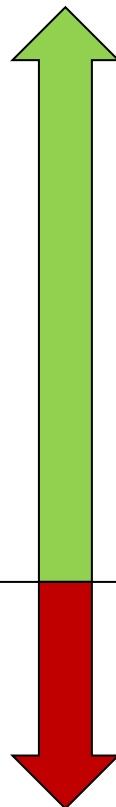

Sistemas Distribuidos

2

**Arquitectura de
los Sistemas
Distribuidos**

Índice

- Modelos de interacción
 - Arquitectura cliente-servidor
 - Variaciones del modelo
 - Aspectos de diseño del modelo cliente/servidor
 - Arquitectura editor-subscriptor
 - Arquitectura productor-consumidor
-
- Arquitectura *peer-to-peer*
 - Arquitecturas para computación distribuida
- 
- 1ª parte
(incluida en examen)
- 2ª parte
(no incluida en examen)

Modelos de interacción en los SD

- Organización lógica de componentes de aplicación distribuida
 - Qué roles ejercen los procesos y cuál es su patrón de interacción
 - La “topología” de la aplicación distribuida
- En principio, tantos como aplicaciones
 - Pero hay patrones que se repiten de forma habitual
- Patrones más frecuentes en SD de propósito general
 - Cliente/servidor (C/S)
 - Editor/subscriptor (EdSu); *Publisher/Subscriber (PubSub)*
 - Productor/consumidor (ProdCons)
 - *Peer-to-peer* (Paritaria)
- Computación distribuida presenta sus propios patrones
 - Maestro/trabajador
- El objetivo de este tema es estudiar estos patrones

Grado de acoplamiento

- Sea cual sea el patrón, 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
 - Vidas de entidades interaccionando **no** tienen que coincidir en tiempo
- 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.”*

Ejemplo de grados de acoplamiento

Ejemplos cotidianos

temporal \ espacial	acoplado	desacoplado
acoplado	llamada telefónica	retransmisión TV en directo
desacoplado	carta postal	anuncio en un tablón

A lo largo del tema iremos revisando esta tabla

temporal \ espacial	acoplado	desacoplado
acoplado	C/S	EdSu comunicación de grupo
desacoplado	no habitual	ProdCons

Ejemplo fuera del ámbito de SD: memoria compartida doble desacoplamiento

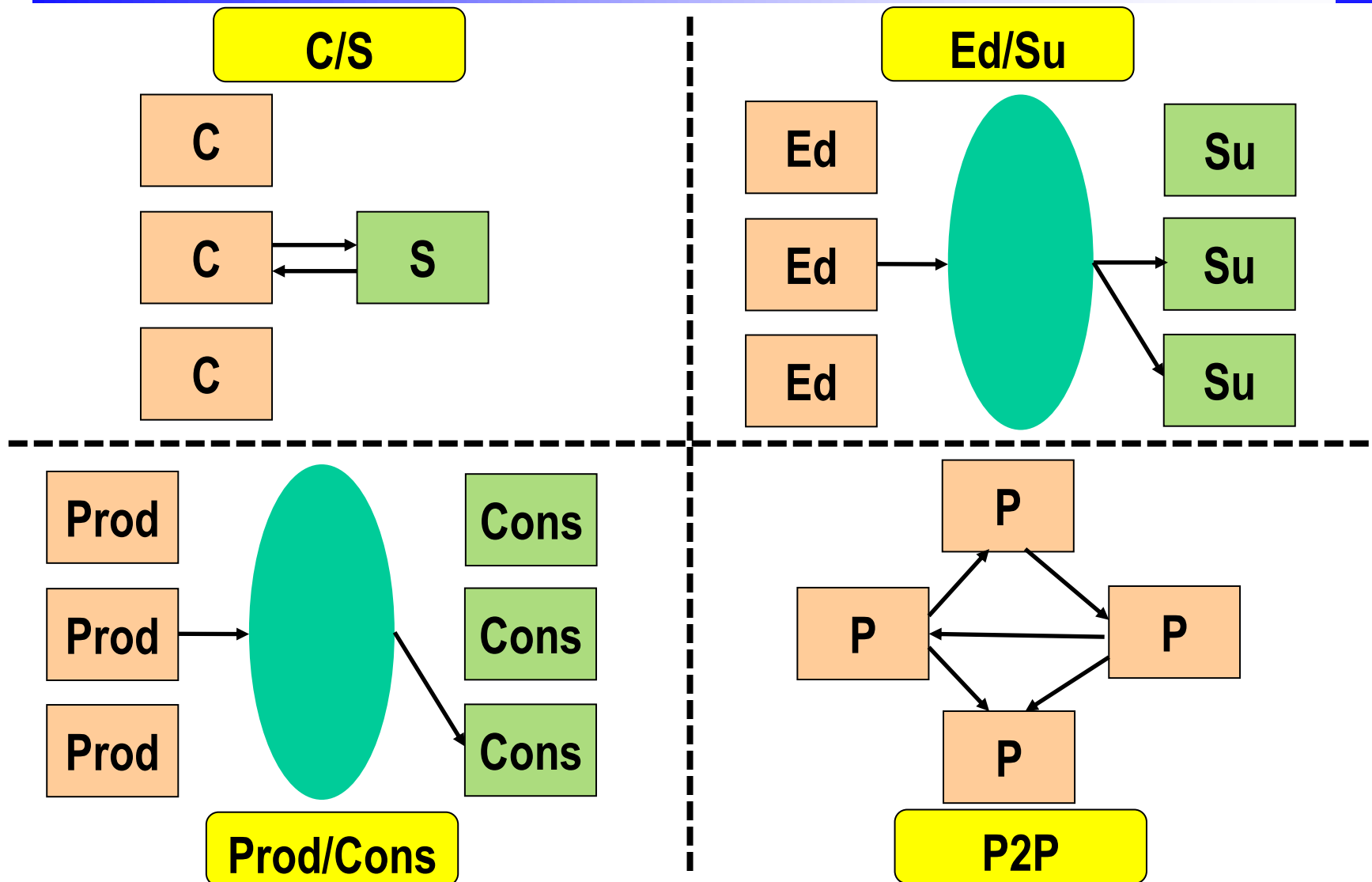
Espacial: escritura en memoria no sabe a qué procesos afectará

Temporal: cuando proceso lee de memoria el escritor puede no existir

Arquitecturas en SD de propósito general

- Cliente/servidor (petición/respuesta); N clientes – 1 servidor
 - Extensión a un SD del esquema biblioteca y programa que la usa
 - Interacción 1 cliente \leftrightarrow 1 servidor
- Editor/subscriptor; M editores – N subscriptores
 - Extensión a un SD de un esquema guiado por eventos
 - Interacción 1 editor \rightarrow N subscriptores
- Productor/consumidor; M productores – N consumidores
 - Extensión a un SD de un esquema de tipo tubería UNIX
 - Interacción 1 productor \rightarrow 1 consumidor
- *Peer-to-peer*; N procesos
 - Procesos cooperantes con el mismo rol
 - Interacción N-N

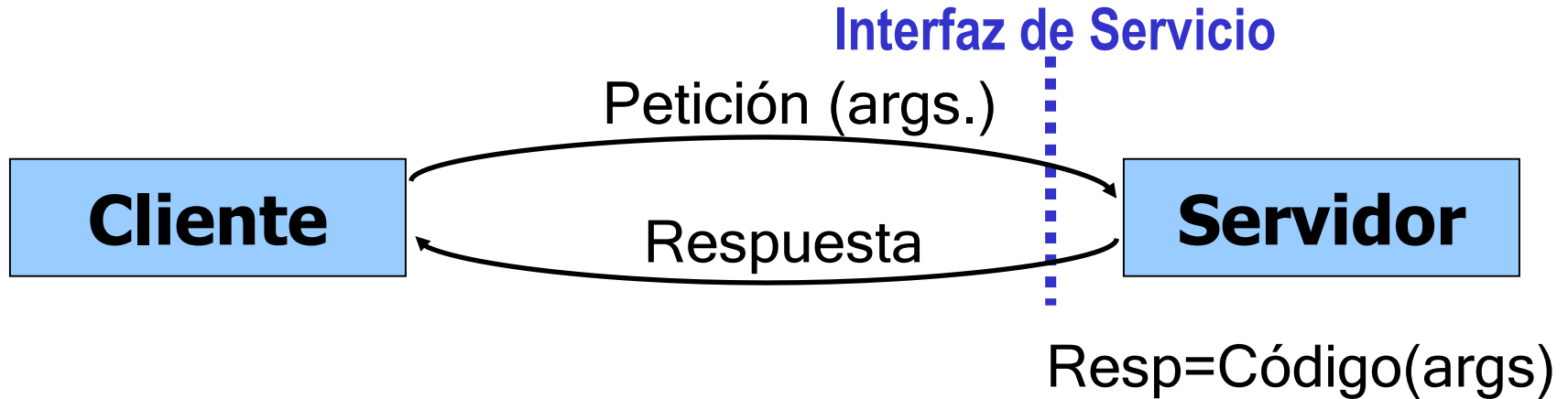
Arquitecturas en SD de propósito general



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
- 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



- Reparto funcionalidad entre C y S
- “Grosor del cliente”: Cantidad de trabajo que realiza
 - Pesados (*Thick/Fat/Rich Client*) vs. Ligeros (*Thin/Lean/Slim Client*)
- Cliente más pesado (más inteligente)
 - Mayor autonomía y más ágil respuesta al usuario
 - Mejor escalabilidad: Cliente gasta menos recursos de red y servidor
 - Pero más coste de mantenimiento y problemas de seguridad

Variaciones del modelo C/S

- C/S con caché
- C/S con proxy
- C/S jerárquico
- C/S con reparto de carga
- C/S con alta disponibilidad
- C/S con código móvil

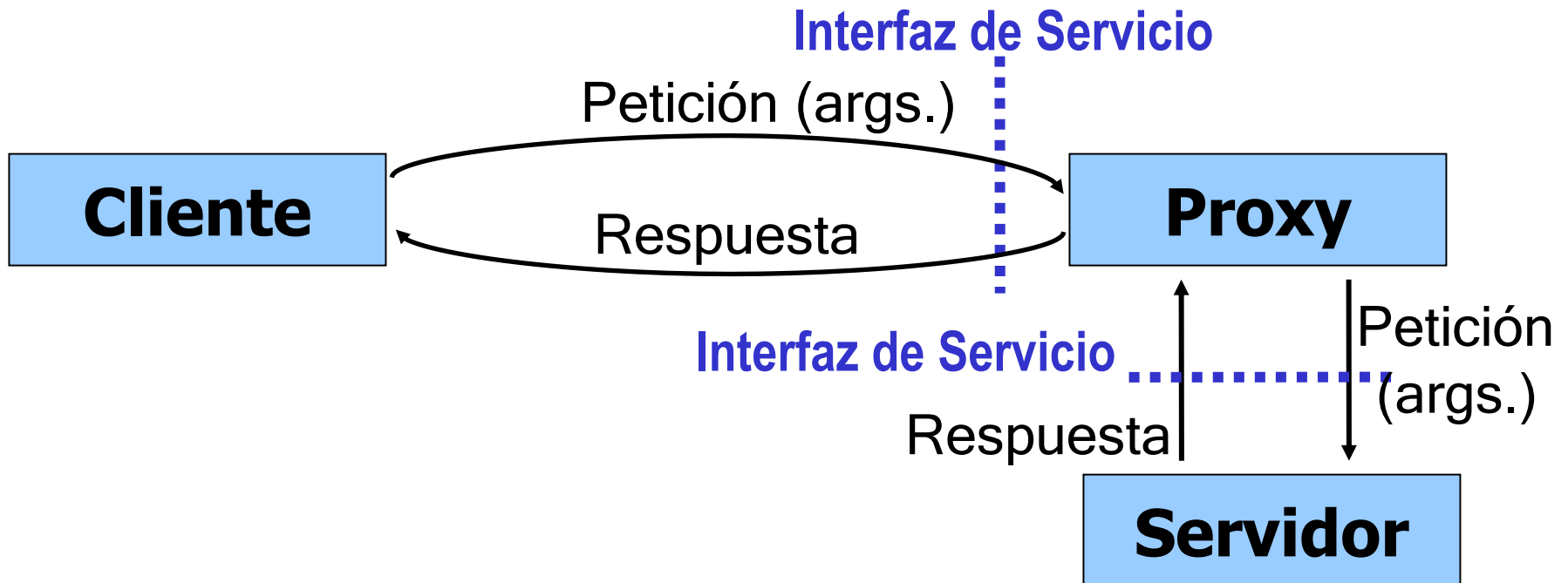
Cliente/servidor con caché

- 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)
 - Temas de sistemas de ficheros y de servicio de nombres
- Puede posibilitar modo de operación desconectado
 - Sistema de ficheros CODA (tema de sistemas de ficheros)
- Y *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
- Se puede considerar caché como un tipo de replicación parcial

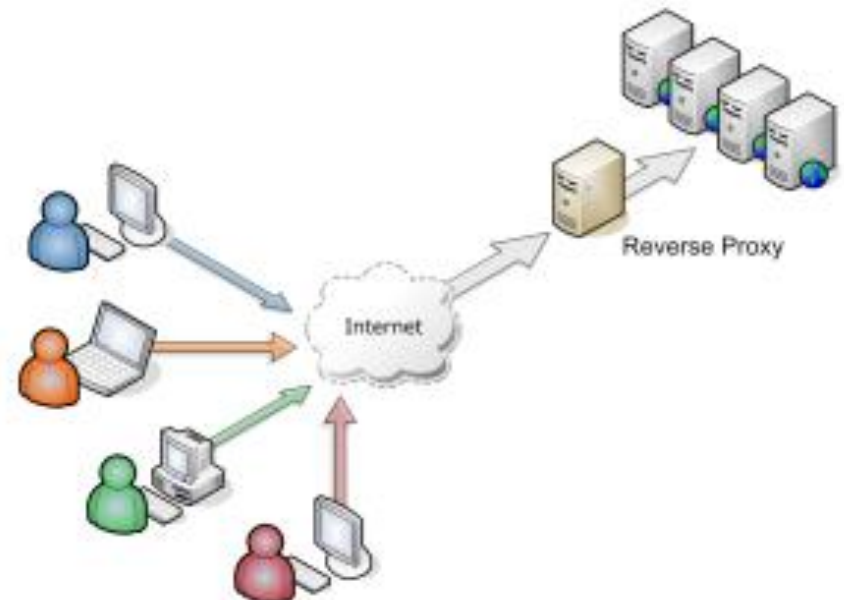
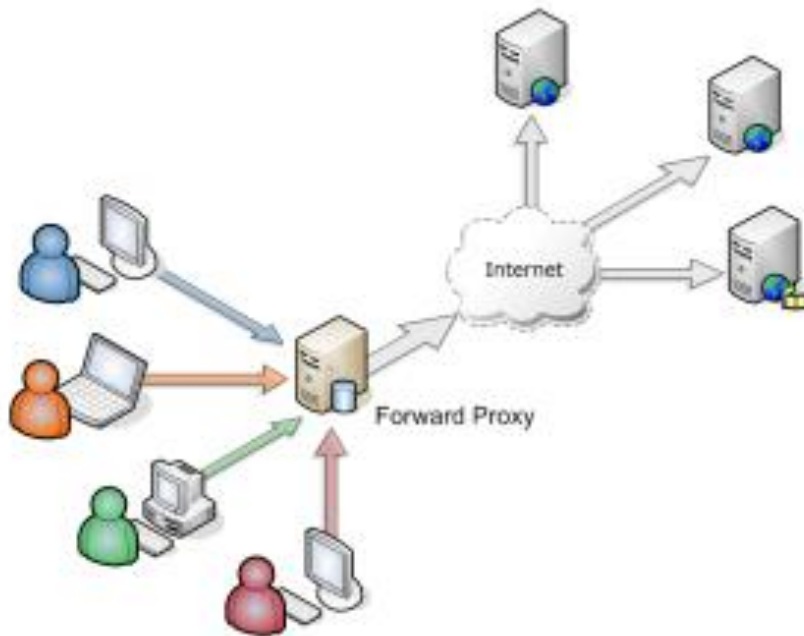
Cliente/servidor con proxy

- Componentes intermediarios entre cliente y servidor
- Interfaz de servicio de proxy debe ser igual que la del servidor:
 - Se comporta como servidor para cliente y como cliente para servidor
 - Se pueden “enganchar” sucesivos *proxies* de forma transparente
- *(Forward) Proxy*
 - Ubicado en la misma organización que los clientes
 - Usuarios desconocen su existencia
 - Uso: caché, punto único de salida a Internet...
- *Reverse Proxy*
 - Ubicado en la misma organización que los servidores
 - Usuarios especifican su dirección (desconocen existencia servidores)
 - Uso: caché, punto único de entrada desde Internet, cortafuegos, reparto de carga...

