

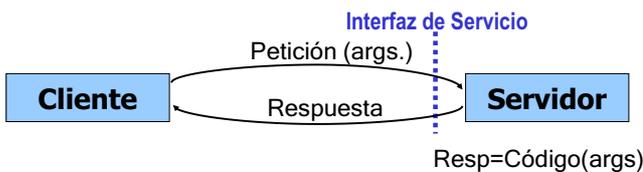
## Grado de acoplamiento

- 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 específica 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`

## 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.

## 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
  - Esta característica es una de las claves del éxito de la Web
- Actúan como “tuberías”
  - Pueden procesar/filtrar información y/o realizar labor de caché
    - Simil con clases `FilterInputStream|FilterOutputStream` de Java
- Diversos tipos: *forward proxy, reverse proxy, gateways, ...*

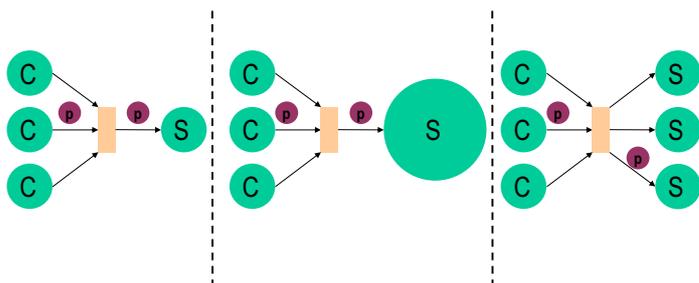
## Cliente/servidor jerárquico

- 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
  - Traducir un nombre de fichero en SFD
  - Traducir de nombre de máquina a IP usando DNS

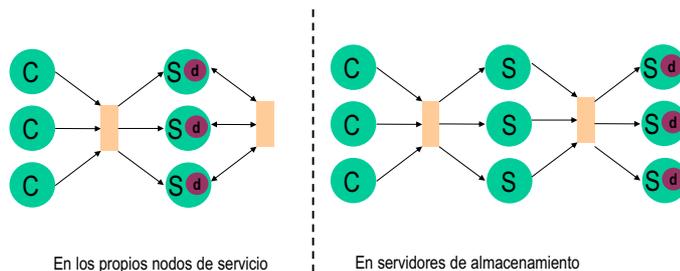
## Cliente/servidor con reparto de carga

- 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 obtiene el valor de la escritura más reciente
  - **A**vailability: los datos están accesibles
  - **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 (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 *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?: sin sentido (no se puede renunciar a P)
  - realmente sólo se puede elegir entre A y C al diseñar un SD

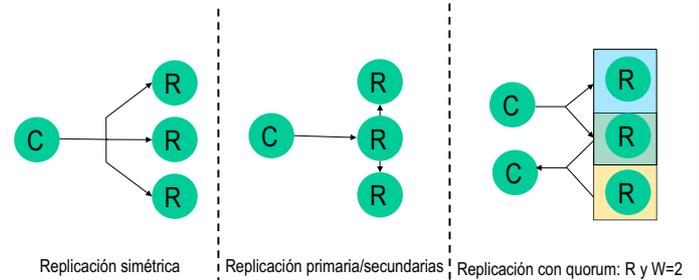
## Latency vs. Consistency

- 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
    - Escritura no puede completarse hasta todas las réplicas actualizadas
    - En ocasiones, puedo renunciar a C por conseguir mejor latencia (L)
      - Lectura puede 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?

## Actualización de réplicas

- Actualización simultánea en todas las réplicas (rep. simétrica)
  - Para C hay que asegurar que se procesan en mismo orden → mala L
- Actualizar copia primaria/maestra que lo propaga a secundarias
  - Por cada dato, una réplica es elegida como primaria/maestra
  - Escritura síncrona: no se completa hasta total propagación
    - Asegura C pero sacrificando L
  - Escritura asíncrona: no se espera hasta total propagación
    - Asegura buena L pero sacrificando C
- Uso de quorum (suponiendo N réplicas)
  - Escritura simultánea síncrona en W réplicas
    - Restantes réplicas se actualizarán asincrónicamente
  - Lectura síncrona de R réplicas (se elige el valor más actual)
  - Si  $R + W > N$  y  $W + W > N$  → se asegura C pero sacrificando L
  - Si  $R + W \leq N$  o  $W + W \leq N$  → mejora L pero no se asegura C

