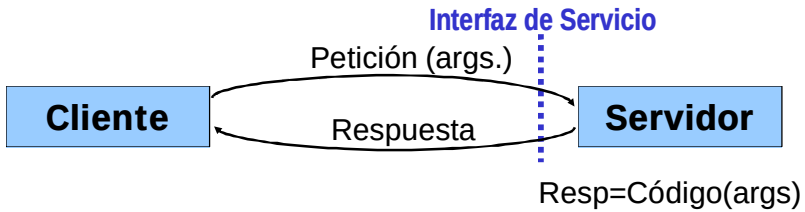
Grado de acoplamiento

- 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 interaccionando **no** tienen que coincidir en tiempo
- Ej. de ambos desacoplamientos: 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."*

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

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)*
- Cierta OP se mueve del servidor al cliente (cliente + pesado)
  - Ej. Javascript en navegador valida letra NIF en formulario
  - Algunas peticiones pueden resolverse sin intervención del servidor
    - Mayor autonomía del cliente y más ágil respuesta al usuario
    - Mejor escalabilidad: Cliente gasta menos recursos de red y de servidor
  - Otras no, pero procesamiento en servidor más rápido (no realiza OP)
    - Mejor escalabilidad: Cliente gasta menos procesamiento de servidor
    - aunque a veces se puede gastar más ancho de banda de red:
      - Datos intercambiados pueden estar menos "procesados" (más volumen)
- Algunas ventajas de clientes ligeros
  - Menos software: menos coste de mantenimiento y mejor seguridad

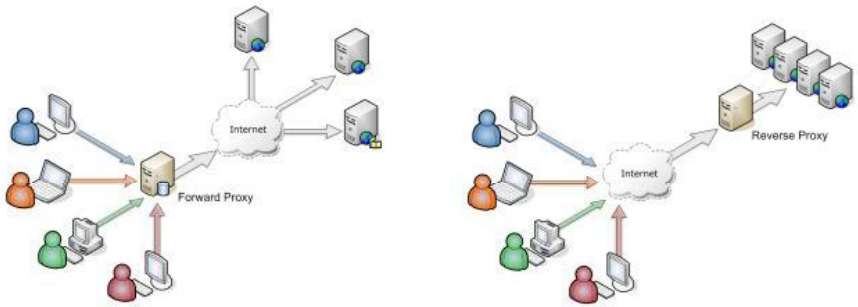
Cliente/servidor con caché

- 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
  - Sistema de ficheros 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
- 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 el del servidor:
  - *Proxy* se comporta como cliente y servidor convencional
  - 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, distribución de servicios específicos,…

Forward vs Reverse Proxy

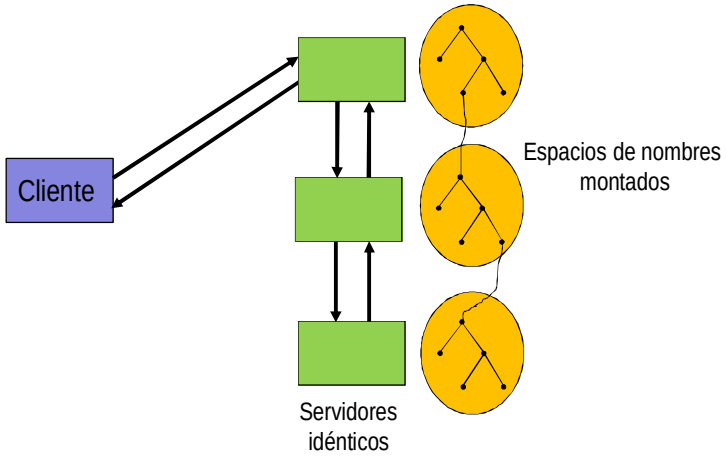


https://www.quora.com/Whats-the-difference-between-a-reverse-proxy-and-forward-proxy

Cliente/servidor jerárquico

- Servidor actúa como cliente de otro servicio
  - Igual que biblioteca usa servicio de otra biblioteca
- División vertical
  - 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
- División horizontal
  - Múltiples servidores idénticos cooperan en servicio
    - Cierta similitud con P2P
  - Traducir nombre de fichero en SFD o nombre de máquina con DNS

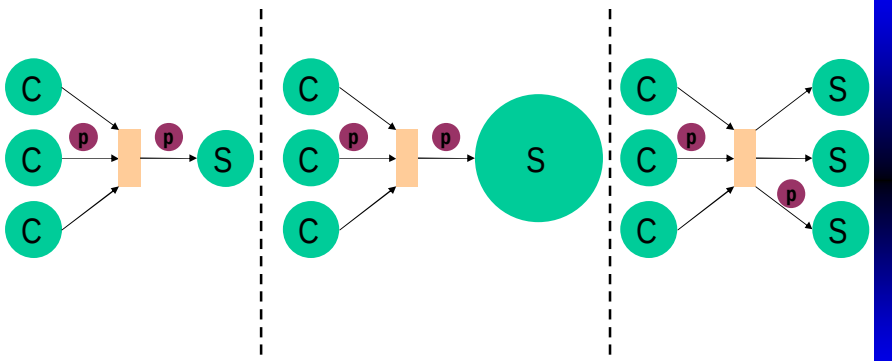
División horizontal



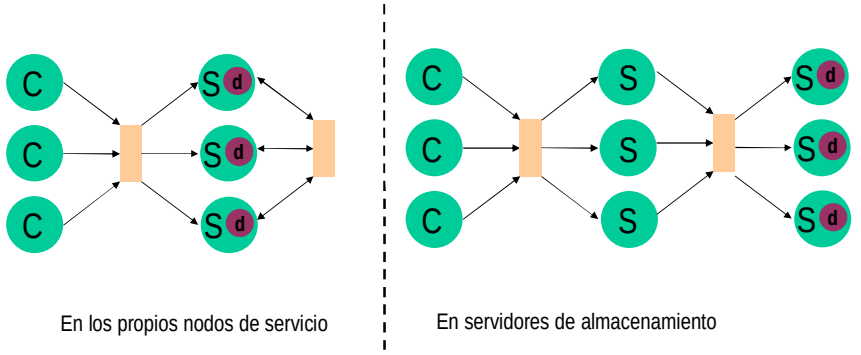
Cliente/servidor con reparto de carga

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

Scale-up vs Scale-out



Scale-out con datos replicados



Teorema CAP (Eric Brewer)

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

Latency vs. *Consistency*

- Teorema CAP sólo 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)
  - Sino (***E****lse*): LC especifica si se elige *consistency* o *latency*
- Posibles sistemas:
  - PCEC: Siempre asegura *consistency*
  - PAEC: Sólo asegura *consistency* si no hay partición
  - PAEL: Nunca asegura *consistency*
  - PCEL: Sólo asegura *consistency* si hay partición→ sin mucho sentido

Actualización de réplicas

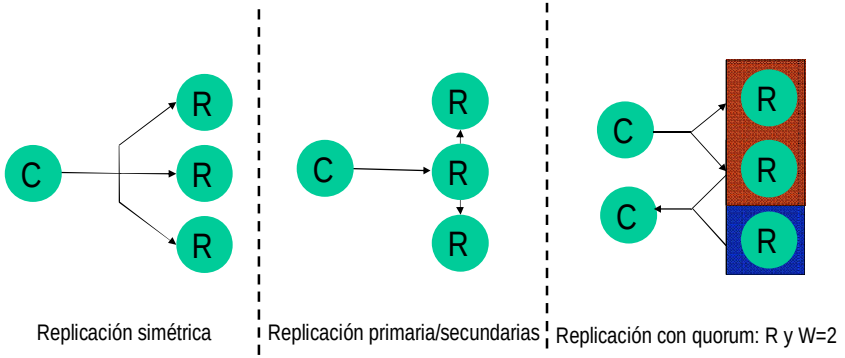
- *Consistency* (C) depende de cómo se actualicen las réplicas
- Tres alternativas:

1. Replic. simétrica: Actualización simultánea de todas las réplicas
   - Para C: asegurar escrituras de clientes se procesan en mismo orden
     - Hay que secuenciar las operaciones de escritura
     - Lo que provoca una sobrecarga → mala latencia (L)
2. Replicación con copia primaria/maestra y secundarias
   - Por cada dato, una réplica es elegida como primaria/maestra
   - Se actualiza copia primaria que lo propaga a secundarias
   - Escritura síncrona: no se completa hasta total propagación
     - No se procesan lecturas ni escrituras de ese dato hasta que termine
     - Asegura C pero sacrificando L
   - Escritura asíncrona: no se espera hasta total propagación
     - Lecturas de copias secundarias pueden obtener valores obsoletos
     - Asegura buena L pero sacrificando C

Actualización de réplicas

3. Uso de quorum (suponiendo N réplicas)
   - Escritura simultánea síncrona en W réplicas
     - Restantes réplicas se actualizarán asíncronamente
   - Lectura síncrona de R réplicas
     - Se elige el valor más actual (cada dato lleva asociado nº versión)
   - Si R + W > N
     - Lectura y escritura simultánea tienen al menos un nodo común
     - No se puede leer un dato obsoleto
     - Se asegura C pero sacrificando L por la necesidad de más sincronía
   - Si R + W ≤ N
     - Lectura y escritura simultánea pueden no tener ningún nodo en común
     - Se puede leer un dato obsoleto
     - Mejora L por menor nº de nodos involucrados, pero no se asegura C

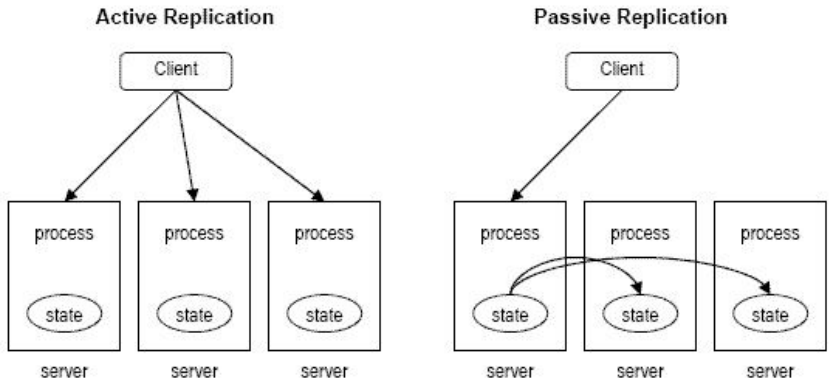
Actualización de réplicas



Cliente/servidor con alta disponibilidad

- Servicio con reparto de carga: cada servidor procesa 1 petición
  - Si se cae, se pierde la petición en curso
- S. alta disponibilidad: debe completarse aunque caiga servidor
  - Requiere uso de múltiples servidores replicados para cada servicio
- Soluciones alternativas (de mejor a peor t. de recuperación)
  - Replicación activa (puede incluir también votación)
    - Todos los servidores reciben y procesan la petición del cliente
    - Requiere procesado determinista para que nodos tengan mismo estado
  - Replicación pasiva: Primario procesa petición; Secundarios (*standby*)
    - *Hot standby:* P envía estado a S; estado de S sincronizado con P
    - *Warm standby*:
      - P guarda estado en almacenamiento persistente replicado; P envía periódicamente estado a S; estado de S no totalmente actualizado;
      - caída P ⟶ S actualiza estado leyendo últimos cambios del almacenamiento
    - *Cold standby*:
      - S desactivado; P guarda estado en almacenamiento persistente replicado;
      - caída P ⟶ Activa S y reconstruye estado leyendo todo el almacenamiento

Repl. activa vs. pasiva con *hot standby*

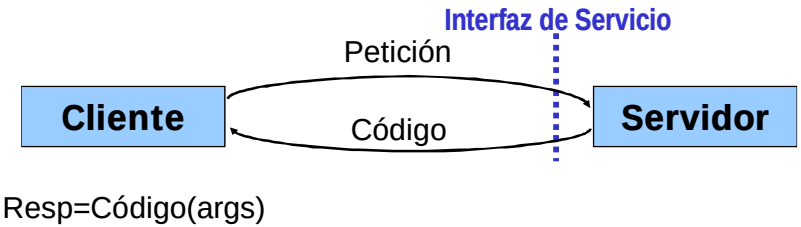


https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/

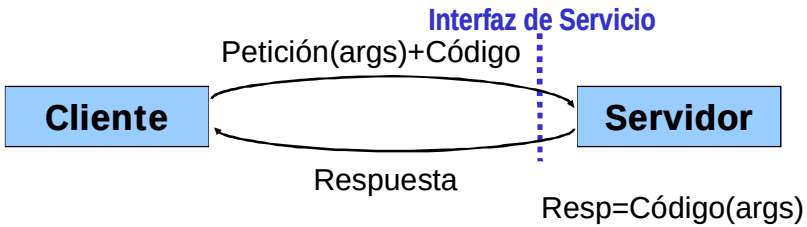
Código móvil

- 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



Evaluación remota



Servicio a múltiples clientes

- Servidor concurrente (p.e. servidor web Apache)
  - Un flujo de ejecución atiende sólo una petición en cada momento
  - Se bloquea esperando datos de ese cliente y envía respuesta
- Servidor basado en eventos (p.e. servidor web Nginx)
  - Un flujo de ejecución atiende múltiples peticiones simultáneamente
  - Espera (y trata) evento asociado a cualquiera de las peticiones
  - Implementación en UNIX de espera simultánea; alternativas:
    - *select/poll/epoll*; uso de señales de t.real; operaciones asíncronas (*aio*)
  - Para aprovechar paralelismo HW: un flujo de ejecución/procesador
- Servidor concurrente vs. basado en eventos:
  - Peor escalabilidad (*The C10K problem*: http://kegel.com/c10k.html)
    - Sobrecarga creación/destrucción/planificación de procesos/*threads*, más cambios de contexto, más gasto de memoria (p.e. pilas de *threads*),...
  - Programación menos "oscura"
  - *Why threads are a bad idea (for most purposes)?*
  - *Why events are a bad idea (for high concurrency servers)?*

Servidor concurrente: alternativas

- *Threads* (**T**) vs. Procesos (**P**)
  - Generalmente *threads*: Más ligeros y comparten más recursos
    - Pero más problemas de sincronización
- Creación dinámica de *T/P* según llegan peticiones
  - Sobrecarga de creación y destrucción
- Conjunto (*pool*) 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

Esquema servicio web secuencial

while (1)
- acepta conexión
- recibe petición
- abre fichero
- lee fichero
- prepara cabecera
- envía cabecera y fichero

Esquema concurrente dinámico

while (1)
- acepta conexión
- crea *thread* pasándole la conexión

thread
- recibe petición
- abre fichero
- lee fichero
- prepara cabecera
- envía cabecera y fichero

Esquema basado en eventos

Registra manejadores: nueva conexión, nuevos datos, fin lectura fichero
while (1)
- espera próximo evento y lo trata