Esquema con *proxy*



Forward vs Reverse Proxy

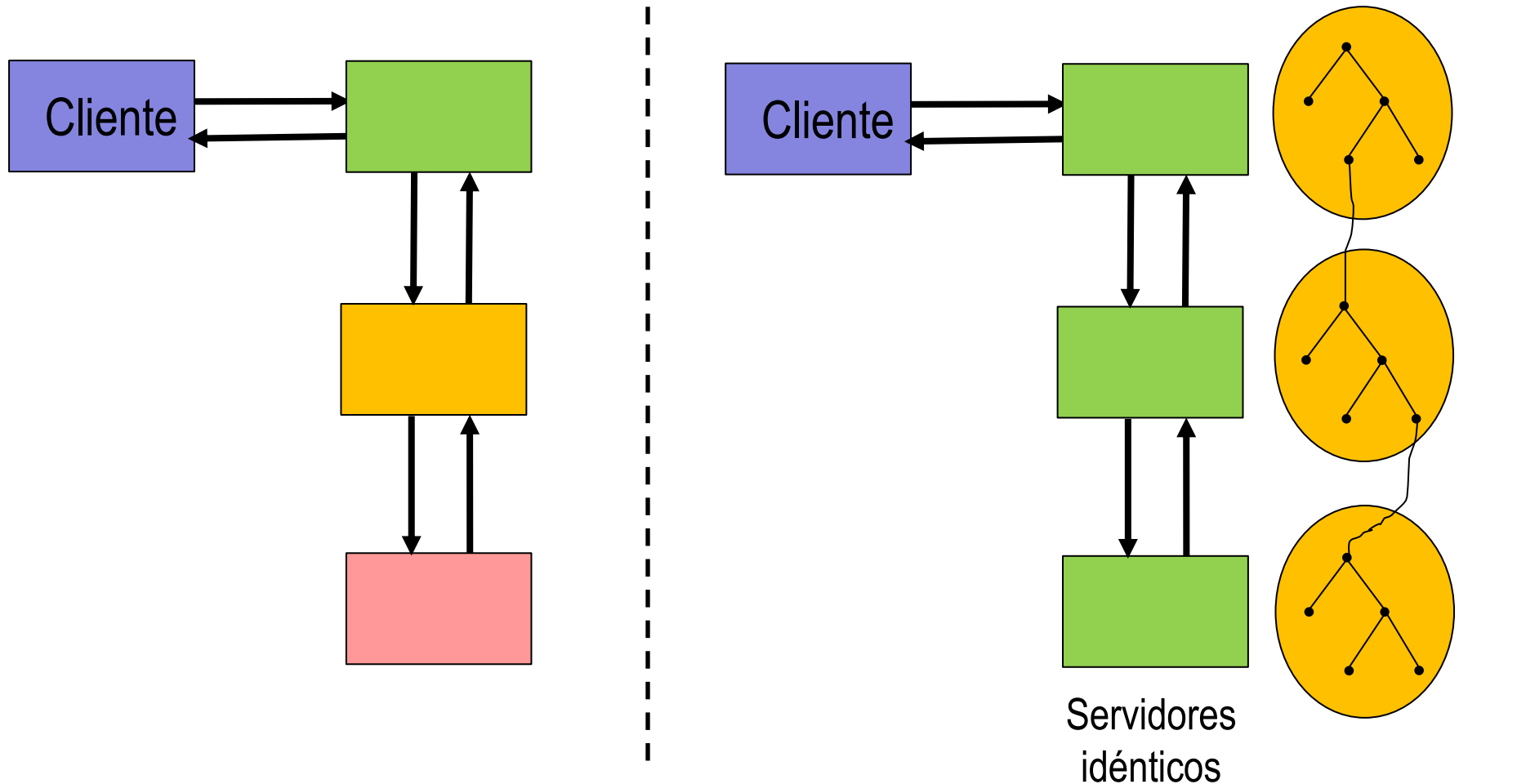


<https://www.quora.com/Whats-the-difference-between-a-reverse-proxy-and-forward-proxy>

Cliente/servidor jerárquico

- Servidor actúa como cliente de otro servidor
 - Igual que biblioteca usa función de otra biblioteca
- División vertical
 - Funcionalidad dividida en varios niveles (*multi-tier*)
 - P. ej. En aplicación típica con 3 capas:
 - Presentación; Aplicación (lógica de negocio) y Acceso a datos
 - cada nivel puede implementarse como un servidor
- División horizontal
 - Múltiples servidores idénticos cooperan en servicio
 - Similitud con P2P
 - Traducir nombre de fichero en SFD o nombre de máquina con DNS
 - Temas de sistemas de ficheros y de servicios de nombres

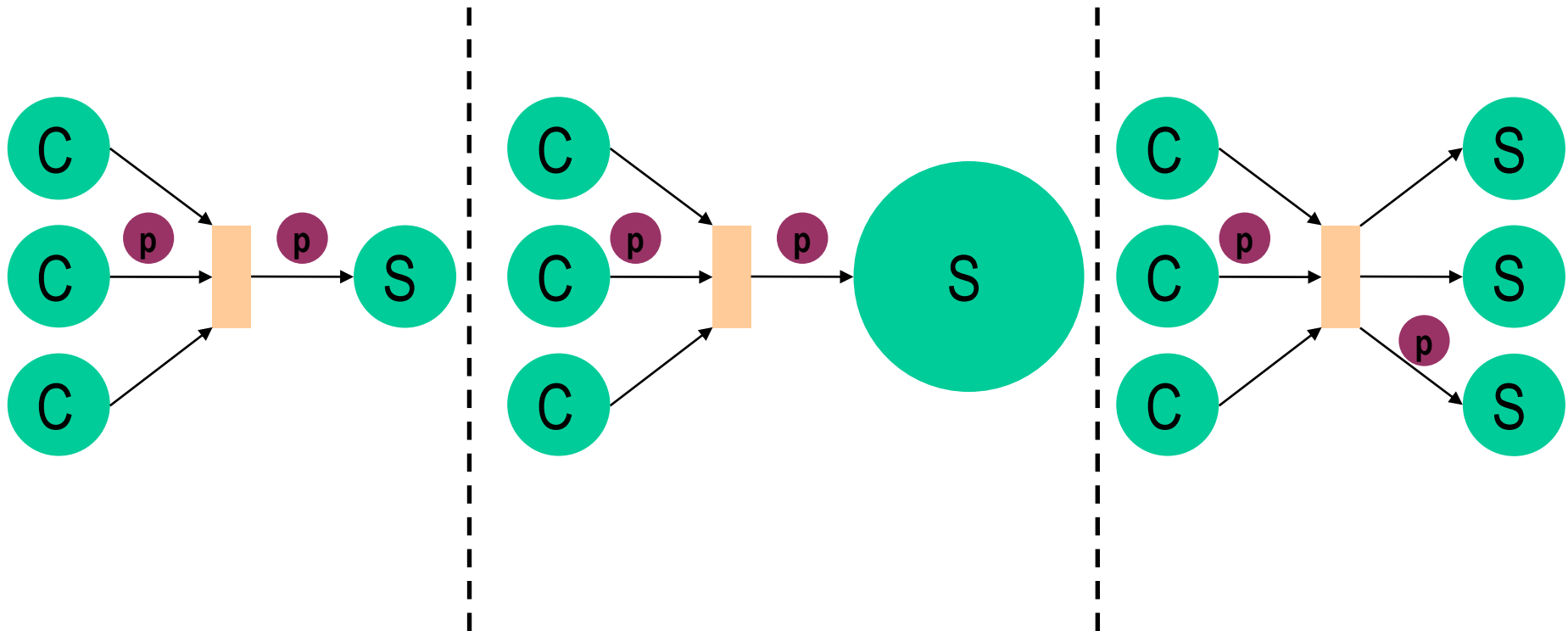
División vertical vs. horizontal



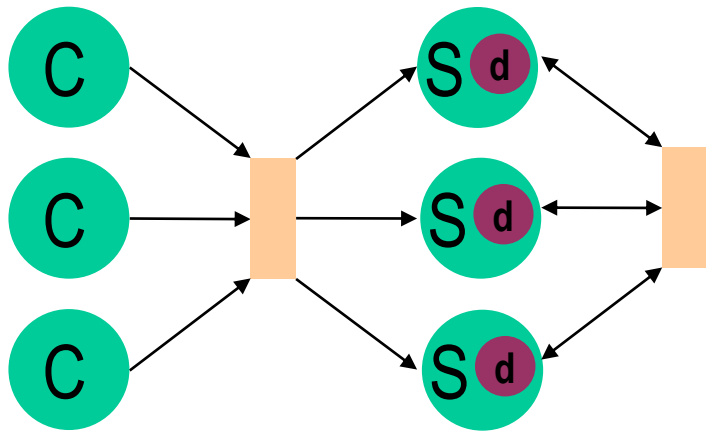
Cliente/servidor con reparto de carga

- Servidor único
 - Cuello de botella: afecta a latencia y ancho de banda
 - Punto único de fallo: afecta a fiabilidad
- Mejorar prestaciones de nodo servidor
 - Escalado vertical (*scale-up*)
 - Mejora rendimiento
 - Pero no escalabilidad del servicio ni su tolerancia a fallos
- Múltiples servidores con reparto de carga (M-N)
 - Peticiones se reparten entre servidores
 - Escalado horizontal (*scale-out*)
 - Mejora latencia, escalabilidad del servicio y tolerancia a fallos
 - Si servicio usa repositorio de datos, necesita replicación de datos
 - Las réplicas necesitan mantenerse sincronizadas
 - ¿Qué ocurre si hay una partición de red que aísla réplicas entre sí?

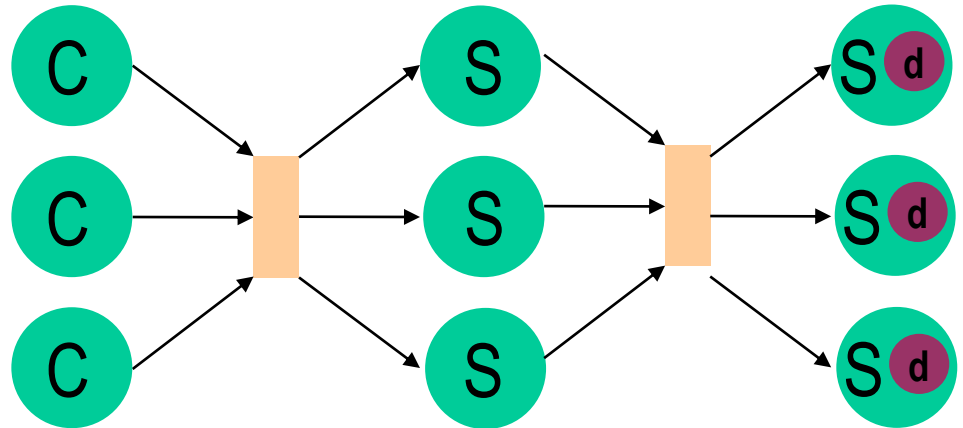
Scale-up vs Scale-out



Scale-out con datos replicados



En los propios nodos de servicio



En servidores de almacenamiento

Teorema CAP (Eric Brewer)

- Un SD puede proporcionar las siguientes propiedades:
 - Consistency: lectura dato **siempre** obtiene valor escritura más reciente
 - Availability: los datos están accesibles para **todos** los procesos
 - Partition tolerance: comportamiento OK a pesar de particiones de red
- **Teorema CAP:** solo se pueden tener 2 de las 3 propiedades
- SD de tipo CP: Ante partición de red (P)
 - Asegura *Consistency* pero no *Availability*: no acceso a dato para todos
 - Lecturas/escrituras sobre réplicas pueden devolver un error o bloquearse
- SD de tipo AP: Ante partición de red (P)
 - Asegura *Availability* (acceso a dato) pero no *Consistency*
 - Lectura puede obtener dato obsoleto
 - Escritura modifica solo réplicas accesibles
 - Al restablecerse red, necesaria reconciliación de cambios en cada partición
- ¿SD de tipo CA?: P irrenunciable: solo se puede elegir A o C

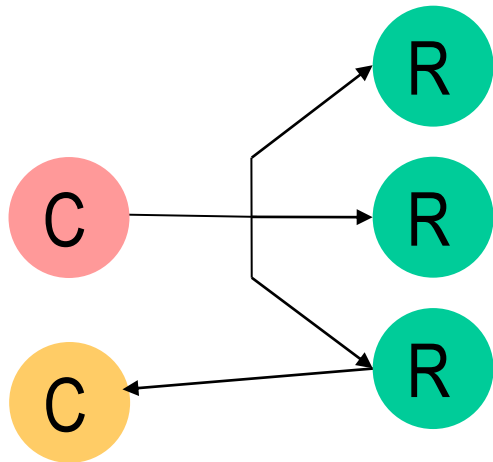
Latency vs. Consistency

- Teorema CAP solo aplicable cuando hay partición en la red
 - Sin partición, SD puede ofrecer C y A simultáneamente
- ¿Es siempre deseable tener C en sistema sin particiones?
 - Mantener consistencia estricta puede aumentar la latencia
 - En ocasiones, puedo renunciar a C por conseguir mejor latencia (L)
 - Aunque lectura pueda no obtener el valor de la última escritura
- **Teorema PACELC** (Abadi): extensión del teorema CAP
 - PAC define el comportamiento cuando hay partición (= CAP)
 - Si no (E/se): LC especifica si se elige *consistency* o *latency*
- Posibles sistemas:
 - PCEC: siempre asegura *consistency*
 - PAEC: solo asegura *consistency* si no hay partición
 - PAEL: nunca asegura *consistency*
 - PCEL: solo asegura *consistency* si hay partición

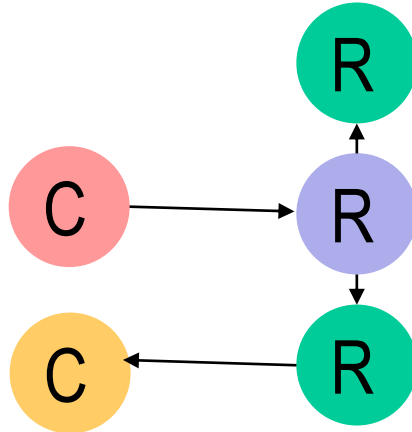
Actualización de réplicas

- Gestión réplicas compleja por caídas nodos y particiones de red
 - Problema del consenso distribuido (tema de sincronización)
- En este tema se presenta solo esbozo de alternativas
 - Replicación simétrica:
 - Lectura de cualquiera réplica y escritura en todas
 - Consistencia: lectura o escritura esperan que complete escritura en curso
 - Replicación con copia primaria y secundarias
 - Lectura de cualquiera y escritura en primaria que propaga a secundarias
 - Consistencia: lectura o escritura esperan que se complete propagación
 - Uso de *quorum*: N réplicas y parámetros N_W y N_R
 - Además del valor del objeto se guarda un n^o de versión
 - Lectura de N_R réplicas y se queda con versión más moderna
 - Escritura en N_W réplicas incrementando el n^o de versión
 - Para consistencia $N_R + N_W > N$ y $2 N_W > N$
 - Asegura que tanto 1 lectura y 1 escritura como 2 escrituras se solapan

