Arquitectura de los Sistemas Distribuidos

Índice

- Introducción
- Arquitecturas para computación distribuida
  - Arquitecturas de computación en Google
    - Modelo *Map-Reduce* y *Pregel*
- Arquitectura cliente-servidor
  - Variaciones del modelo
  - Aspectos de diseño del modelo cliente/servidor
- Arquitectura editor-subscriptor
- Arquitectura *Peer-to-peer*
  - Sistemas P2P desestructurados
  - Sistemas P2P estructurados
    - Protocolo Chord

Arquitecturas de los SD

- Organización lógica de componentes de aplicación distribuida
  - Cómo es su patrón de interacción
  - Qué roles ejercen los procesos involucrados
  - Y cuál es su correspondencia con nodos de SD físico
  - "Topología" de la aplicación distribuida
- En principio, tantas como aplicaciones
  - Pero hay patrones que se repiten de forma habitual
- Arquitecturas más frecuentes en SD de propósito general
  - Cliente/servidor
  - Editor/subscriptor
  - *Peer-to-peer* (Paritaria)
- Computación distribuida (**CD**) presenta sus propios patrones
  - Maestro/trabajador
  - Arquitecturas guiadas por la "geometría" de los datos

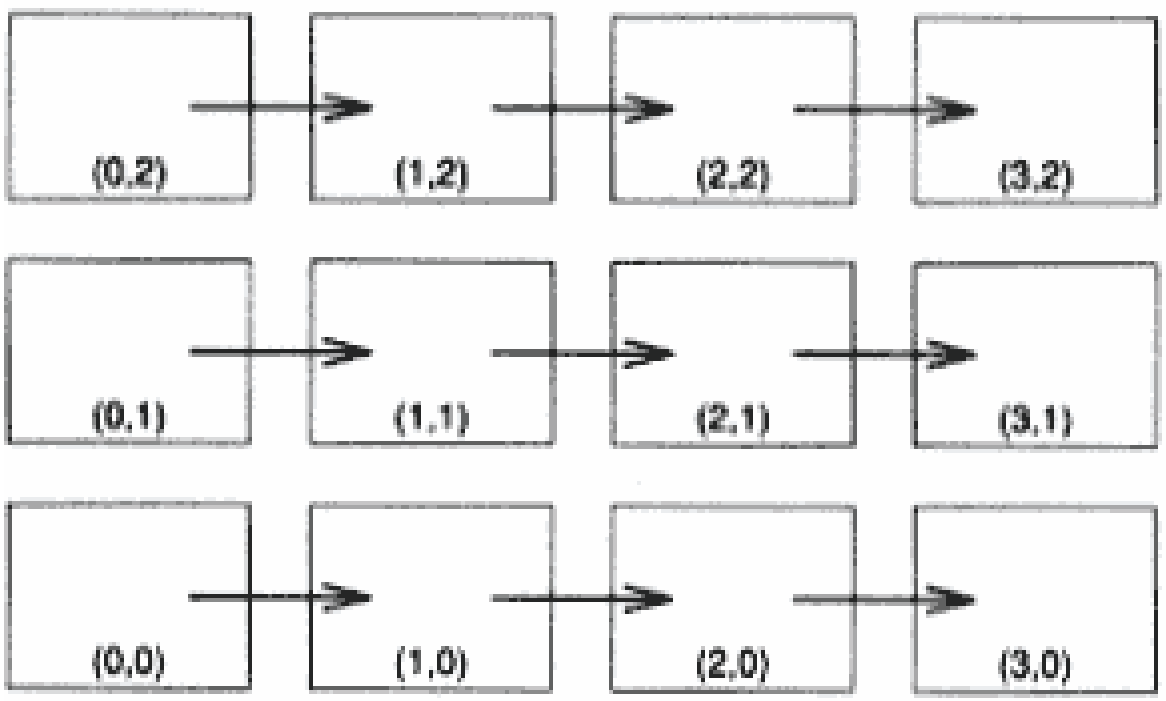
Grado de acoplamiento

- Sea cual sea el modelo, conlleva interacción entre entidades
- Interacción tradicional implica acoplamiento espacial y temporal
- Desacoplamiento espacial (de referencia)
  - Entidad inicia interacción **no** hace referencia directa a la otra entidad
    - No necesitan conocerse entre sí
- Desacoplamiento temporal (menos frecuente)
  - "Vidas" de entidades que interaccionan **no** tienen que coincidir
- Ej. Uso de memoria compartida
- 2 desacoplamientos son independientes entre sí
- Estos modos de operación "indirectos" proporcionan flexibilidad
- David Wheeler (el inventor de la subrutina):
  - "*All problems in computer science can be solved by another level of indirection...except for the problem of too many layers of indirection.*"

Arquitecturas para CD

- Maestro-trabajador *M/T* (*aka* maestro-esclavo)
  - *M* va repartiendo trabajos entre nodos trabajadores *T*
    - Si nº trabajos >> nº trabajadores → reparto automático de carga
  - Tolerancia a fallos:
    - Caída de *T*: *M* reasigna sus trabajos pendientes (¡efectos laterales!)
    - Caída de *M*: punto crítico de fallo
- Arquitecturas guiadas por "geometría" de los datos
  - Matrices multidimensionales, grafos, etc.
  - P.e. Matriz 2D
    - Cada nodo se encarga de sub-matriz
    - Comunicación más frecuente con nodos "vecinos cartesianos"
  - MPI facilita uso mediante "topologías virtuales" (*Cartesian* y *Graph*)
    - Permite localizar fácilmente "vecinos"
    - Implementación puede optimizar asignación a plataforma

Topología *Cartesian* de MPI



int **MPI_Cart_create**(MPI_Comm comm, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);

int **MPI_Cart_shift**(MPI_Comm comm_cart, int direction, int disp, int *rank_source, int *rank_dest);

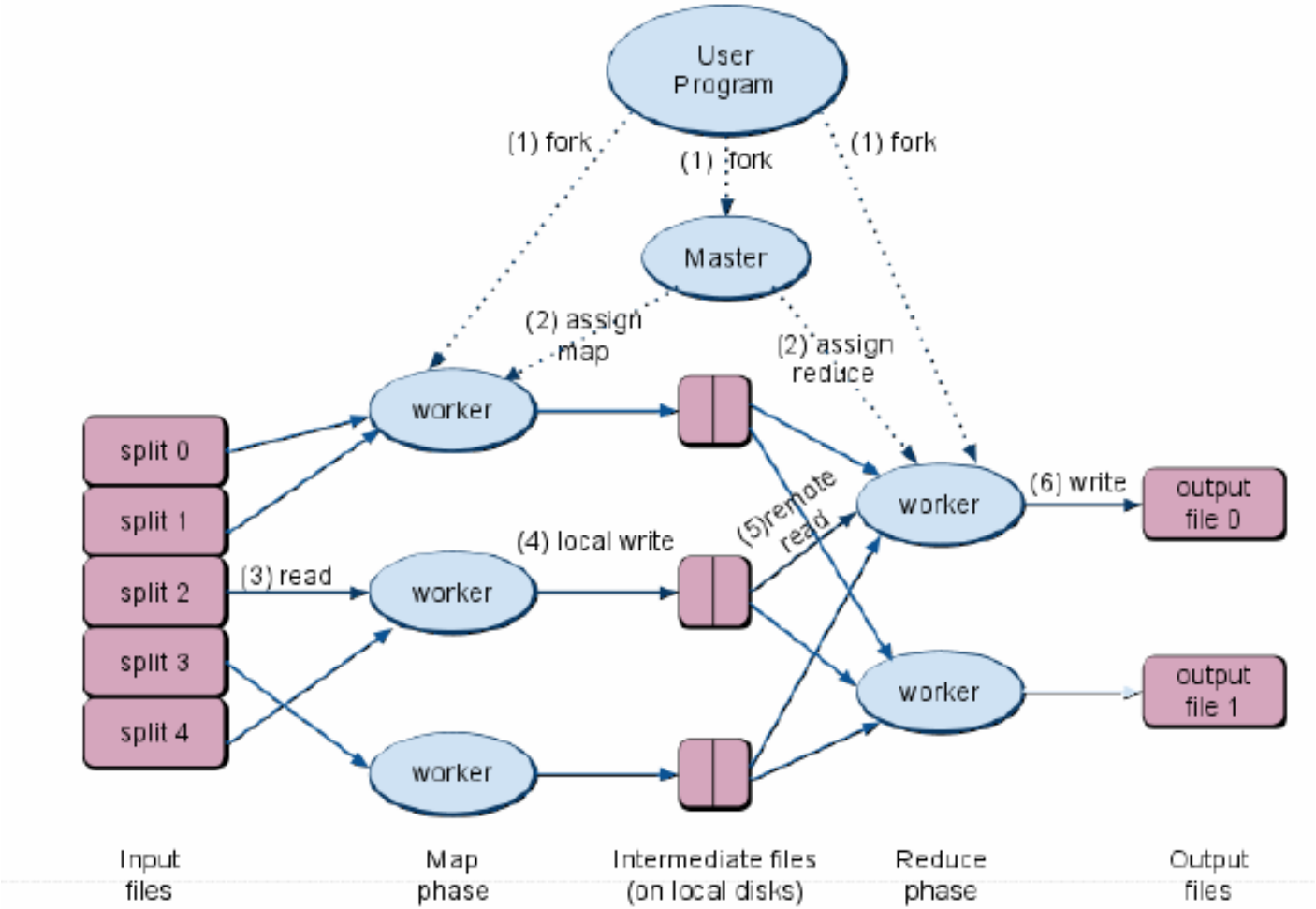
Using MPI

William Gropp, Ewing Lusk y Anthony Skjellum (*MIT Press*)

Arquit. de computación en Google

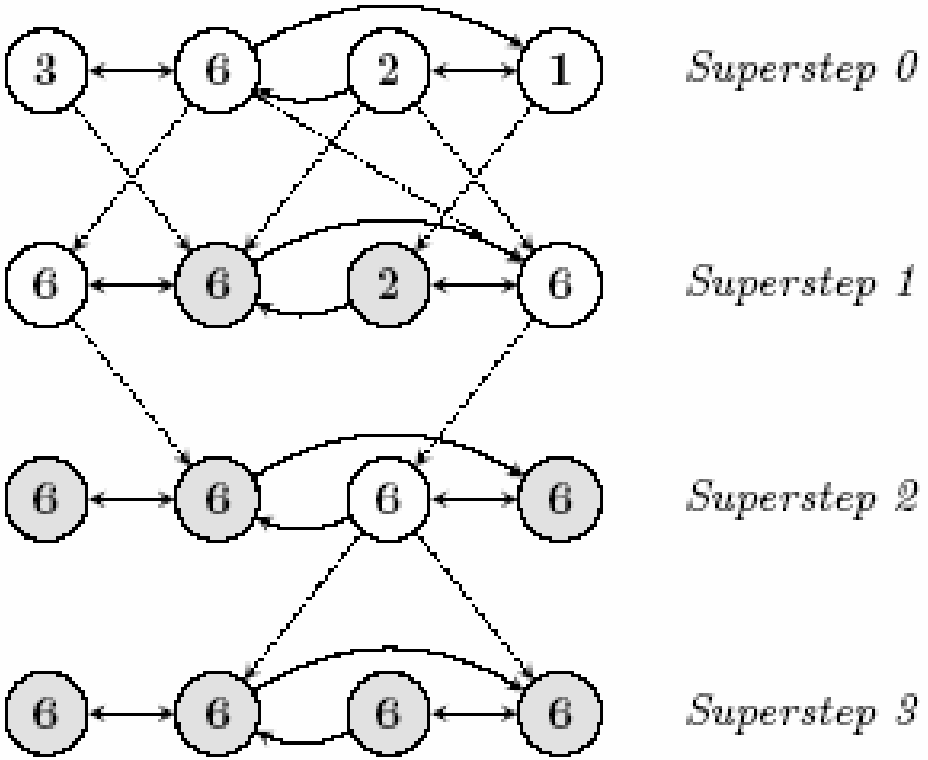
- *MapReduce* (≈ 80% de computaciones en Google)
  - Maestro-trabajador con dos fases: *Map* y *Reduce*
  - *Map*: *T* procesa su parte de datos de entrada y genera (*clave*, *valor*)
    - P.ej. Palabras que aparecen en su parte de datos de entrada
  - *Reduce*: *T* procesa valores asociados a una clave
    - P.ej. Frecuencia de palabras en todos los datos de entrada
- *Pregel* (≈ 20% de computaciones en Google)
  - Modelo diseñado para procesar grafos de gran escala
  - Computación organizada en "*supersteps*" síncronos:
    - Cada vértice recibe datos de otros vértices por aristas de entrada
    - Cambia su estado y genera datos por vértices de salida
    - Incluso puede cambiar topología del grafo
  - Inspirado en modelo "*Bulk Synchronous Parallel*" de Valiant
  - Implementado como arquitectura maestro/trabajador
    - *M* reparte grafo entre *T* y controla sincronización de "*supersteps*"

Modelo de computación *Map-Reduce*



Extraído de tutorial sobre MapReduce **de Jerry Zhao y Jelena Pjesivac-Grbovic**

Modelo de computación *Pregel*



Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz et al.; *SIGMOD '10*

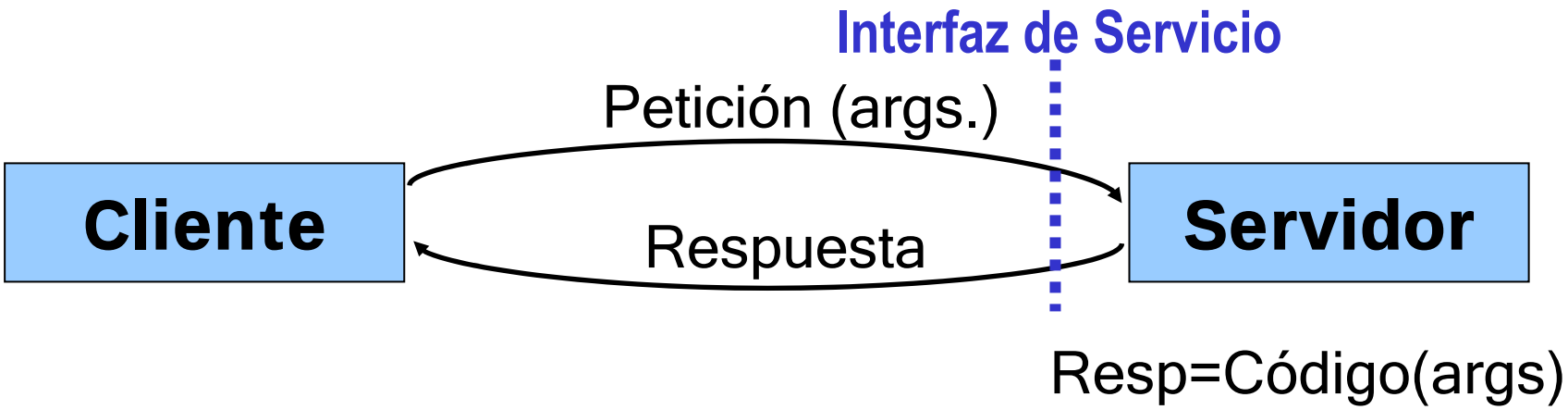
Arquitecturas en SD de propósito general

- Cliente-servidor
  - Extensión del modelo proveedor/consumidor de servicio a SD
    - Similar a biblioteca de servicio y programa que la usa pero en SD
  - Interacción 1-N
- Editor/subscriptor
  - Extensión de esquema guiado por eventos a SD
  - Facilita el desacoplamiento espacial y, potencialmente, el temporal
  - Interacción M-N
- *Peer-to-peer*
  - Procesos cooperantes con el mismo rol
  - Interacción N-N

Modelo cliente/servidor

- Arquitectura asimétrica: 2 roles en la interacción
  - Cliente: Solicita servicio
    - Activo: inicia interacción
  - Servidor: Proporciona servicio
    - Pasivo: responde a petición de servicio
- Desventajas de arquitectura cliente/servidor
  - Servidor "cuello de botella" → problemas de escalabilidad
  - Servidor punto crítico de fallo
  - Mal aprovechamiento de recursos de máquinas cliente
- Normalmente, acoplamiento espacial y temporal
- Servidor ofrece colección de servicios que cliente debe conocer
- Normalmente, petición especifica recurso, operación y args.
  - NFS: *READ, file_handle, offset, count*
  - HTTP: *GET /index.html HTTP/1.1*

Esquema cliente/servidor



- Alternativas en diseño de la interfaz de servicio
  - Operaciones específicas para cada servicio
    - Énfasis en operaciones (“acciones”)
  - Mismas ops. para todos servicios pero sobre distintos recursos (REST)
    - Énfasis en recursos: ops. CRUD (HTTP GET, PUT, DELETE, POST)
  - Ejemplo:
    - AddBook(nb) vs. PUT /books/ISBN-8448130014 HTTP/1.1

Cliente/servidor con caché

- Mejora latencia, reduce consumo red y recursos servidor
- Aumenta escalabilidad
  - Mejor operación en SD → La que no usa la red
- Necesidad de coherencia: sobrecarga para mantenerla
  - ¿Tolera el servicio que cliente use datos obsoletos?
    - SFD normalmente no; pero servidor de nombres puede que sí (DNS)
- Puede posibilitar modo de operación desconectado
  - CODA
  - HTML5 *Offline Web Applications*
- *Pre-fetching*: puede mejorar latencia de operaciones pero
  - Si datos anticipados finalmente no requeridos: gasto innecesario
    - Para arreglar la falacia 2 hemos estropeado la 3

Reparto funcionalidad entre C y S

- ¿Qué parte del trabajo realiza el cliente y cuál el servidor?
- "Grosor del cliente": Cantidad de trabajo que realiza
  - Pesados *(Thick/Fat/Rich Client)* vs. Ligeros (*Thin/Lean/Slim Client)*
- Ventajas de clientes ligeros
  - Menor coste de operación y mantenimiento
  - Mejor seguridad
- Ventajas de clientes pesados
  - Mayor autonomía
  - Mejor escalabilidad
    - Cliente gasta menos recursos de red y de servidor
  - Más ágil en respuesta al usuario
- Ej. "inteligencia en cliente": Javascript valida letra NIF en form.

Posibles repartos entre C y S

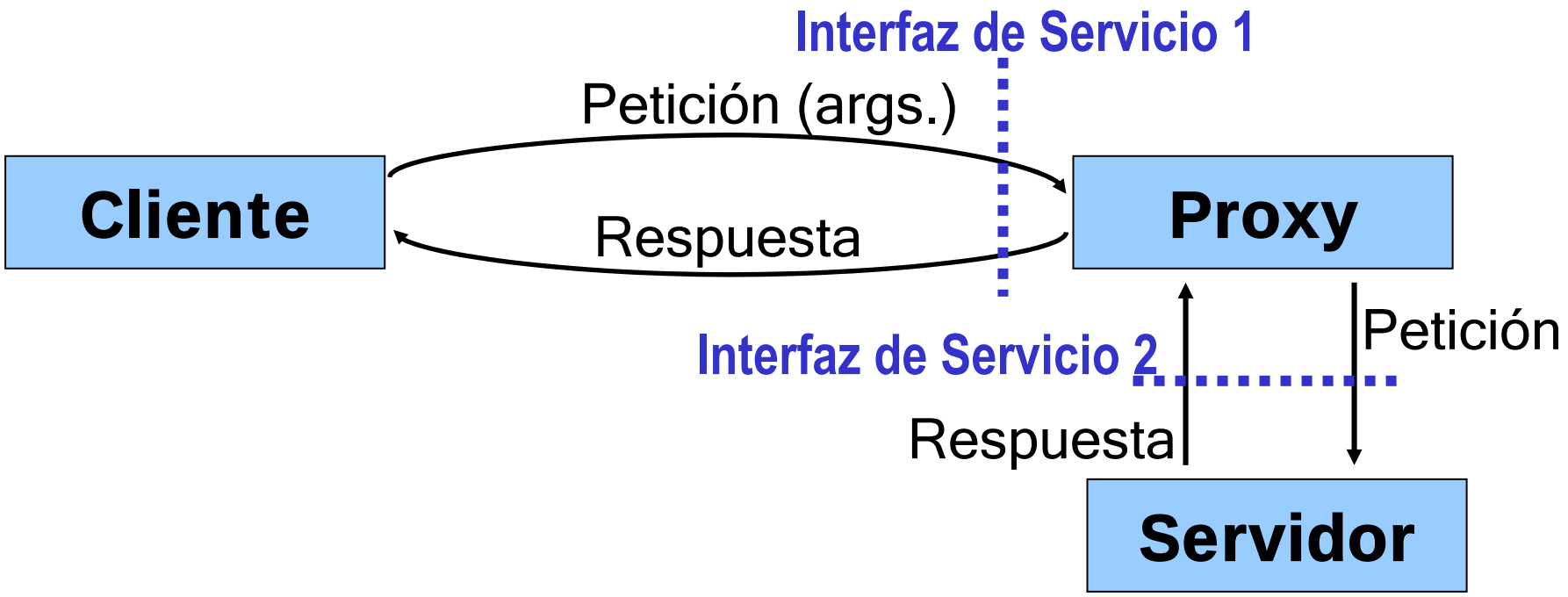
- Arquitectura típica de aplicación basada en 3 capas:
  - Presentación (interfaz de usuario gráfica: GUI)
  - Aplicación: lógica de negocio
  - Acceso a datos
- ¿Qué labor se asigna a máquina cliente? (“grosor” creciente)
  - Envía eventos de ratón/teclado y recibe info. a dibujar en forma de:
    - Píxeles (VNC) o Primitivas gráficas (X11)
    - Cambio de roles: servidor gráfico en máquina cliente
  - GUI
  - GUI + parte de la lógica de negocio
  - GUI + lógica de negocio
  - GUI + lógica de negocio + parte de lógica de acceso