Evento de nueva conexión
- acepta conexión (crea un contexto asociado a la misma)

Evento de nuevos datos
- recibe petición
- abre y lee fichero asíncronamente
- actualiza contexto para vincular lectura de fichero y conexión

Evento de fin de lectura de fichero
- prepara cabecera
- extrae del contexto la conexión asociada
- envía cabecera y fichero asíncronamente

Esquema concurrente con *pool*

crea N *threads*
while (1)
- acepta conexión
- selecciona un *thread* y le asigna la conexión

thread
- while (1)
  - espera asignación
  - recibe petición
  - abre fichero
  - lee fichero
  - prepara cabecera
  - envía cabecera y fichero

Esquema concurrente híbrido

crea N *threads*
while (1)
    acepta conexión
    si todos *threads* ocupados y no se ha llegado a umbral máximo
        crea *thread*
    selecciona un *thread* y le asigna la conexión
thread
    while (1)
        espera asignación o plazo de tiempo
        si se cumple plazo y no se ha llegado a umbral mínimo
            termina el *thread*
        recibe petición
        abre y lee fichero
        prepara cabecera
        envía cabecera y fichero

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!)
  - Protocolos de transporte orientados a C/S (T/TCP, TCP *Fast Open*)
- Conexiones persistentes: *N* peticiones cliente misma conexión
  - Más complejo pero menor sobrecarga
  - Dado que servidor admite nº limitado de conexiones
    - Dificulta reparto de servicio entre clientes
  - Implica que servidor mantiene un estado
  - 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
  - Una media de 100 objetos por página (https://daniel.haxx.se/http2)
- HTTP/1.0
  - Una conexión por cada petición
- HTTP/1.0 con *keep-alive extension*
  - 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 (*round trip*) de cada petición
- ¿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: nºmáx conexiones simultáneas/cliente limitado
  - 2 en estándar original; actualmente navegadores suelen limitar 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 *Keep-alive Extension*

- 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*

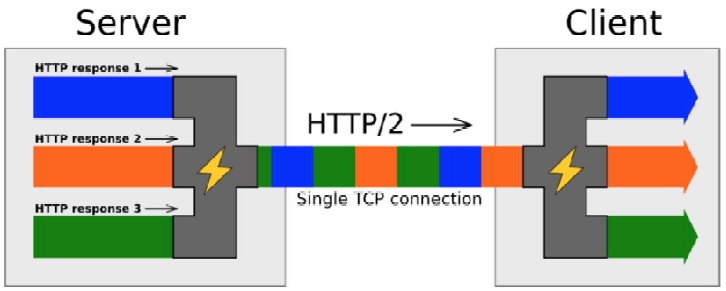
Gestión de conexiones en HTTP/1.1

- 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*
    - Cada petición debe esperar a la anterior aunque sea mucho más larga
- 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

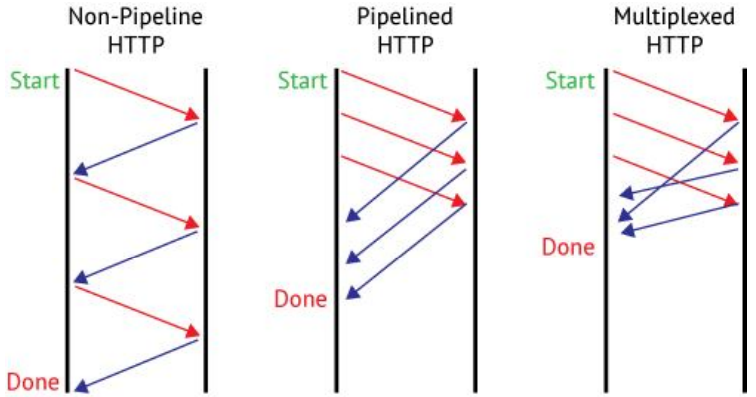
- Uso de *multiplexing* en vez de *pipelining*
  - Elimina el problema de *Head-of-line blocking*
  - Respuestas pueden llegar en cualquier orden
- Permite crear múltiples flujos dentro de cada conexión
  - Cada paquete lleva un identificador de flujo único
  - Paquetes de peticiones/respuestas se mezclan 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



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 (web: HTTP *refresh*; Ajax *polling*)
    - Servidor responde inmediatamente, con nuevos datos si los hay
  - *Long Polling*: Servidor no responde hasta que tenga datos
  - *Server Push*: servidor "empuja" datos hacia el cliente
    - Cliente mantiene conexión persistente y servidor envía actualizaciones
    - Web: HTTP *Server Push*, *Server-Sent Events*, *Web Sockets*
  - Usar editor/subscriptor en vez de cliente/servidor

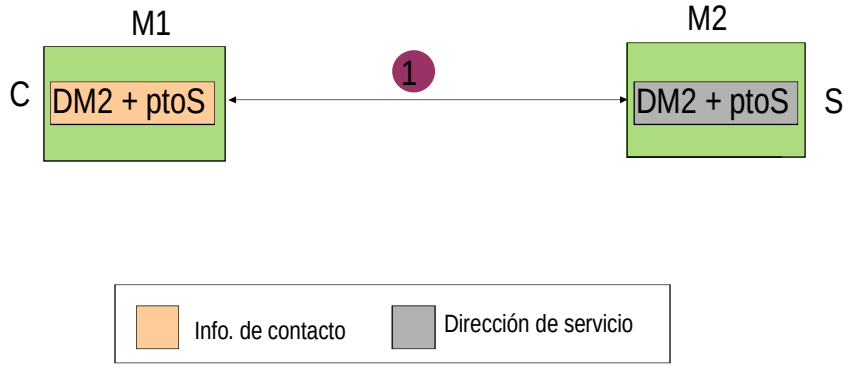
Localización del servidor

- Servidor en máquina con dirección *DM* y usando puerto *PS*
  - ¿Cómo lo localiza el cliente? → *Binding*
  - Otro servidor proporciona esa información → problema huevo-gallina
- *Binder* mantiene correspondencias ID servicio → (*DM*, *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 → (*DM1*, *PS1*) | (*DM2*, *PS2*) ....
  - 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

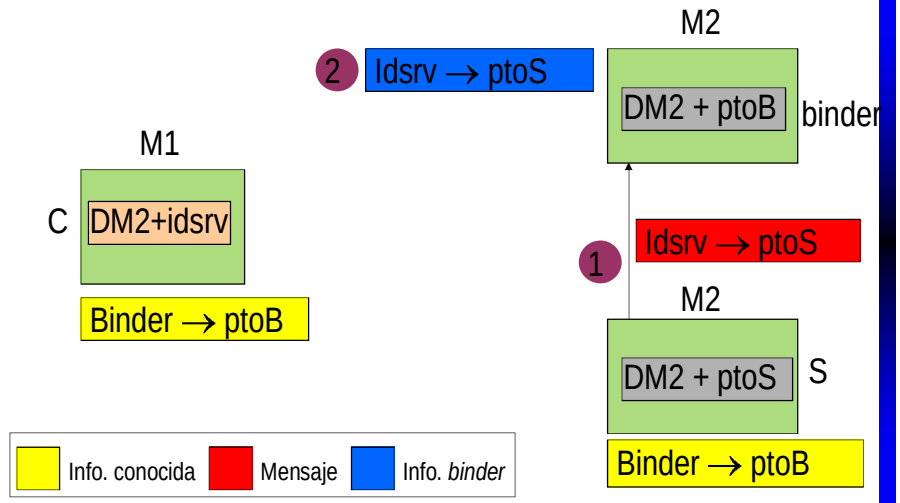
Alternativas en la ID del servicio

1. Uso directo de dirección *DM* y puerto *PS*
   - No proporciona transparencia
2. Nombre servicio + dir servidor (Java RMI *Registry, Sun RPC*)
   - Servidor (*binder*) en cada nodo: nombre de servicio → puerto
   - Impide migración del servidor
3. Nombre de servicio con ámbito global (DCE, CORBA, Mach)
   - Servidor con ubicación conocida en el sistema
   - Dos opciones:
     a) Sólo *binder* global: nombre de servicio → [*DM+PS*]
     b) Opción: *binder* global (BG) + *binder* local (BL) en puerto conocido
        - BG: ID → [*DM*] ; BL: ID → [*PS*]
- Uso de caché en clientes para evitar repetir traducción
  - Mecanismo para detectar que traducción en caché ya no es válida