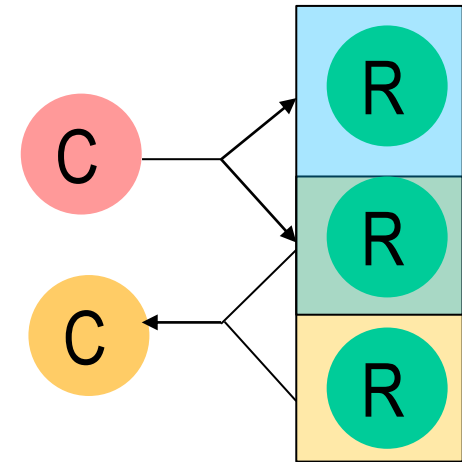
Actualización de réplicas



Replicación simétrica



Replicación primaria/secundarias

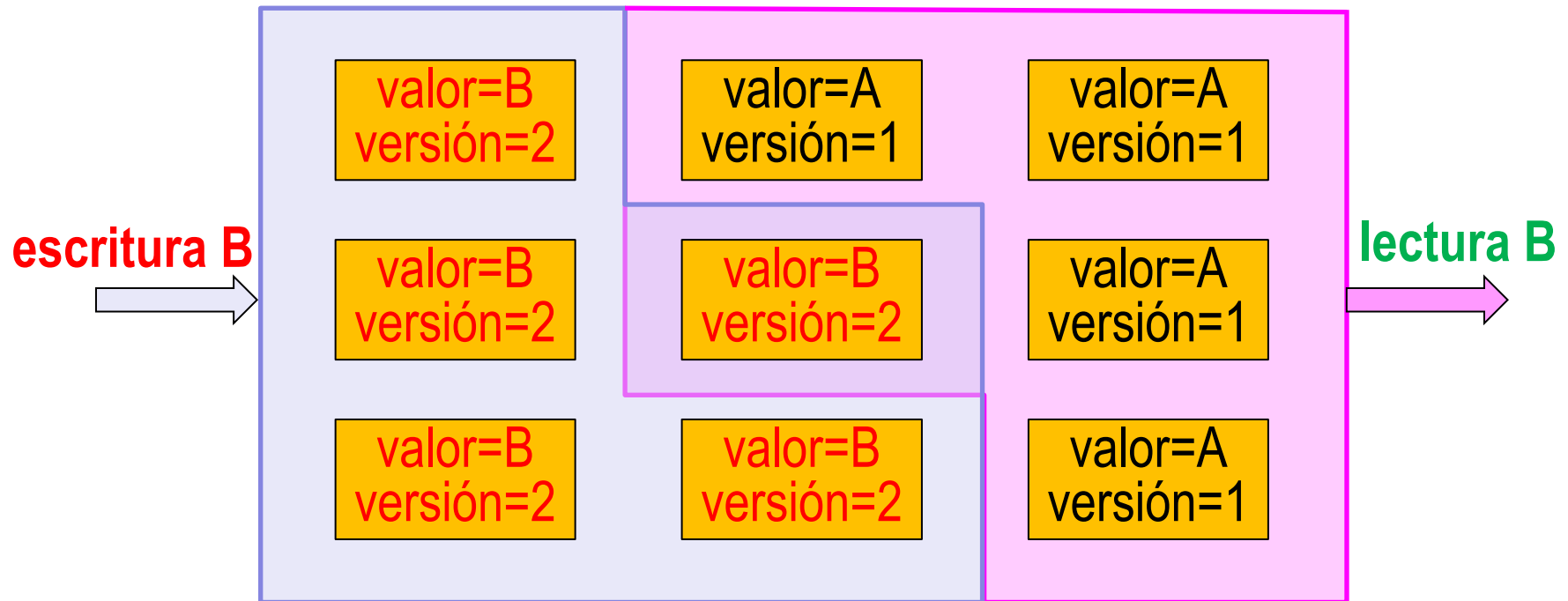


Replicación con *quorum*: N_R y $N_W = 2$

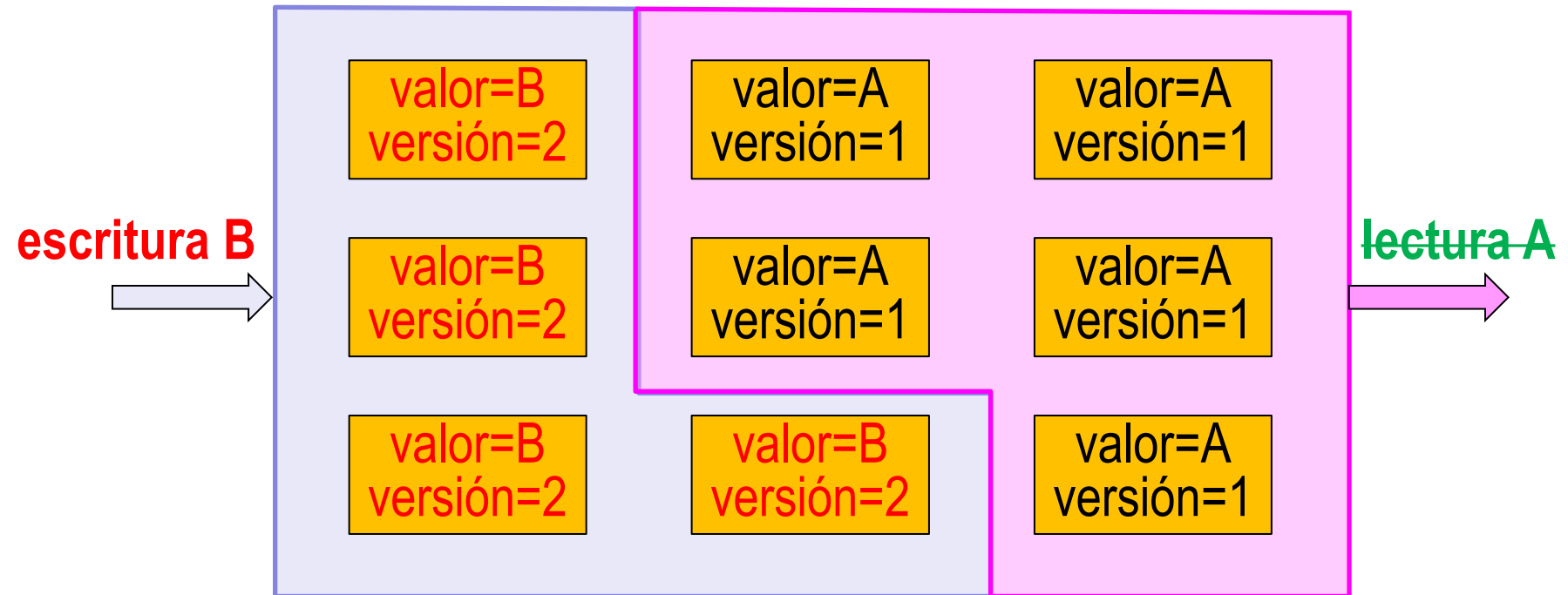
Ejemplo de *quorum*: $N=9$

valor=A versión=1	valor=A versión=1	valor=A versión=1
valor=A versión=1	valor=A versión=1	valor=A versión=1
valor=A versión=1	valor=A versión=1	valor=A versión=1

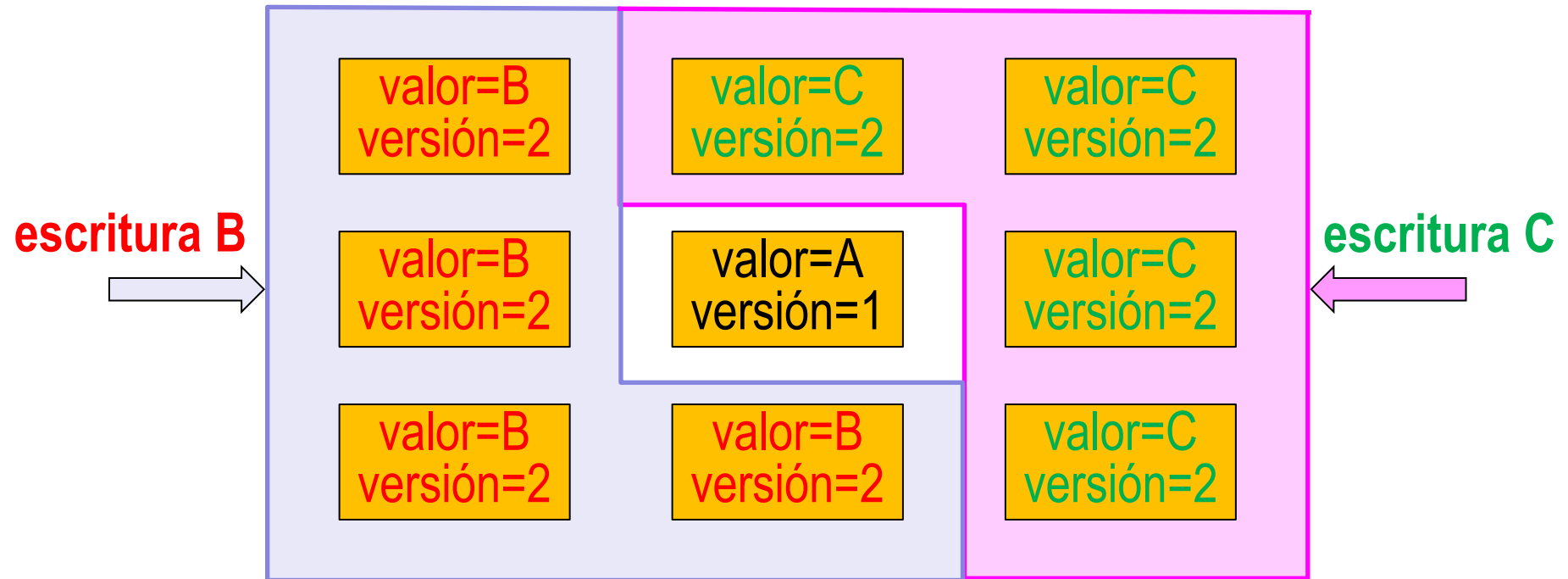
$N=9$; $N_W=5$; $N_R=5$; lectura y escritura



$N=9$; $N_W=4$; $N_R=5$; lectura y escritura



$N=9$; $N_W=4$; $N_R=5$; 2 escrituras

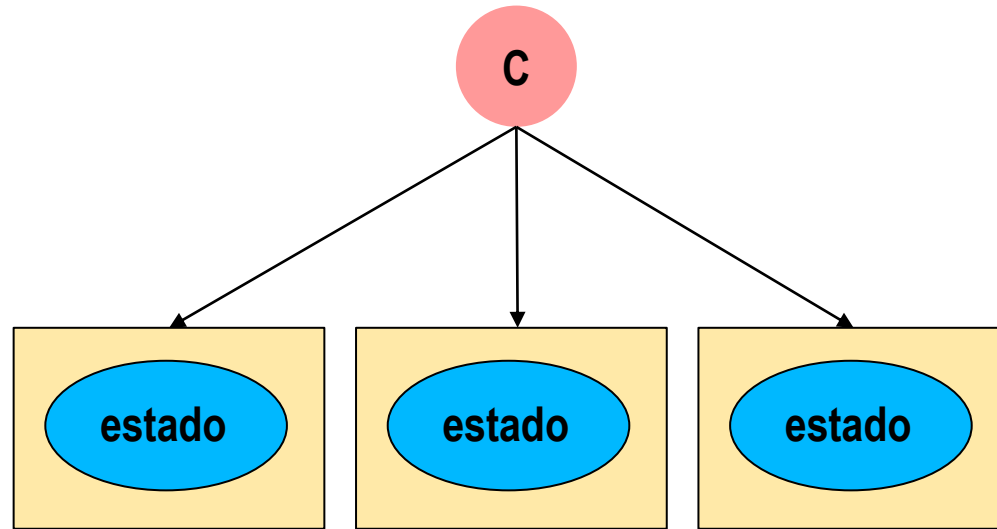


suponiendo que se producen sobre la situación inicial
conducen a un estado incoherente

Cliente/servidor con alta disponibilidad

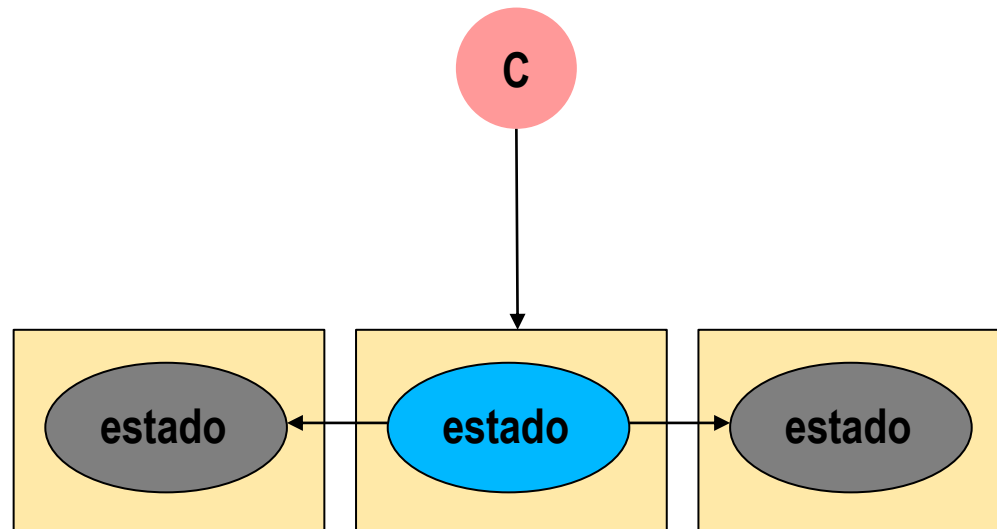
- Servicio con reparto de carga: cada servidor procesa 1 petición
 - Si se cae, se pierde la petición en curso, que puede ser muy larga
- S. alta disponibilidad: debe completarse aunque caiga 1 servidor
 - Requiere uso de múltiples servidores replicados para cada servicio
 - ¿Cómo mantener sincronizado estado de los servidores?
 - Nuevamente problema del consenso distribuido
- Soluciones alternativas (solo esbozadas)
 - Replicación activa: todos los servidores procesan cada petición
 - Replicación pasiva: primario procesa petición; secundarios en *standby*
 - *Hot standby* y *Cold standby*
 - *Warm standby*: Mejora tiempo de recuperación de *Cold standby*
 - Nodo *standby* lee periódicamente estado del almacenamiento
 - Caída primario, nuevo líder solo lee del almacenamiento últimos cambios

Servicio con replicación activa



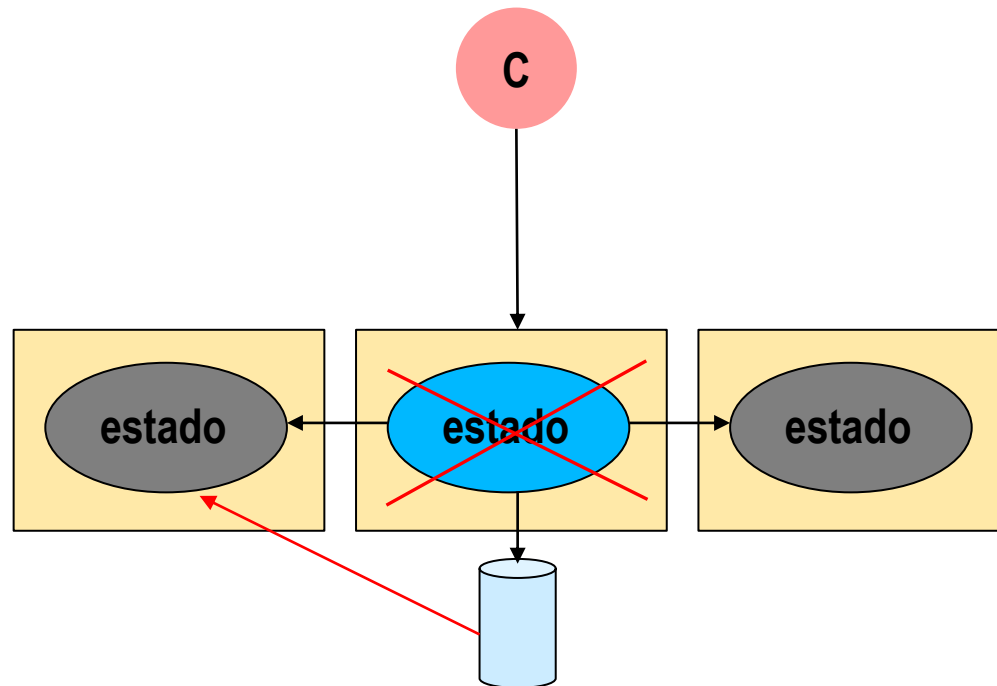
- Todos los servidores reciben y procesan la petición del cliente
- Requiere procesamiento determinista para que los nodos tengan el mismo estado
- Solo un servidor envía la respuesta al cliente
 - Aunque por tolerancia a fallos se puede realizar una votación
- Recuperación inmediata ante caída de un servidor. Si se cae el que respondía al cliente
 - Hay que elegir quién lo hará ahora (elección de líder: tema de sincronización)

Replicación pasiva con *hot standby*



- Solo primario recibe, procesa y responde a la petición del cliente
 - No se requiere servicio determinista
- Primario envía a secundarios los cambios de estado durante el procesado de la petición
- Rápida recuperación ante caída de servidor primario
 - Hay que elegir nuevo primario (elección de líder: tema de sincronización)
 - que continuará inmediatamente con procesado de la petición y responderá al cliente

Replicación pasiva con cold standby

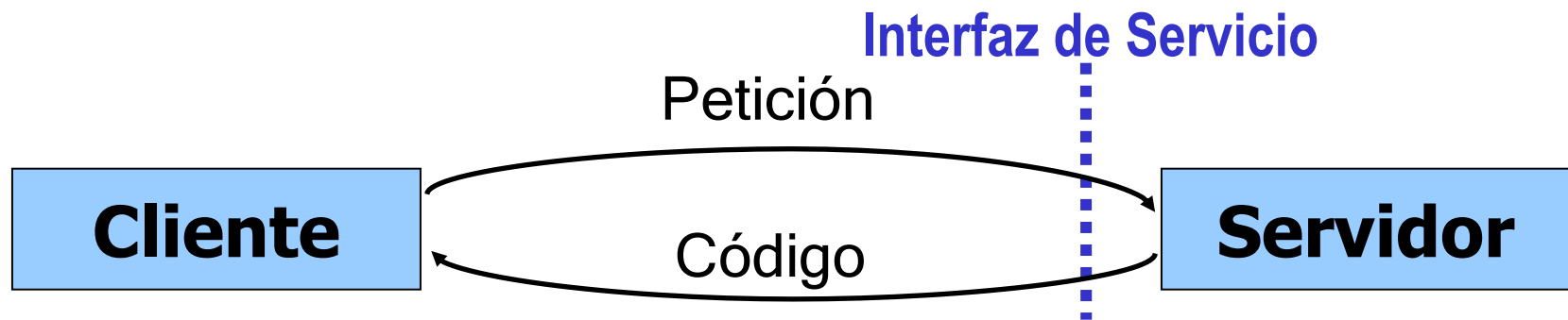


- Primario va guardando estado de la petición en almacenamiento persistente replicado
- Peor tiempo de recuperación ante caída de servidor primario:
 - Hay que elegir nuevo primario (elección de líder: tema de sincronización)
 - que leerá del almacenamiento los cambios de estado desde el inicio de ese servicio
 - y completará la petición respondiendo al cliente

Código móvil

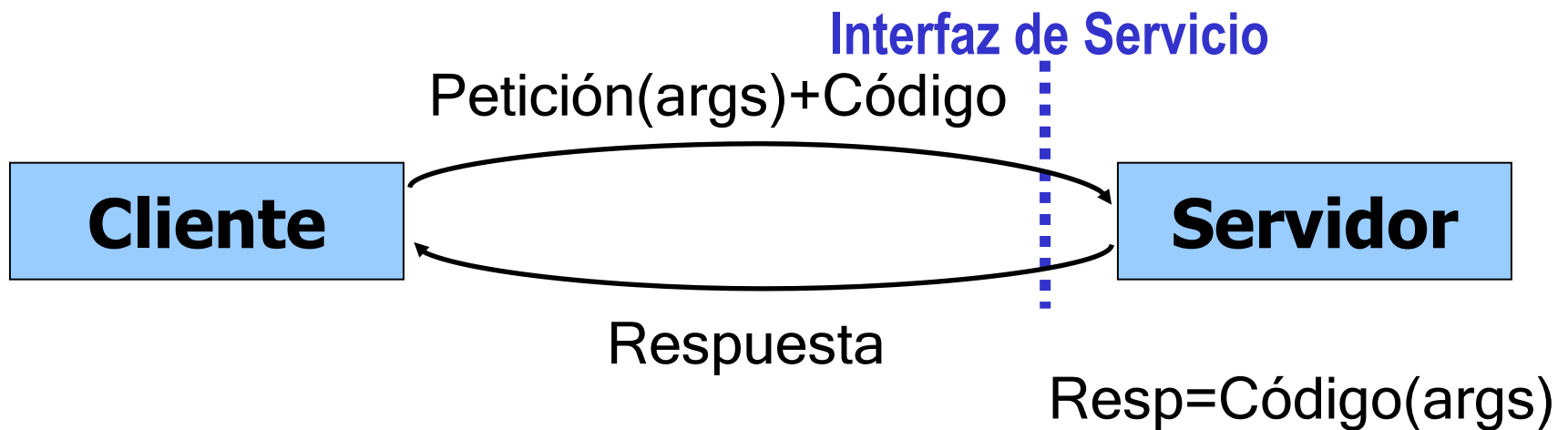
- Viaja el código en vez de los datos y/o resultados
- Código de poder ejecutarse en máquina destino; Requiere:
 - Arquitecturas homogéneas o
 - Interpretación de código o
 - Máquinas virtuales con emulación
- Modelos alternativos
 - Código por demanda (COD)
 - Servidor envía código a cliente
 - P.e. applets o javascript insertados en página web
 - Evaluación remota (REV)
 - Cliente dispone de código pero usa recursos del servidor para ejecución
 - P.ej. *Cyber-Foraging*
 - Agentes móviles
 - Componente autónomo proactivo que viaja por SD

