquitectura de tres capas

Arquitecturas basadas en objetos/servicios

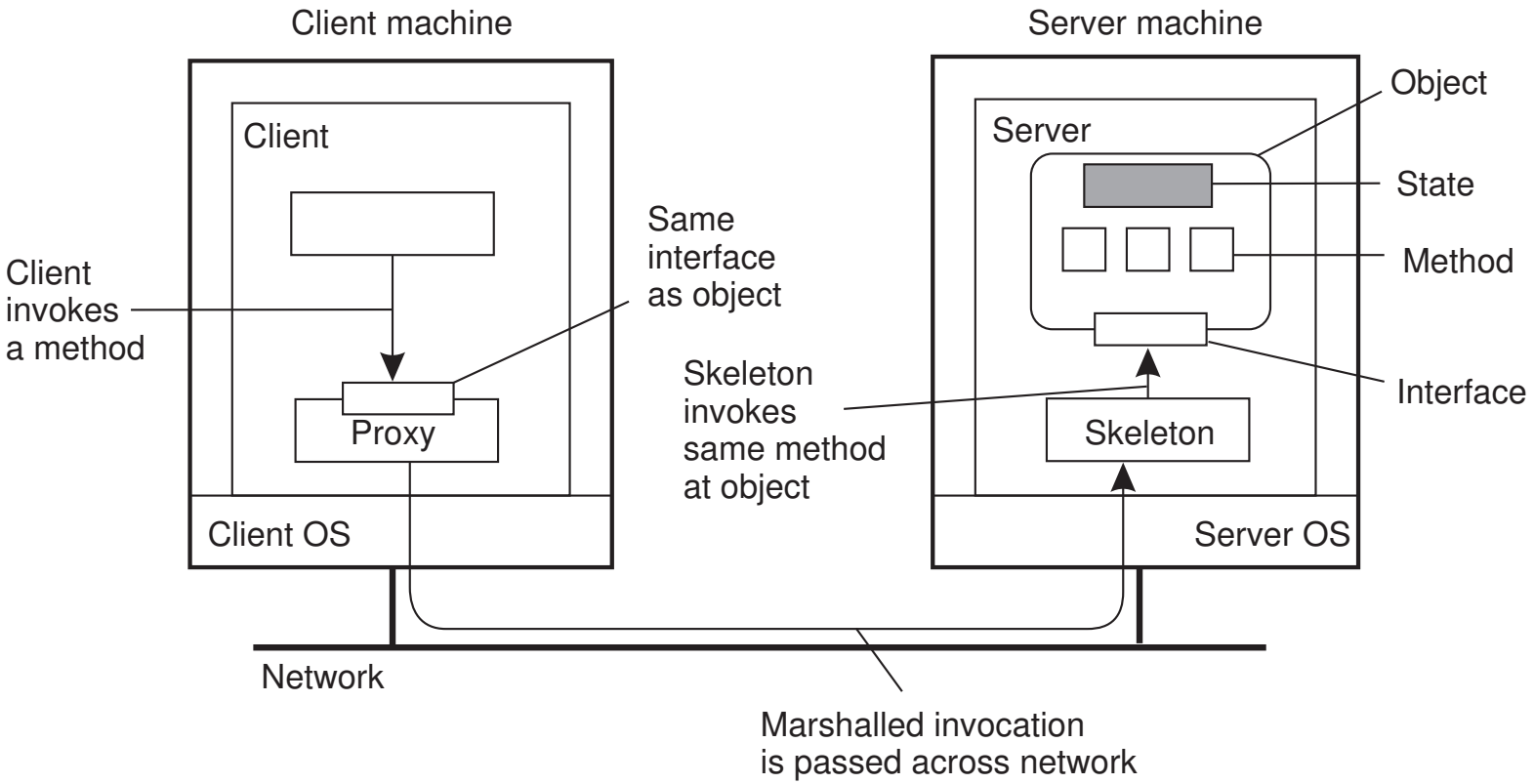
Aquí los componentes son objetos conectados mediante esquema de llamadas remotas (RPC, RMI, Web Services, etc)
Objetos pueden ser remotos.





Objetos **encapsulan** datos y **ofrecen métodos** sobre esos datos.
Objetos **ocultan implementación** interna.

Arquitecturas basadas en objetos/servicios



Arquitecturas basada en recursos

Sistema distribuidos es una **colección de recursos** manejados individualmente por componentes, mediante una interfaz de **acciones sobre esos recursos**

Desafíos sobre la arquitectura

- Identificación mediante esquema de nombres único
- Todos los servicios usan la misma interfaz
- Mensajes totalmente auto-descriptivos (no hay contexto)
- Luego de la operación, el servicio olvida al invocador (no hay estado)

Operaciones típicas

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

Arquitecturas basada en recursos

Ejemplo: Amazon Simple Storage Service
Objetos (archivos) se encuentran en *buckets* (directorios).
Operación sobre `ObjectName` en el *bucket* `BucketName` se describe como:

`http://BucketName.s3.amazonaws.com/ObjectName`

Operaciones típicas mediante HTTP request

- Crear *bucket*/objeto: `PUT` + URI
- Listar objetos: `GET` + *bucket*
- Leer objeto: `GET` + URI

Arquitecturas basada en recursos

Interfaz simple. Alta cantidad de información en el espacio de parámetros.