Cliente/servidor con *proxy*

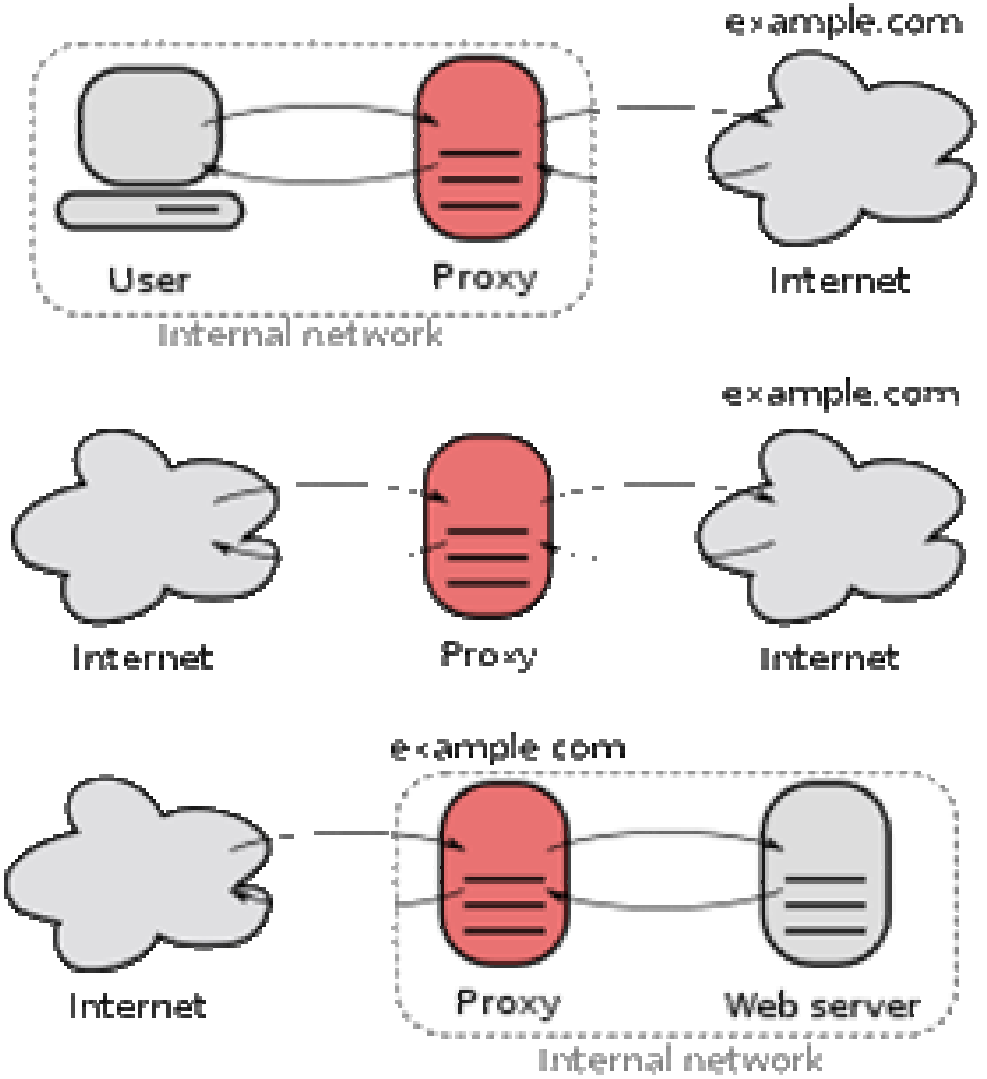
- Componentes intermediarios entre cliente y servidor
  - NOTA: Término corresponde también a un patrón de diseño
- Actúan como “tuberías”
  - Pueden procesar/filtrar información y/o realizar labor de caché
    - Símil con clases *FilterInputStream*|*FilterOutputStream* de Java
- Diversos tipos: *forward proxy*, *reverse proxy*, *gateways*, ...
- Mejor si interfaz de servicio uniforme:
  - *Proxy* se comporta como cliente y servidor convencional
  - Se pueden “enganchar” sucesivos *proxies* de forma transparente
  - Esta característica es una de las claves del éxito de la Web

Esquema con *proxy*

- Mejor si *Interfaz de Servicio 1* = *Interfaz de Servicio 2*



Wikipedia: *Proxy server*



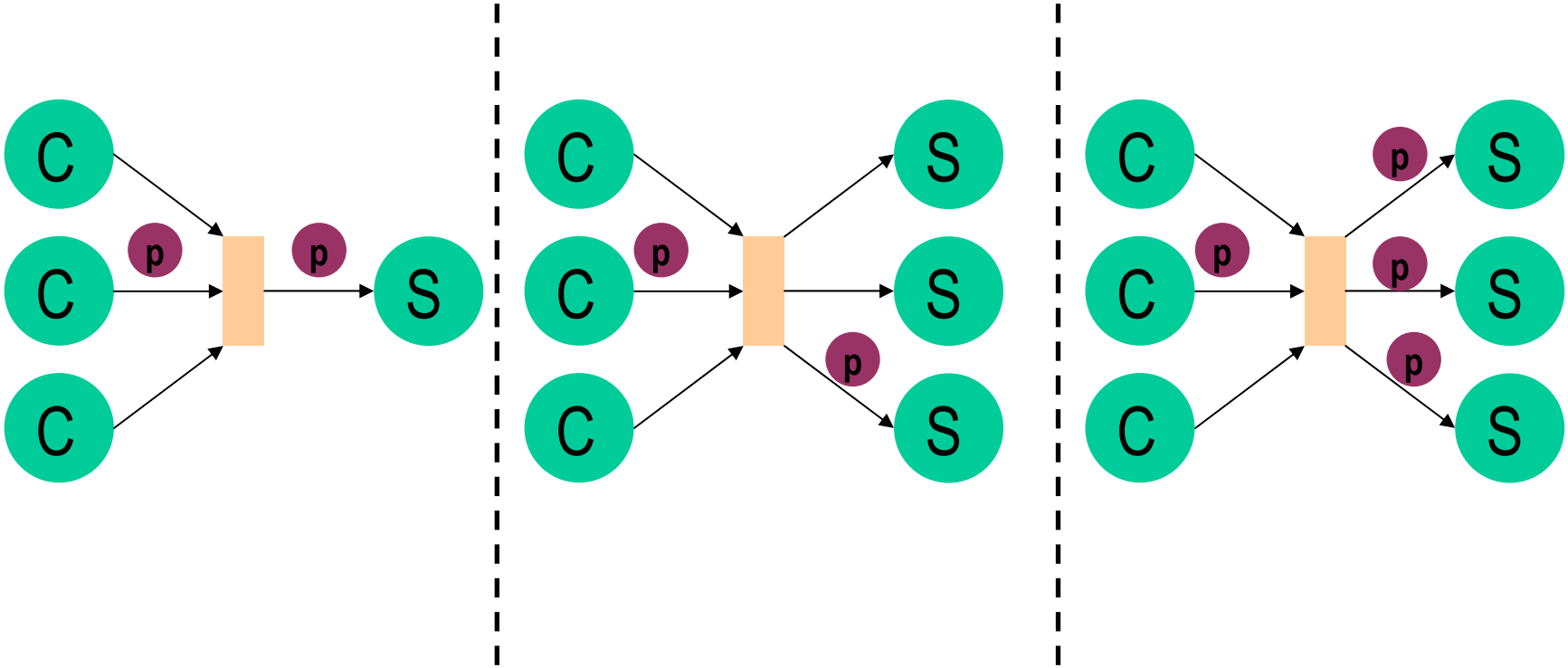
Cliente/servidor jerárquico

- Servidor actúa como cliente de otro servicio
  - Igual que biblioteca usa servicio de otra biblioteca
- Funcionalidad dividida en varios niveles (*multi-tier*)
  - P. ej. Aplicación típica con 3 capas:
    - Presentación
    - Aplicación: lógica de negocio
    - Acceso a datos
  - Cada nivel puede implementarse como un servidor
- Múltiples servidores idénticos cooperan en servicio
  - Traducir un nombre de fichero en SFD
  - Traducir de nombre de máquina a IP usando DNS

Cliente/servidor con replicación

- Servidor único:
  - Cuello de botella: afecta a latencia y ancho de banda
  - Punto único de fallo: afecta a fiabilidad
- Uso de múltiples servidores (interacción M-N)
- Si sólo uno implicado en servicio → reparto de carga
  - P.ej. leer el valor de un dato replicado en varios servidores
  - Mejora latencia, escalabilidad y tolerancia a fallos
- Si múltiples servidores deben cooperar para ofrecer servicio
  - P. ej. actualizar simultáneamente dato replicado en varios servidores
  - Mejora tolerancia a fallos pero no latencia y escalabilidad
- Necesidad de coherencia (sobrecarga para mantenerla):
  - Esquema simétrico: Actualizar simultánea en todas las réplicas
  - Esquema asimétrico: Actualizar en primario y propagar a secundarios

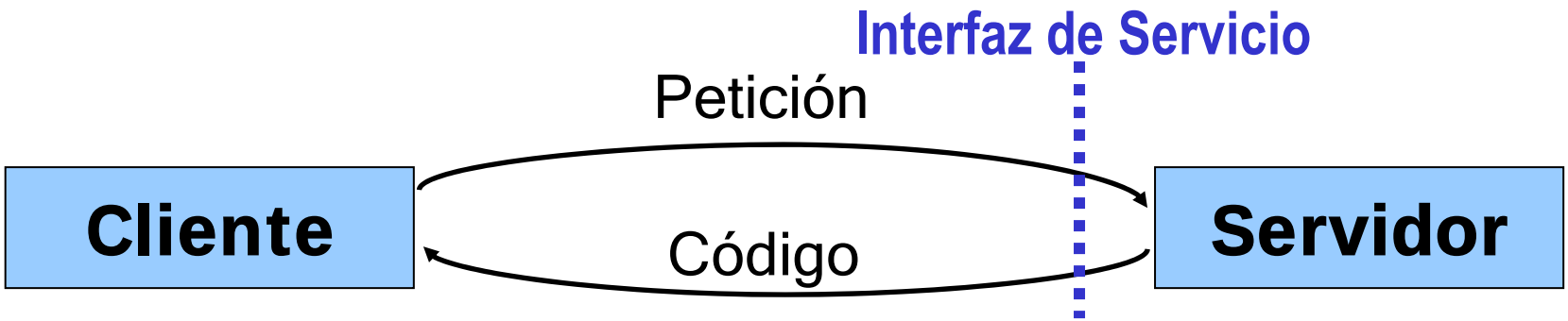
Cliente/servidor con replicación



Código móvil

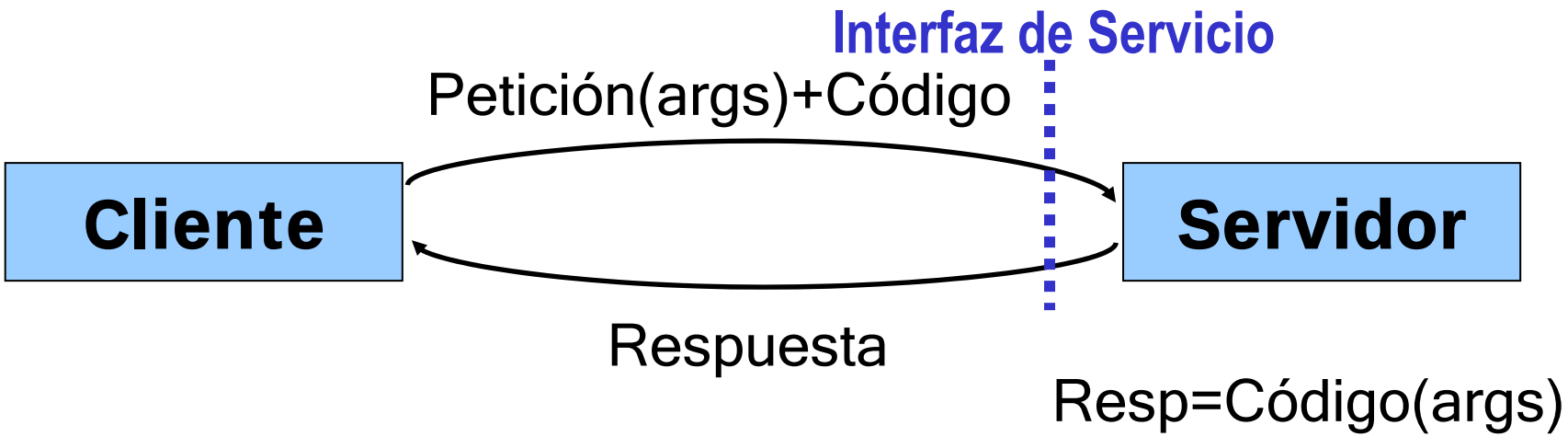
- Viaja el código en vez de los datos y/o resultados
- Requiere:
  - Arquitecturas homogéneas o
  - Interpretación de código o
  - Máquinas virtuales
- Modelos alternativos
  - Código por demanda (COD)
    - Servidor envía código a cliente
    - P.e. applets
  - Evaluación remota (REV)
    - Cliente dispone de código pero ejecución debe usar recursos servidor
    - P.ej. *Cyber-Foraging*
  - Agentes móviles
    - Componente autónomo y activo que viaja por SD

Código por demanda



Resp=Código(args)

Evaluación remota



Aspectos de diseño de cliente/servidor

Se van a considerar 5 aspectos específicos:

- Localización del servidor
- Esquemas de servicio a múltiples clientes
- Gestión de conexiones
- Servicio con estado o sin estado
- Comportamiento del servicio ante fallos

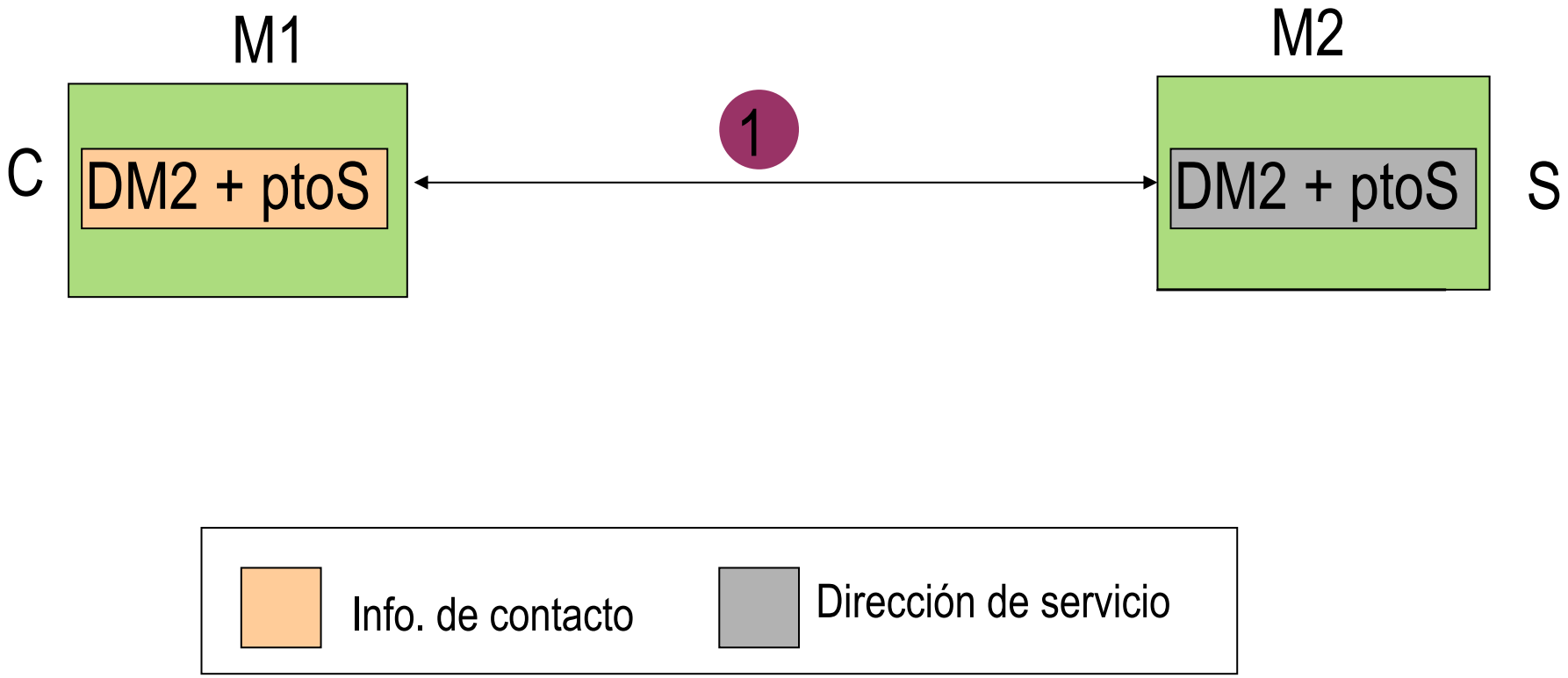
Localización del servidor

- Servidor en máquina con dirección *DMS* y usando puerto *PS*
  - ¿Cómo lo localiza el cliente? → *Binding*
  - Otro servidor proporciona esa información → problema huevo-gallina
- *Binder*: mantiene correspondencias ID servicio → (*DMS*, *PS*)
  - Cliente debe conocer dirección y puerto del *Binder*
- Características deseables de ID de servicio:
  - Ámbito global
  - Mejor nombre de texto de carácter jerárquico (como *pathname*)
  - Transparencia de ubicación
  - Posible replicación: ID servicio → (*DMS1*, *PS1*) | (*DMS2*, *PS2*) ....
  - Posibilidad de migración de servicio (incluso en mitad de un servicio)
  - Convivencia de múltiples versiones del servicio
- Suele estar englobado en un mecanismo más general
  - Servicio de nombres (tema 5): Gestiona IDs de todos los recursos

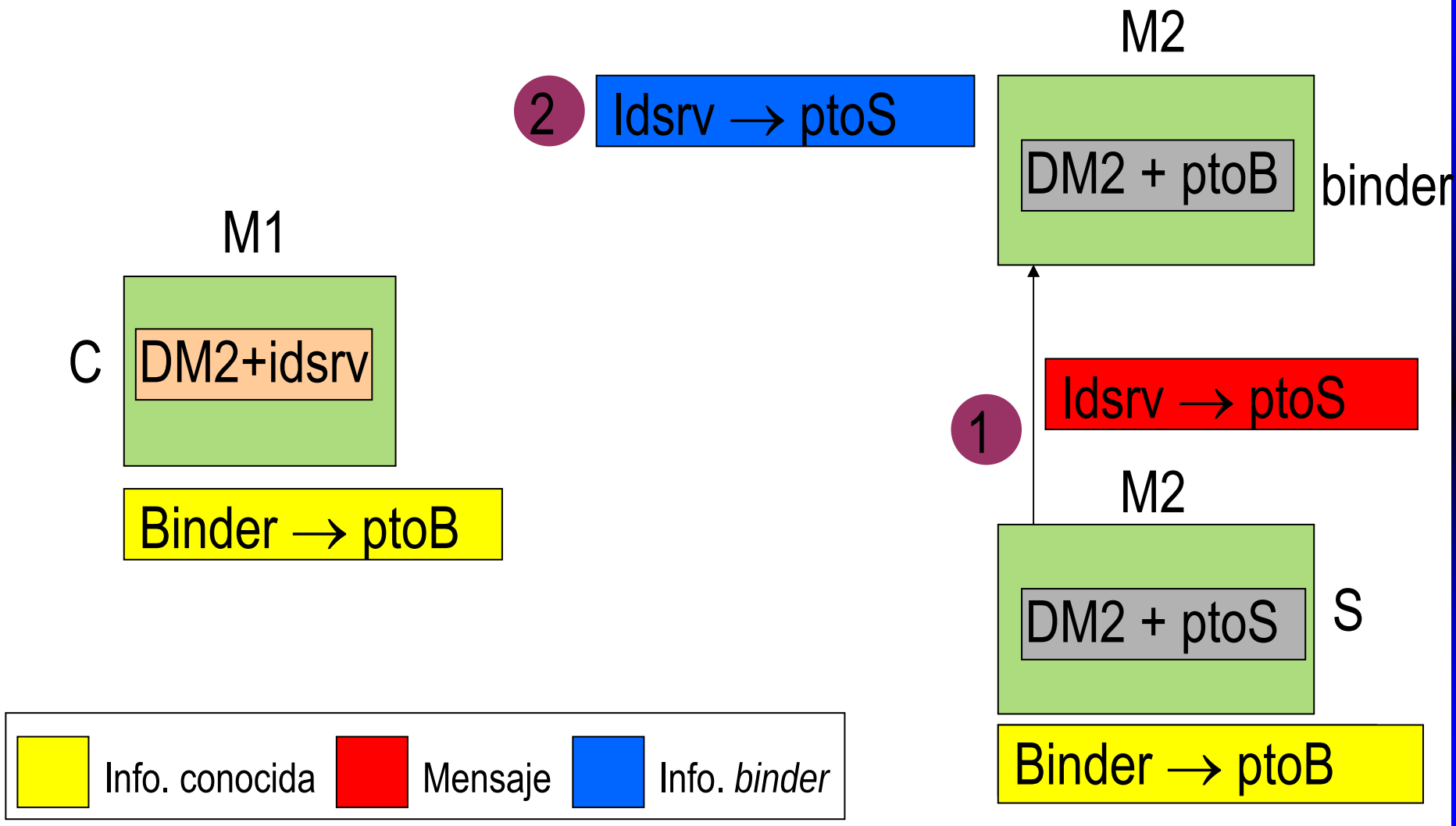
Alternativas en la ID del servicio

- Uso directo de dirección *DMS* y puerto *PS*
  - No proporciona transparencia
- Nombre servicio + dir servidor (Java RMI *Registry*, *Sun RPC*)
  - Servidor (*binder*) en cada nodo: nombre de servicio → puerto
  - Impide migración del servidor
- Nombre de servicio con ámbito global (DCE, CORBA, Mach)
  - Servidor con ubicación conocida en el sistema
  - Opción 1. Sólo *binder* global: nombre de servicio → [*DMS*+*PS*]
  - Opción 2. *binder* global (BG) + *binder* local (BL) en puerto conocido
    - BG: ID → [*DMS*] ; BL: ID → [*PS*]
  - Uso de caché en clientes para evitar repetir traducción
  - Puede haber nivel adicional para facilitar migración durante servicio
    - nombre de servicio → [ID binario interno] → [*DMS*+*PS*]
    - Necesidad de localización: *Broadcast* o Servidor de localización