Código por demanda



$\text{Resp} = \text{Código}(\text{args})$

Evaluación remota



Aspectos de diseño de cliente/servidor

Se van a considerar 5 aspectos específicos:

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

Servicio a múltiples clientes: alternativas

- Servidor secuencial
 - Solo factible si servicios muy cortos y no bloqueantes
- Servidor concurrente (p.e. servidor web Apache)
 - Un flujo de ejecución atiende una petición en cada momento
 - Se bloquea esperando datos de ese cliente y envía respuesta
 - *Threads* (**T**) vs. Procesos (**P**)
 - Generalmente *threads*: Más ligeros y comparten más recursos
 - Pero más problemas de sincronización
 - Pueden requerirse procesos si se usa algún módulo no reentrante
- Servidor basado en eventos (p.e. servidor web Nginx)
 - Un flujo de ejecución atiende múltiples peticiones simultáneamente

Esquema servicio web secuencial

while (1)

 acepta conexión

 recibe petición

 lee fichero

 prepara cabecera

 envía cabecera y fichero

Servicio concurrente: alternativas

- Creación dinámica de T/P
 - Cuando llega una petición se crea un T/P
 - Cuando se completa el servicio, se destruye ese T/P
 - Sobrecarga de creación y destrucción
- Conjunto (*pool*) estático de T/P
 - Al iniciarse el servidor crea N T/P
 - Cuando llega una petición se asigna a un T/P libre
 - Al finalizar trabajo, el T/P se queda en espera de más peticiones
 - Poca carga \rightarrow gasto innecesario; Mucha carga \rightarrow insuficientes
- Esquema híbrido
 - Se establece un umbral mínimo n y uno máximo N
 - Al iniciarse el servidor crea n T/P
 - Si llega petición, ningún T/P libre y $n^0 < N \rightarrow$ Se crea un nuevo T/P
 - Si T/P inactivo tiempo prefijado y $n^0 > n \rightarrow$ Se destruye ese T/P

Esquema concurrente dinámico

while (true)

- acepta conexión

- crea *thread* pasándole la conexión

thread

- recibe petición

- lee fichero

- prepara cabecera

- envía cabecera y fichero

- termina el thread

Esquema concurrente con pool

crea N *threads*

while (true)

- acepta conexión

- selecciona un *thread* libre y le asigna la conexión

- rechaza petición si ninguno libre

thread

- while (true)

 - espera asignación

 - recibe petición

 - lee fichero

 - prepara cabecera

 - envía cabecera y fichero

Esquema concurrente híbrido

crea n *threads*

while (true)

- acepta conexión

- si todos *threads* ocupados y no se ha llegado a umbral máximo N

 - crea *thread*

 - rechaza petición si se ha llegado al umbral

- selecciona un *thread* libre y le asigna la conexión

thread

- while (true)

 - espera asignación o plazo de tiempo

 - si se cumple plazo y no se ha llegado a umbral mínimo n

 - termina el *thread*

 - recibe petición

 - lee fichero y prepara cabecera

 - envía cabecera y fichero

Servicio basado en eventos

- 1 flujo ejecución atiende múltiples peticiones simultáneamente
- Pide ser notificado cuando se produzcan eventos relevantes
 - Petición de conexión, llegada de datos por una conexión...
- No usa operaciones bloqueantes
- Si servicio requiere operación bloqueante (p.e. leer fichero)
 - La realiza de forma asíncrona siendo notificado cuando se completa
- Flujo se queda a la espera de ser notificado de algún evento
- Cuando se produce, lo trata y vuelve a esperar
- Para aprovechar paralelismo HW: un flujo/procesador
- Servidor concurrente vs. basado en eventos:
 - Peor escalabilidad
 - Sobrecarga creación/destrucción/planificación de procesos/*threads*, cambios de contexto, más gasto de memoria (p.e. pilas de *threads*)...

Esquema basado en eventos

Pide ser notificado cuando llegue una nueva conexión

while (true)

espera próximo evento y lo trata

Evento de nueva conexión

acepta conexión y pide ser notificado cuando lleguen datos por ella

Evento de nuevos datos en una conexión

lee la petición

lectura asíncrona del fichero y pide ser notificado cuando termine

Evento de fin de lectura de fichero

prepara cabecera

envía asíncronamente por conexión correspondiente cabecera y fichero

Gestión de conexiones

- 2 opciones si C/S usa esquema de comunicación con conexión
- Una conexión para cada petición de un cliente
 - Cada operación cliente-servidor conlleva
 - conexión, envío de petición, recepción de respuesta, cierre de conexión
 - Más sencillo pero mayor sobrecarga (¡9 mensajes con TCP!)
 - 3 para conexión + 4 petición/respuesta y sus ACKs + 2 desconexión
- Conexiones persistentes: N peticiones cliente misma conexión
 - Más complejo pero menor sobrecarga
 - Dado que servidor admite nº limitado de conexiones C
 - C clientes pueden acaparar el servicio
 - Dificulta reparto de servicio entre clientes
 - Dificulta reparto de carga en esquema con escalado horizontal
 - Posibilita el *pipeline* de las peticiones
 - Enviar la siguiente petición sin esperar la respuesta de la previa
 - Facilita *server push*

Evolución gestión de conexiones en HTTP

- Cliente accede a una página web:
 - Debe de solicitar múltiples objetos al mismo servidor
 - Todos los objetos *inline* en esa página
 - Habitualmente, más de 100 objetos por página
- HTTP/1.0
 - Una conexión para cada petición
- HTTP/1.0 con extensión *keep-alive*
 - Conexiones persistentes
- HTTP/1.1
 - *Pipeline* de peticiones
- HTTP 2
 - Multiplexación de peticiones

Gestión de conexiones en HTTP/1.0

- Una conexión por cada objeto
 - *Connect | GET 1 | Resp 1 | Close | Connect | GET 2 | Resp 2 | Close | ...*
 - Sobrecarga y latencia (*round trip*) de cada conexión
 - Latencia de cada petición: tiene que esperar que se complete previa
- ¿Cómo mejorar el rendimiento de la descarga en esta versión?
- Uso de conexiones simultáneas: objetos se piden en paralelo
 - *Connect | GET 1 | Resp 1 | Close*
 - *.....*
 - *Connect | GET n | Resp n | Close*
- Pero en tandas: n^omáx conexiones simultáneas/cliente limitado
 - Algunos navegadores las limitan a 6
 - *Connect | GET 1 | Resp 1 | Close | Connect | GET 2 | Resp 2 | Close | ...*
 - *.....*
 - *Connect | GET n | Resp n | Close | Connect | GET n+1 | Resp n+1 | Close | ...*

HTTP/1.0 con extensión *Keep-alive*

- Ante problemas de rendimiento de HTTP/1.0
 - Extensión que permite a cliente mantener conexión activa
- Se usa una conexión para pedir todos los objetos de la página
 - *Connect | GET 1 | Resp 1 | GET 2 | Resp 2 | ... | Close*
 - Se elimina sobrecarga y latencia de conexiones adicionales
 - Se mantiene latencia de cada petición
- Uso combinado con conexiones simultáneas:
 - *Connect | GET 1 | Resp 1 | GET 2 | Resp 2 | ... | Close*
 - *.....*
 - *Connect | GET n | Resp n | GET n+1 | Resp n+1 | ... | Close*

Gestión de conexiones en HTTP/1.1

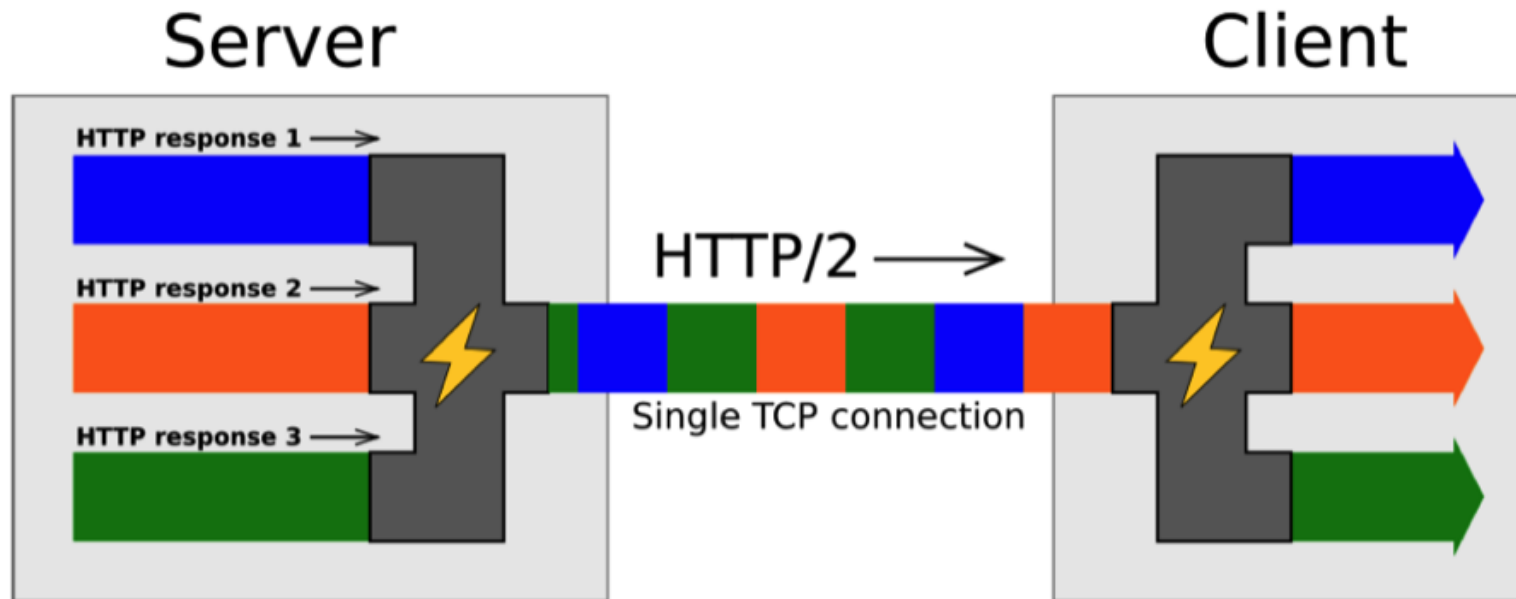
- Además de usar una conexión para pedir todos los objetos
- Se usa pipeline de peticiones
- Se envían todas las peticiones sin esperar su respuesta
 - *Connect | GET 1 | GET 2 | ... | Resp 1 | Resp 2 | ... | Close*
 - No hay latencia acumulada de peticiones
- Estándar exige que respuestas lleguen en orden de petición
 - Orden FIFO → Problema de *Head-of-line blocking*
 - Envío de respuesta de petición corta debe esperar a las anteriores
- Uso combinado con conexiones simultáneas:
 - *Connect | GET 1 | GET 2 | ... | Resp 1 | Resp 2 | ... | Close*
 - *.....*
 - *Connect | GET n | GET n+1 | ... | Resp n | Resp n+1 | ... | Close*

Gestión de conexiones en HTTP/2

- Uso de *multiplexing* en vez de *pipelining*
 - Elimina el problema de *Head-of-line blocking*
 - Respuestas pueden llegar en cualquier orden
- Permite crear múltiples flujos dentro de cada conexión
 - Cada paquete lleva un identificador de flujo único
 - Paquetes de respuestas se mezclan en misma conexión
 - *Connect | GET 1 | GET 2 | ... | Resp 2 | Resp 1 | ... | Close*
- Uso combinado con conexiones simultáneas:
 - *Connect | GET 1 | GET 2 | ... | Resp 2 | Resp 1 | ... | Close*
 -
 - *Connect | GET n | GET n+1 | ... | Resp n+1 | Resp n | ... | Close*

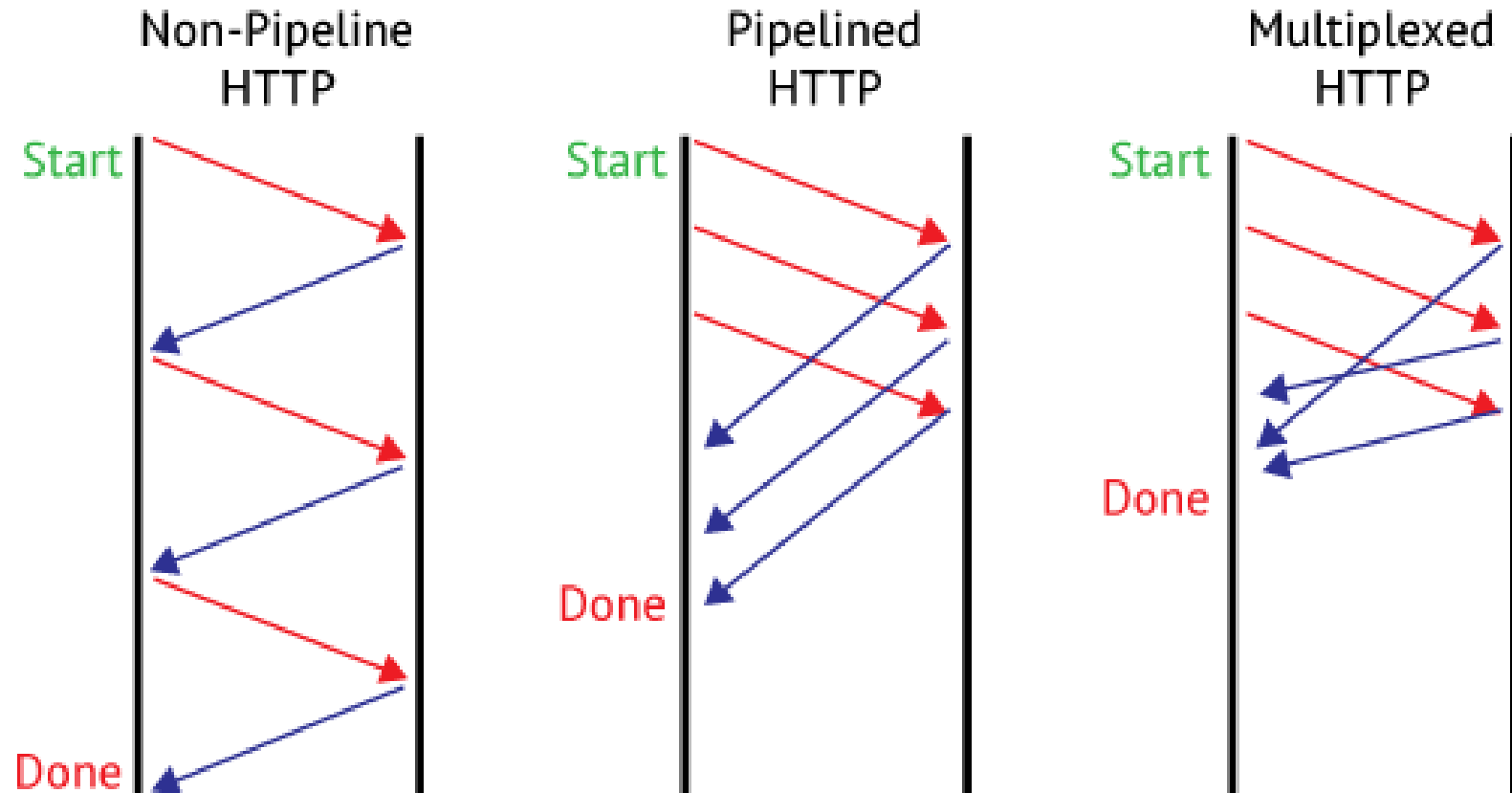
HTTP/2: multiplexación

Envío de respuestas multiplexado



<https://www.nginx.com/blog/7-tips-for-faster-http2-performance/>

Evolución de HTTP



<https://kemptechnologies.com/solutions/http2/>

Client Pull vs Server Push

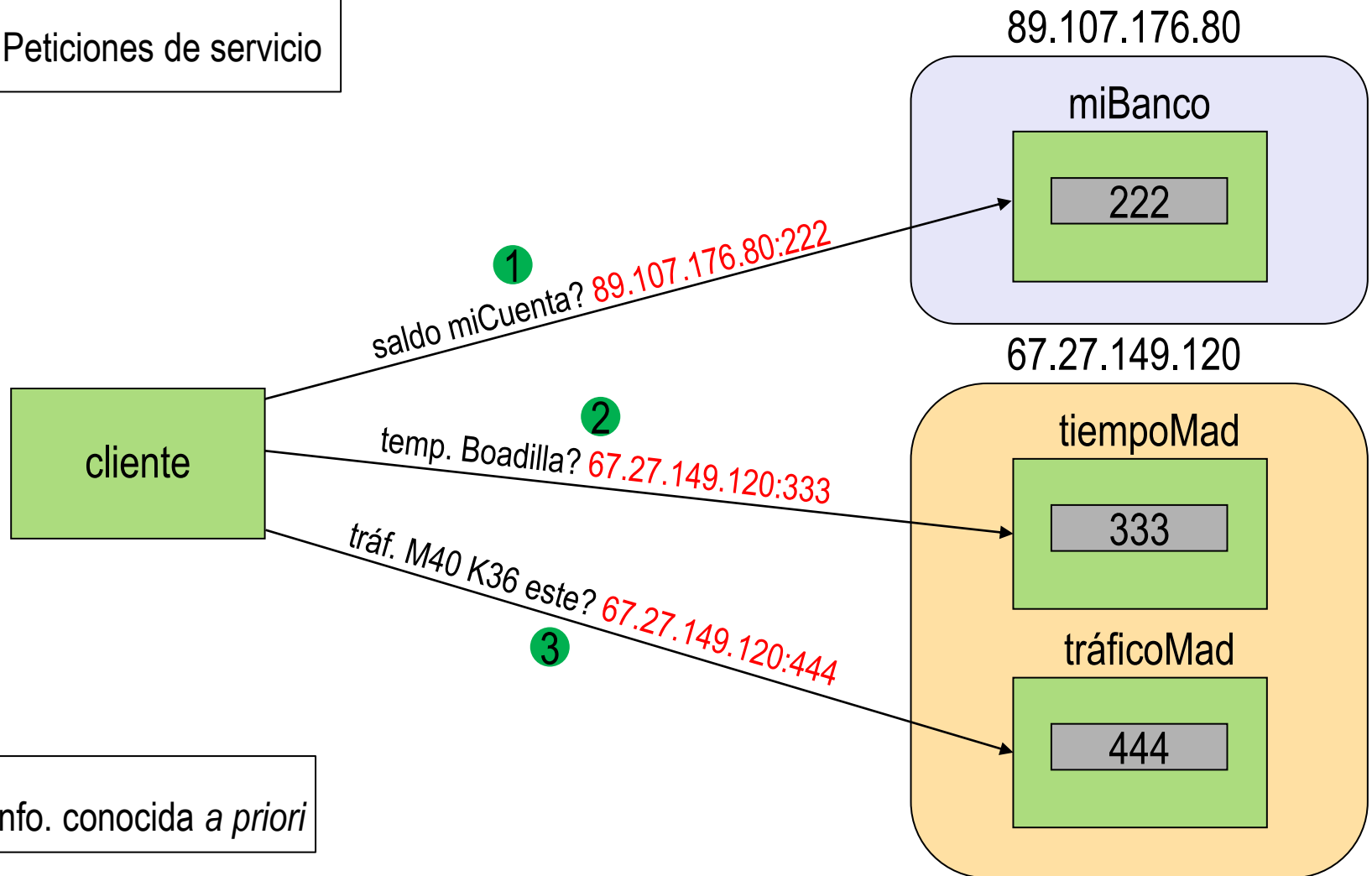
- C/S: modo *pull* → cliente “extrae” datos del servidor
- Escenario: servidor dispone de información actualizada
 - P.e. retransmisión web en modo texto de acontecimiento deportivo
 - P.e. servicio de chat basado en servidor centralizado
- ¿Cómo recibe cliente actualizaciones? Alternativas:
 - Cliente *polling* periódico al servidor
 - Servidor responde inmediatamente, con nuevos datos si los hay
 - *Long Polling*
 - Igual pero servidor no responde hasta que tenga datos
 - *Server Push*:
 - Servidor “empuja” datos hacia el cliente
 - Cliente mantiene conexión persistente y servidor envía actualizaciones
 - Uso de editor/subscriptor en vez de cliente/servidor