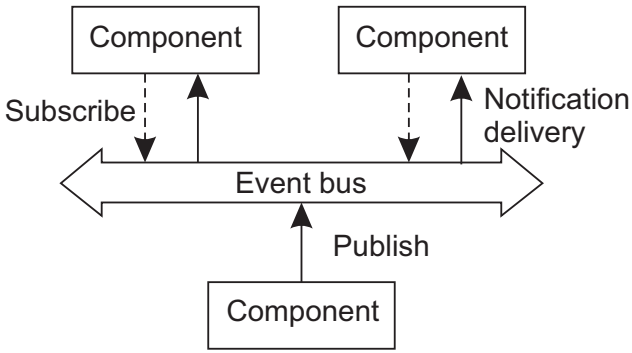
Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

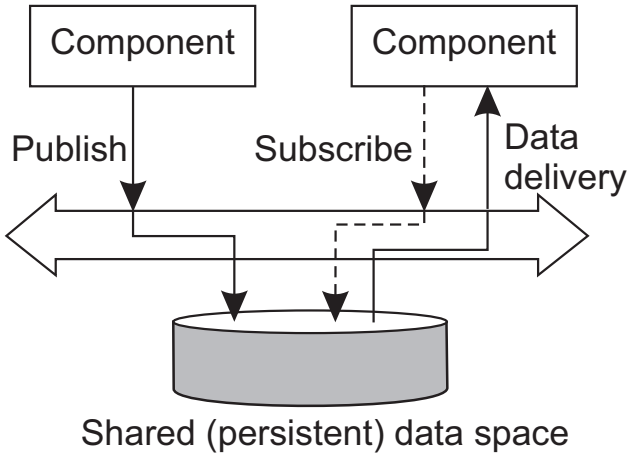
Arquitecturas basadas en publish/suscribe

Conceptos de acoplamiento espacial y temporal.

	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Coordinación basada en eventos y espacios compartidos





Arquitecturas basadas en publish/suscribe

Ejemplo: Linda

Espacio de tuplas con tres operaciones

- `in(t)`: borra tupla con template `t`
- `rd(t)`: obtiene copia de una tupla que hace *match* con `t`
- `out(t)`: agrega tupla `t` al espacio compartido

- Tuplas pueden estar repetidas
- Operaciones `in` y `rd` son **bloqueantes**

Arquitecturas basadas en publish/suscribe

Bob

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("bob","distsys","I am studying chap 2"))
4 blog._out(("bob","distsys","The linda example's pretty simple"))
5 blog._out(("bob","gtcn","Cool book!"))
```

Alice

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("alice","gtcn","This graph theory stuff is not easy"))
4 blog._out(("alice","distsys","I like systems more than graphs"))
```