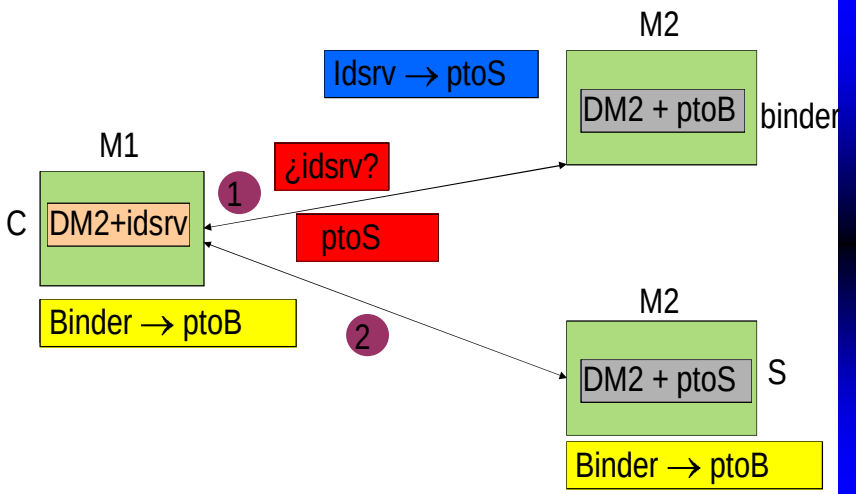
(1) ID servicio = [DM+pto]



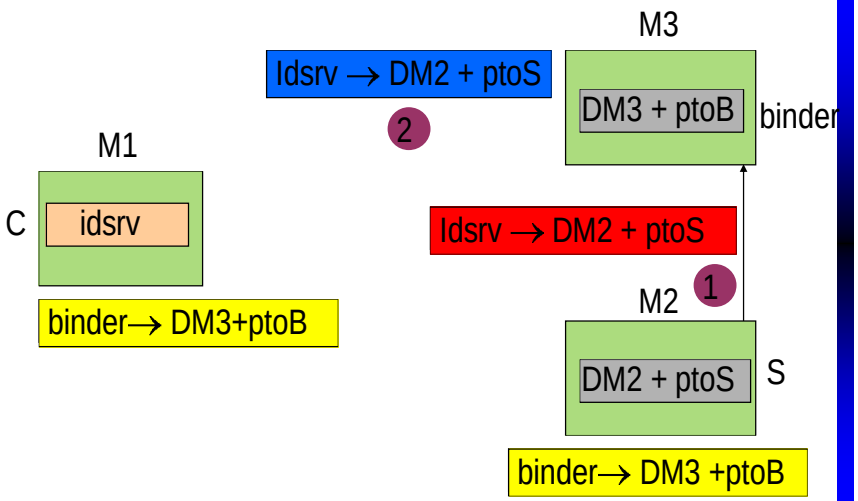
(2) ID servicio = [DM+idsrv]: Alta



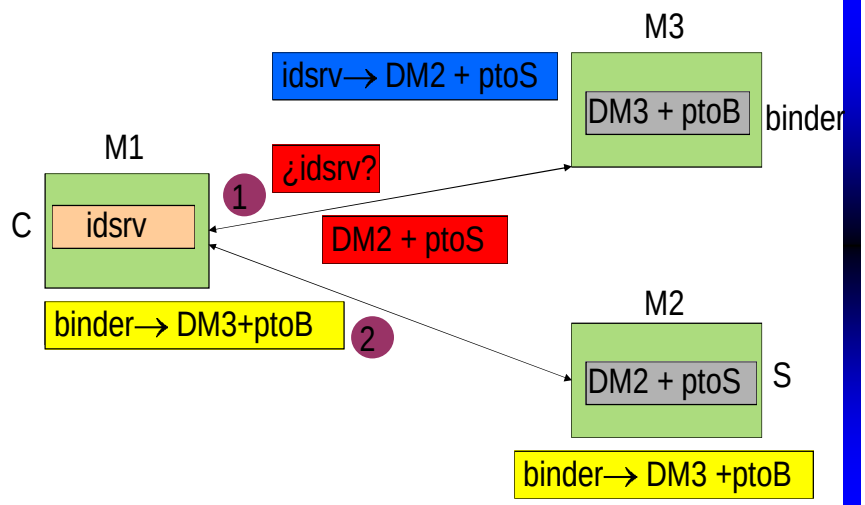
(2) ID servicio = [DM+idsrv]: Consulta



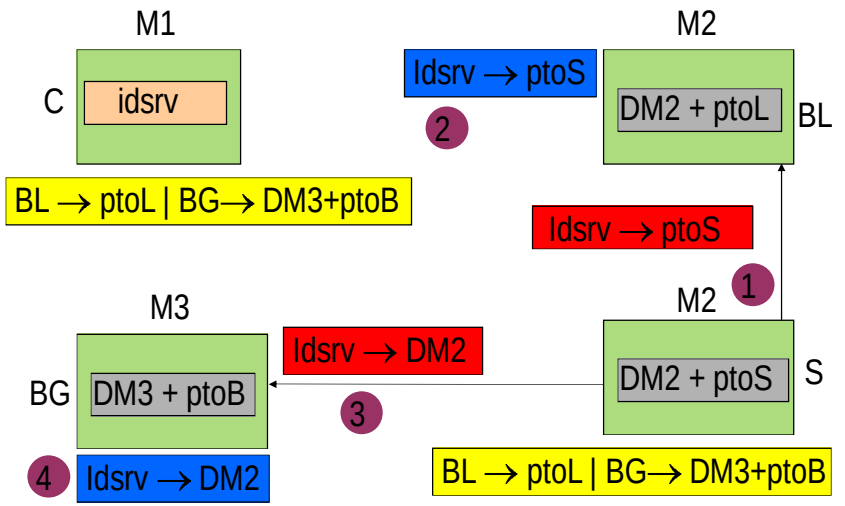
(3a) ID servicio = [idsrv]; Sólo BG: Alta