Localización del servidor

- Servidor en máquina con dirección *MS* y usando puerto *PS*
 - Además de *MS* y *PS*, protocolo *tcp* o *udp*, pero lo obviemos
- Cliente usa *MS* y *PS* para solicitar servicio: ¿cómo los conoce?
 - Parámetro: no transparencia; no permite migración del servidor
- Uso de componente que guarde información de los servicios
 - Diversos nombres: *binder*, *mapper*, *registry*, *directory*...
 - Huevo-gallina: Cliente debe conocer dir. y puerto de ese componente
- Modo de operación: cada servicio tiene *ID* único
 - Servidor contacta con *binder* y da de alta el servicio *ID*
 - Cliente consulta *binder* para obtener información del servicio *ID*
- Uso de caché en clientes para evitar repetir traducción
 - Necesidad de mantener la coherencia
- Se estudia en tema de servicio de nombres

sin Binder

● Peticiones de servicio

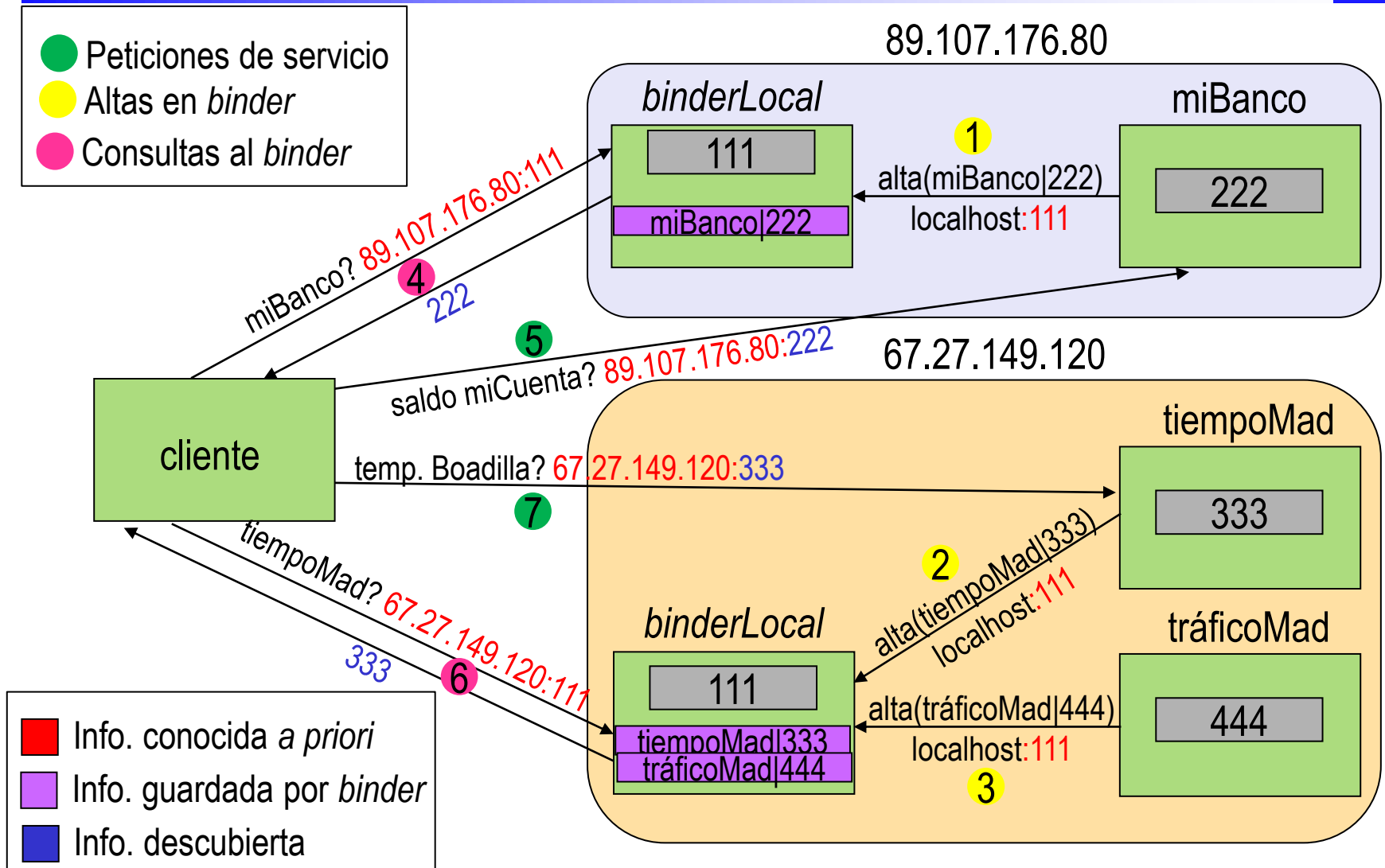


■ Info. conocida a priori

Binder local

- 1 *Binder* en cada máquina (*Java RMI Registry*)
- Guarda correspondencias *ID* servicio y *PS* en esa máquina
- Servidor elige puerto cualquiera (*PS*) e informa a *binder* local
 - Debe conocer *a priori* el puerto que usa el *binder* (*PB*)
 - *Binder* almacena esa información: *ID* servicio \rightarrow *PS*
- Cliente conoce *a priori* máquina servidor *MS* y pto. *binder* *PB*
 - Contacta con *binder* de esa máquina y le pregunta por *ID* de servicio
 - Obtiene *PS*
 - Ya puede enviar la petición a (*MS*,*DS*)
- Servidor puede usar puerto diferente en cada activación
- Pero no permite la migración de un servidor a otra máquina
 - Cliente requiere dir. máquina de servicio para contactar con su *binder*

Modo de operación de *binder* local



Binder global

- 1 *Binder* único para todo el sistema (CORBA)
- Guarda correspondencias *ID* servicio y (*MS*, *PS*)
- Servidor elige puerto cualquiera (*PS*) e informa a *binder* global
 - Debe conocer *a priori* máquina y puerto del *binder* (*MB*, *PB*)
 - *Binder* almacena esa información: *ID* servicio \rightarrow (*MS*, *PS*)
- Cliente conoce *a priori* máquina y puerto del binder *MB* y *PB*
 - Contacta con *binder* global y le pregunta por *ID* de servicio
 - Obtiene *MS*, *PS*
 - Ya puede enviar la petición a (*MS*, *DS*)
- Servidor puede usar puerto diferente en cada activación
- Servidor puede activarse en una máquina diferente cada vez

Modo de operación de *binder* global

- Peticiones de servicio
- Altas en *binder*
- Consultas al *binder*

IP1: 89.107.176.80

miBanco

222

IP2: 67.27.149.120

tiempoMad

333

tráficoMad

444

6.6.6.6

binderGlobal

666

miBanco|222|IP1
tiempoMad|333|IP2
tráficoMad|444|IP2

cliente

5 saldo miCuenta? 89.107.176.80:222

miBanco? 6.6.6.6:666

4 89.107.176.80:222

tráficoMad? 6.6.6.6:666

6 67.27.149.120:444

7 tráf. M40 K36 este? 67.27.149.120:444

1 alta(miBanco|89.107.176.80:222)
6.6.6.6:666

2 alta(tiempoMad|67.27.149.120:333)
6.6.6.6:666

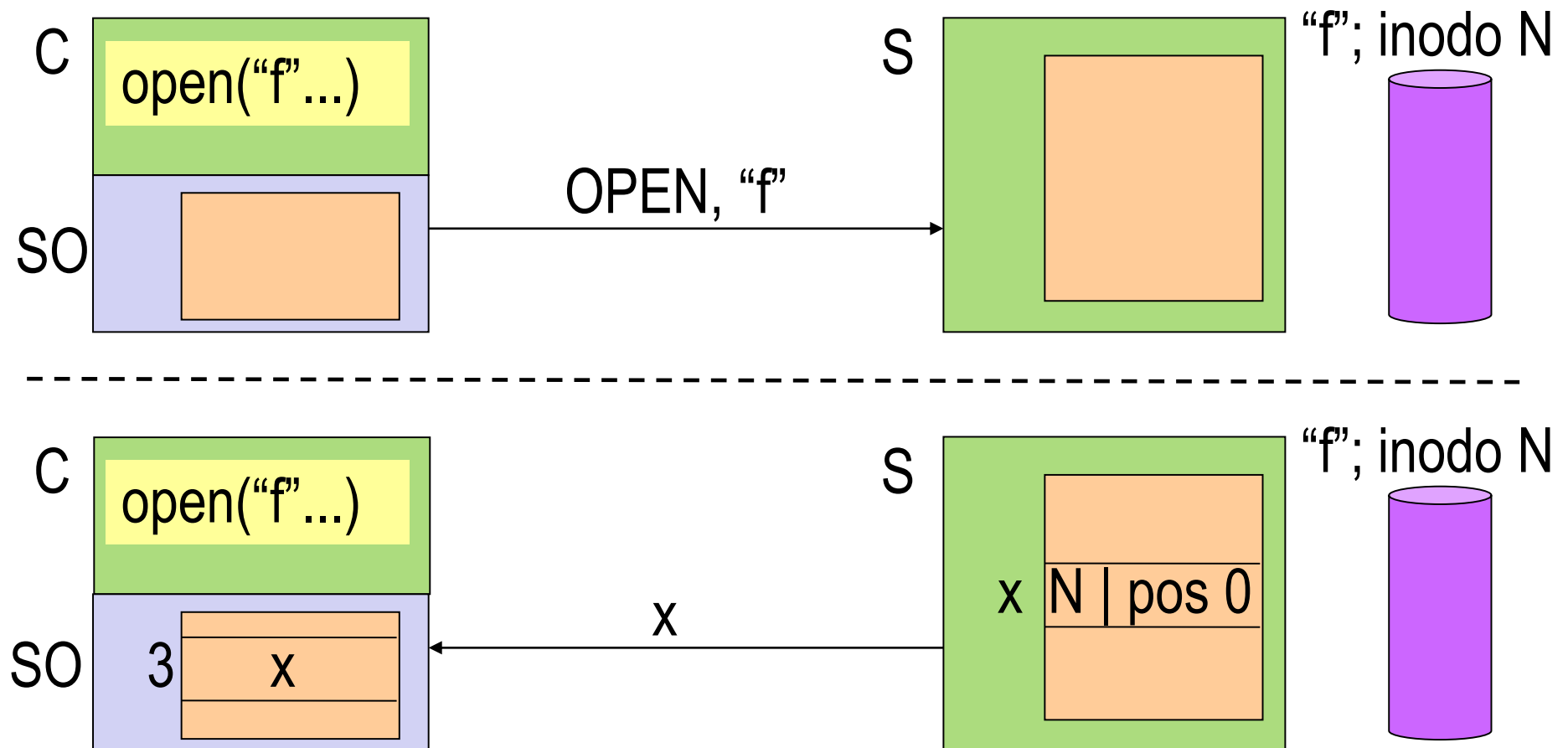
3 alta(tráficoMad|67.27.149.120:444)
6.6.6.6:666

- Info. conocida *a priori*
- Info. guardada por *binder*
- Info. descubierta

Servicio con/sin estado

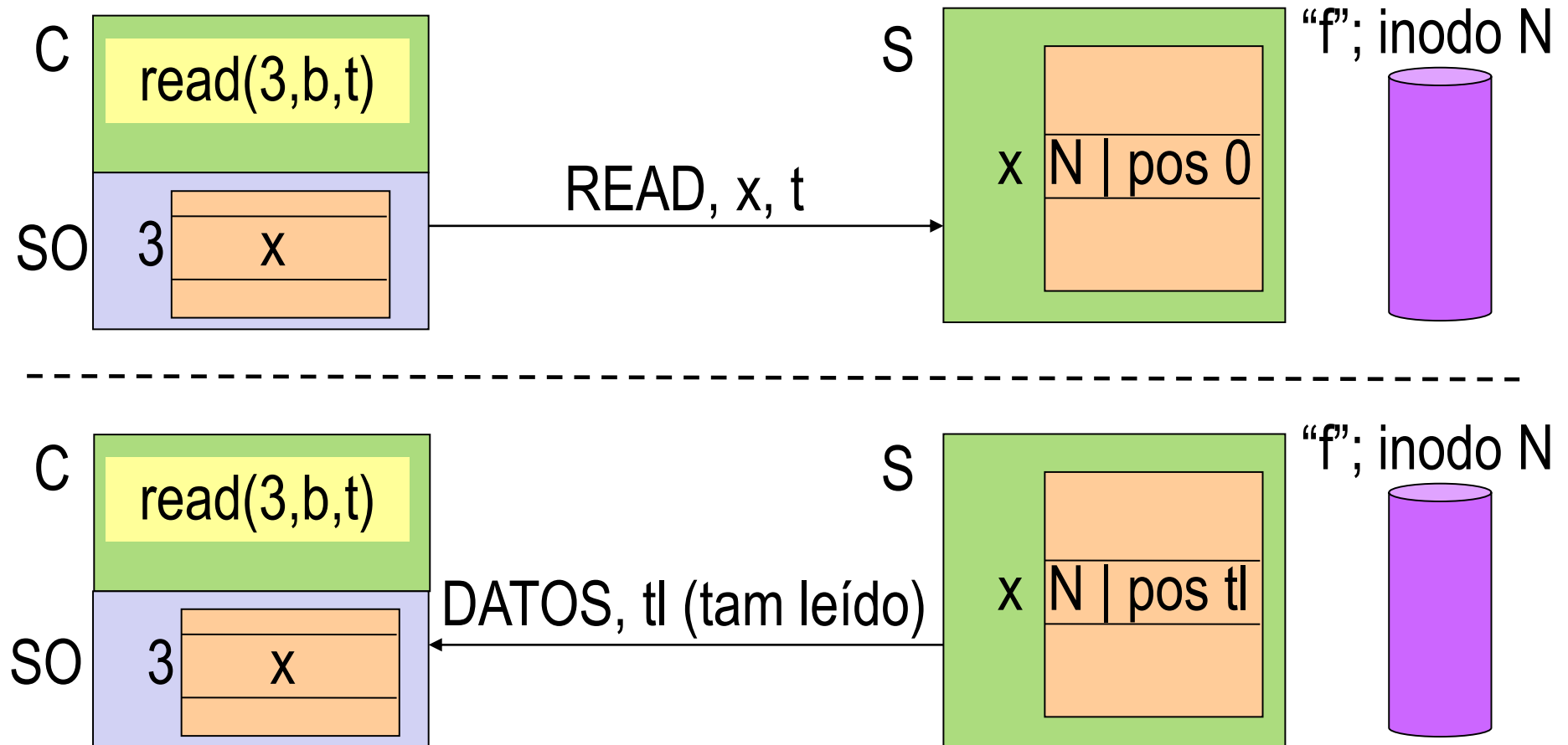
- ¿Servidor mantiene información de clientes? Véase ejemplo
- Ventajas de servicio con estado:
 - Mensajes de petición más cortos
 - Mejor rendimiento (se mantiene información en memoria)
 - Favorece optimización de servicio: estrategias predictivas
- Ventajas de servicio sin estado:
 - Más tolerantes a fallos: servidor reinicia y recibe petición (p.e. READ)
 - Con estado: error no puede interpretarla porque ha perdido el estado
 - Sin estado: OK ya que las peticiones son autocontenidas
 - Reduce nº de mensajes: no comienzos/finales de sesión.
 - Más económico para servidor (no consume memoria)
 - Mejor reparto carga y fiabilidad en esquema con escalado horizontal
 - Cada petición puede ir a un servidor diferente
- Estado sobre servicios sin estado
 - Cliente guarda el estado y lo envía al servidor (p.e. HTTP+cookies)