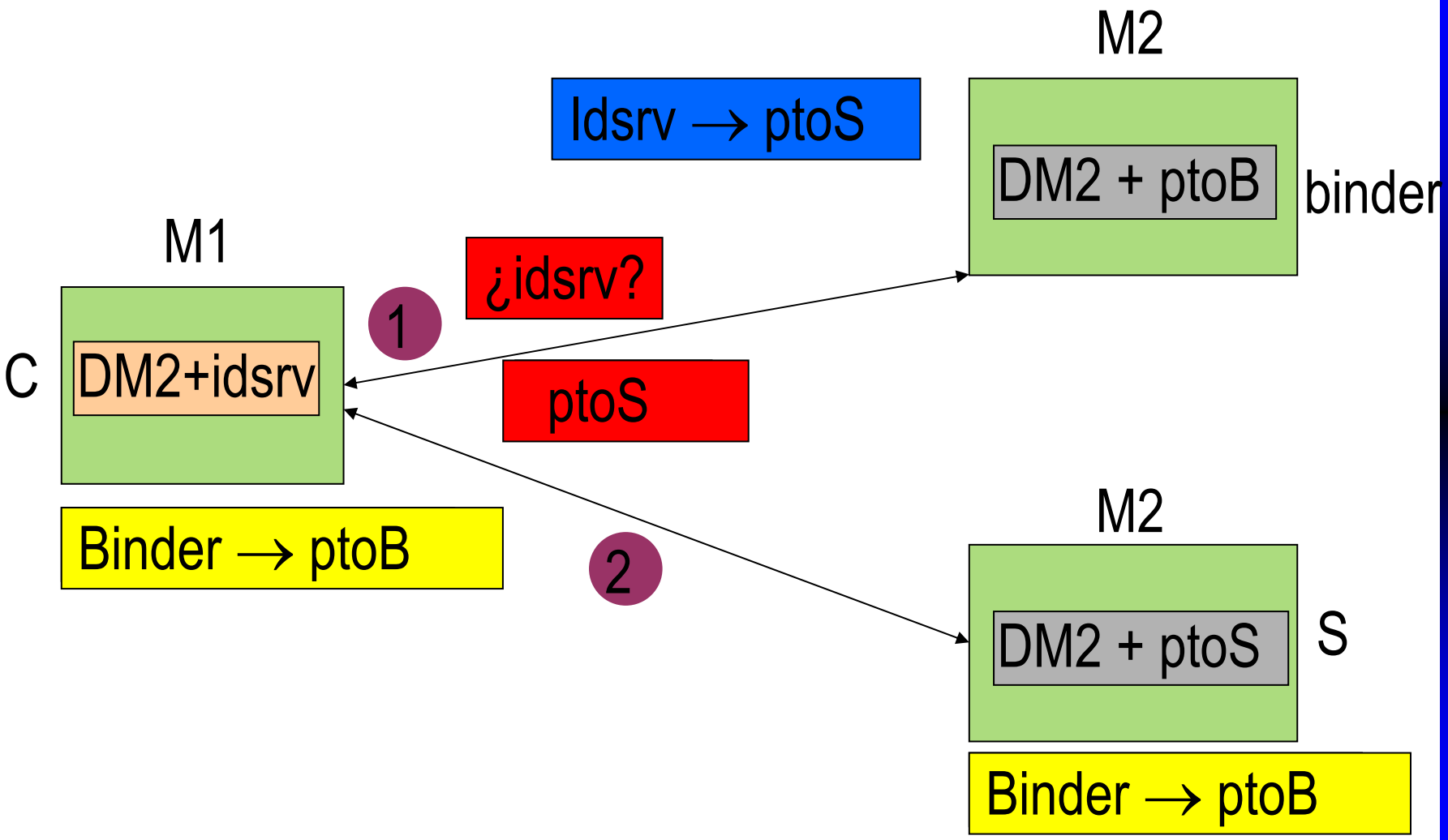
ID servicio = [DM+pto]



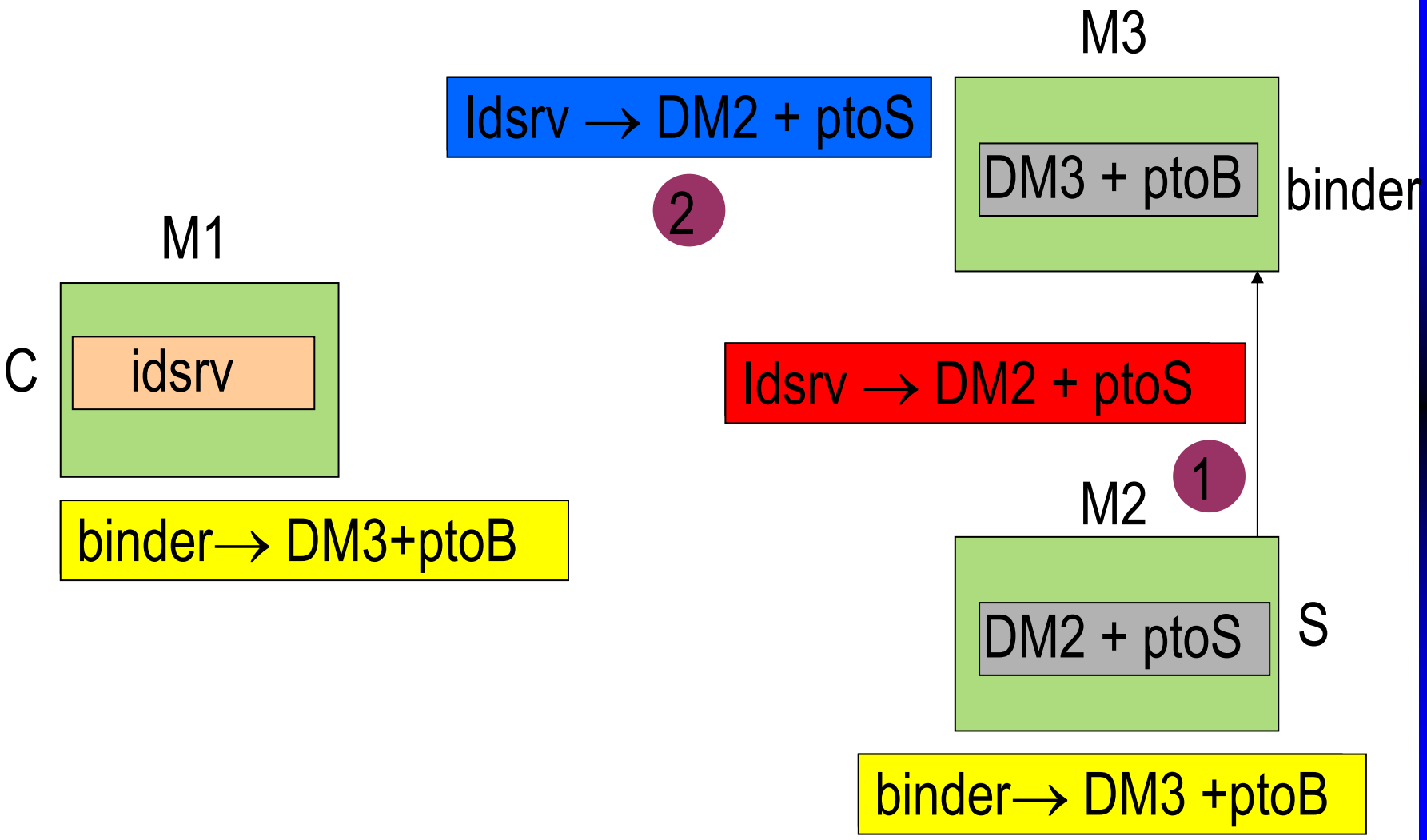
ID servicio = [DM+idsrv]: Alta



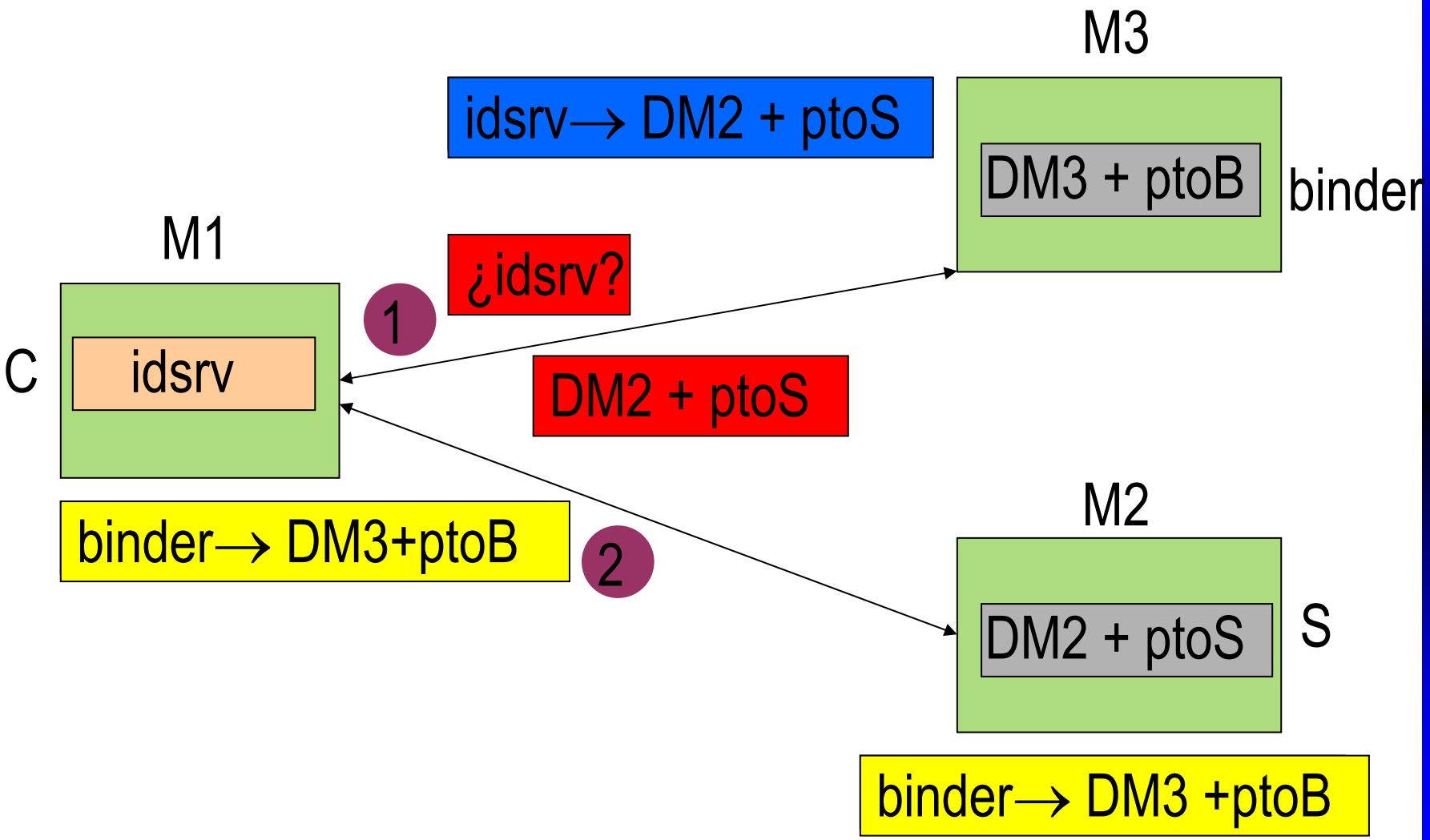
ID servicio = [DM+idsrv]: Consulta



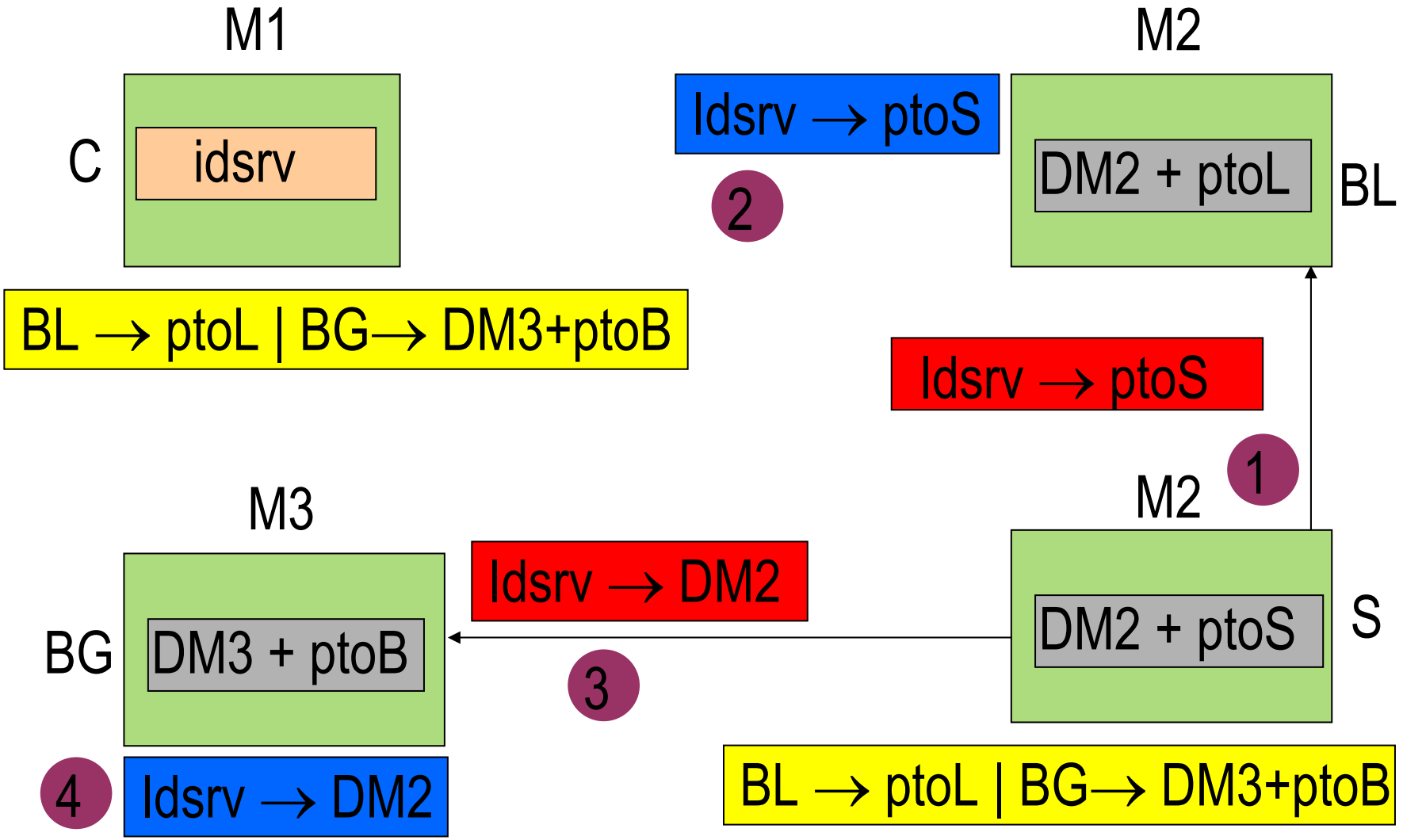
ID servicio = [idsrv]; Sólo BG: Alta



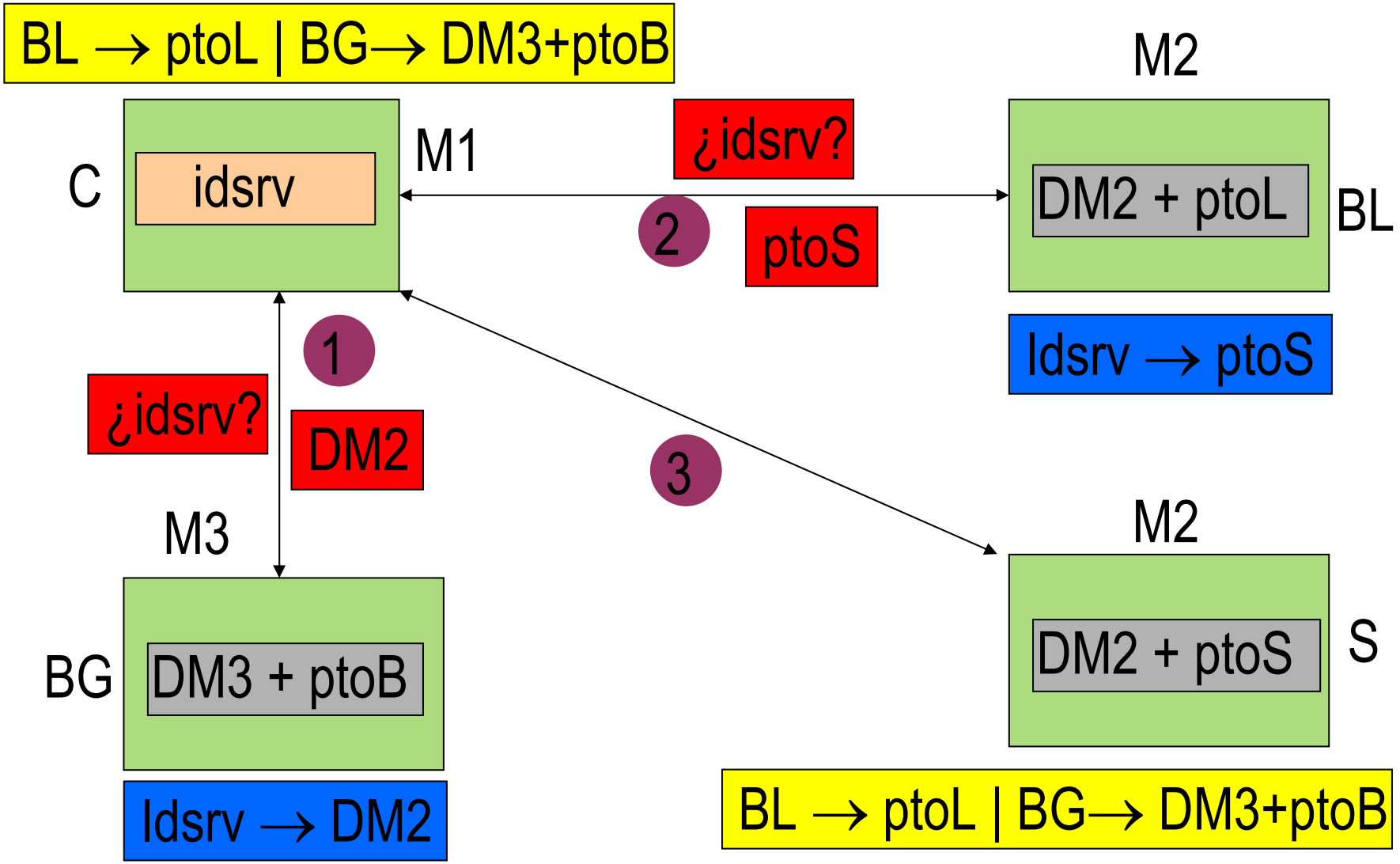
ID servicio = [idsrv]; Sólo BG: Consulta



ID servicio = [idsrv]; BG+BL: Alta



ID servicio = [idsrv]; BG+BL: Consulta



Binding

- Caso con BG y BL + versiones:
  - Servidor:
    - Elige puerto local
    - Informa a *binder* local del alta:
      - (id. servicio + versión) = puerto
    - Informa a *binder* global del alta:
      - (id. servicio + versión) = dir máquina
    - Al terminar, notifica la baja a ambos *binder* :
      - Ambos eliminan (id. servicio + versión)
  - Cliente:
    - Consulta a *binder* global
      - (id. servicio + versión) → dir. máquina
    - Consulta a *binder* en dir. máquina
      - (id. servicio + versión) → puerto

Servicio a múltiples clientes

- Servidor secuencial
  - Único flujo de ejecución atiende múltiples peticiones
    - Operaciones asíncronas y/o espera por múltiples eventos (*select/poll*)
  - Uso de “máquina de estados” para seguimiento de clientes
  - Solución compleja y que no aprovecha paralelismo HW
- Servidor concurrente
  - Solución más natural y que aprovecha paralelismo HW
  - *Threads* (**T**) vs. Procesos (**P**)
    - Generalmente *threads*: Más ligeros y comparten más recursos