Chuck

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 t1 = blog._rd(("bob","distsys",str))
4 t2 = blog._rd(("alice","gtcn",str))
5 t3 = blog._rd(("bob","gtcn",str))
```

Contenidos

Middleware

Un *middleware* para un sistema distribuido debe ser abierto (*openness*)
Sin embargo, existen muchos componentes *legacy* que no se adaptan a este propósito.
Dos patrones de diseño para construir *middlewares*:

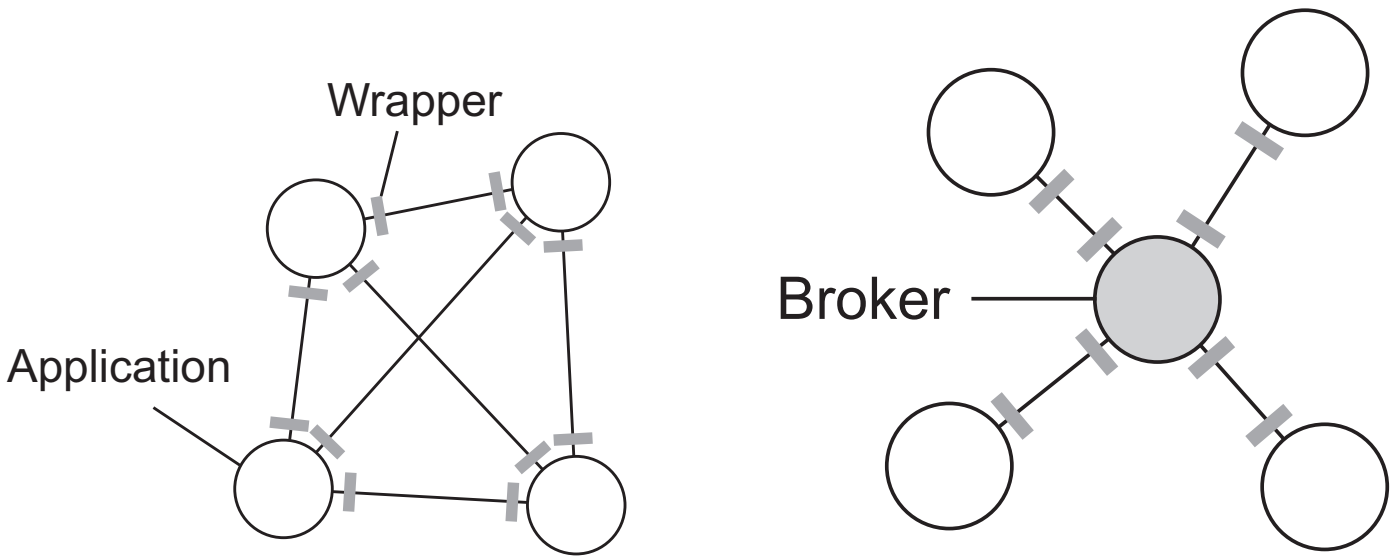
- *Wrappers*
- *Interceptors*

Middleware: *Wrappers*

Wrappers ó _adapters_

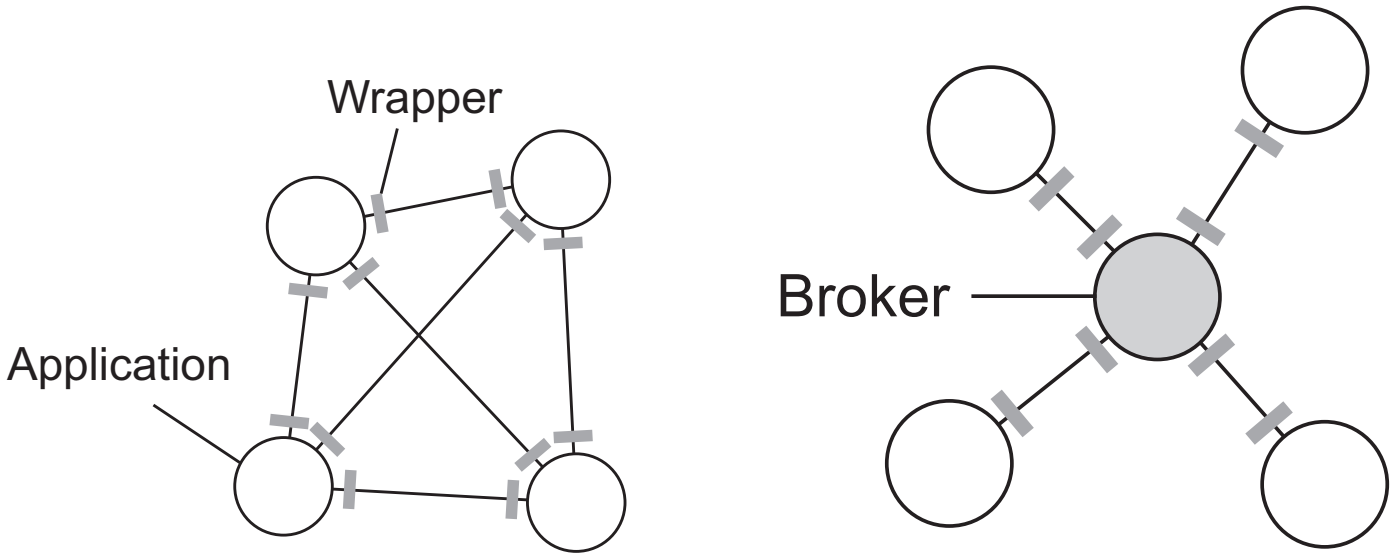
Componente que ofrece una interfaz aceptable para el cliente, transformando esas llamadas a las que están disponible en el componente envuelto.

Solución uno-a-uno, o través de *broker*



Middleware: *Wrappers*

Solución uno-a-uno, o través de *broker*



Complejidad con N aplicaciones

- 1 a 1: requiere $N \times (N-1) = O(N^2)$ *wrappers*
- *broker*: requiere $2N = O(N)$ *wrappers*