Servicio de ficheros con estado: OPEN

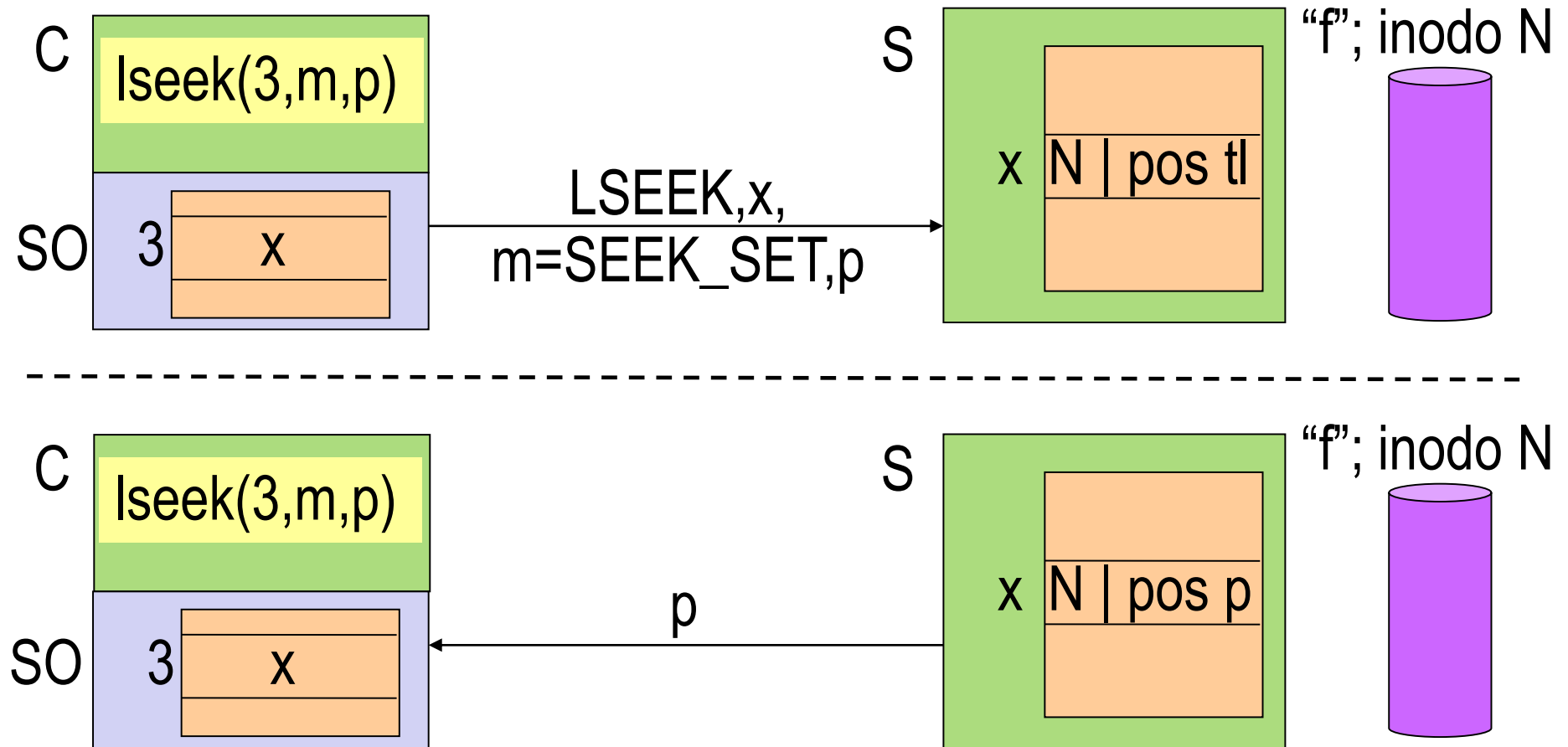


Servidor de ficheros hipotético: versión con estado

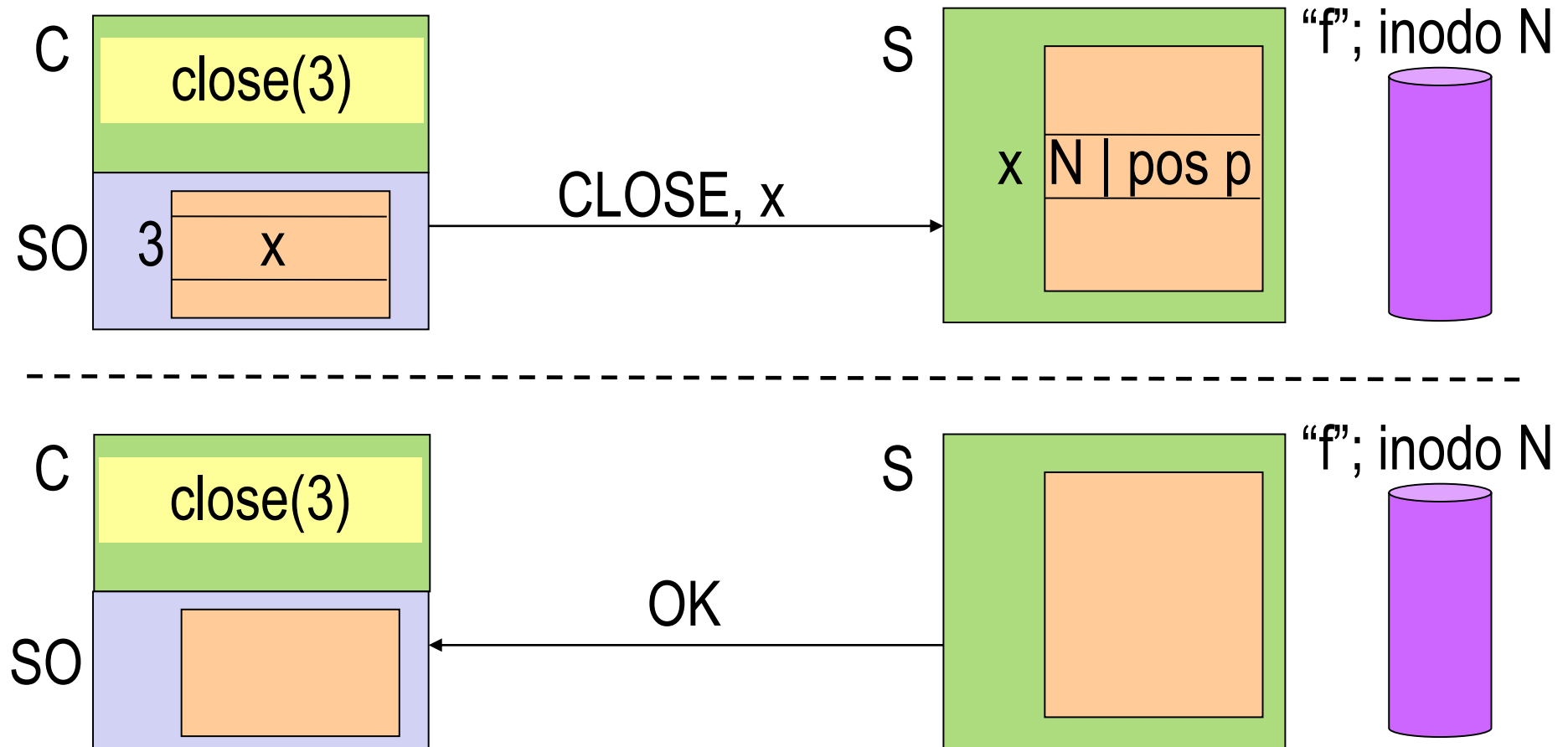
Servicio de ficheros con estado: READ



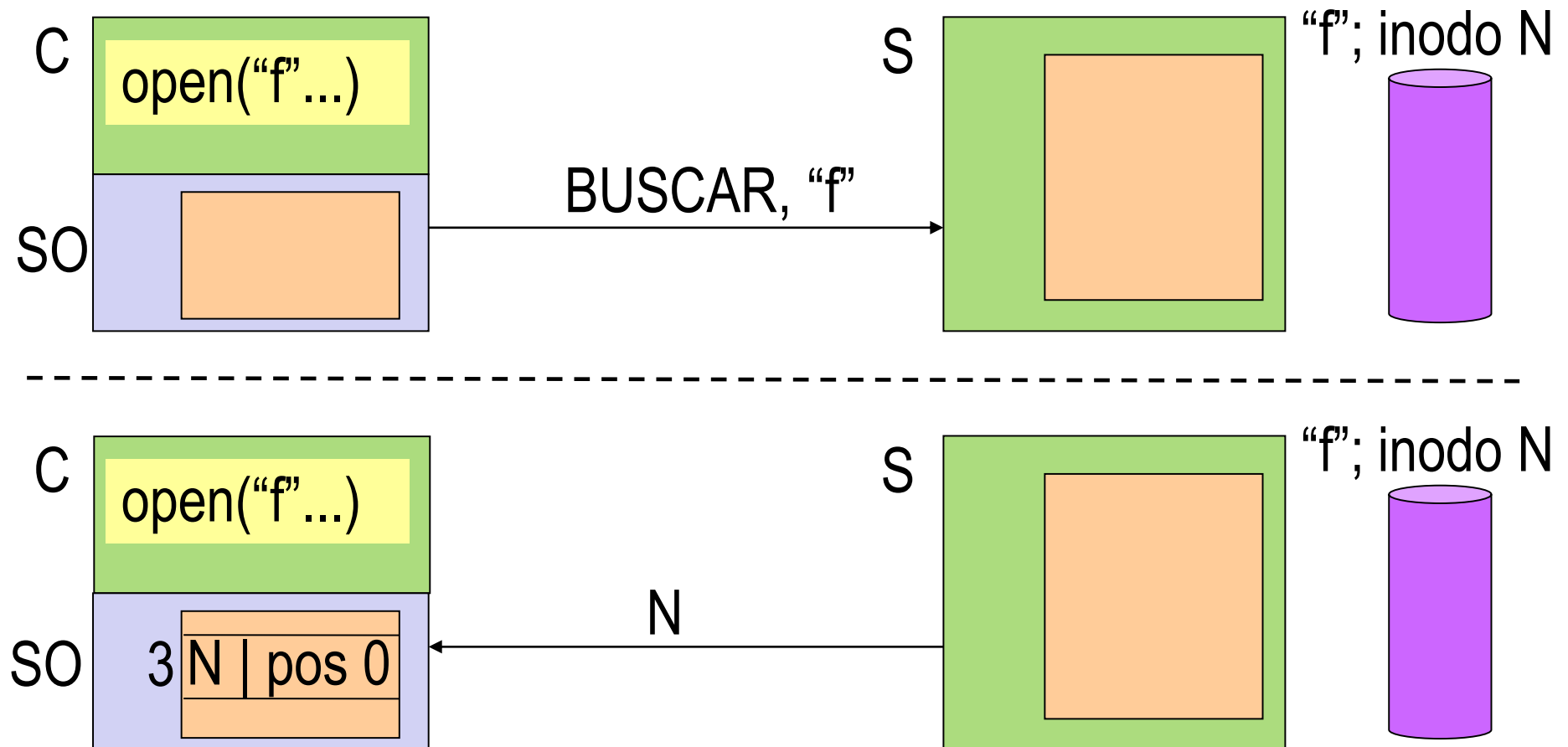
Servicio de ficheros con estado: LSEEK



Servicio de ficheros con estado: CLOSE

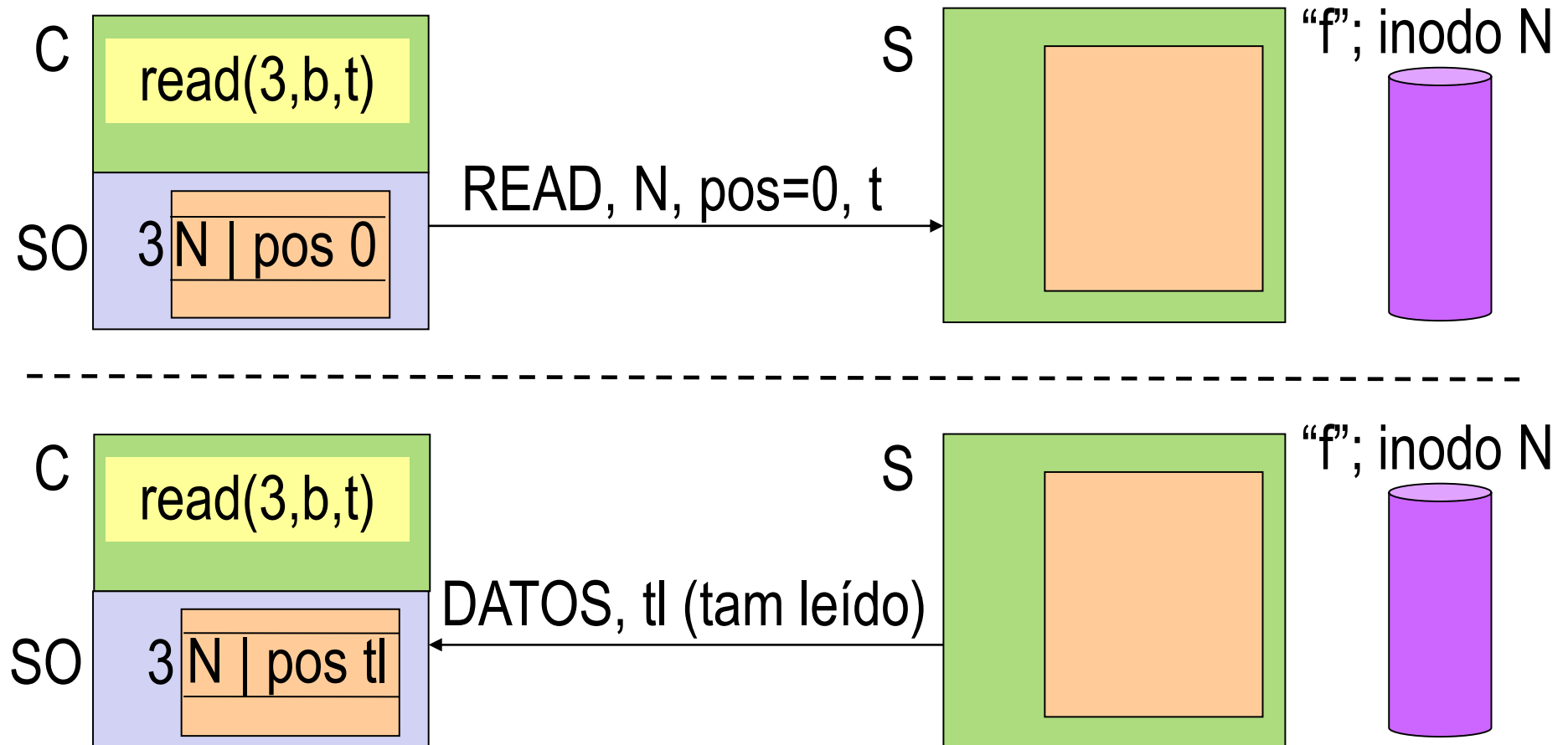


Servicio de ficheros sin estado: OPEN

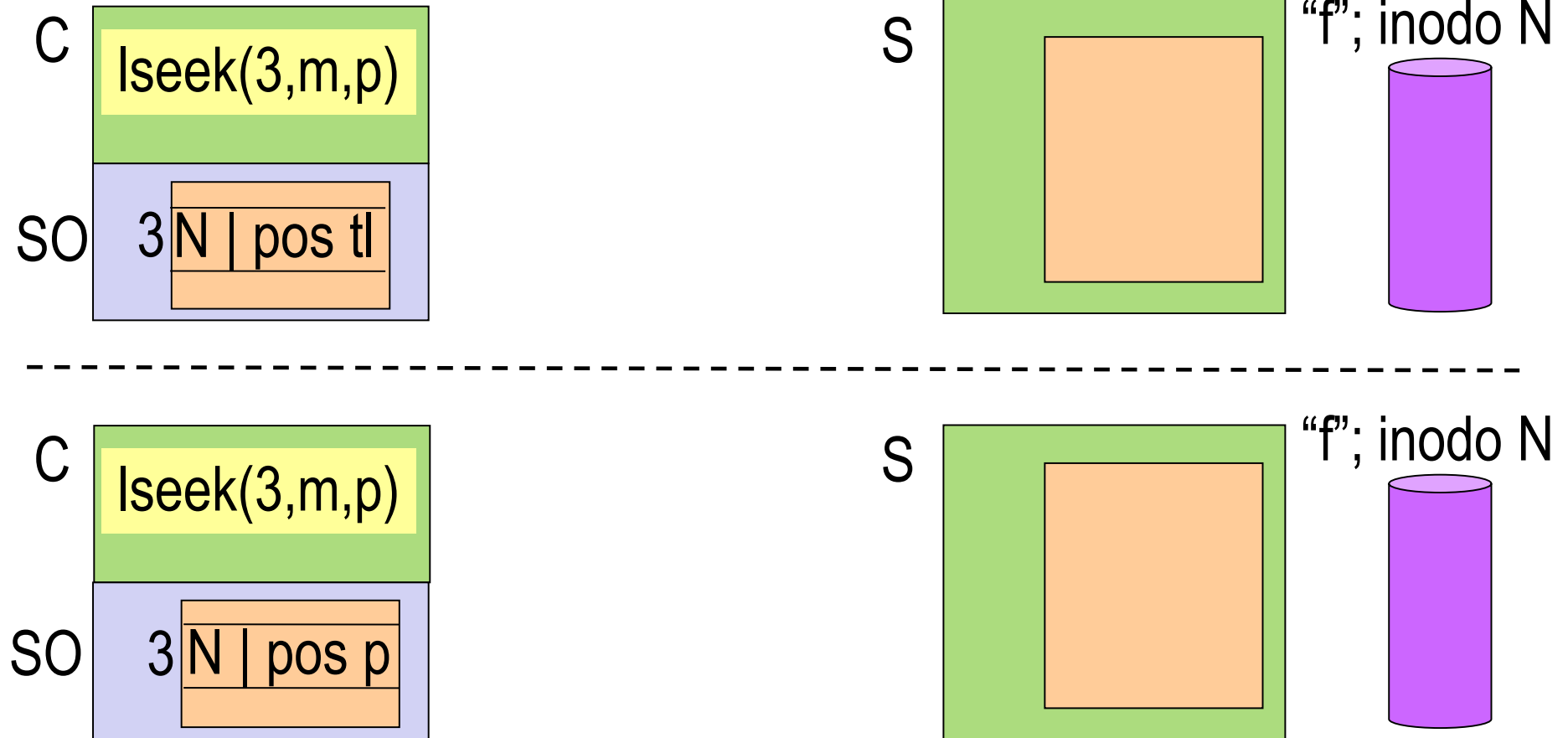


Servidor de ficheros hipotético: versión sin estado (\approx NFS)