Servicio concurrente: alternativas

- Creación dinámica de *T/P* según llegan peticiones
  - Sobrecarga
- Conjunto de *N T/P* pre-arrancados:
  - Al finalizar trabajo, en espera de más peticiones
  - Poca carga $\rightarrow$ gasto innecesario
  - Mucha carga $\rightarrow$ insuficientes
- Esquema híbrido con mínimo *m* y máximo *M*
  - *m* pre-arrancados; $m \leq T/P \leq M$
  - Si petición, ninguno libre y nº < *M* $\rightarrow$ se crea
  - Si inactivo tiempo prefijado y nº > *m* $\rightarrow$ se destruye

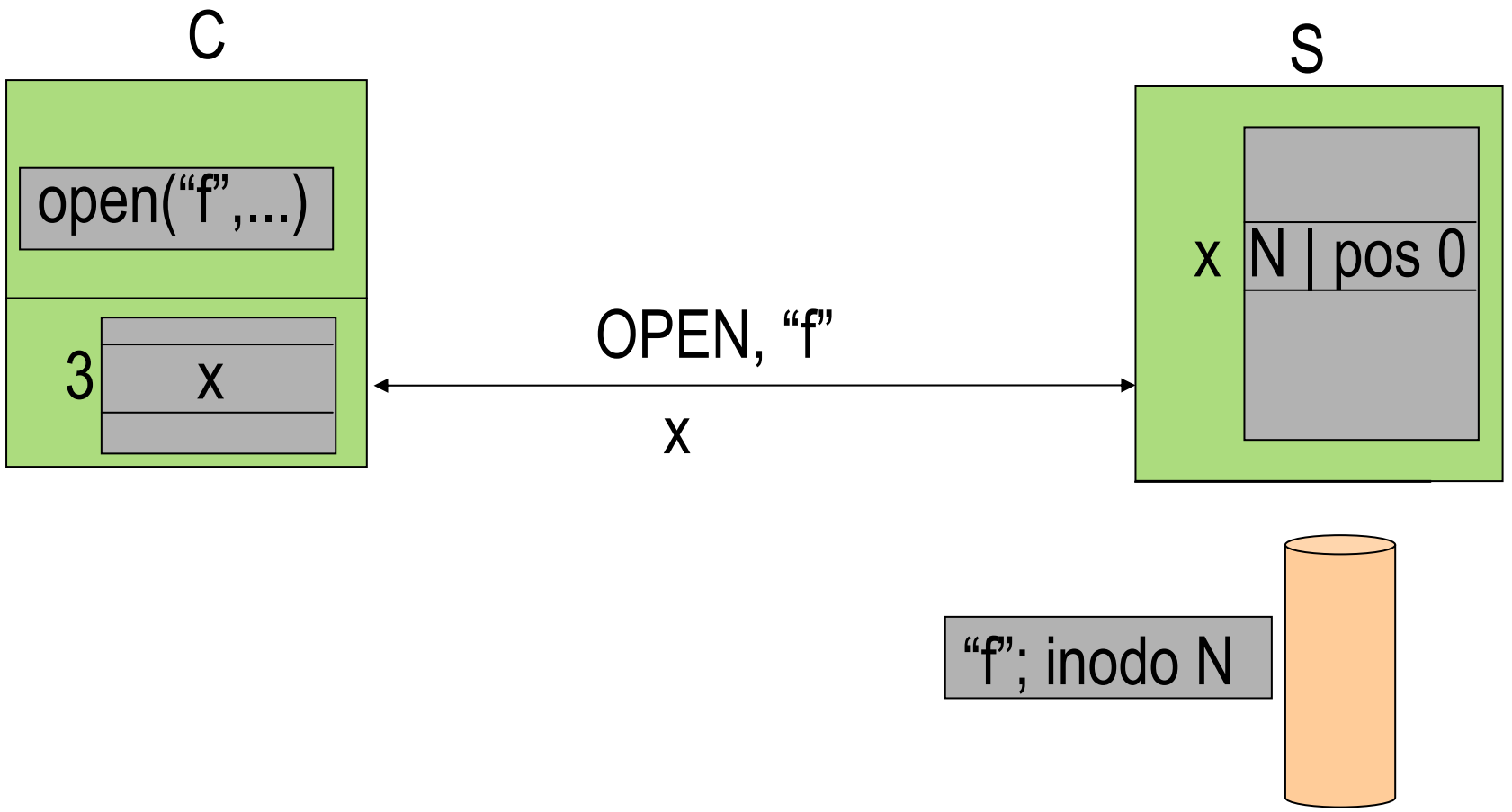
Gestión de conexiones

- En caso de que se use un esquema con conexión
- 1 conexión por cada petición
  - 1 operación cliente-servidor
    - conexión, envío de petición, recepción de respuesta, cierre de conexión
  - Más sencillo pero mayor sobrecarga (¡9 mensajes con TCP!)
  - Propuestas de protocolos de transporte orientados a C/S (T/TCP)
- Varias peticiones de cliente usan misma conexión
  - Más complejo pero menor sobrecarga
  - Esquema usado en HTTP/1.1
    - Además, *pipeline* de peticiones
  - Implica que servidor mantiene un estado

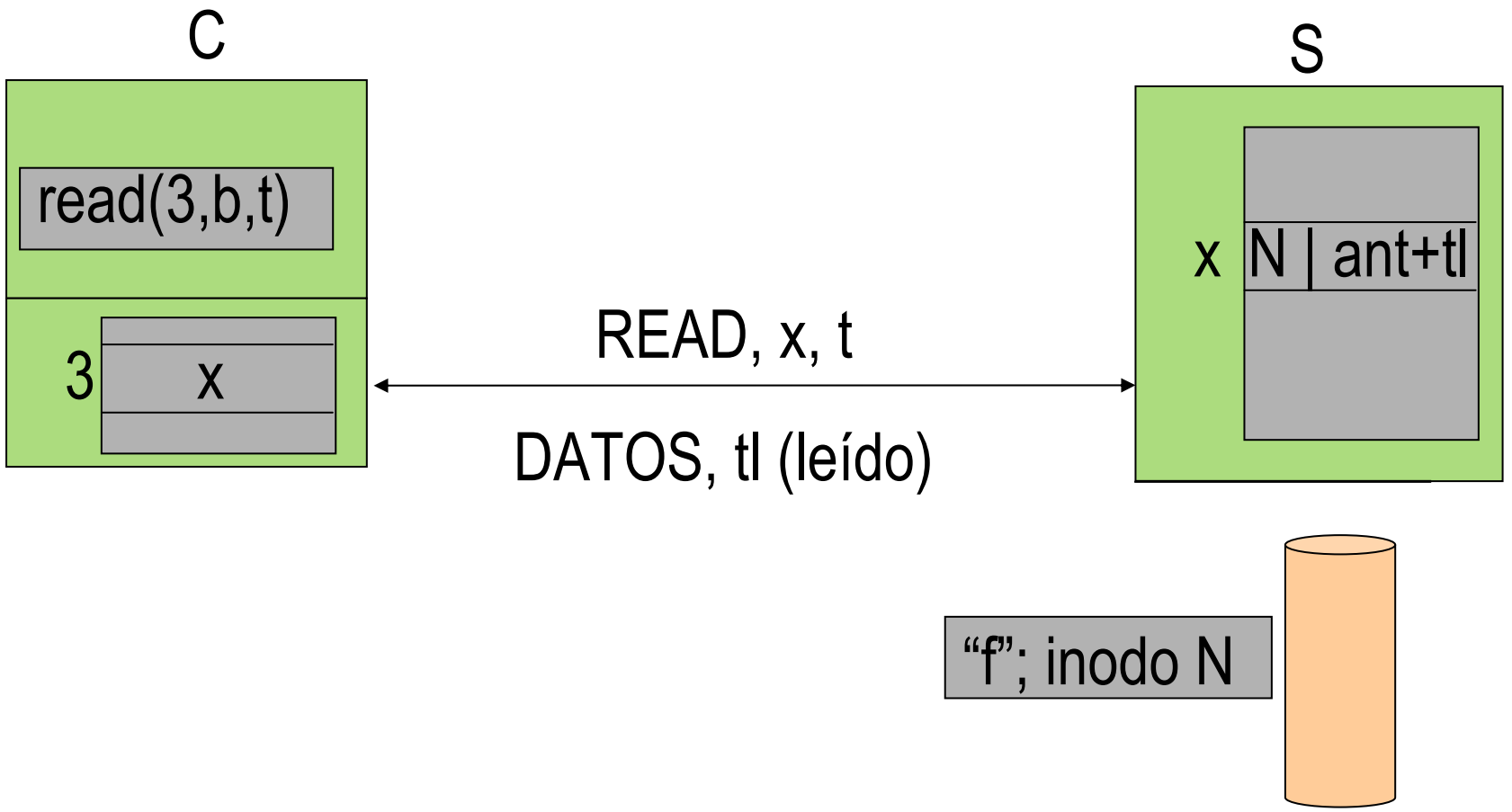
Servicio con/sin estado

- ¿Servidor mantiene información de clientes?
- Ventajas de servicio con estado (*aka* con sesión remota):
  - Mensajes de petición más cortos
  - Mejor rendimiento (se mantiene información en memoria)
  - Favorece optimización de servicio: estrategias predictivas
- Ventajas de servicio sin estado:
  - Más tolerantes a fallos (ante rearranque del servidor)
    - Peticiones autocontenidas.
  - Reduce nº de mensajes: no comienzos/finales de sesión.
  - Más económicos para servidor (no consume memoria)
- Servicio sin estado base de la propuesta REST
- Estado sobre servicios sin estado
  - Cliente almacena estado y lo envía al servidor (p.e. HTTP+*cookies*)