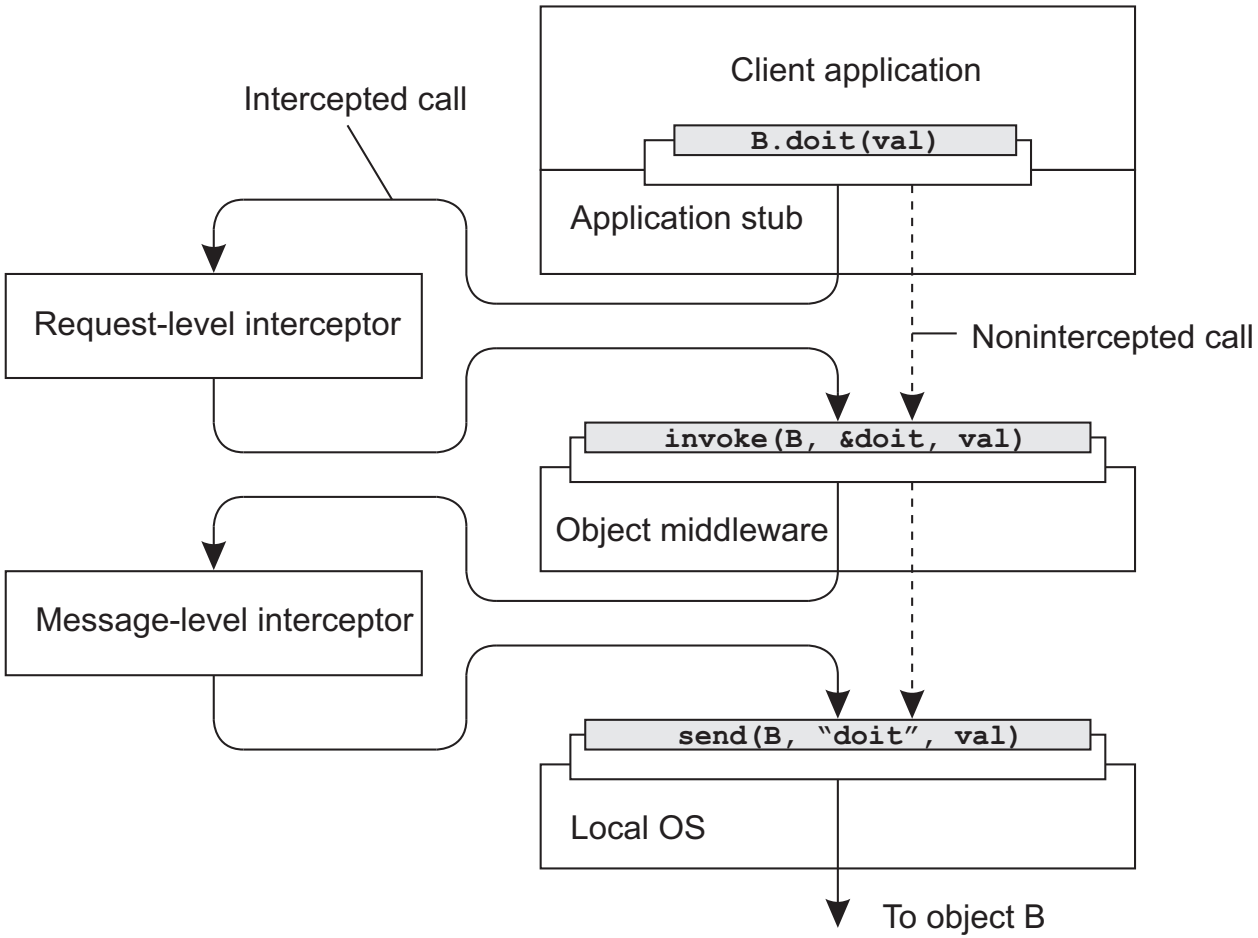
Middleware: *Interceptors*

Interceptor

Elemento de *software* que interviene el flujo natural de control, y permite la ejecución de código específico entre medio de ese flujo.

Métodos de comunicación de invocación remota como RPC ó RMI usan este patrón.

Middleware: *Interceptors*



Middleware modificable

Muchas veces un sistema distribuido (*middleware*) no puede ser completamente detenido. Se necesita que sea **adaptable** en tiempo de ejecución (*runtime*).

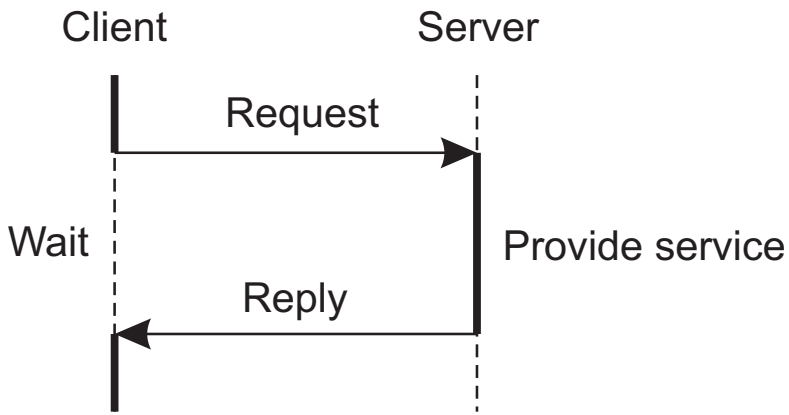
Modelos basados en componentes permiten adaptar parte de su comportamiento mediante técnicas como **late binding**.

Contenidos

Arquitecturas centralizadas

Modelo básico: cliente-servidor

- Procesos que ofrecen servicios: **servidores**
- Procesos que usan servicios: **clientes**
- Clientes y servidores pueden estar en máquinas distintas
- Clientes usan servicio mediante modelo **request/reply**