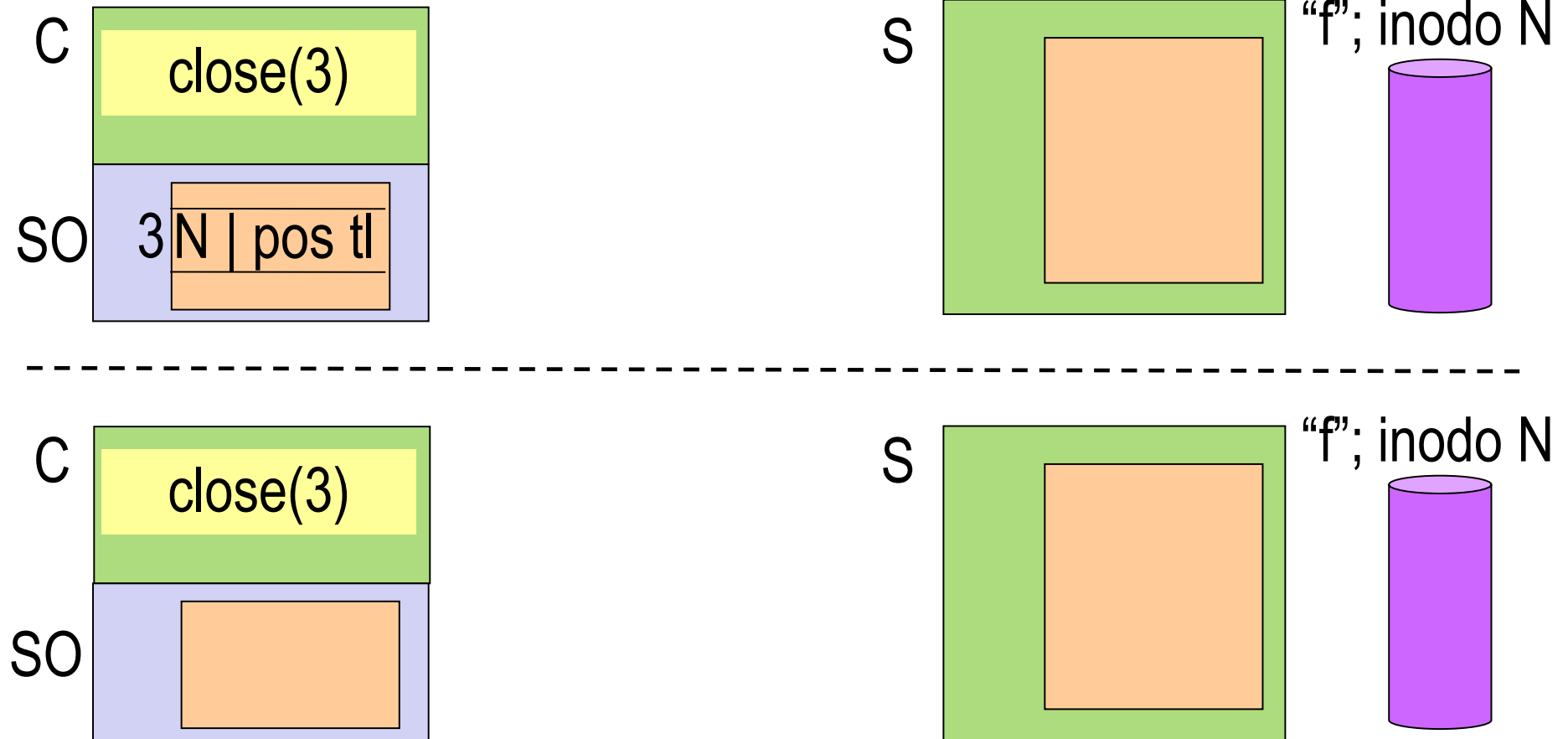
Servicio de ficheros sin estado: READ



Servicio de ficheros sin estado: LSEEK



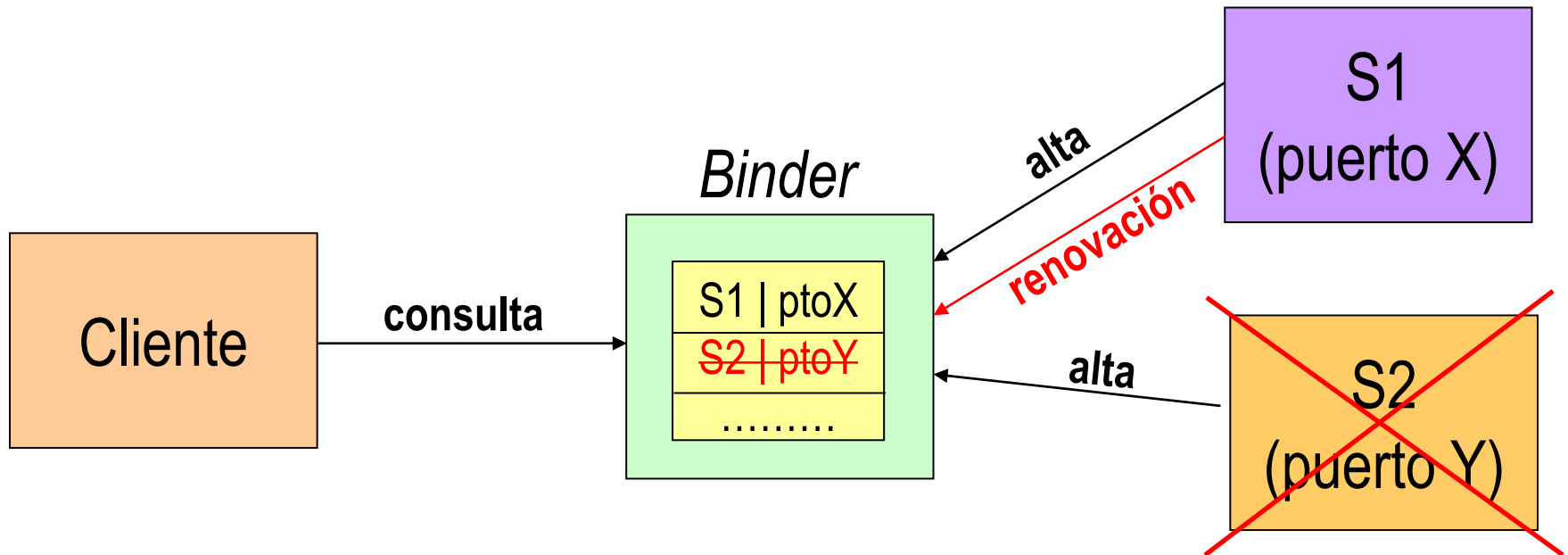
Servicio de ficheros sin estado: CLOSE



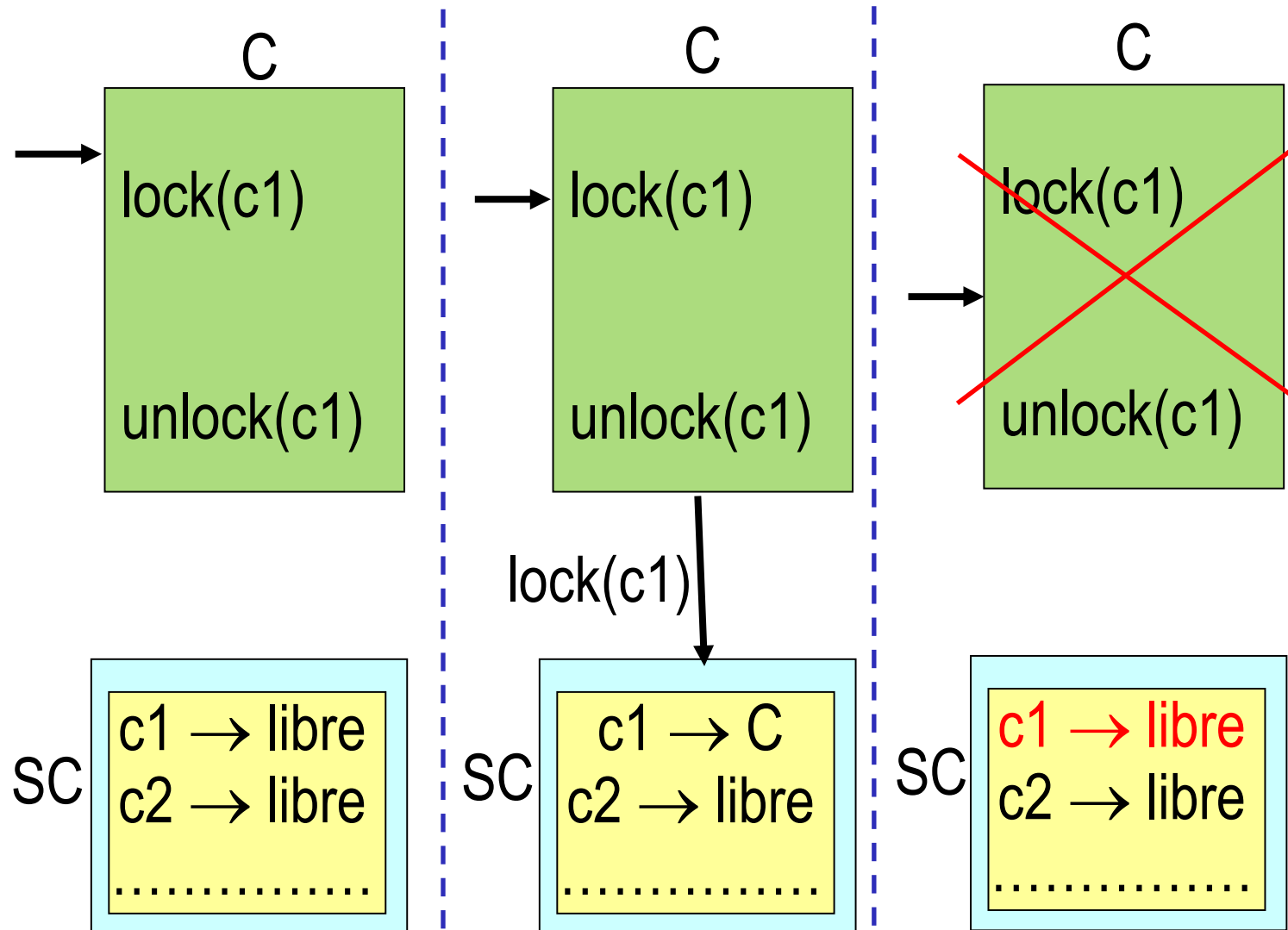
Leases

- Mecanismo para mejorar tolerancia a fallos ante caída de nodo
- Escenario de uso:
 - Proceso/nodo *A* necesita saber si proceso/nodo *B* se cae
 - P.e. servidor DHCP quiere saber si caído nodo al que le asignó una IP
- Solución para detectarlo:
 - *B* debe enviar periódicamente mensajes “estoy vivo”
 - Tiene que renovar el “alquiler” → **lease**
 - Si no renueva: *A* asume que *B* está caído y actúa en consecuencia
 - En DHCP, servidor considera que la IP asignada a ese equipo está libre
- Uso muy frecuente en sistemas distribuidos
 - Puede usarse en *binding* para dar de baja servicios caídos
 - Se utiliza en servicios con estado
 - Si cliente no renueva *lease*, servidor elimina el estado del cliente
 - Ejemplo: servicio de cerrojos distribuidos (tema de sincronización)

Leasing en el binding



Servicio cerrojos distribuido con *leases*



Comportamiento del servicio ante fallos

- Cliente envía petición y no recibe respuesta. ¿Qué ha pasado?
 - Se ha perdido petición o respuesta, solo si comunicación no fiable UDP
 - Se ha caído el servidor
- ¿Qué garantiza el servicio si no llega respuesta? ¿Y si llega?
 - Nada (*maybe*): Servicio puede haberse ejecutado de 0 a N veces
 - Al menos una vez (*at-least-once*): de 1 a N veces
 - Como mucho una vez (*at-most-once*) 0 o 1 vez
 - Exactamente una vez: Sería lo deseable
- Operaciones repetibles (**idempotentes**)
 - Estado resultante es el mismo aunque se repita la operación
 - Cuenta += cantidad (**NO**); Cuenta = cantidad (**SÍ**)
- Operación idempotente + al menos 1 vez \approx exactamente 1
- En HTTP operación GET es idempotente pero POST no

Estrategias ante fallos de comunicación

- Con comunicaciones no fiables (UDP)
 - Después de enviar el cliente activa un temporizador
 - Si se cumple el plazo sin respuesta, retransmisión hasta N veces
 - Petición con n^o secuencia; servidor puede guardar caché respuestas:
 - Si n^o de secuencia duplicado, no procesa petición pero manda respuesta
- Garantías:
 - Si finalmente no llega respuesta, nada (0 a N); ¿Cómo puede ser N ?
 - Sin caché de respuestas, se han perdido las N respuestas
 - Con caché, se ha caído N veces el servidor antes de enviar la respuesta
 - Si finalmente llega la respuesta, *at-least-once* (de 1 a N)
- Con comunicaciones fiables (TCP)
 - Cliente no retransmite: Protocolo asegura no pérdida de mensajes
- Garantías:
 - si respuesta, exactamente 1; Si no, *at-most-once* (0 o 1) por caída

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 n° de época
 - En rearranque de cliente C : transmite ($++n^{\circ}$ de época) a servidores
 - Servidor aborta servicios de C con n° de época menor
 - Uso de *leases*:
 - Servidor asigna *lease* mientras dura el servicio
 - Si cliente no renueva *lease* se aborta el servicio
 - Solo tiene sentido para servicios muy largos
- Abortar un servicio no es trivial
 - Puede dejar incoherente el estado del servidor (p.e. cerrojos)
 - En ocasiones puede ser mejor no abortar

Leasing para tratar caída del cliente



Modelo editor/subscriptor

- Sistema de eventos distribuidos con dos roles:
 - Subscriptor *S* (*subscriber*):
 - Expresa su interés por cierto tipo de eventos (establece un **filtro**)
 - Editor *E* (*publisher*): Genera un evento
 - Se envía a subscriptores interesados en ese tipo de evento
 - Un proceso puede ser editor y subscriptor
- Paradigma asíncrono y desacoplado en espacio
 - Editores y subscriptores no se conocen entre sí (\neq cliente/servidor)
- Normalmente, *push*:
 - Subscriptor recibe inmediatamente los eventos de forma asíncrona
- Alternativa, *pull* (requiere que se almacenen los eventos)
 - Subscriptor recoge los eventos que le corresponden
 - Periódicamente o porque ha sido notificado
- Diversos aspectos de diseño que no estudiamos
 - orden de entrega, fiabilidad, persistencia, prioridad...

Operaciones modelo editor/subscriptor

- Estructura típica de un evento: [*atrib1=val1; atrib2=val2; ...*]
 - Ejemplos de domótica, chat y retransmisión deportiva en texto:

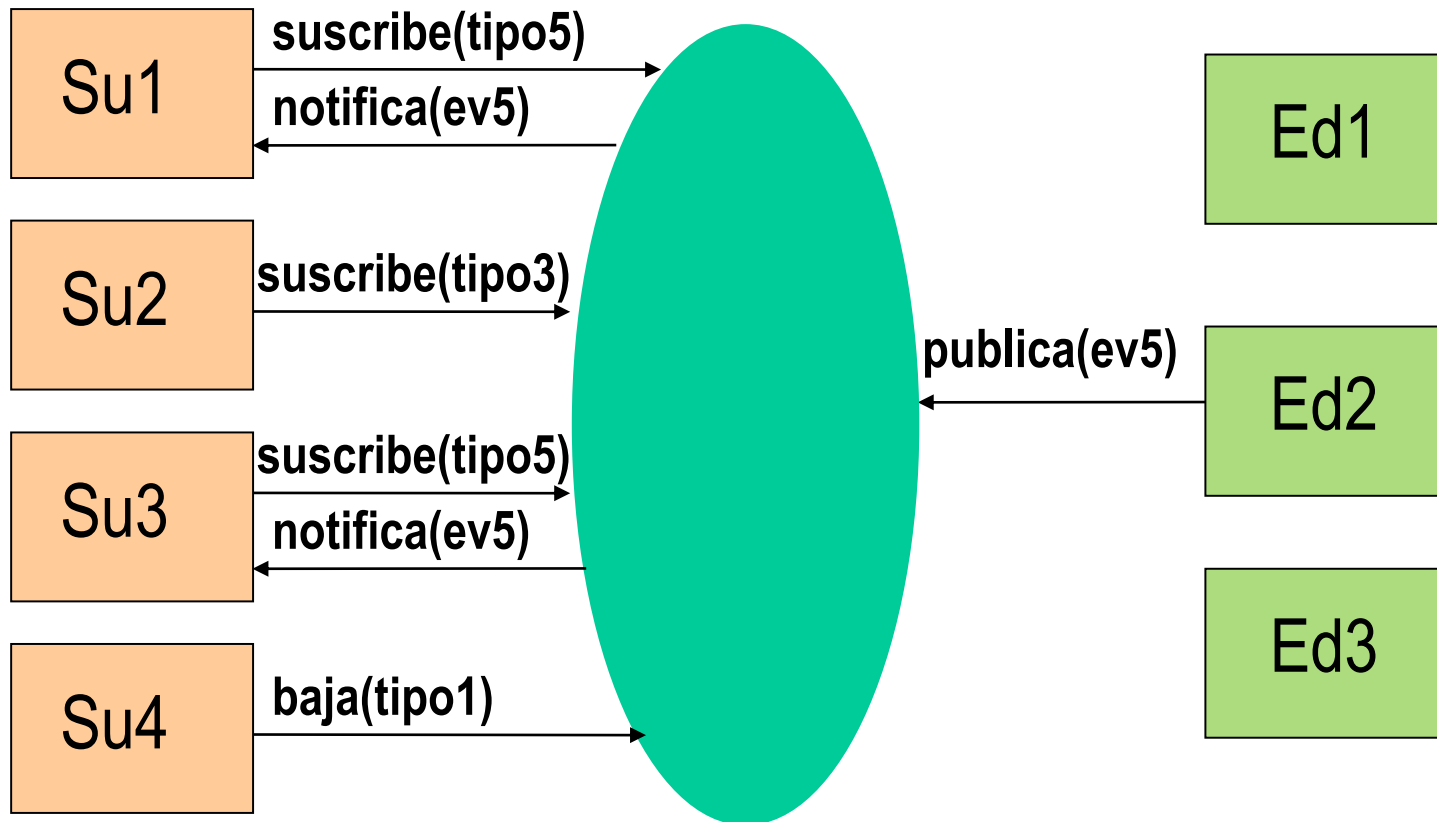
[tipoSensor=temperatura; modelo=DZ1; ubicación=H123; lectura=20º; fecha=....]