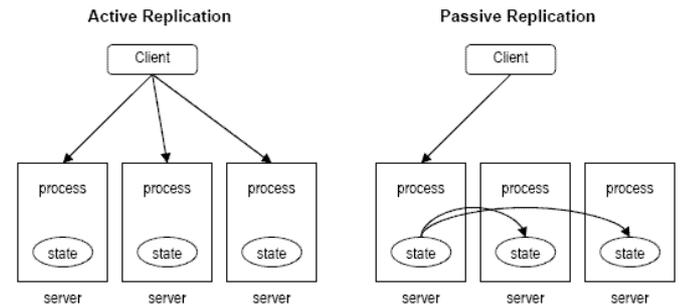
## Actualización de réplicas



## Cliente/servidor con alta disponibilidad

- Servicio debe completarse aunque se caiga servidor
  - Requiere uso de múltiples servidores replicados
  - No confundir con servicio sólo con reparto de carga
    - En ese caso si se cae servidor se pierde petición en curso
- Soluciones alternativas (de mejor a peor t. de recuperación)
  - Replicación activa:
    - 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

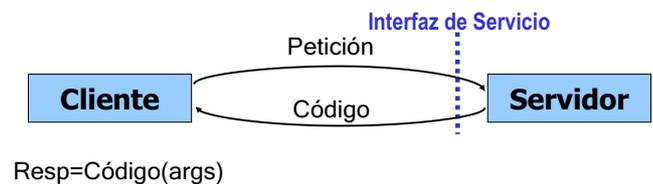


<https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/>

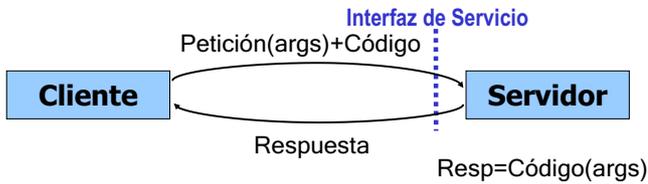
## Código móvil

- 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



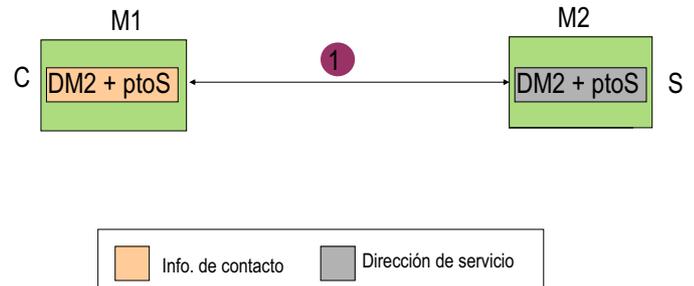
## Localización del servidor

- 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 4): Gestiona IDs de todos los recursos