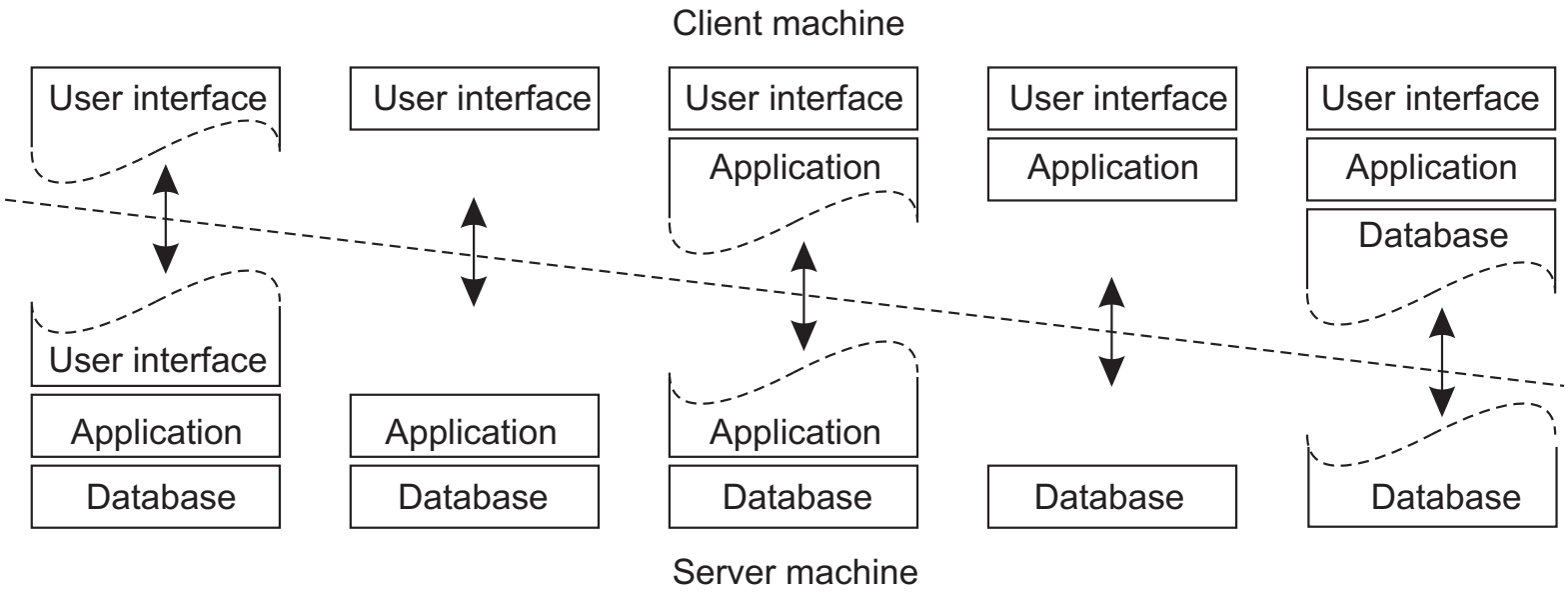


Desafíos

- Manejo de mensajes perdidos. Operaciones idempotentes o no.
- Costo de protocolos de establecimiento de conexión

Arquitecturas *multitiered*

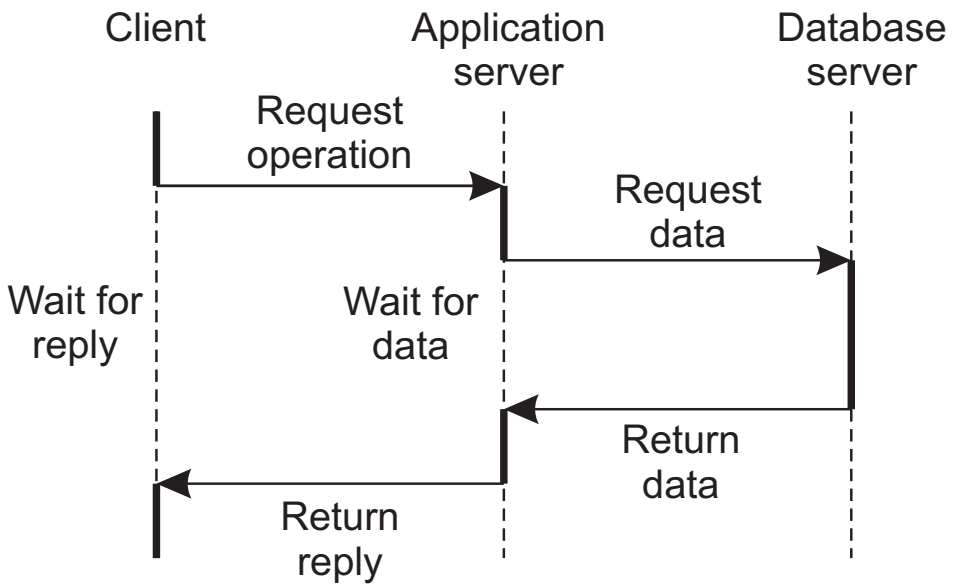
Podemos implementar la arquitectura de *software* de tres capas, en una arquitectura física *two-tiered*, con un cliente y un servidor.



¿Dónde está la división entre cliente y servidor?

Arquitecturas *multitiered*

También podemos implementar la arquitectura de *software* de tres capas en una arquitectura física *three-tiered*.



Algunas procesos son clientes y servidores simultáneamente.

Más alternativas de organización

Distribución vertical

Cada capa lógica se ejecuta en máquinas diferentes.

Distribución horizontal

Cada capa lógica puede separarse en distintas partes replicadas o distribuidas. Permite balancear carga.

Arquitecturas *peer-to-peer*

Todos los procesos son iguales. Todos actúan como cliente y como servidor (*servant*).