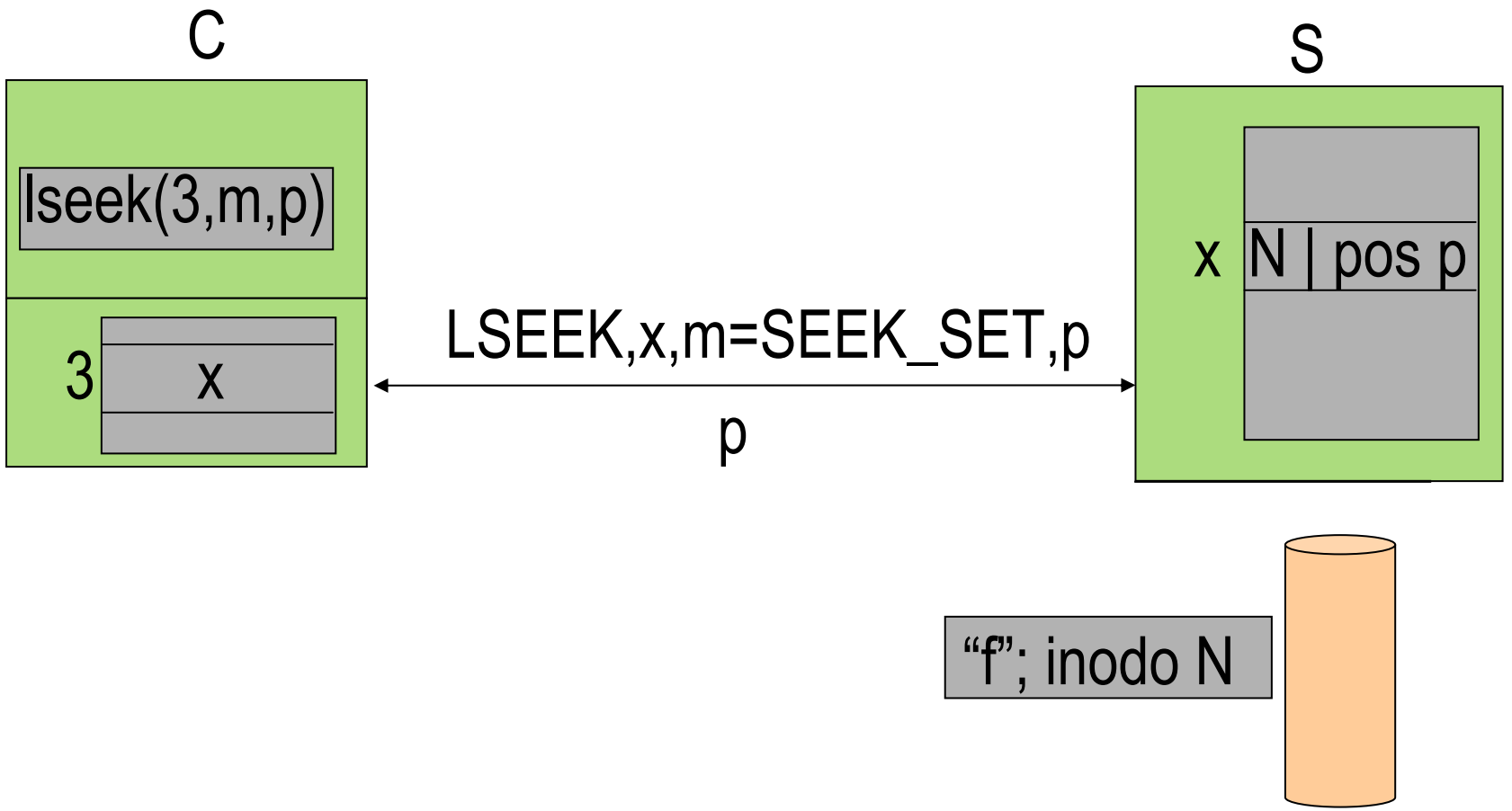
Servicio de ficheros con estado: OPEN



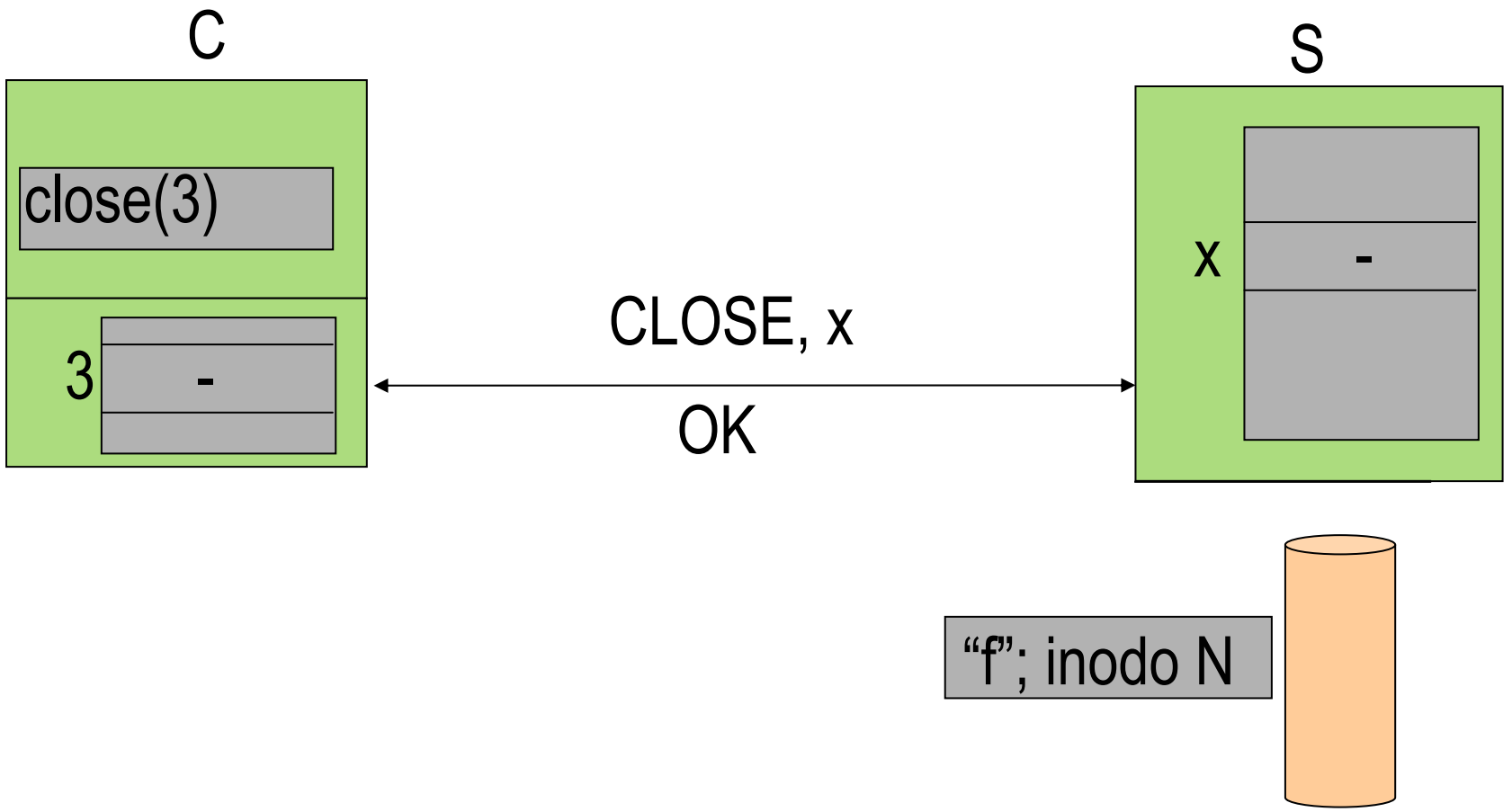
Servicio de ficheros con estado: READ



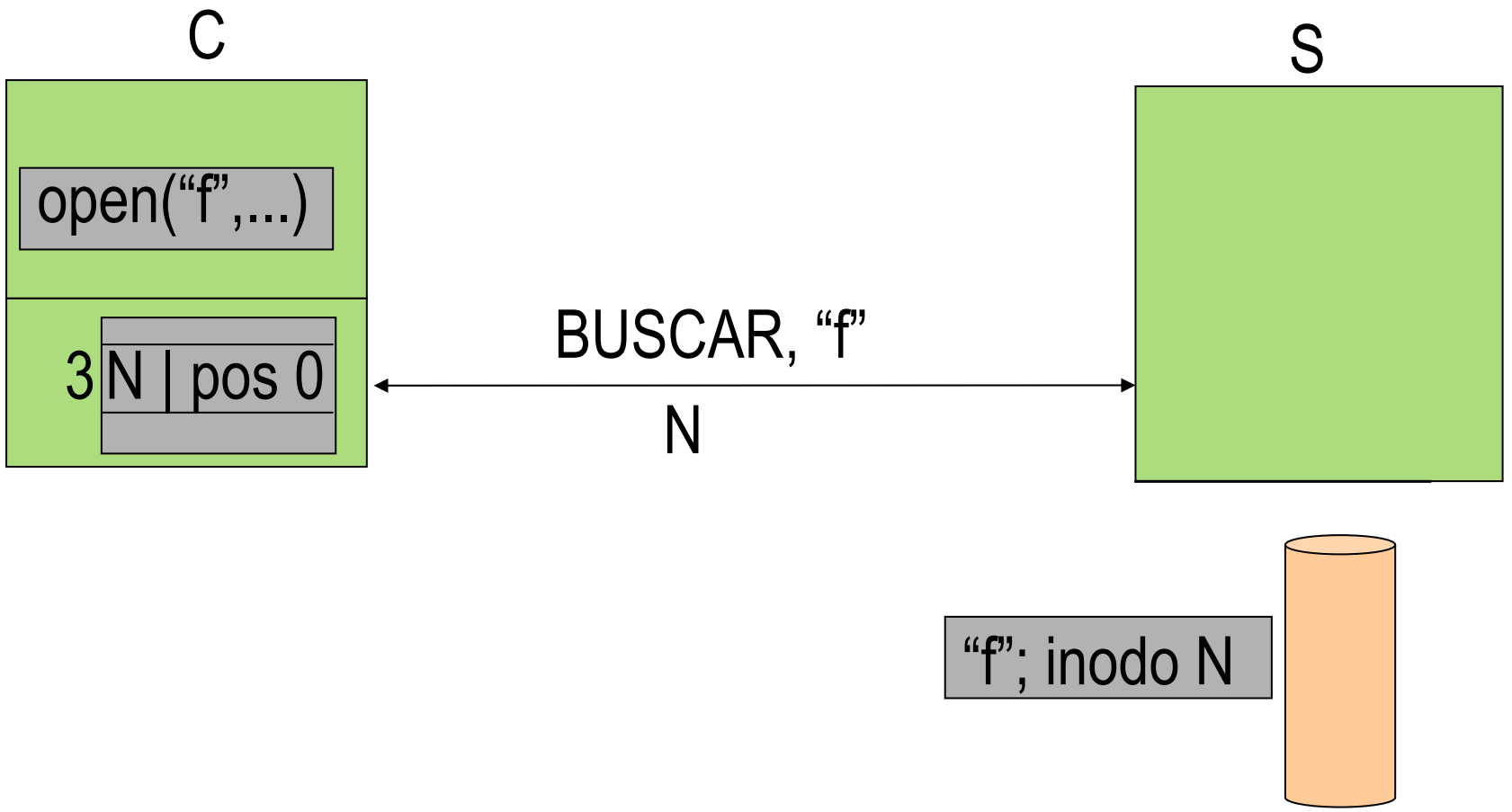
Servicio de ficheros con estado: LSEEK



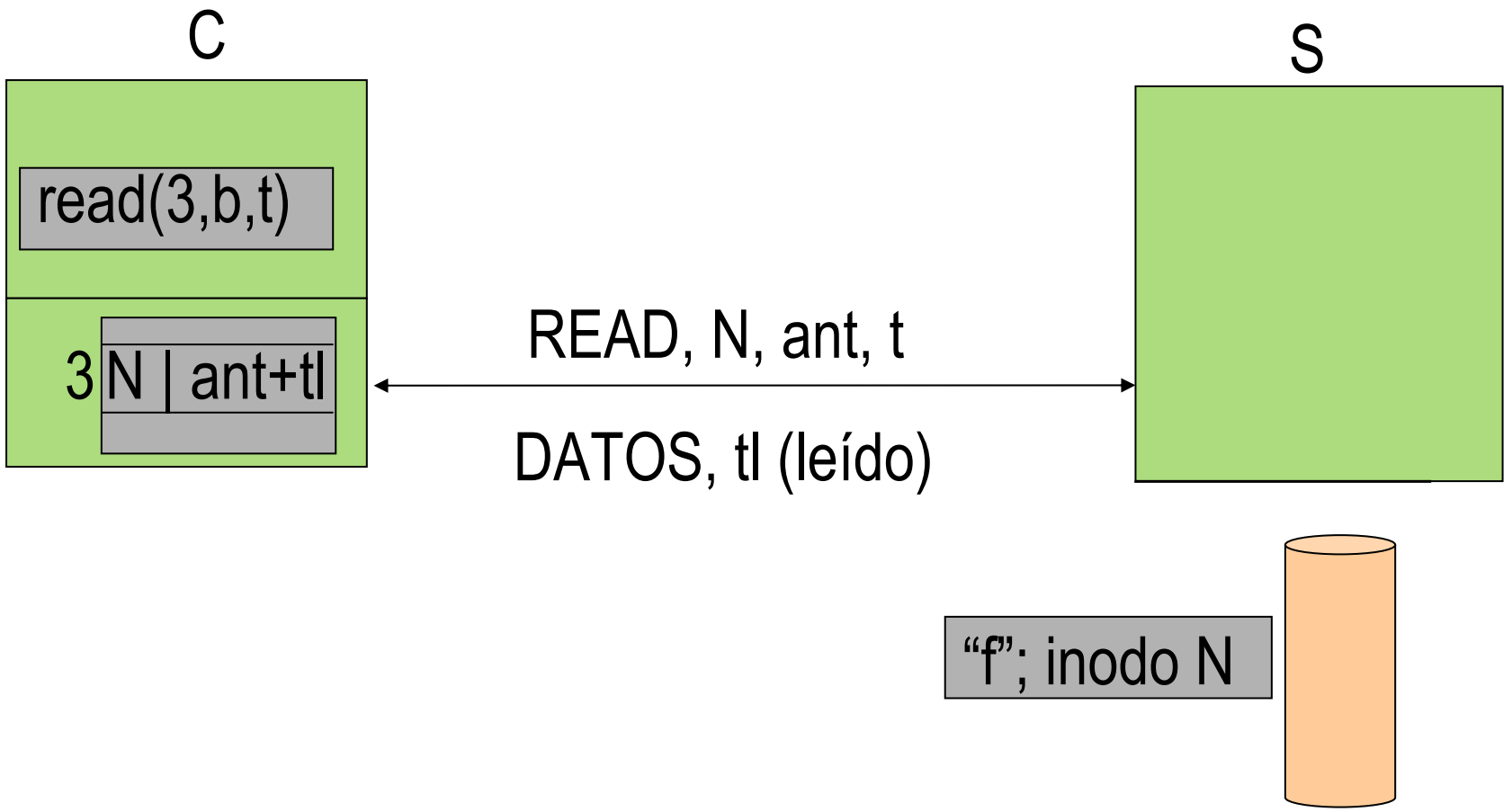
Servicio de ficheros con estado: CLOSE



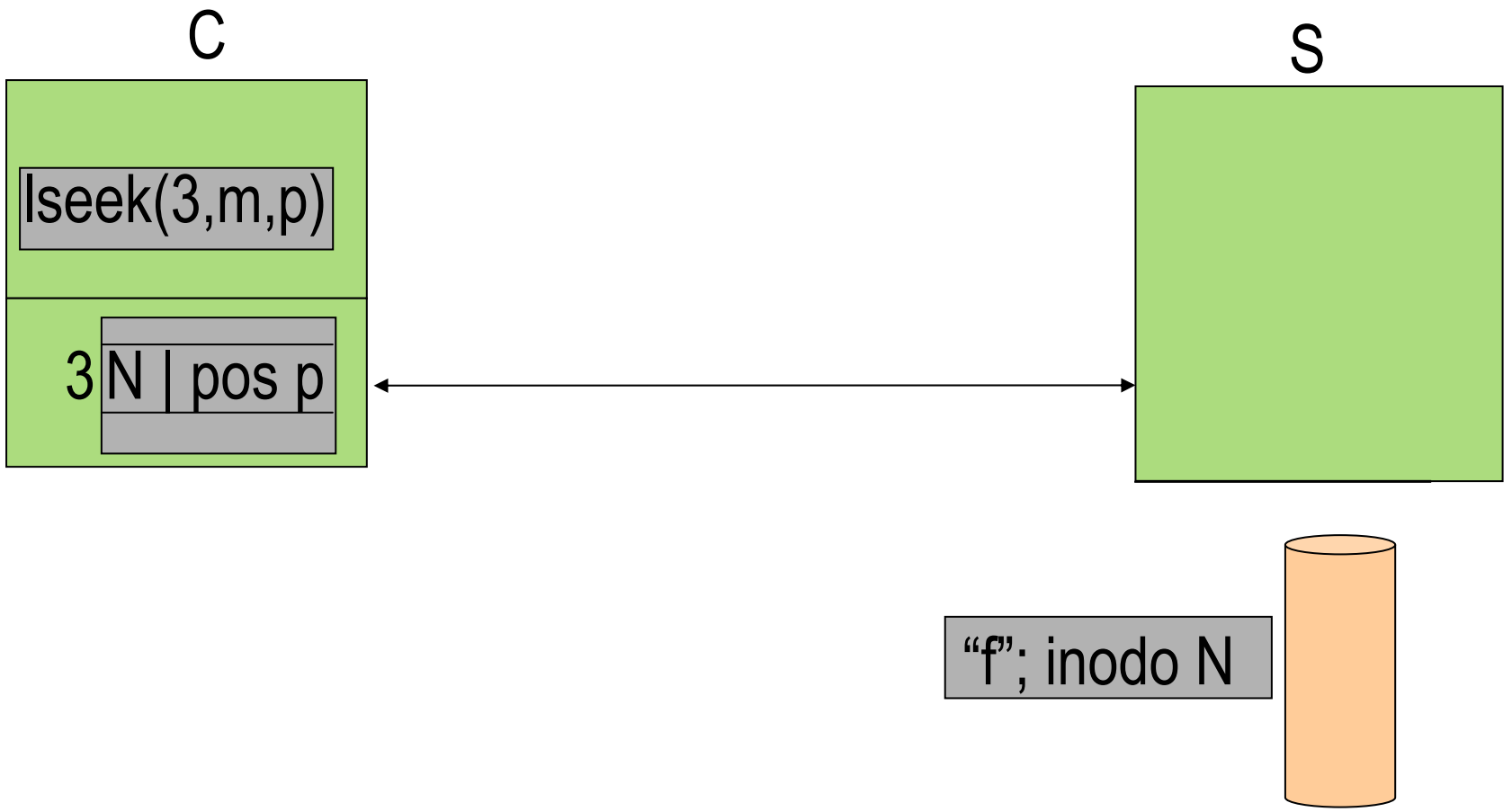
Servicio de ficheros sin estado: OPEN



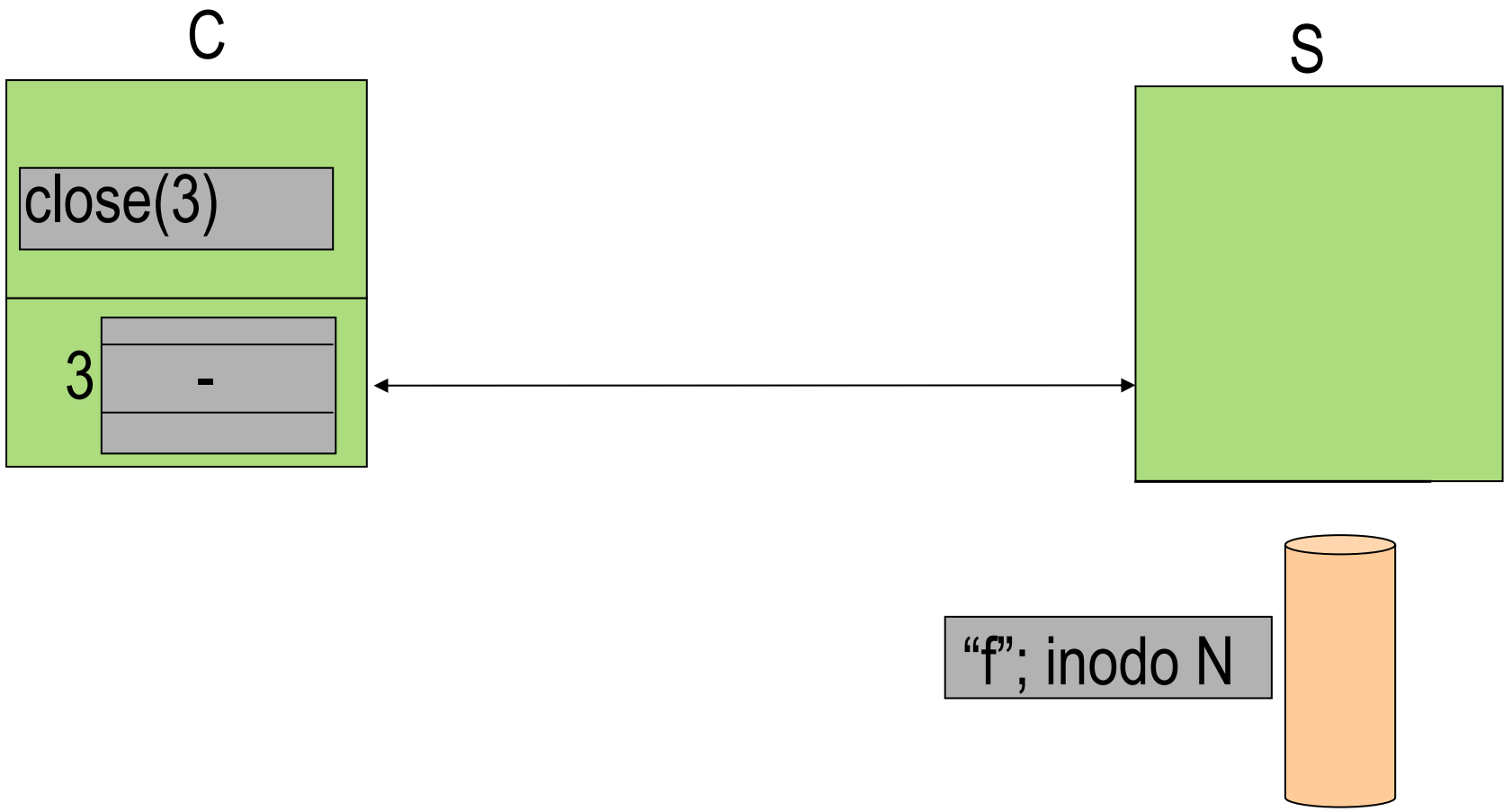
Servicio de ficheros sin estado: READ



Servicio de ficheros sin estado: LSEEK



Servicio de ficheros sin estado: CLOSE



REST (*Representational state transfer)*

- Éxito de Web vs. sistemas con interfaces específicos
- Arquitectura abstracta C/S base de HTTP/1.1 (Fielding)
- Características principales
  - Servicios sin estado
  - Interfaz uniforme centrado en recursos en vez de acciones
    - Recursos con ID único (p.e. URI para la web)
  - Sistema jerárquico: conectores transparentes entre cliente y servidor
- Beneficios
  - Facilita integración y despliegue independiente de componentes
  - Facilita incorporación de técnicas de *caching* o políticas de seguridad

Convenios uso HTTP en servicio RESTful

- Ops. **C**reate**R**etrieve**U**pdate**D**elete → Ops. HTTP
- URI representa un recurso o una colección de recursos
- GET (C**R**UD)
  - Si URI representa recurso → Lo obtiene
  - Si URI representa colección → obtiene URIs miembros de colección
- DELETE (CRU**D**): Borra recurso o colección
- PUT (**C**R**U**D): Crea (sobrescribe) recurso o colección
- ¿POST (**C**R**U**D)? PUT vs. POST → asunto polémico y confuso
  - URI de PUT identifica el recurso que se quiere crear /sobrescribir
  - URI de POST identifica recurso que manejará contenido de POST
    - Como parte de procesado puede crear/actualizar recurso
  - PUT sustituye completamente contenido previo de recurso
  - POST (no idempotente) permite actualizaciones parciales

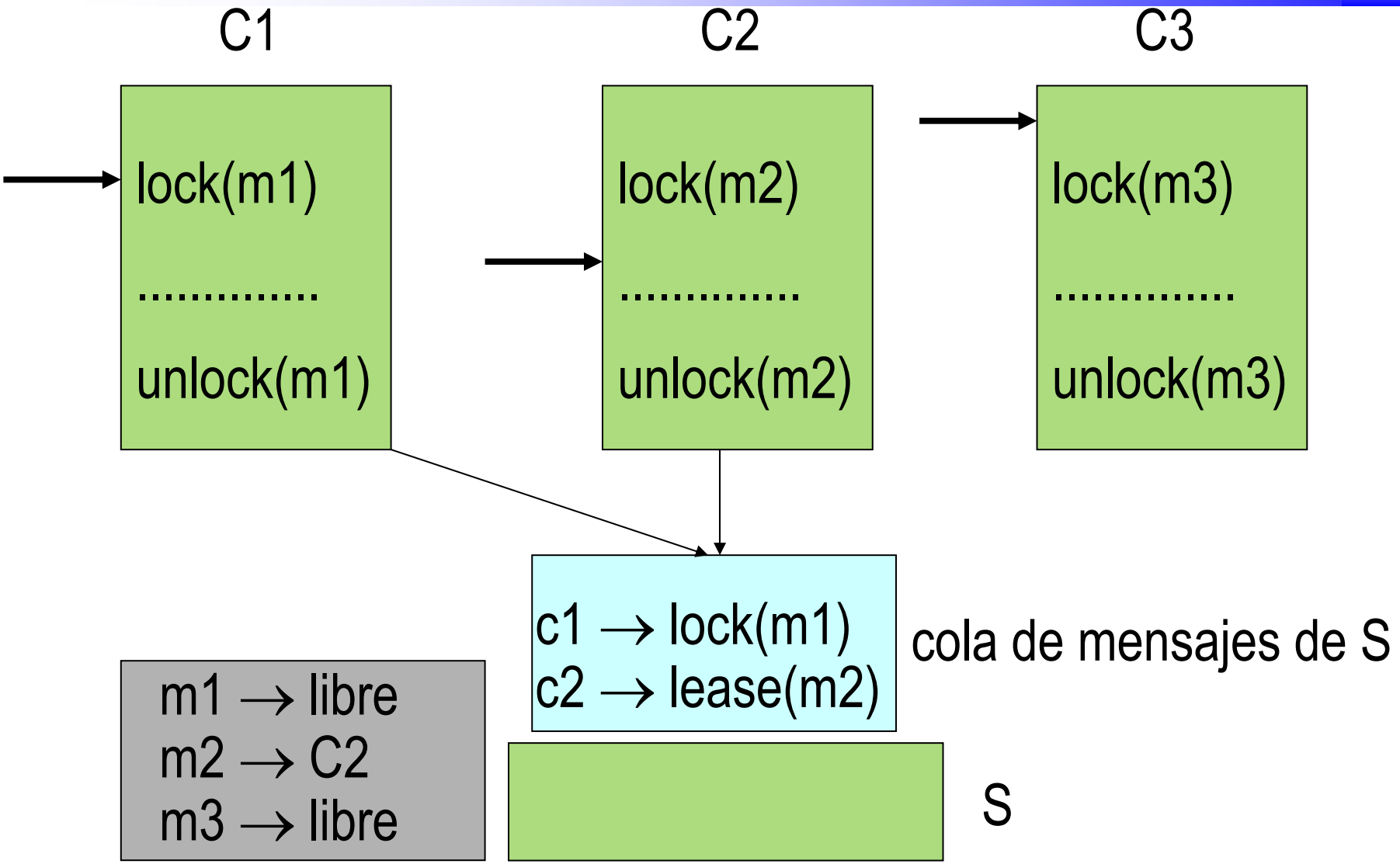
Leases

- Mecanismo para mejorar tolerancia a fallos en SD
  - Detección y tratamiento de caídas de clientes y servidores
- Modo de operación
  - Servidor otorga a cliente un *lease* que dura un plazo de tiempo
  - Cliente debe renovar *lease* antes de fin de plazo
- Aplicación típica (genérica) de *leases*:
  - Servidor gestiona algún tipo de recurso vinculado con un cliente
    - Excepto por *leases*, cliente no tiene por qué contactar con servidor
  - Si cliente cae y no renueva el *lease*, recurso "abandonado"
  - Si servidor cae, en reinicio obtiene renovaciones
    - Puede "reconstruir" los recursos
- No confundir con un simple temporizador
  - Cliente envía petición a servidor y arranca temporizador
    - Si se cumple antes de ACK, vuelve a enviar petición (≠ *lease*)

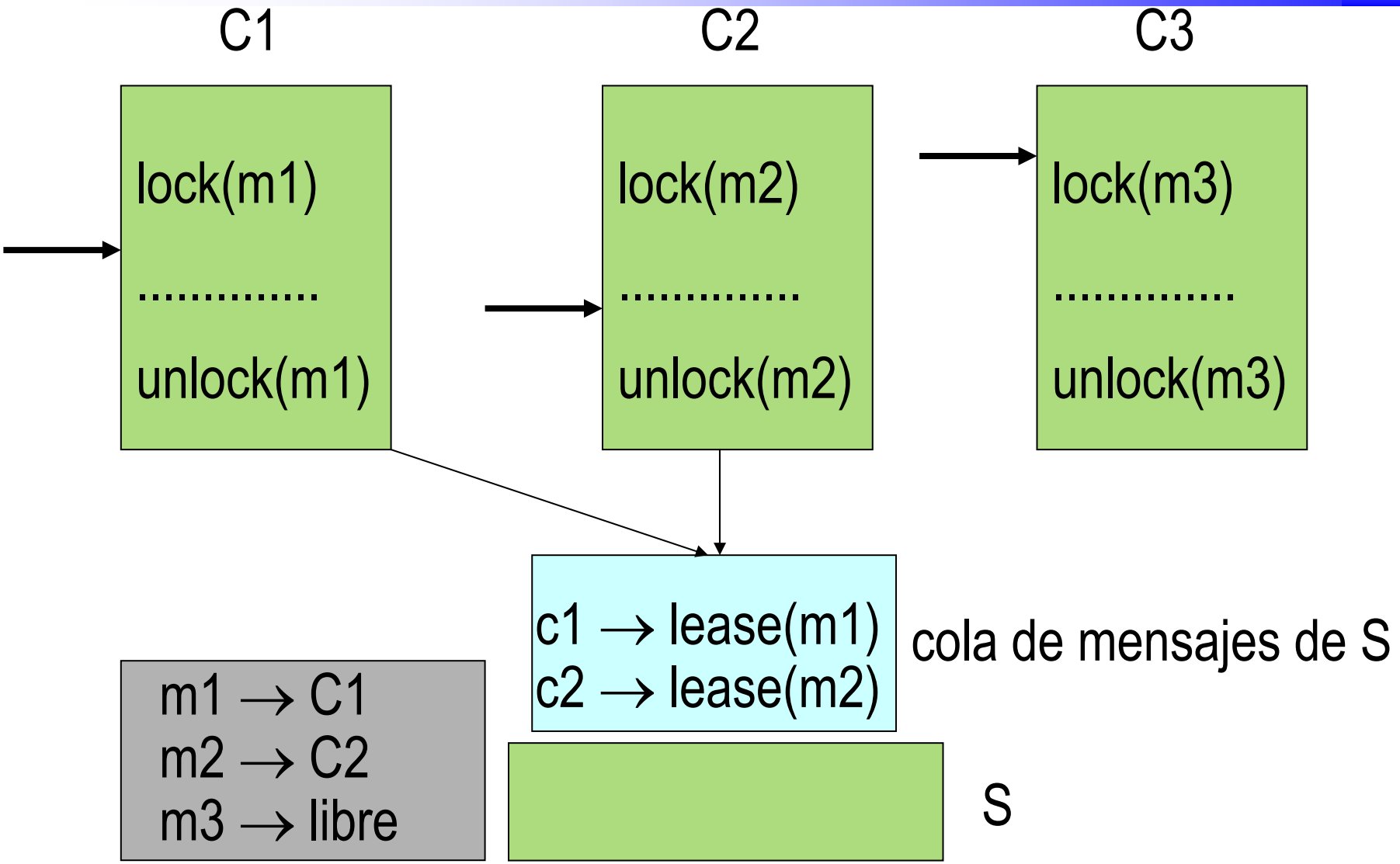
Leases en servicios con estado

- Algunos servicios son inherentemente con estado
  - P. ej. cerrojos distribuidos
- Uso de *leases* en servicio de cerrojos distribuido
  - Servidor asigna *lease* a cliente mientras en posesión de cerrojo
  - Clientes en posesión de cerrojos deben renovar su *lease*
  - Rearranque de *S*: debe procesar primero peticiones de renovación
    - Tiempo de reinicio de servicio > tiempo de renovación
  - Reconstrucción automática de estado después de rearranque de *S*
  - Caída de cliente: falta de renovación
    - Revocación automática de cerrojos de ese cliente