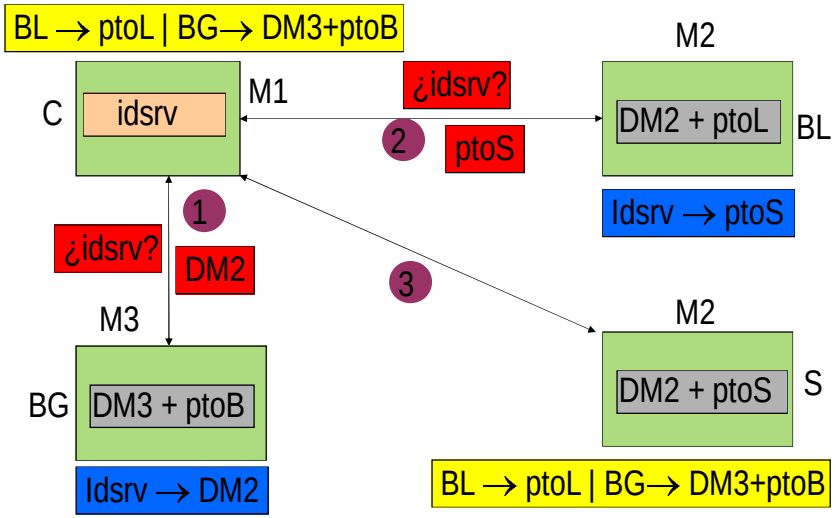
(3a) ID servicio = [idsrv]; Sólo BG: Consulta



(3b) ID servicio = [idsrv]; BG+BL: Alta



(3b) ID servicio = [idsrv]; BG+BL: Consulta



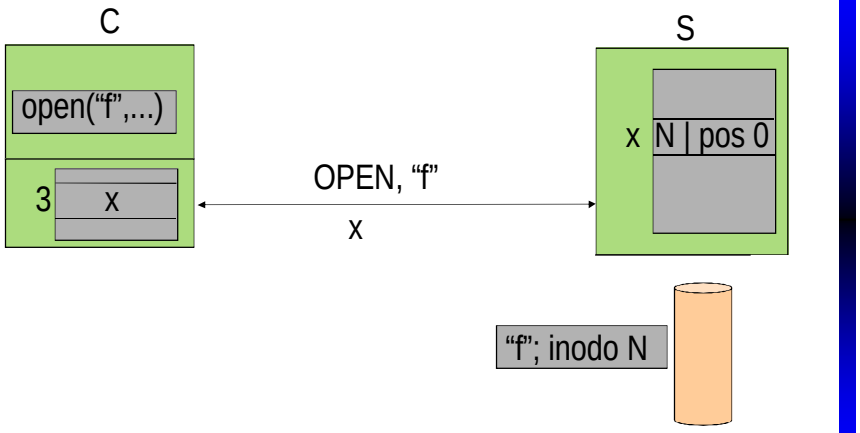
Recapitulación del *Binding*

- 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

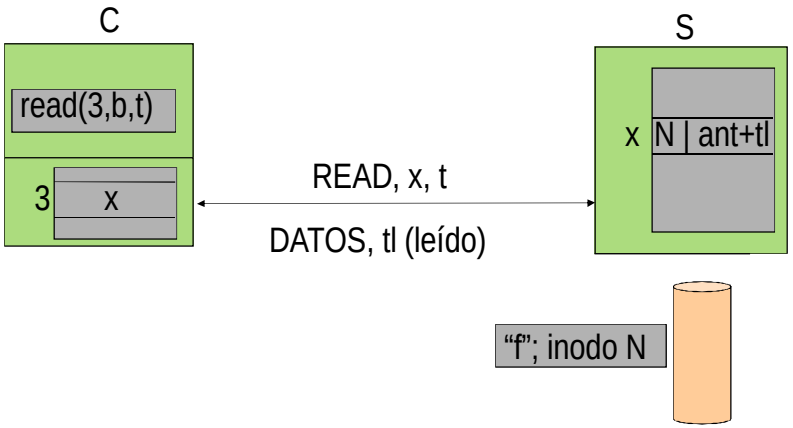
Servicio con/sin estado

- ¿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)
  - Mejor reparto carga y fiabilidad en esquema con escalado horizontal
- 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*)