Arquitecturas P2P implementan distribución horizontal: *overlay networks*

- Estructuradas
- No Estructuradas

Redes P2P estructuradas

Redes P2P estructuradas

Nodos (procesos) organizados en una topología.

- La topología permite ubicar eficientemente recursos
- Usan un *semantic-free index*: datos asociados a *key*s únicas (índices)

$$key(item) = hash(valor\ del\ item)$$

- Sistemas P2P almacenan pares *key-value*.
- Implementan una **Distributed Hash Table (DHT)**

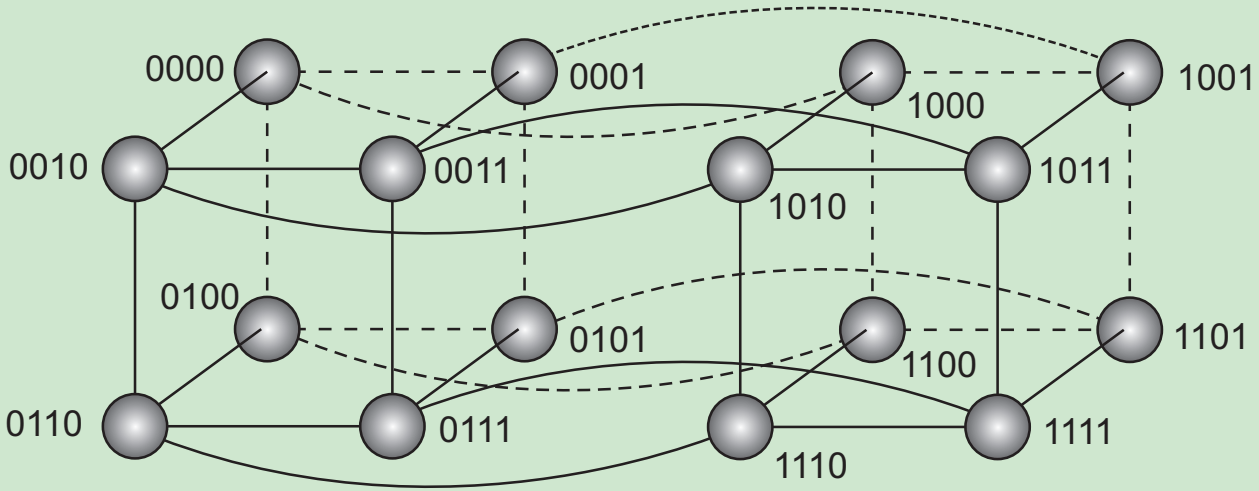
Funcionamiento

El principal objetivo de una red P2P estructurada es proveer una implementación eficiente de:

$$nodo\ existente = lookup(key)$$

Redes P2P estructuradas: ejemplo

Arquitectura de hipercubo de 4 dimensiones



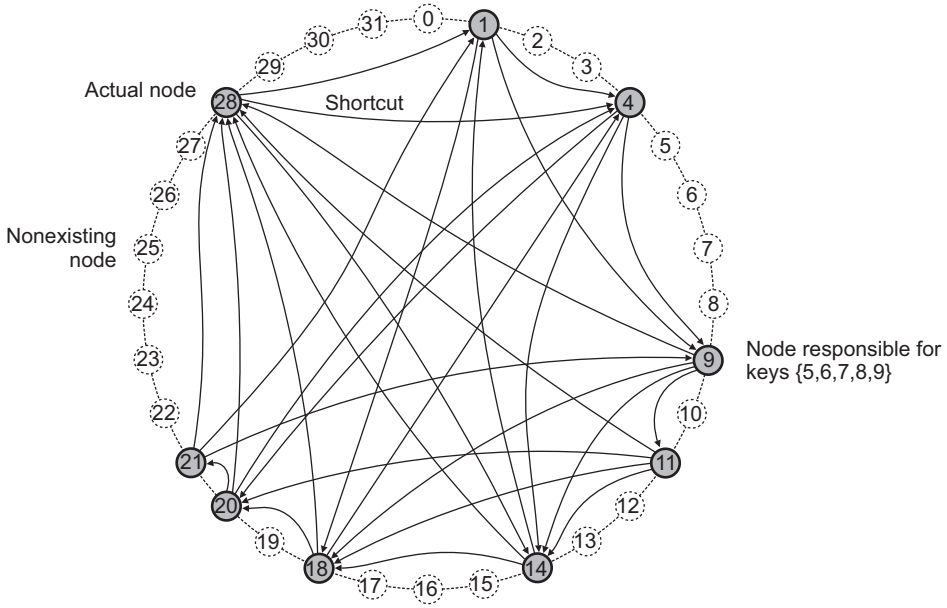
- Cada nodo poseee una *key* $k \in \{0, 1, 2, \ldots, 2^4 - 1\}$
- La operación *lookup*(k) consiste en **enrutar** la solicitud hasta el nodo con identificador k