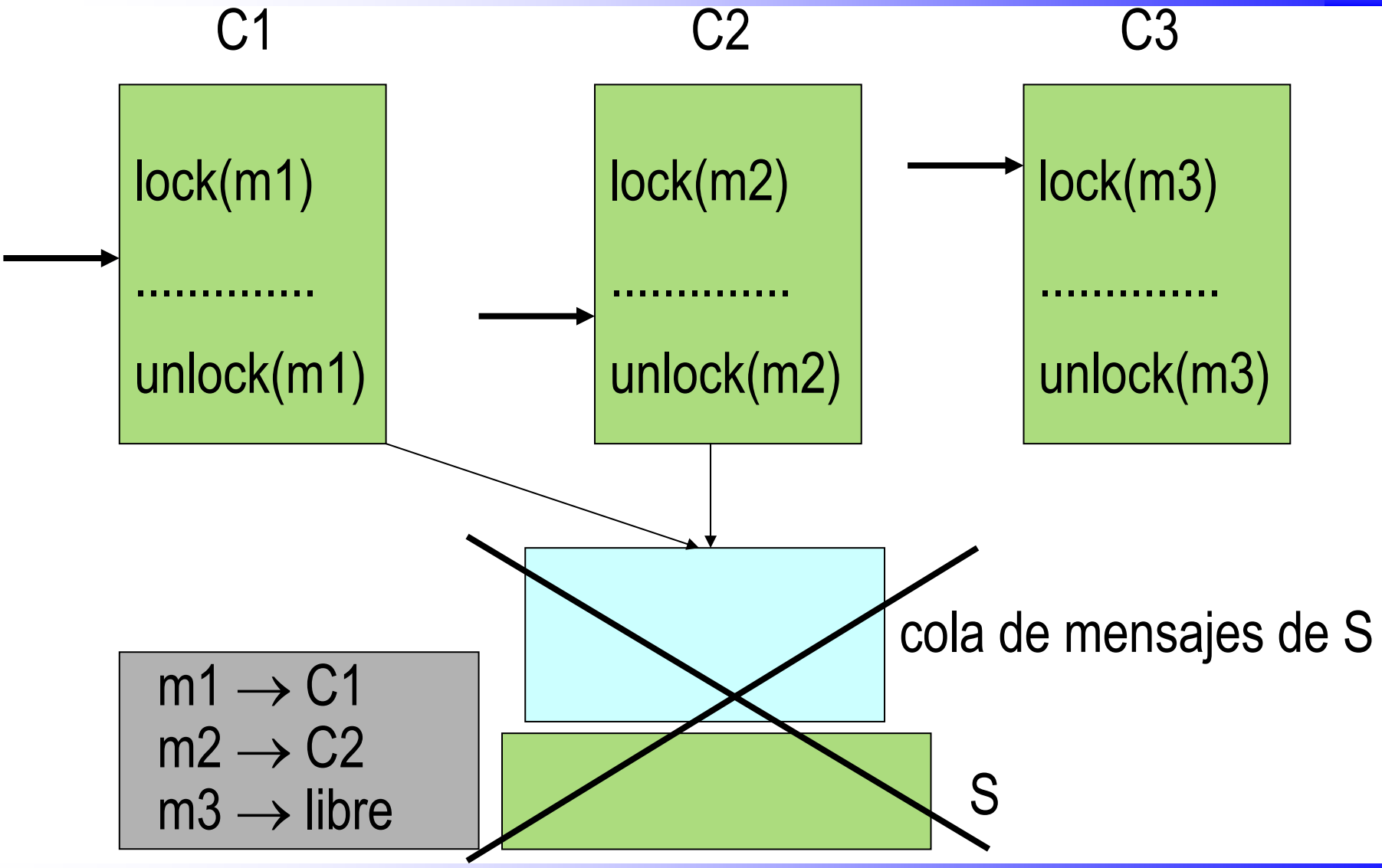
Serv. cerrojos con estado: *leases* (1)



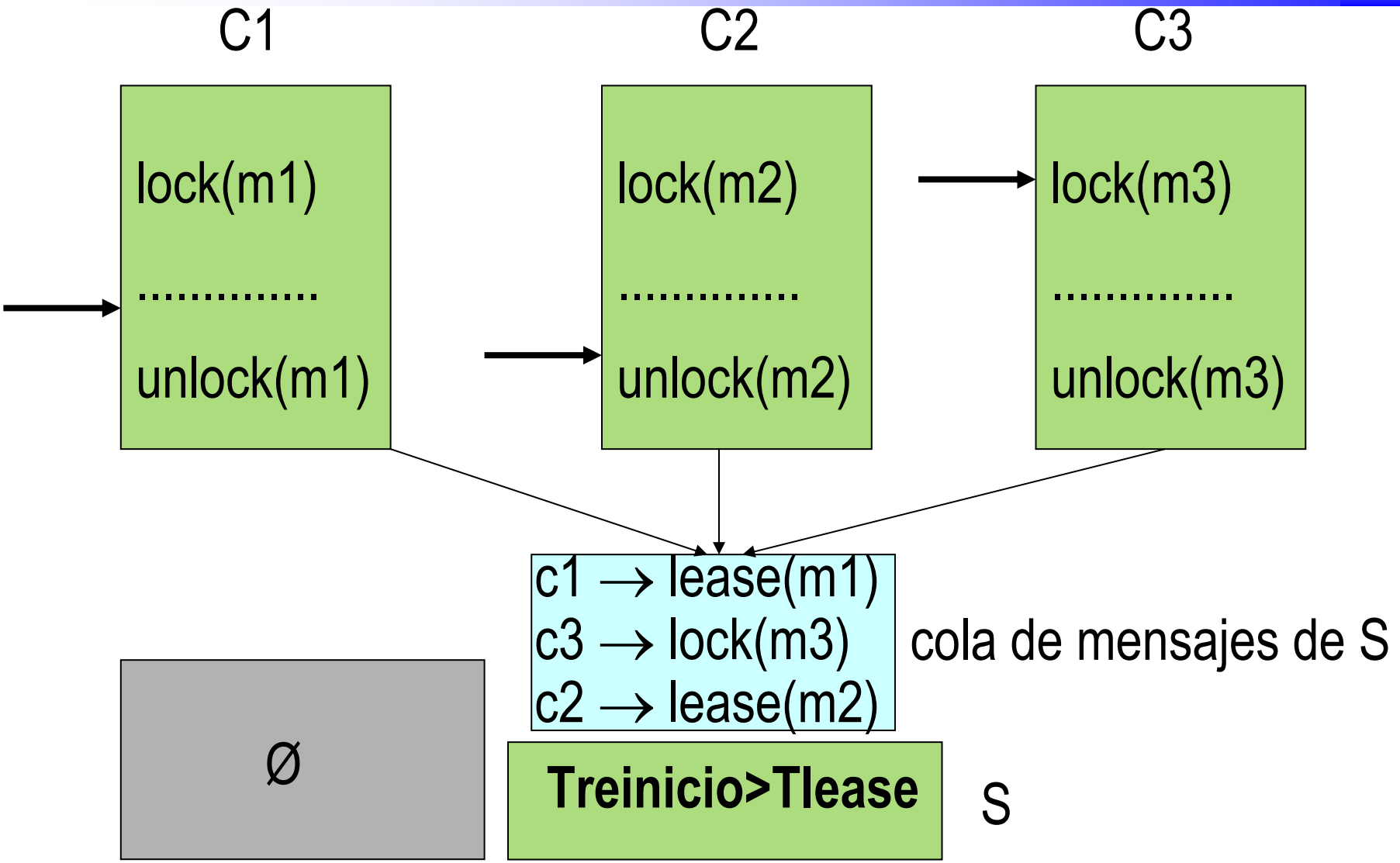
Serv. cerrojos con estado: *leases* (2)



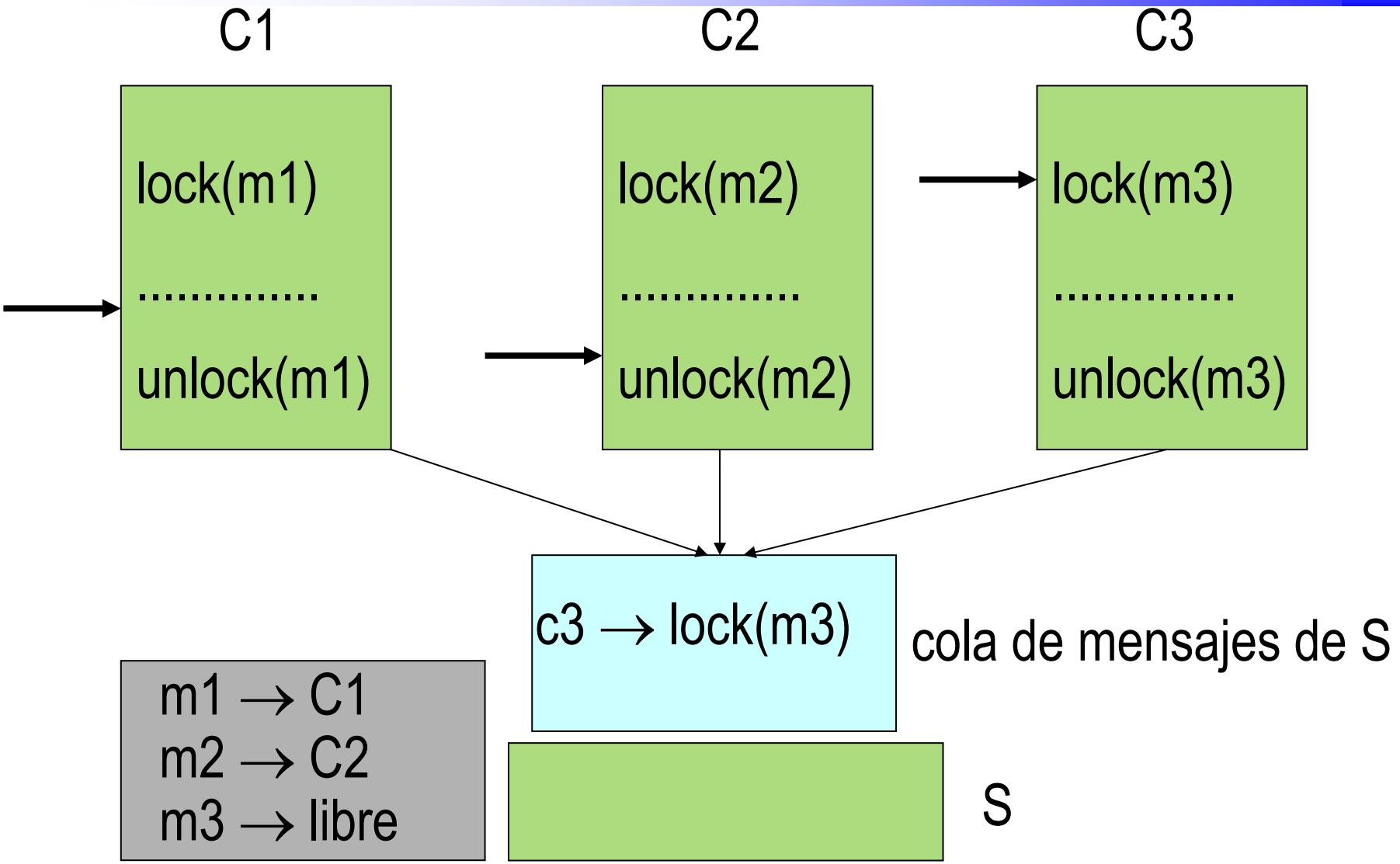
Serv. cerrojos con estado: *leases* (3)



Serv. cerrojos con estado: *leases* (4)



Serv. cerrojos con estado: *leases* (5)



Comportamiento del servicio ante fallos

- ¿Qué se garantiza con respecto al servicio ante fallos?
  - Nada: Servicio puede ejecutar 0 a N veces
  - Al menos una vez (1 a N veces)
  - Como mucho una vez (0 ó 1 vez)
  - Exactamente una vez: Sería lo deseable
- Operaciones repetibles (**idempotentes**)
  - Cuenta += cantidad (**NO**)
  - Cuenta = cantidad (**SÍ**)
- Operación idempotente + al menos 1 vez ≈ exactamente 1
- Tipos de fallos:
  - Pérdida de petición o de respuesta (sólo si comunicación no fiable)
  - Caída del servidor
  - Caída del cliente

Fallos con comunicación fiable

- Si cae servidor no siempre puede saber si ejecutado servicio
- Semántica de como mucho una vez
  - Si llega respuesta, se ha ejecutado exactamente una vez
  - Si no llega (servidor caído), se ha ejecutado 0 ó 1 vez
- Para semántica al menos una vez (con ops. idempotentes)
  - Retransmitir hasta respuesta (servidor se recupere) o fin de plazo
  - Usar un sistema de transacciones distribuidas

Fallos con comunicación no fiable

- Pérdida de petición/respuesta
  - Si no respuesta, retransmisión cuando se cumple plazo
  - Nº de secuencia en mensaje de petición
  - Si pérdida de petición, retransmisión no causa problemas
  - Si pérdida de respuesta, retransmisión causa re-ejecución
    - Si operación idempotente, no es erróneo pero gasta recursos
    - Si no, es erróneo
  - Se puede guardar histórico de respuestas (caché de respuestas):
    - Si nº de secuencia duplicado, no se ejecuta pero manda respuesta
- Caída del servidor
  - Si llega finalmente respuesta, semántica de al menos una vez
  - Si no llega, no hay ninguna garantía (0 a N veces)

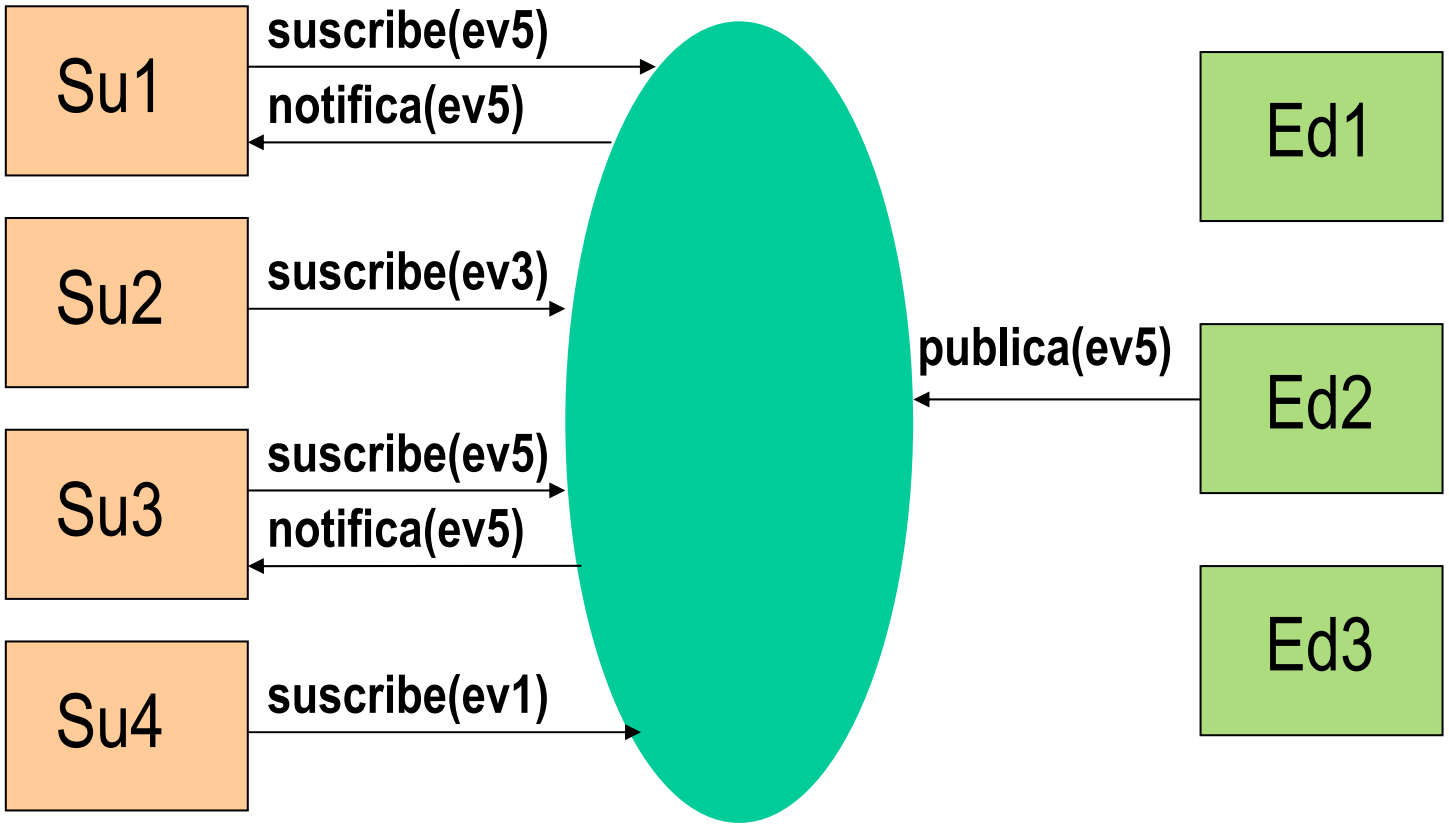
Caída del cliente

- Menos “traumática”: problema de computación huérfana
  - Gasto de recursos inútil en el servidor
- Alternativas:
  - Uso de épocas:
    - Peticiones de cliente llevan asociadas un nº de época
    - En rearranque de cliente *C*: transmite (++nº de época) a servidores
    - Servidor aborta servicios de *C* con nº de época menor
  - Uso de *leases*:
    - Servidor asigna *lease* mientras dura el servicio
    - Si cliente no renueva *lease* se aborta el servicio
- Abortar un servicio no es trivial
  - Puede dejar incoherente el estado del servidor (p.e. cerrojos)

Modelo editor/subscriptor

- Sistema de eventos distribuidos (Jini, s. eventos/notifi. CORBA)
- Suscriptor *S* (*subscriber*) muestra interés por eventos
  - Se subscribe a ciertos eventos: filtro por tipo, por tema, por contenido
- Editor *E* (*publisher*) genera un evento
  - Se envía a subscriptores interesados en el mismo
- Ops.: suscribir [alta/baja] ($S\rightarrow$); publicar ($E\rightarrow$); notificar ($\rightarrow S$)
- Paradigma asíncrono y desacoplado en espacio
  - Editores y subscriptores no se conocen entre sí (≠ cliente/servidor)
  - En algunos casos también desacoplado en el tiempo
- Normalmente, *push*: suscriptor recibe notificaciones
  - Alternativa, *pull*: suscriptor pregunta si hay notificaciones
- Calidad de servicio (orden entrega o pérdida de notificaciones)
- Posible uso de *leases* en suscripción