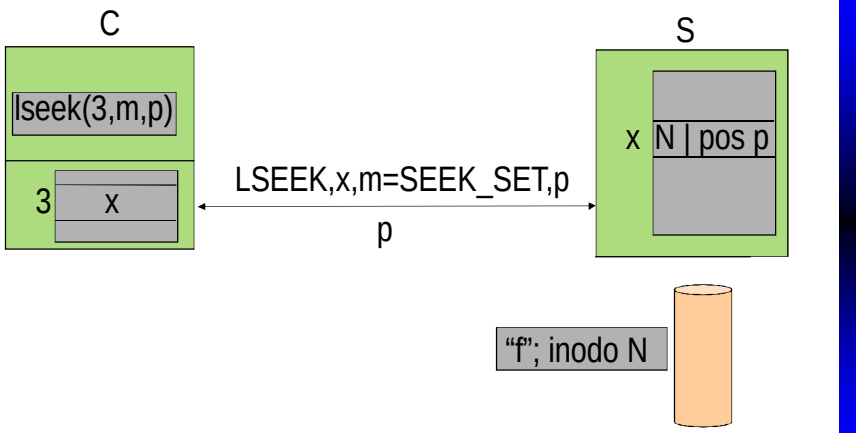
Servicio de ficheros con estado: OPEN



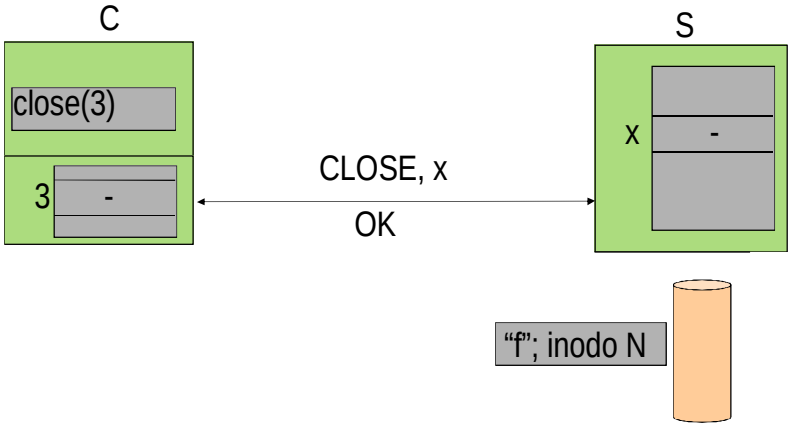
Servicio de ficheros con estado: READ



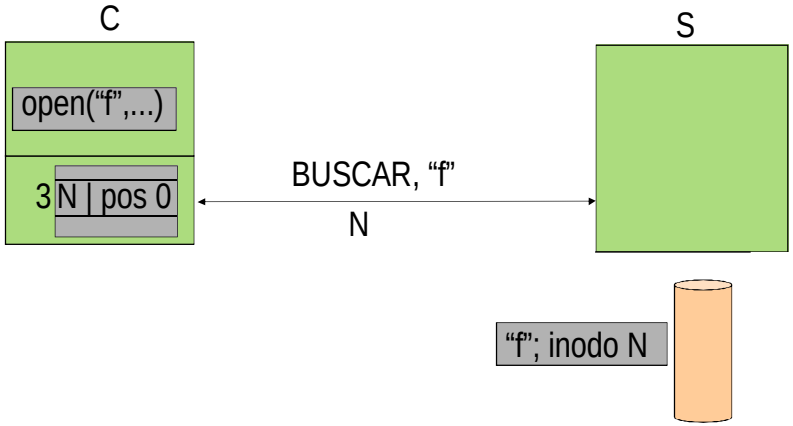
Servicio de ficheros con estado: LSEEK



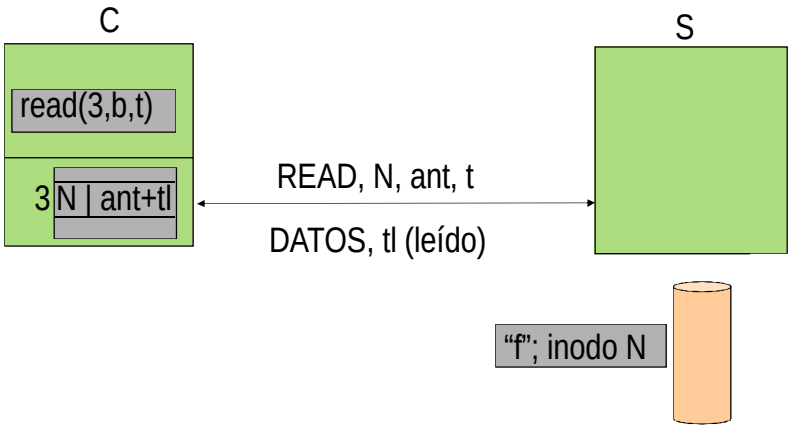
Servicio de ficheros con estado: CLOSE



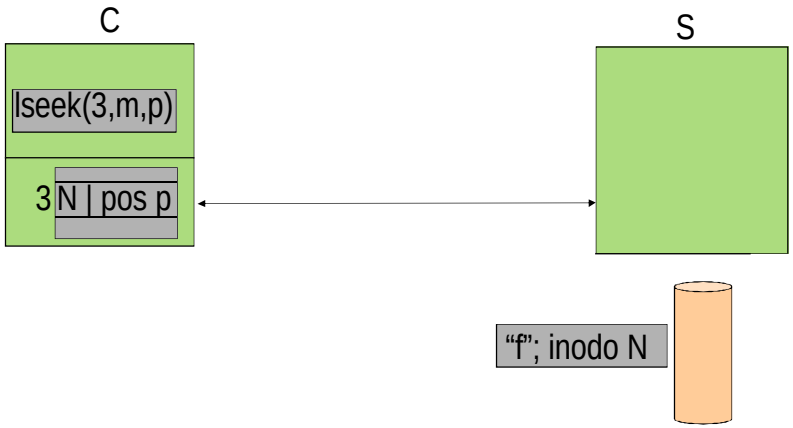
Servicio de ficheros sin estado: OPEN



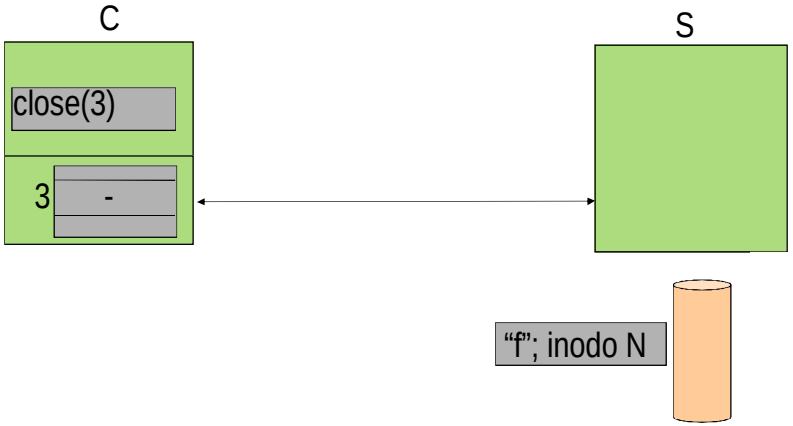
Servicio de ficheros sin estado: READ



Servicio de ficheros sin estado: LSEEK



Servicio de ficheros sin estado: CLOSE



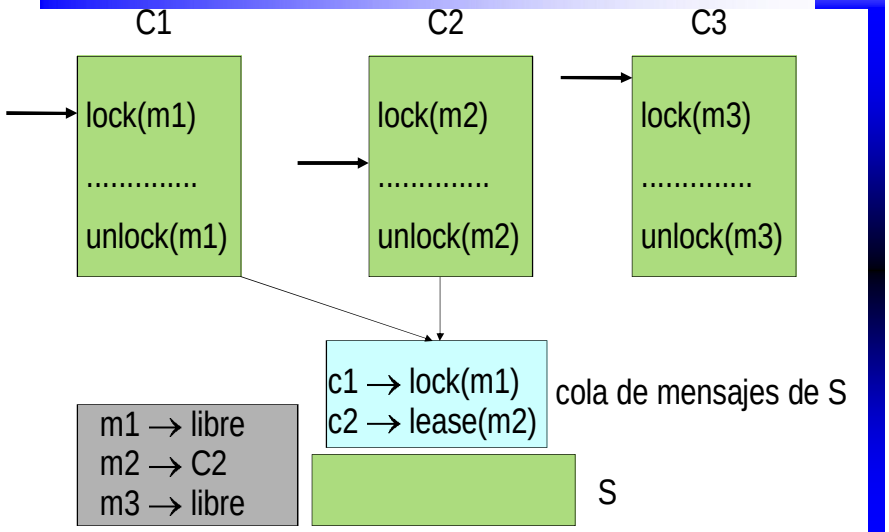
Leases

- Mecanismo para mejorar tolerancia a fallos en SD
  - Detección y tratamiento de caídas de nodos
- Aplicación típica (genérica) de *leases*:
  - Proceso A gestiona algún tipo de recurso vinculado con proceso B
    - Proceso B no tiene por qué contactar de nuevo con A
  - Si B cae, A no lo detecta y el recurso queda "abandonado"
- Modo de operación
  - A otorga a B un *lease* que dura un plazo de tiempo
  - B debe enviar a A mensaje de renovación *lease* antes de fin de plazo
  - Si B cae y no renueva *lease*, se considera recurso "abandonado"
  - Si A cae, en reinicio obtiene renovaciones
    - Con suficiente información, puede "reconstruir" los recursos
- No confundir con un simple temporizador
  - Proceso envía petición a otro y arranca temporizador
    - Si se cumple antes de ACK, vuelve a enviar petición (**≠ *lease***)

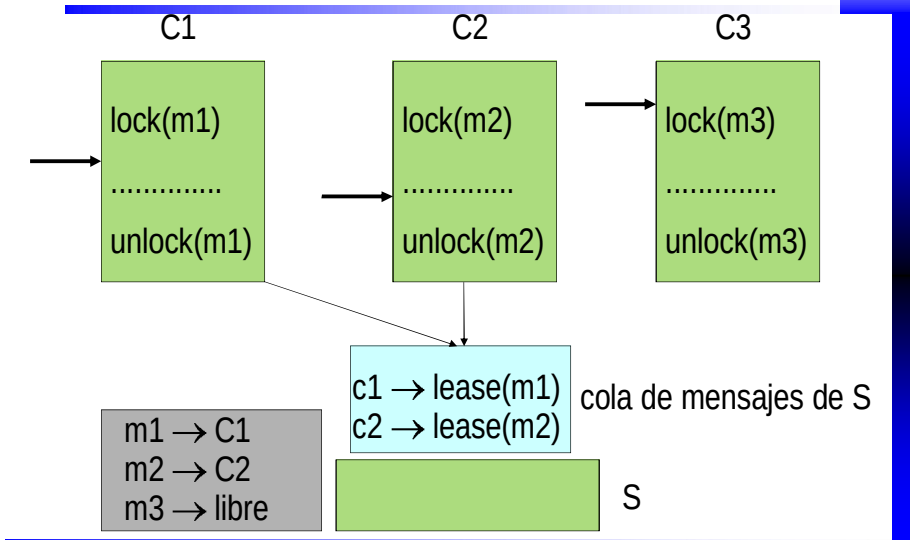
Aplicaciones de *leases*

- Aparecerán a menudo:
  - *Binding*, caídas del cliente, suscripción en Ed/Su, caché de SFD, etc.
  - Ejemplo: Aplicación a *binding*
    - *Binder* asigna *lease* a servidor y servidor renueva *lease*
- *Leases* en servicios con estado
  - Servicios inherentemente con estado: p.e. 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 re-arranque de *S*
  - Caída de cliente: falta de renovación
    - Revocación automática de cerrojos de ese cliente