¿Cómo encontramos un nodo eficientemente a partir de cualquier otro nodo?

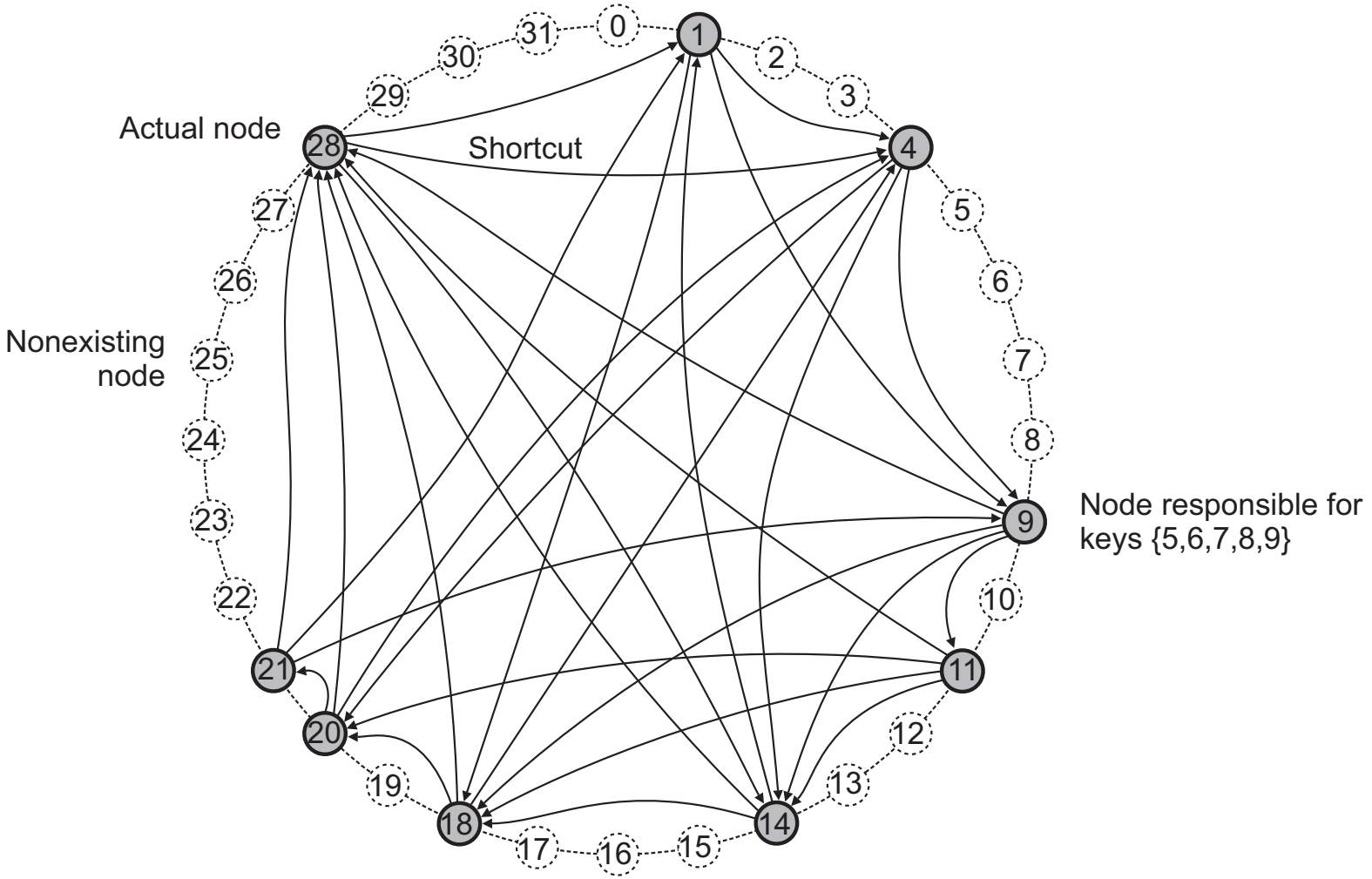
Redes P2P estructuradas: ejemplo Chord

Chord

- Nodos organizado como un anillo (*ring*)
- Cada ítem se asocia un *hash* de *m*-bit. (2^m keys)
- El ítem con llave k se almacena en el nodo con identificador más pequeño tal que $id \geq k$. Este nodo es el **sucesor** de k
- El anillo posee *shortcuts* a otros nodos.



Redes P2P estructuradas: ejemplo Chord



Redes P2P no estructuradas

Redes P2P no estructuradas

Nodos mantienen listas *ad-hoc* a vecinos.

- La red *overlay* parecer ser un grafo aleatorio.
- Cada arista $\langle u, v \rangle$ posible tiene una probabilidad $P(\langle u, v \rangle)$ de existir.

Funcionamiento

Dos maneras de búsqueda:

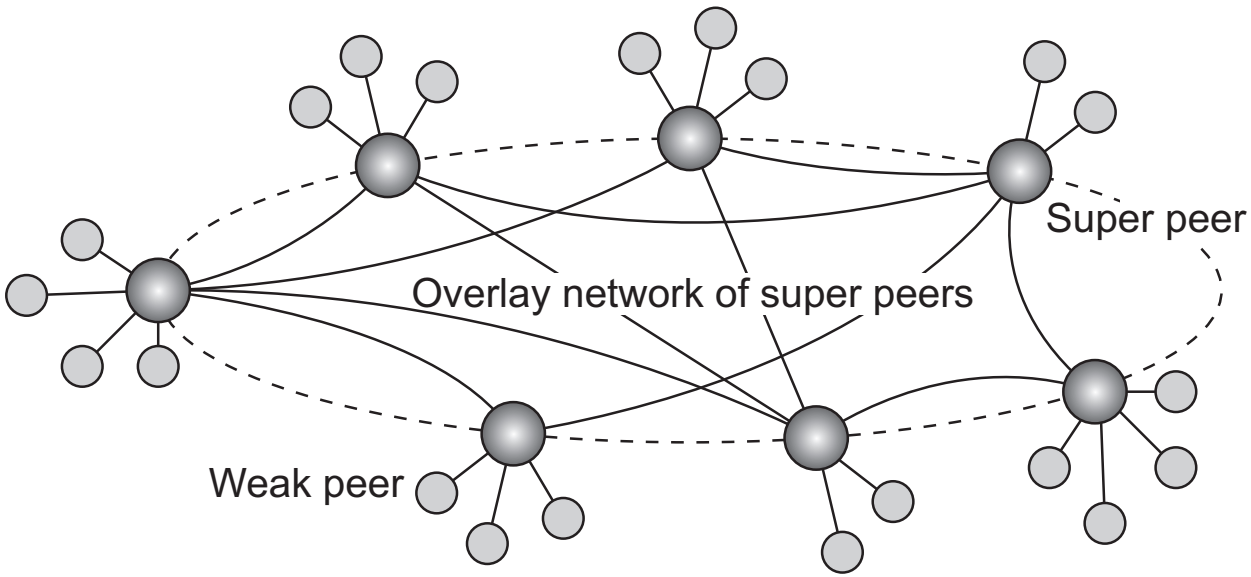
- **Flooding**: nodo origen u envía *request* a d vecinos $\{v_1, v_2, \ldots, v_d\}$. *Request*s duplicadas se ignoran. Si v_i no tiene el dato, hace una búsqueda local recursiva. Usa límite (TTL) de saltos (*hops*).