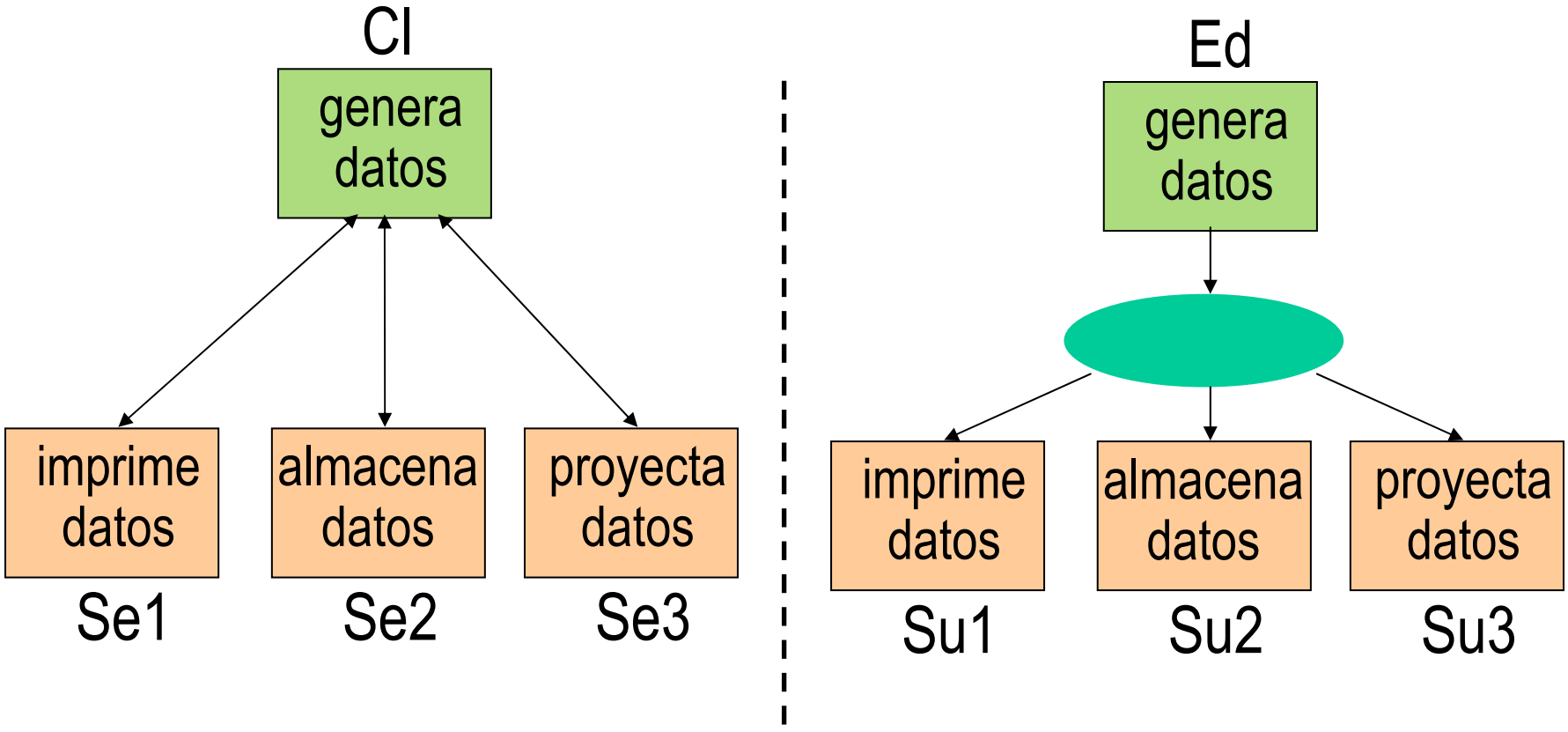
Modelo editor/subscriptor



Ejemplo editor/suscriptor

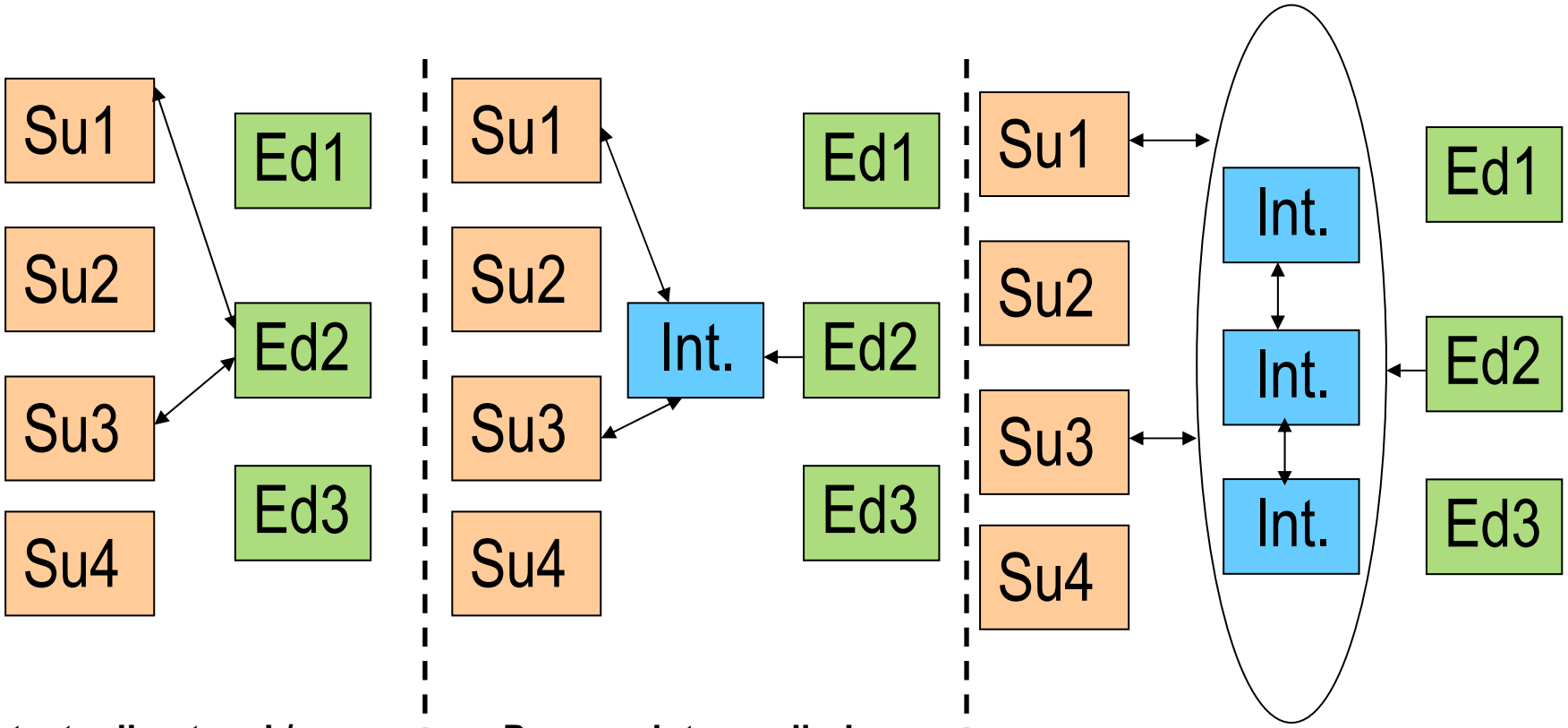
- Mercado bursátil
- Tipo de evento
  - Cada valor (V) del mercado
- Subscriptor
  - Proceso interesado (PI) en operaciones sobre un determinado valor
- Editores
  - Proceso que realiza operaciones (PO) sobre un determinado valor
- POi realiza operación sobre Vj
  - Envío de notificación a todo PIk suscrito a Vj

Cliente/servidor vs. Editor/suscriptor



¿En cuál es más fácil añadir nuevo consumidor de datos?

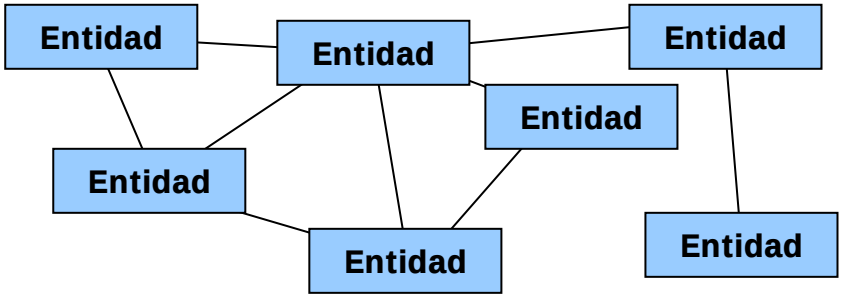
Implementaciones editor/suscriptor

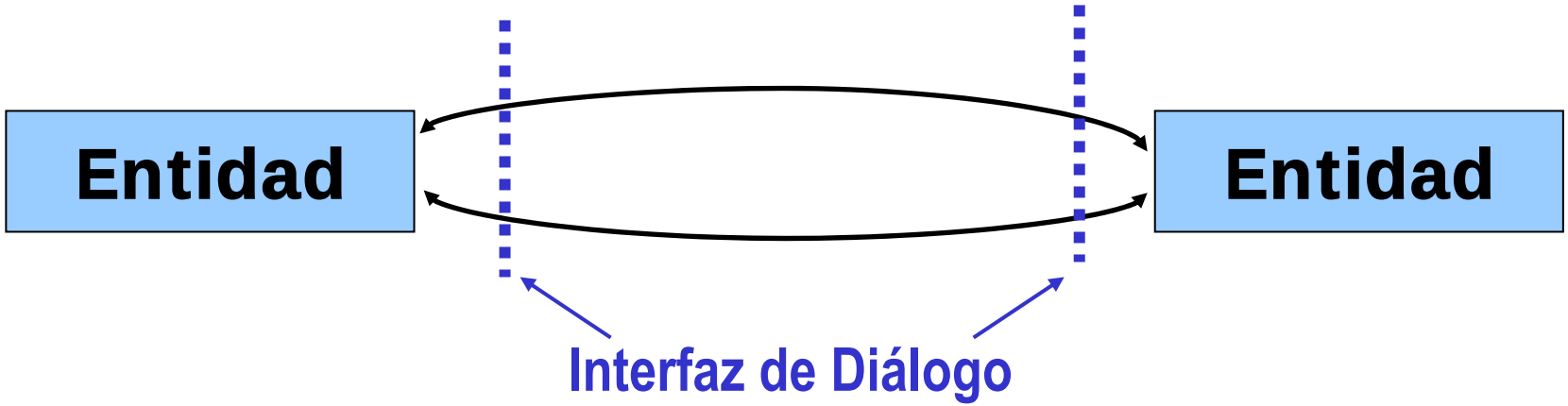


Modelo *Peer-to-Peer (P2P)*

- Todos los nodos tienen mismo rol y funcionalidad (*servents*)
  - No hay cuellos de botella ni puntos críticos de fallo
  - Se aprovechan recursos de todas las máquinas
- Se suelen caracterizar además por:
  - Volatilidad: Nodos entran y salen del SD; variabilidad en conectividad
  - Capacidad de autogestión sin una autoridad global centralizada
- Dedicados a compartir recursos (contenidos,UCP,almacén,...)
  - Y/O a colaboración y comunicación entre usuarios
- Uso de red superpuesta (*overlay*): Red lógica sobre la física
  - Nodos de proceso como nodos de red
  - Esquemas de encaminamiento y localización de recursos
- Desventajas de arquitectura P2P
  - Difícil administración y mayores problemas de seguridad

Esquema *Peer-to-Peer (P2P)*





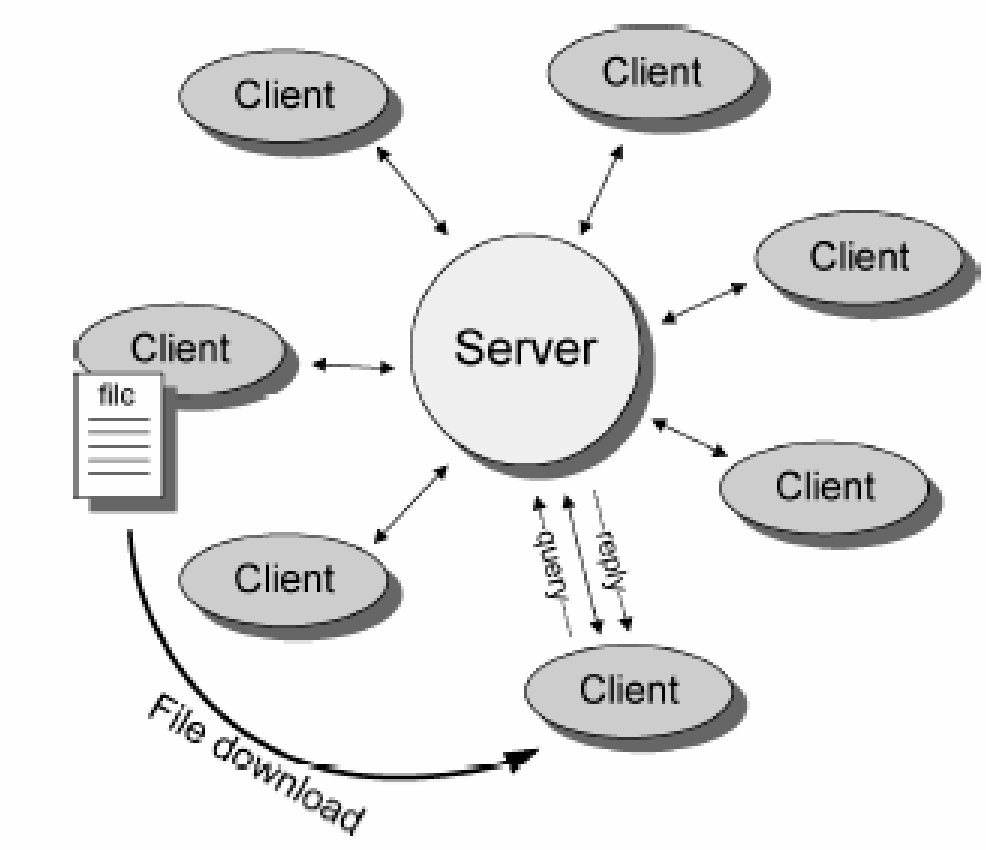
Tipos de sistemas P2P

- Desestructurados:
  - Topología de conexión lógica arbitraria
  - Ubicación de recursos impredecible e independiente de la topología
    - Cada nodo posee un conjunto de recursos
  - Corresponden a sistemas +volátiles con nodos más autónomos
- Estructurados:
  - Topología de conexión prefijada (p.e. anillo en protocolo *Chord*)
  - Ubicación de recursos predecible y dependiente de la topología
  - Generalmente definida por función *hash* distribuida
    - Única op.: lookup(clave recurso) → dir IP máquina que posee recurso
    - Protocolos *Chord* (Stoica et al. MIT 2001), *CAN*, *Tapestry* y *Pastry*
  - Corresponden a sistemas -volátiles con nodos más cooperativos
    - Posesión de recursos cambia según sistema evoluciona

Tipos de sistemas P2P desestructuradas

- Híbridos: P2P + Cliente/servidor (p.e. Napster)
  - Servidor central guarda información encaminamiento y localización
  - Altas y bajas de nodos contactan con servidor
  - Localización de recursos consulta al servidor
- Puramente descentralizados (p.e. versión original de Gnutella)
  - Todos los nodos con la misma funcionalidad
  - Nodos propagan entre sí conocimiento de otros nodos
  - Localización de recursos por “inundación”
- Parcialmente centralizados (p.e. Kazaa)
  - Súper-nodos con info. encaminamiento y localización de grupo nodos
  - Pero dinámicamente elegidos y asignados

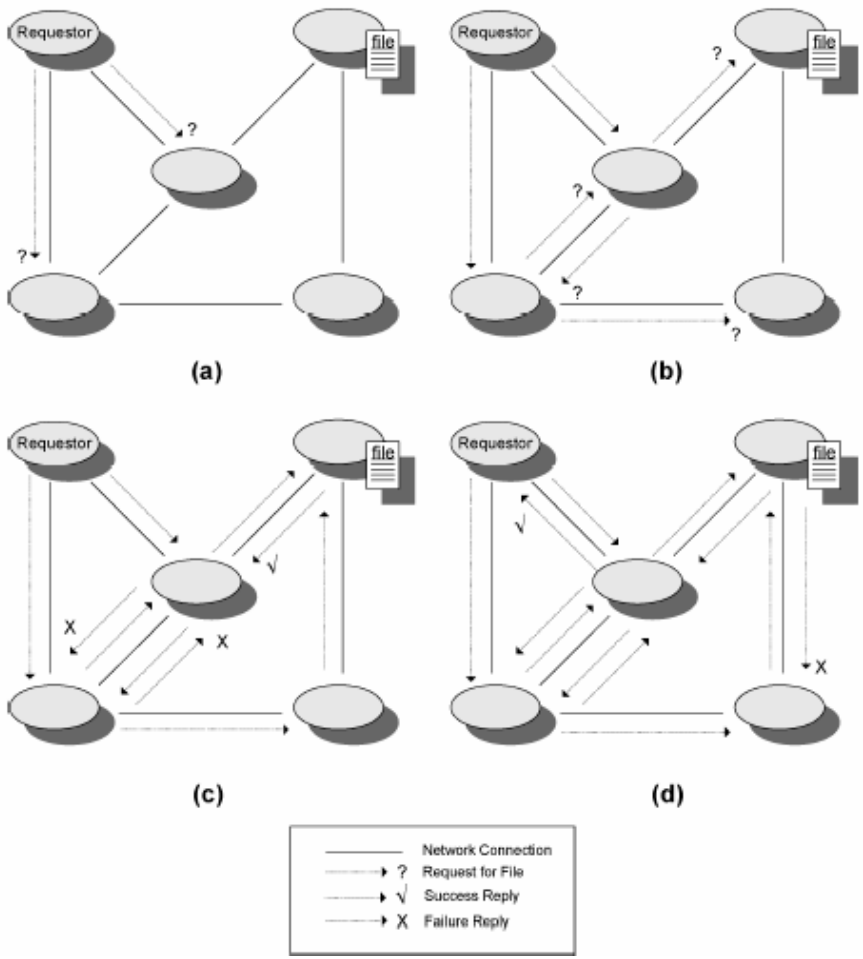
Localización recursos en redes híbridas



A Survey of Peer-to-Peer Content Distribution Technologies

S. Androutsellis-Theotokis y D. Spinellis; ACM Computing Surveys, 2004

Localización recursos en redes puras



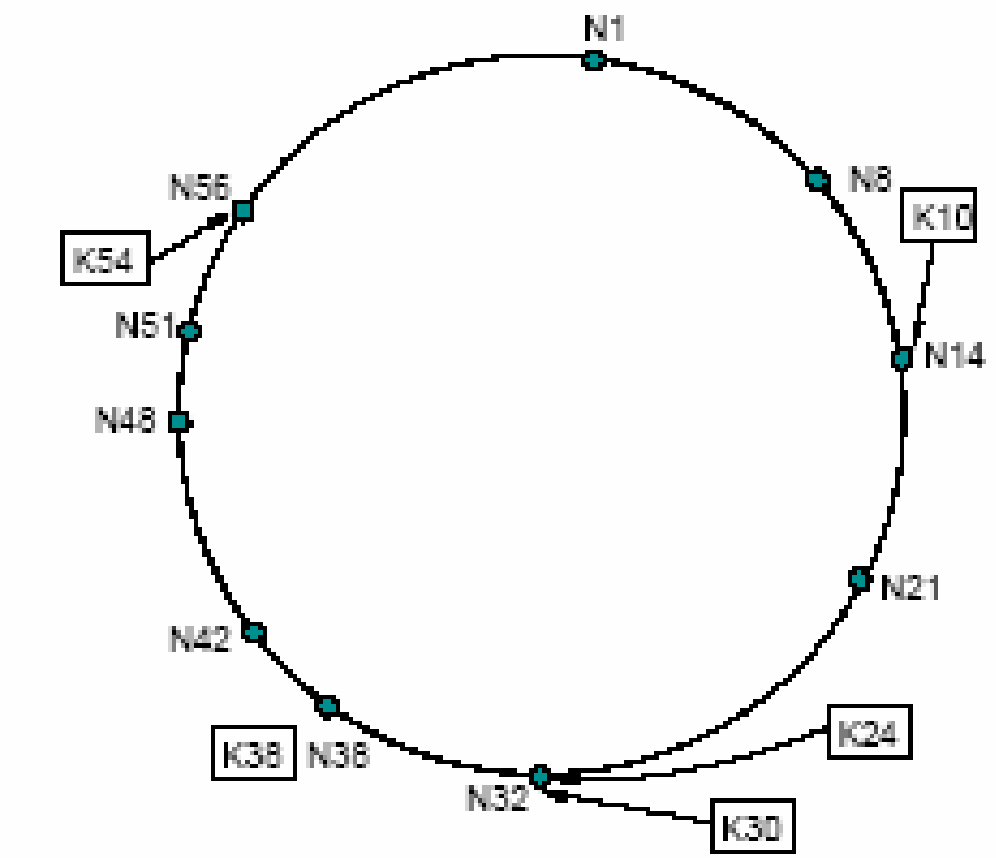
A Survey of Peer-to-Peer Content Distribution Technologies

S. Androutsellis-Theotokis y D. Spinellis; ACM Computing Surveys, 2004

Protocolo *Chord*

- *Hashing* "consistente" asigna ID a clave recurso y a IP de nodo
  - ID con *m* bits tal que nº recursos (K) + nº nodos (N) << 2^m
  - IDs organizados en anillo módulo 2^m
  - Proporciona distribución uniforme
- Asignación de recurso *K* a nodo *N*
  - 1er nodo ID(*N*) ≥ ID(*K*) $\rightarrow$ sucesor(*K*) (NOTA: ≥ siguiendo módulo)
- Localización de recurso: *N* busca *K;* algoritmo simple
  - Cada nodo sólo necesita almacenar quién es su sucesor directo
    - NOTA: nodo almacena también predecesor
  - Búsqueda lineal de sucesor a sucesor
  - Hasta encontrar par de nodos *a*, *b* tal que *ID* $\in$ (*a*, *b*]
  - Ineficiente y no escalable *O(N)*