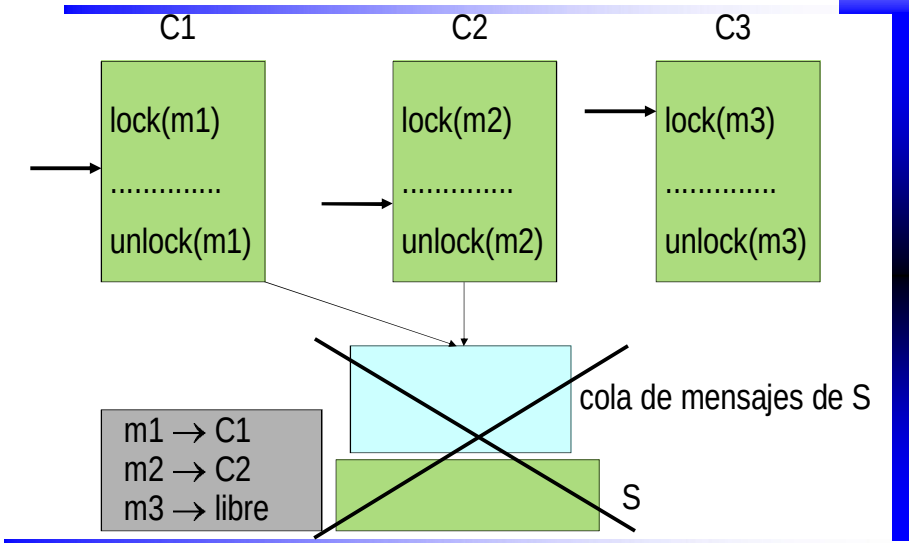
Serv. cerrojos con estado: *leases* (1)



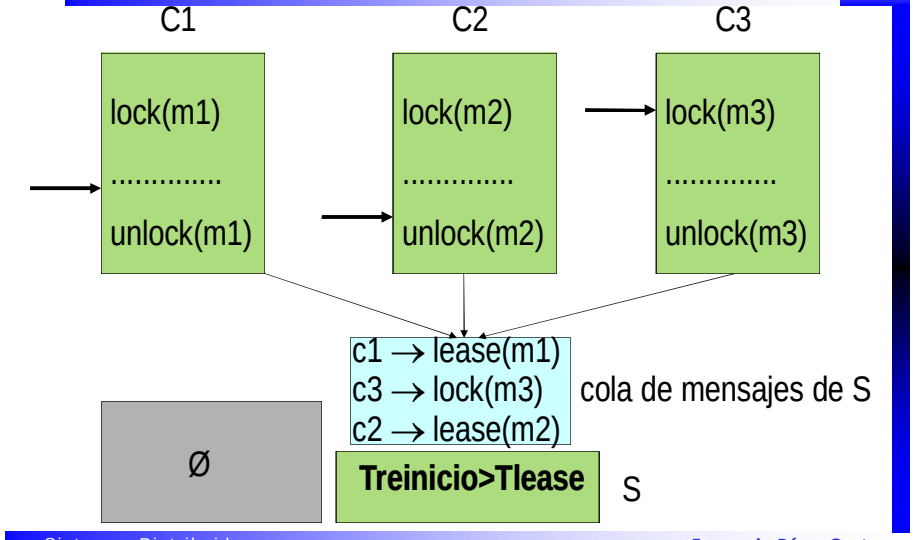
Serv. cerrojos con estado: *leases* (2)



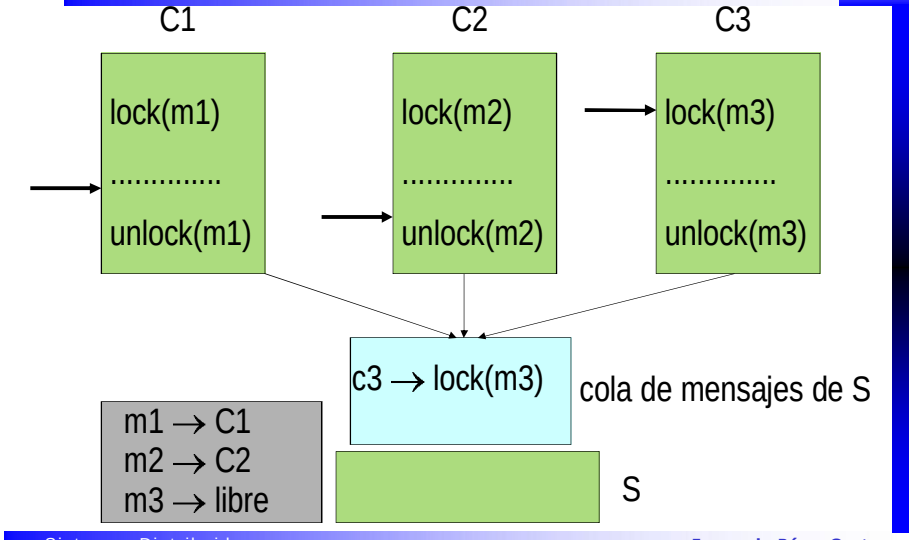
Serv. cerrojos con estado: *leases* (3)



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)
  - En ocasiones puede ser mejor no abortar

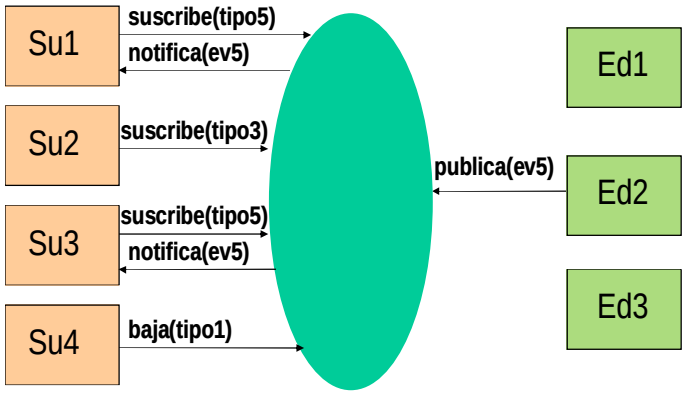
Modelo editor/subscriptor

- Sistema de eventos distribuidos
- Suscriptor *S* (*subscriber*): interés por ciertos eventos (**filtro**)
- Editor *E* (*publisher*) genera un evento
  - Se envía a subscriptores interesados en el mismo
- Paradigma asíncrono y desacoplado en espacio
  - Editores y subscriptores no se conocen entre sí (≠ cliente/servidor)
- Normalmente, *push*: suscriptor recibe evento
  - Alternativa, *pull*: suscriptor pregunta si hay eventos de interés
  - *Pull* requiere que se almacenen eventos (+ complejo)
    - Posibilita mecanismo desacoplado en el tiempo
- Facilita uso en sistemas heterogéneos
- Diversos aspectos relacionados con la calidad de servicio
  - orden de entrega, fiabilidad, persistencia, prioridad, transacciones,...
- Ejemplos: Mercado bursátil, subastas, *chat*, app domótica, etc.