[canal=festivales; nick=pepe; texto=hola; fecha=....]

[evento=finalCopaMundo; reportero=juan; comentario=gol; fecha=....]

- Operaciones:
 - *suscribe(tipo)*: suscriptor expresa interés por un tipo de eventos
 - *baja(tipo)*: suscriptor indica cese del interés
 - *publica(evento)*: editor genera un evento
- Aplicaciones:
 - IoT, chat, mercado bursátil, subastas, etc.

Modelo editor/subscriptor (*push*)



Filtro de eventos por tema

- Un atributo se define como el **tema** del evento
[tema]=temperatura; modelo=DZ1; ubicación=H123; lectura=20°; fecha=....]
[tema]=festivales; nick=pepe; texto=hola; fecha=....]
[tema]=finalCopaMundo; reportero=juan; comentario=gol; fecha=....]
- S se suscribe a tema y recibe los eventos con ese atributo
 - P.e. suscripción “temperatura”: eventos de todos sensores de ese tipo
- Organización jerárquica del espacio de temas:
 - P.e. *planta1/apartamentoB/salón/temperatura*
- Uso de comodines en suscripción
- Si se requiere filtrar por otro atributo debe hacerlo la aplicación
 - Ej. Interés mensajes escritos por “pepe” en cualquier canal del chat
 - Aplicación debe suscribirse a todos los canales y
 - descartar todos los mensajes que no sean de “pepe”

Ejemplo de uso de comodines (MQTT)

- Cliente 1 se subscribe a *planta1/+/+/temperatura*
- Cliente 2 se subscribe a *planta1/apartamentoB/#*
- ¿Quién recibe los eventos? **C1**, **C2**, **ambos** o **ninguno**
 - sensor temperatura cocina del apto B de la planta 1 informa de 15°
 - sensor humedad salón del apto B de la planta 1 informa de 50%
 - sensor temperatura baño del apto A de la planta 1 informa de 35°
 - sensor temperatura del salón del apto B de la planta 3 informa de 0°

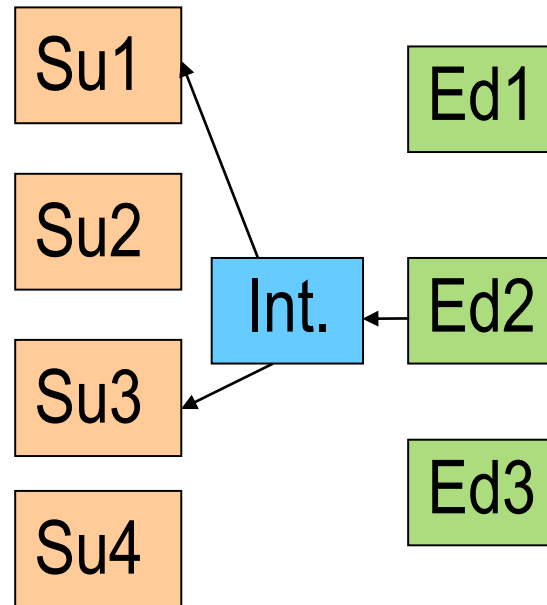
encaja con cualquier valor en múltiples niveles

+ encaja con cualquier valor en ese nivel

Filtro de eventos por contenido

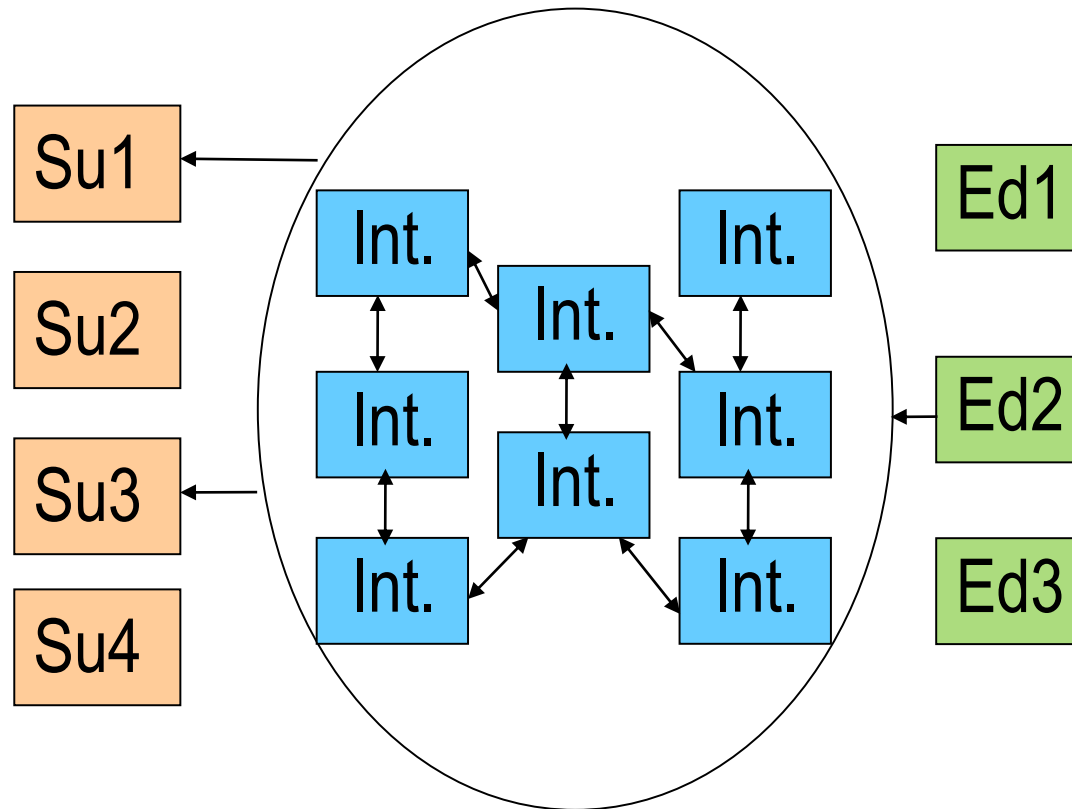
- Suscripción expresa una condición sobre atributos del evento
- Similar a una consulta a una base de datos
 - Uso de lenguaje para expresar la condición (\approx SQL)
- Filtrado de grano más fino y dinámico que usando temas
 - Ej. Interés mensajes escritos por “pepe” en el chat posteriores a X
`nick=pepe & fecha > X`
- Simplifica la aplicación subscriptora
 - Puede expresar mejor lo que necesita
 - Y recibir solo los eventos que realmente le interesan
 - Reduciendo el consumo de ancho de banda
- Pero complica esquema Ed/Su
 - ¿Cómo gestionar todas esas condiciones?
 - Algunas pueden tener expresiones comunes

Implementación EdSu con intermediario



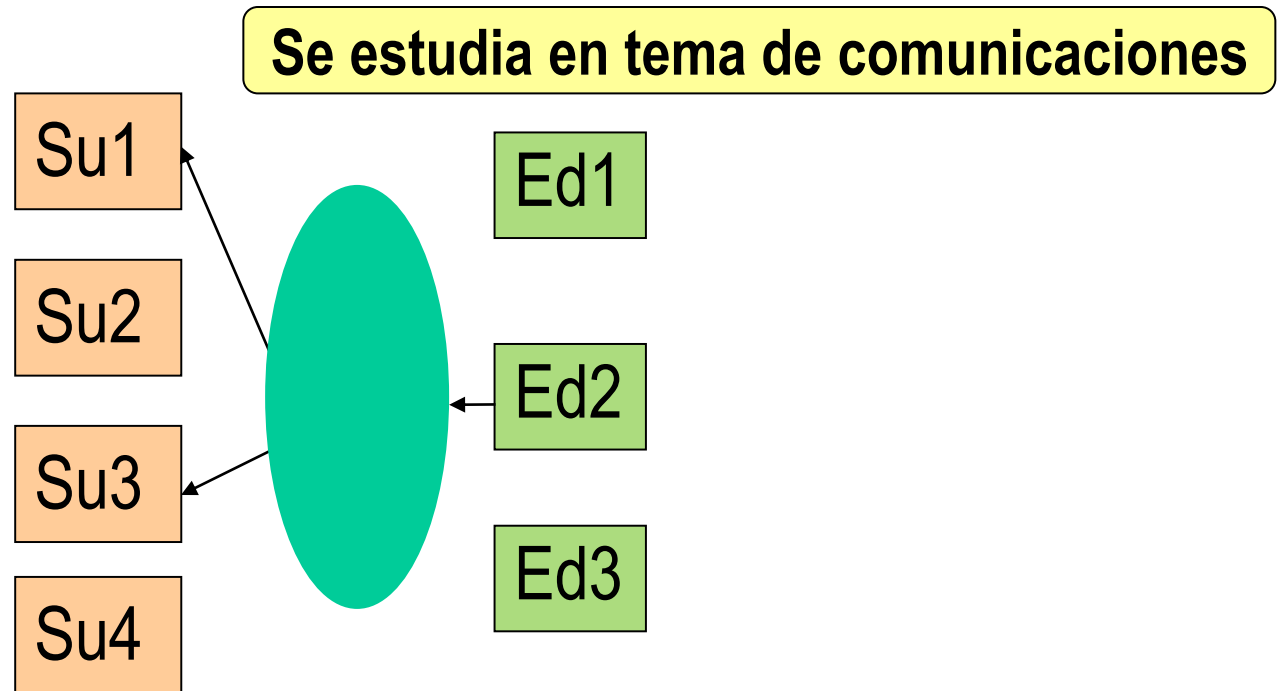
Cuello de botella y punto único de fallo

Implementación EdSu con red interm.



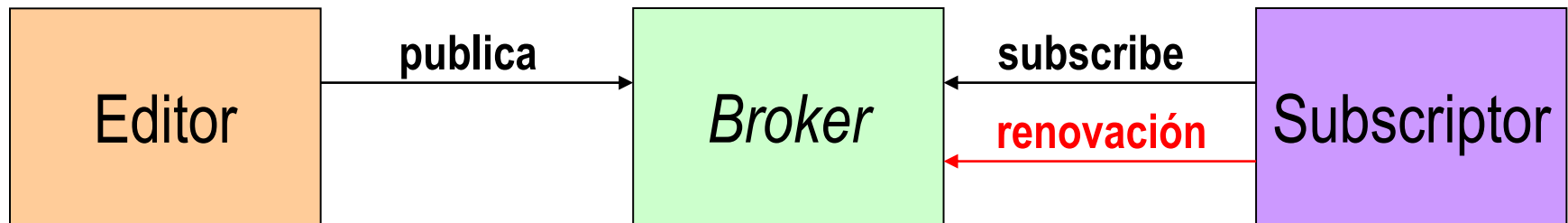
**Red de intermediarios puede optimizar encaminamiento de eventos
Y pueden repartirse cálculo de las condiciones si filtro por contenido**

Implementación EdSu con común. grupo



Comunicación de grupo permite implementar EdSu basado en temas asignando una dirección de grupo a cada tema

Leasing en editor/subscriptor



***Broker* guarda información de los suscritos a cada tema**
Si no hay renovación se elimina ese proceso de todas sus suscripciones

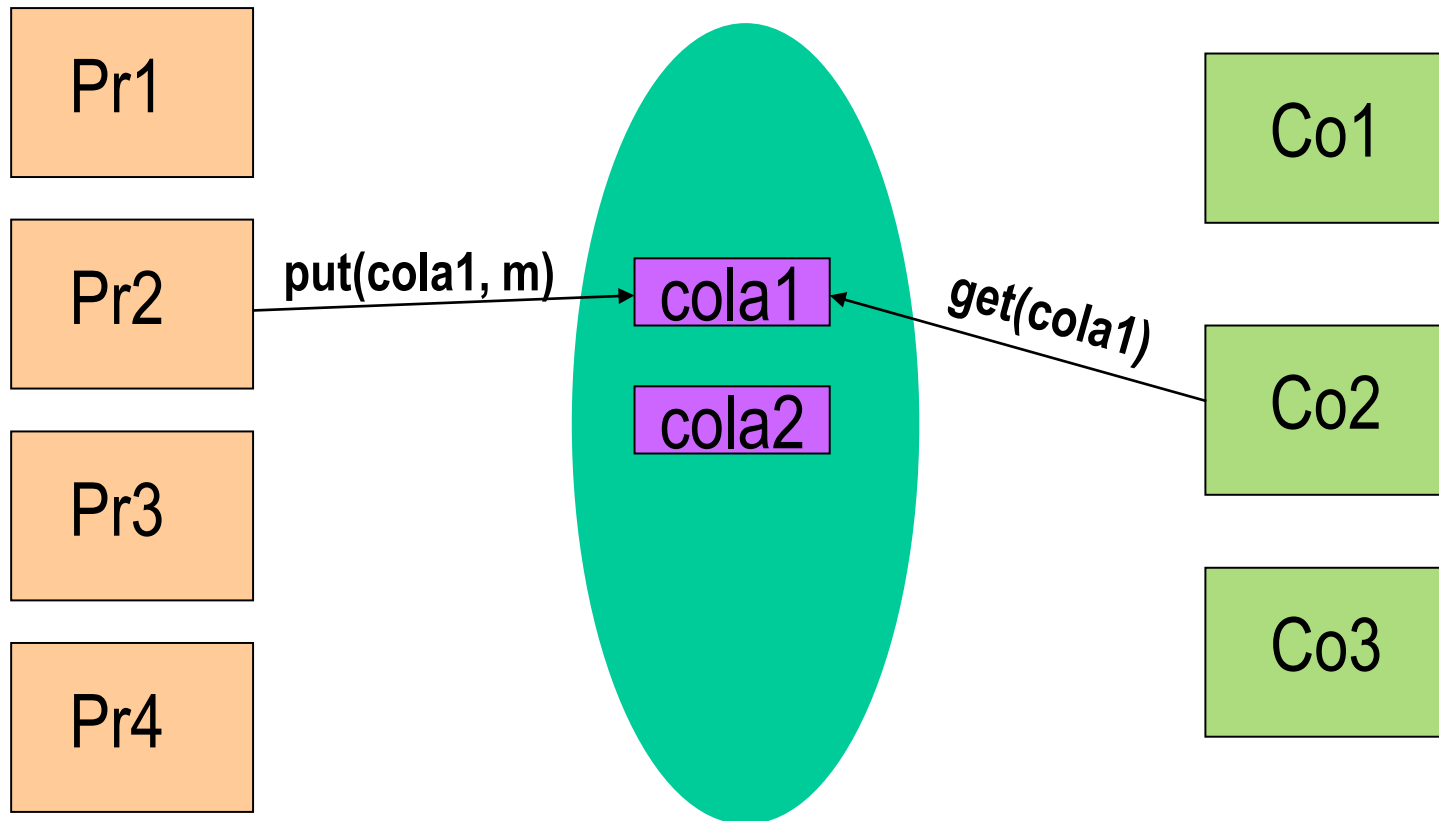
Ejemplos de uso EdSu basado en temas

- Chat:
 - Roles: aplicación de chat es editor y subscriptor
 - Temas: las salas o canales
 - Suscripción|baja: entrar|salir de la sala; Publicar: escribir
- Domótica: apps (climatización, seguridad) interés en sensores
 - Roles: sensor es editor y aplicación es subscriptor
 - Temas: el tipo de sensores
 - Baja|Sus.: (des)interés en tipo sensor; Publicar: sensor envía valor
- Retransmisión en texto de jornada en torneo de tenis
 - Varios reporteros por partido: deben “escucharse” para no “pisarse”
 - Roles: usuario es subscriptor; reportero es editor y subscriptor
 - Temas: cada partido
 - Baja|Sus.: de(seleccionar) partido; Publicar: reportero escribe

Modelo productor/consumidor

- Gestión de almacenes/colas de mensajes
- Dos roles:
 - Productor *Prod* envía (*put*) mensaje a una determinada cola
 - Consumidor *Cons* solicita leer (*get*) mensaje de una cierta cola
 - Solicitud bloqueante (si no hay mensaje, espera) o no bloqueante
- Mensaje enviado por un *Prod* lo recibe un único *Cons*
- Reparto de carga automático entre consumidores
- Facilita escalado automático: añadir más consumidores
- Paradigma desacoplado en el espacio y en el tiempo
 - Productores y consumidores no se conocen entre sí
 - Consumidor puede leer un mensaje de un productor que ya no existe
- Diversos aspectos de implementación similares a EdSu
 - orden de entrega, fiabilidad, persistencia, prioridad...
 - uso de un único intermediario vs múltiples

Modelo productor/consumidor



Encadenando dos colas