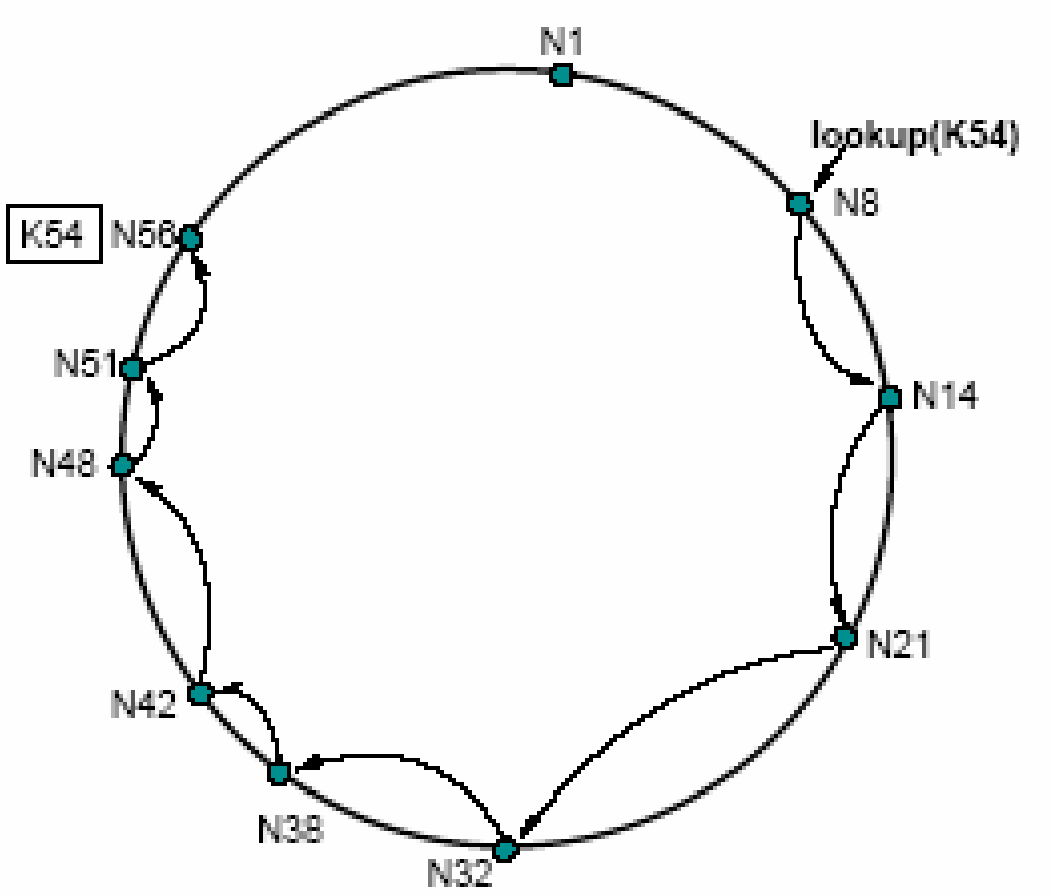
Anillo 2^6 con 10 nodos y 5 recursos



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica et al.; ACM SIGCOMM'01

Búsqueda simple lineal



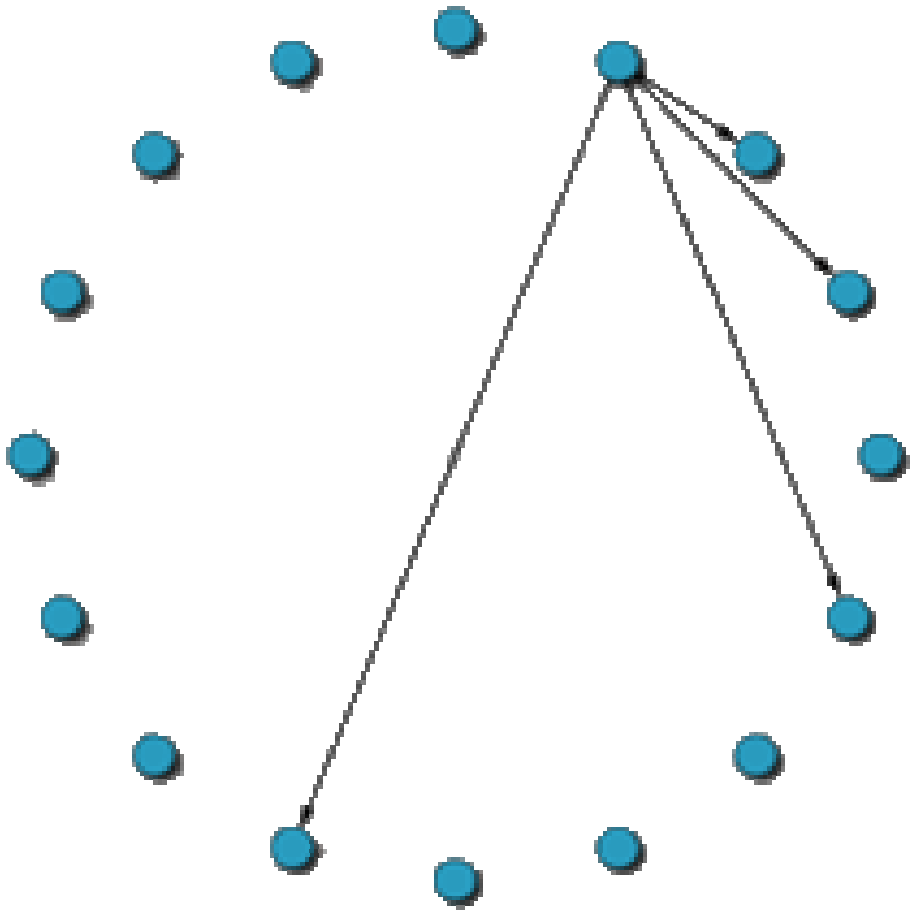
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica et al.; ACM SIGCOMM'01

Búsqueda basada en *fingers*

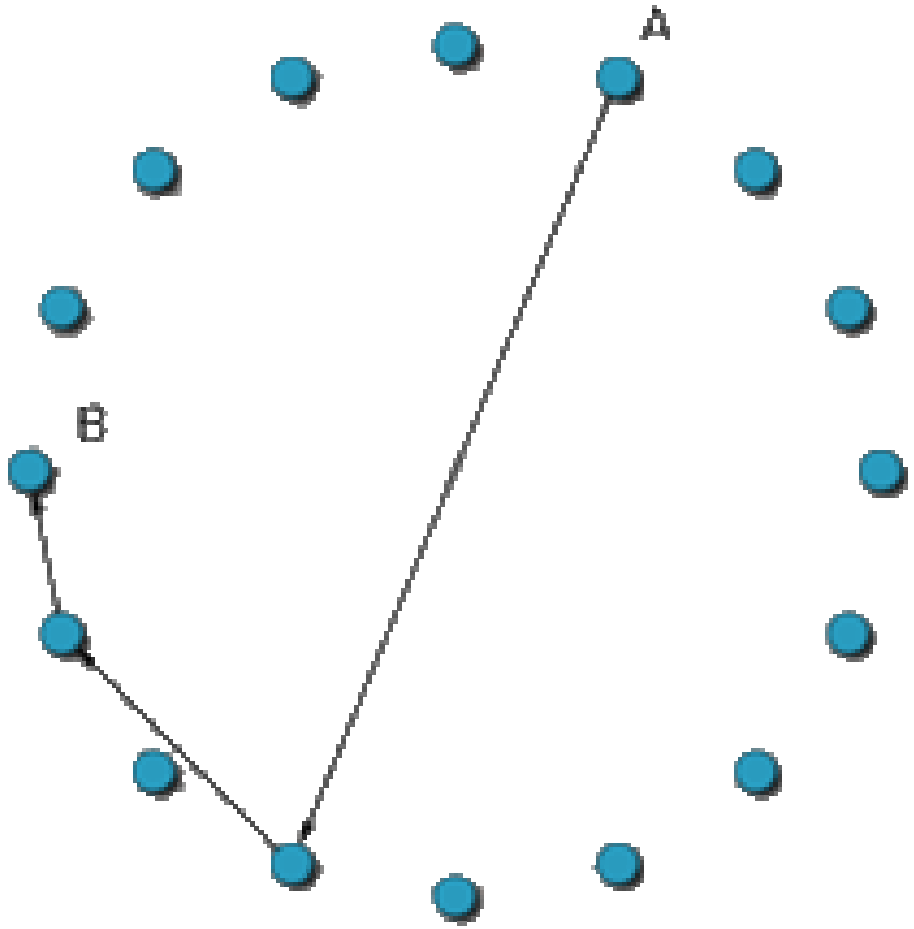
- Cada nodo con ID *n* incluye tabla *fingers TF* con *m* entradas:
  - Entrada *i*: primer nodo a una distancia ≥ 2^{i-1}
    - *sucesor*$(n + 2^{i-1})$ tal que $1 \leq i \leq m$
    - Entrada 0: sucesor directo
- Búsqueda de *K* desde nodo *n*:
  - Si *ID* $\in$ (*n*, *s*: sucesor directo de *n*] $\rightarrow$ *K* asignado a *s*
  - Sino: buscar en *TF* empezando por Entrada *m*:
    - Nodo más inmediatamente precedente
    - Pedirle que continúe la búsqueda
- Tiempo localización e info. requerida: *O(logN)*

Fingers en anillo 2^4



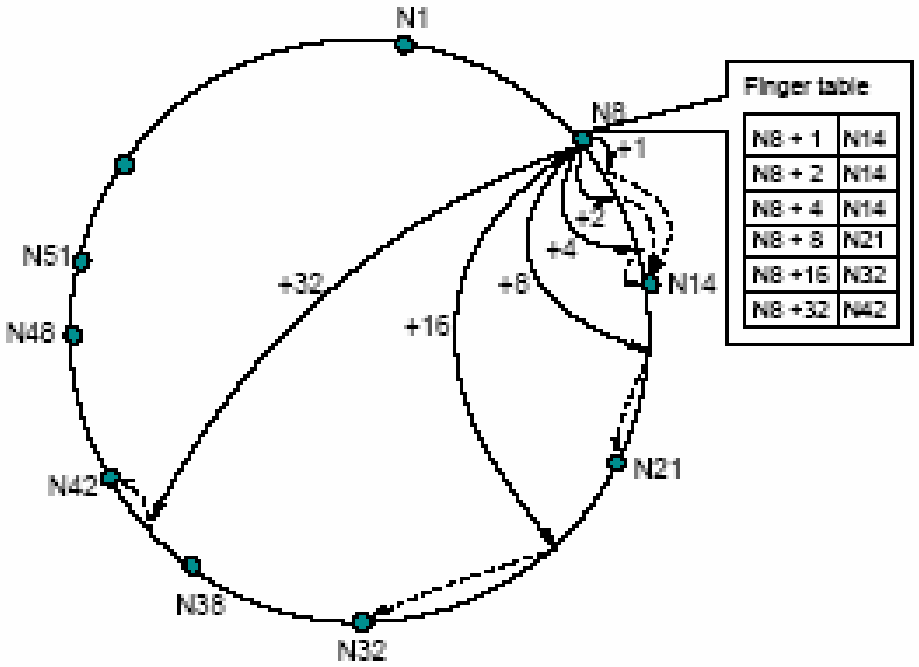
Wikipedia Chord

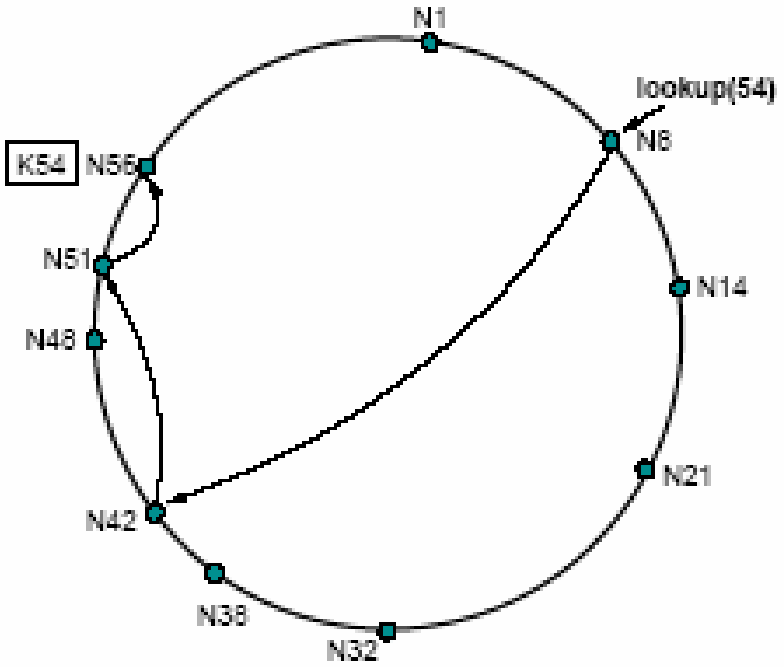
Búsqueda en anillo 2^4



Wikipedia Chord

Búsqueda con *fingers*





Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica et al.; ACM SIGCOMM'01

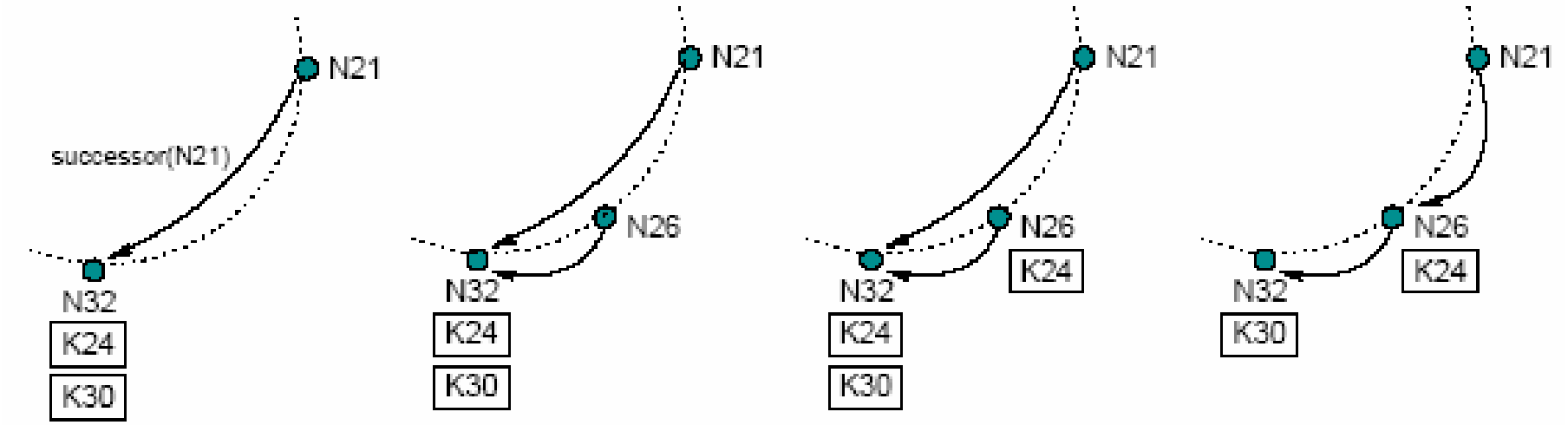
Mantenimiento del anillo

- Carácter dinámico del anillo
  - Alta de nodos
  - Baja voluntaria de nodos
  - Baja involuntaria de nodos (caídas)
- Operaciones deben asegurar estabilidad del anillo
  - Descompuestas en varios pasos cuidadosamente ideados
- Procesos periódicos de actualización del anillo
  - Aseguran estabilización antes cambios continuos

Alta de un nodo

- Operación *join* de nodo *N*:
  - Conoce de alguna forma dir. de cualquier nodo existente *N'*
  - *N* calcula su ID y pide a *N'* búsqueda de su sucesor
  - *N* anota su sucesor (por ahora predecesor = NULO)
- Operación periódica en cada nodo *stabilize*:
  - Pregunta a su sucesor *S* por su predecesor *P*
  - Si *P* mejor sucesor de *N* que *S*, fija *P* como sucesor de *S*
  - Notifica a su sucesor para que reajuste predecesor, si necesario
- Operación periódica en cada nodo *fix_fingers*:
  - Actualización de tabla de *fingers* si necesario
- Operación periódica en cada nodo *check_predecessor*:
  - Comprueba si sigue vivo predecesor: No $\rightarrow$ predecesor = NULO
- Alta incluye transferencia de recursos asociados ahora a *N*

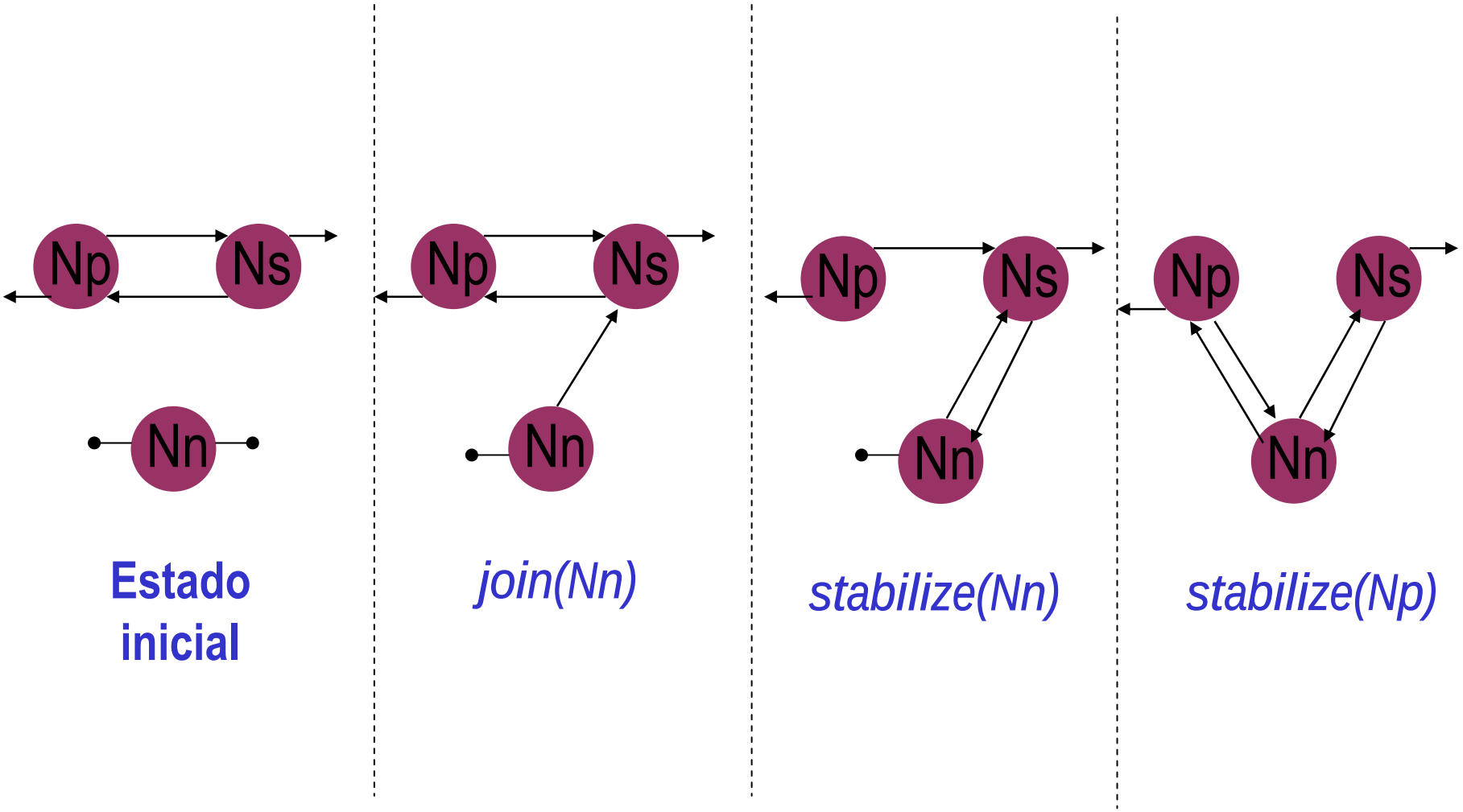
Alta de un nodo



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica et al.; ACM SIGCOMM'01

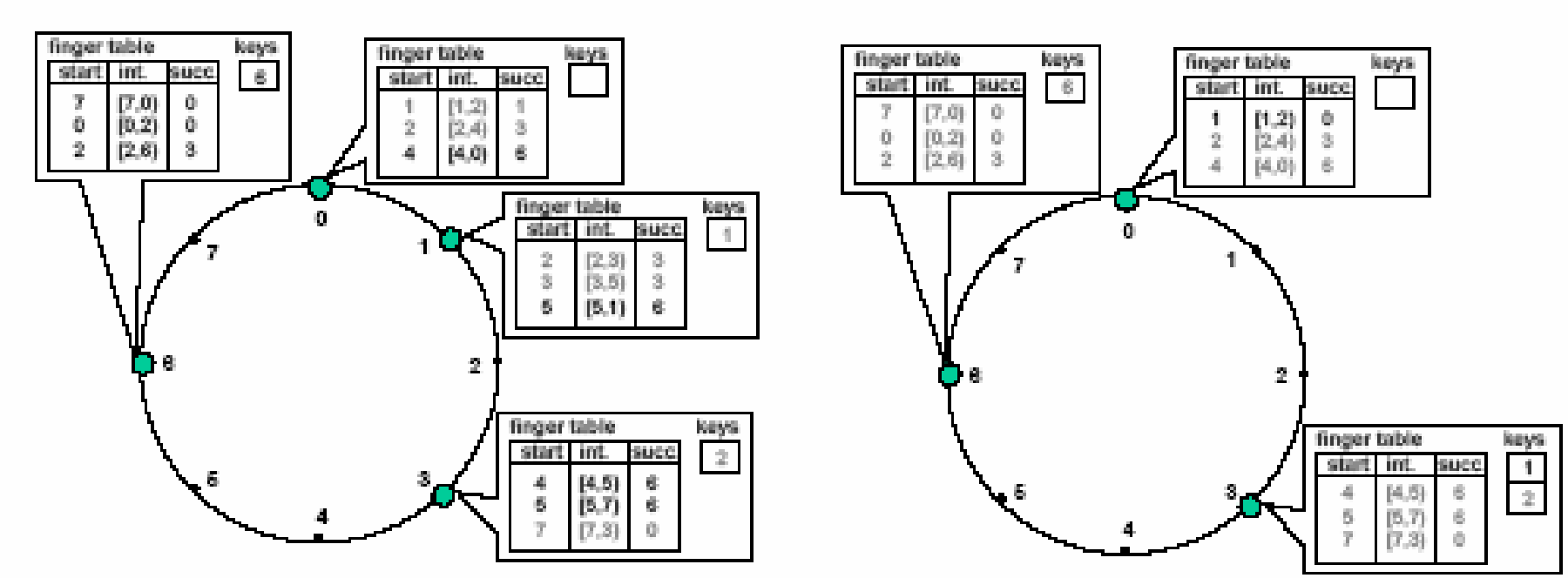
Alta de un nodo: paso a paso



Baja de un nodo

- Baja voluntaria de nodo implica acciones complementarias
  - Devolver recursos a nuevo sucesor
  - Informar a predecesor y sucesor para que reajusten estado
- Caída de nodo (baja involuntaria) más problemática
  - Operaciones periódicas de estabilización van reajustando el anillo
  - Pero puede haber problemas en búsqueda hasta reajuste
    - Nodo sucesor caído hasta que se actualiza nuevo sucesor
  - Solución: Cada nodo guarda lista de sus *m* sucesores
  - ¿Qué pasa con los recursos del nodo caído?
    - Protocolo no especifica política de replicación de recursos
- Algoritmo estable ante altas y bajas simultáneas
  - Es capaz de trabajar con info. no totalmente actualizada

Tablas *finger* después de alta y baja



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica et al.; ACM SIGCOMM'01