- **Random walk**: nodo origen u envía *request* a vecino aleatoria v. Si v no tiene el dato, lo reenvía a vecino aleatorio, y así sucesivamente.

Redes P2P jerárquicas

Redes P2P jerárquicas

No todos los pares son igual de pares

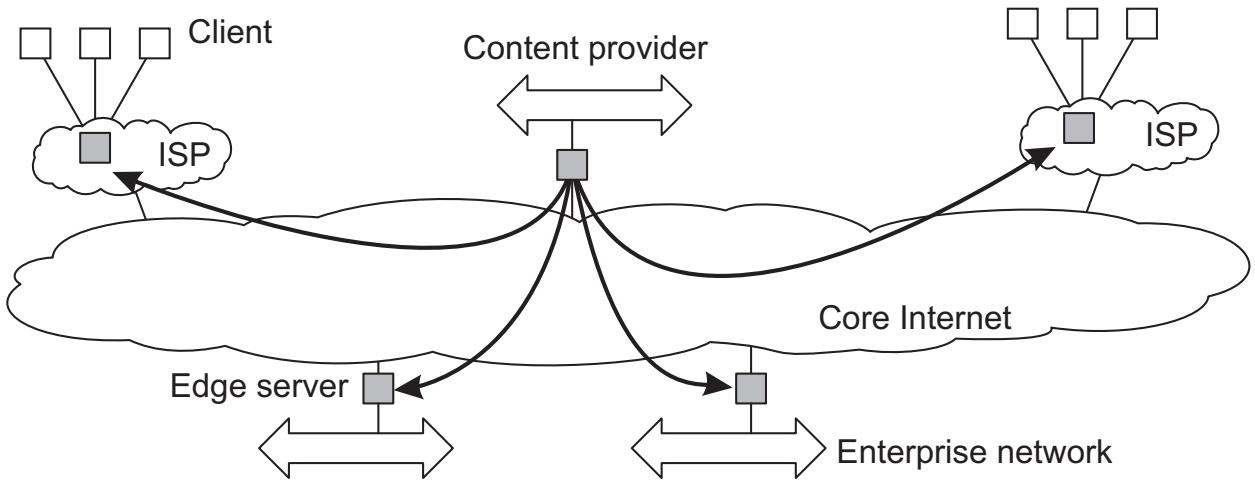
- **Super-peers**: mantienen **índices** de ubicaciones y actúan como **brokers**
- Clientes regulares son **weak peers**



Redes *Edge-Server*

Redes Edge-Server

Usuarios se conectan a través de *edge-servers*



Redes colaborativas

Redes colaborativas

Búsqueda de un archivo *F*

- Se busca en un directorio global que retorna un **torrent file**
- **Torrent file** referencia a un *tracker* que mantiene una lista dinámica de nodos **activos** que contiene **partes** (*chunks*) de *F*
- *P* puede unirse a un *swarm*, obtener un *chunk*, cambiarlo con otro nodo *Q* a través de la red.