Operaciones modelo editor/subscriptor

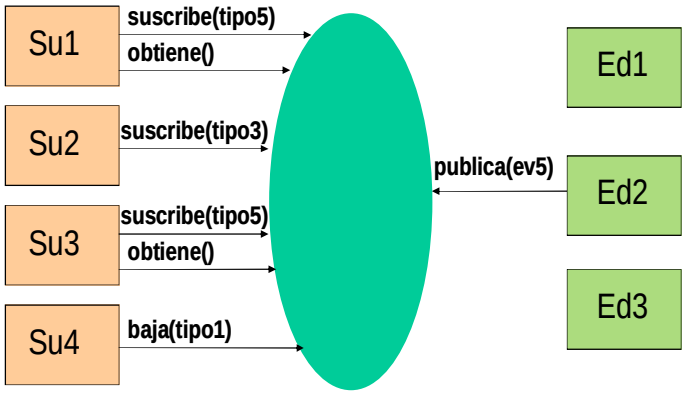
- Estructura típica del evento: [*atrib1=val1; atrib2=val2;* ...]
  - Un atributo puede ser el **tema** del evento
- *suscribe(tipo)* [*S*→]: interés por cierto tipo de eventos
  - Posible uso de *leases* en suscripción
    - Subscriptor renueva *lease*
- *baja(tipo)* [*S*→]: cese del interés
- *publica(evento)* [*E*→]: generación de evento
- *notifica(evento)* [→*S*]: envío de evento (esquema *push*)
- *obtiene()* [*S*→]: lee siguiente(s) evento(s) (esquema *pull*)
  - Puede ser bloqueante o no (si no hay eventos, respuesta inmediata)
- Extensión de modelo: creación dinámica de tipos de eventos
  - *anuncia(tipo)* : se crea un nuevo tipo de evento
  - *baja_tipo(tipo)*: se elimina tipo de evento
  - *notifica_tipo(tipo)* [→*S*]: aviso de nuevo tipo de eventos

Modelo editor/subscriptor (*push*)



Posible extensión: anuncio de nuevo tipo de evento (→S)

Modelo editor/subscriptor (*pull*)



Posible extensión: anuncio de nuevo tipo de evento (→S)

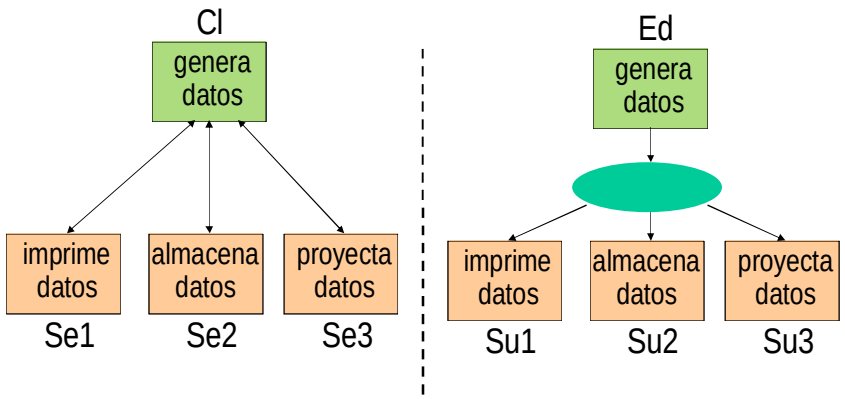
Filtro de eventos por tema

- *S* se subscribe a tema y recibe notificaciones sobre el mismo
- Temas disponibles:
  - Carácter estático: implícitamente conocidos
  - Carácter dinámico: uso de operación de anuncio
    - Ej. Creación de un nuevo valor en el mercado
- Organización del espacio de temas:
  - Plano
  - Jerárquico: (Ej. *bolsas_europeas/españa/madrid*)
  - Uso de comodines en suscripción (Ej. *bolsas_europeas/españa/**)
- Filtrados adicionales deben hacerse en aplicación
  - Ej. Interesado en valores inmobiliarios de cualquier bolsa española
    - Aplicación debe subscribirse a todas las bolsas españolas y
    - descartar todos los eventos no inmobiliarios

Filtro de eventos por contenido

- Debe cumplirse condición sobre atributos del evento
  - Extensión del esquema previo: tema es un atributo del evento
- Uso de lenguaje para expresión de la condición (≈ SQL)
- Filtrado de grano más fino y dinámico que usando temas
  - Ej. Interés en valores inmobiliarios de cualquier bolsa española
- Menor consumo de ancho de banda
  - Llegan menos datos a nodos subscriptor
- Simplifica *app*. subscriptora pero complica esquema Ed/Su
  - Puede involucrar varios tipos de eventos de forma compleja
  - Ejemplo (Tanenbaum):
    - "*Avísame cuando la habitación H420 esté desocupada más de 10 segundos estando la puerta abierta*"

Cliente/servidor vs. Editor/suscriptor



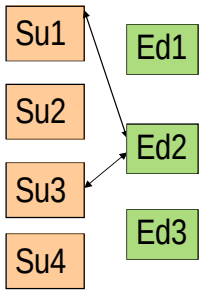
¿En cuál es más fácil añadir nuevo consumidor de datos?

¿Y si queremos que generador sepa cuándo ha procesado datos cada consumidor?

Implementaciones editor/suscriptor

- Comunicación directa
  - No proporciona desacoplamiento espacial
- Uso de intermediario (*broker*)
  - Desacoplamiento espacial pero cuello de botella y único punto de fallo
- Uso de red de intermediarios
  - Distribución de eventos y aplicación de filtros "inteligente"
- Alternativa: uso comunicación de grupo
  - Ed/Su basado en temas: tema = dirección de grupo

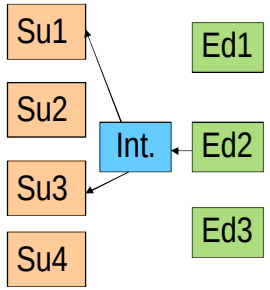
Implementación ed/su sin intermediario



Contacto directo ed./ suscr.

↓ Acoplamiento espacial

Implementación ed/su con intermediario

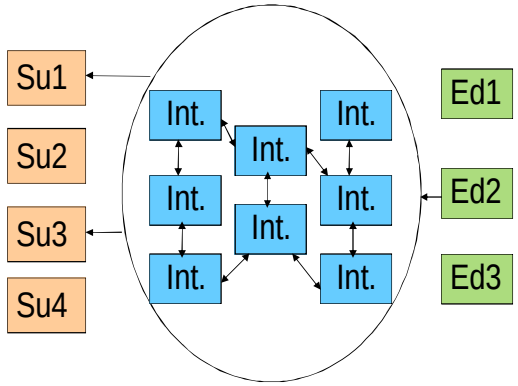


Proceso intermediario

↑ Desacoplamiento espacial

↓ Cuello botella y punto fallo

Implementación ed/su con red interm.

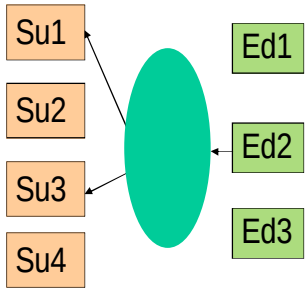


Red de intermediarios

↑ Desacoplamiento espacial

↑ Escalabilidad y fiabilidad

Implementación ed/su con c. grupo



Comunicación de grupo

↑ Desacoplamiento espacial