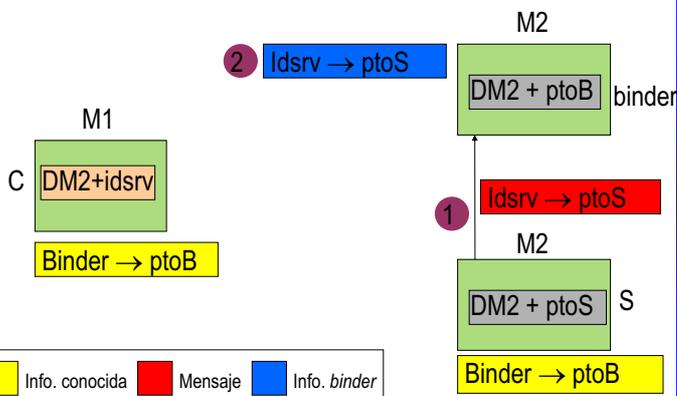
## Alternativas en la ID del servicio

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

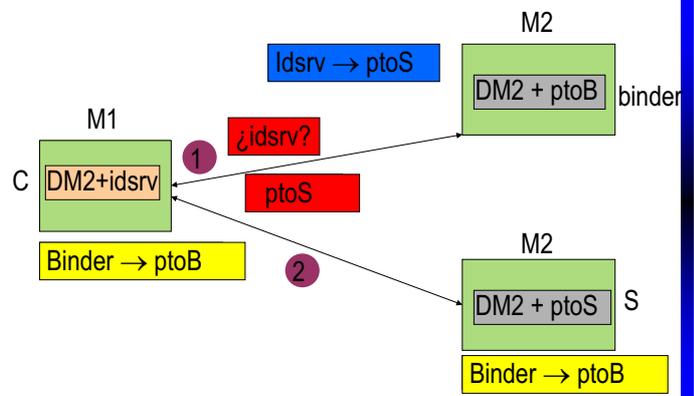
## (1) ID servicio = [DM+pto]



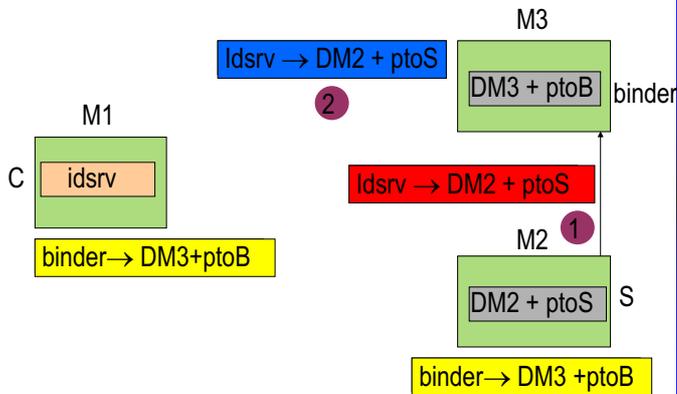
## (2) ID servicio = [DM+idsrv]: Alta



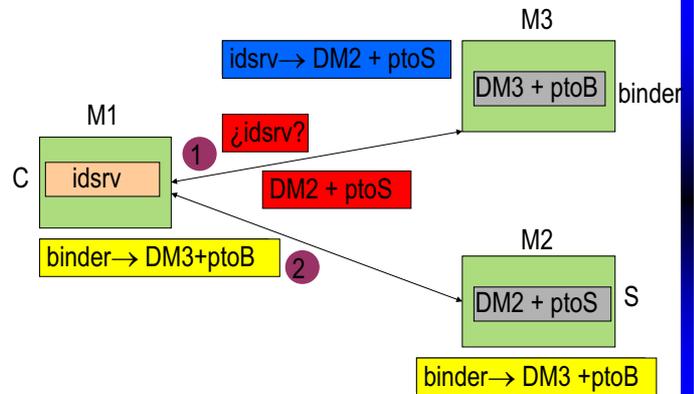
## (2) ID servicio = [DM+idsrv]: Consulta



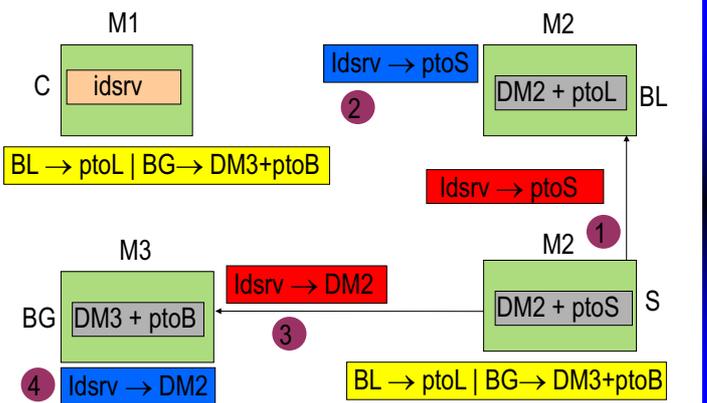
### (3a) ID servicio = [idsrv]; Sólo BG: Alta



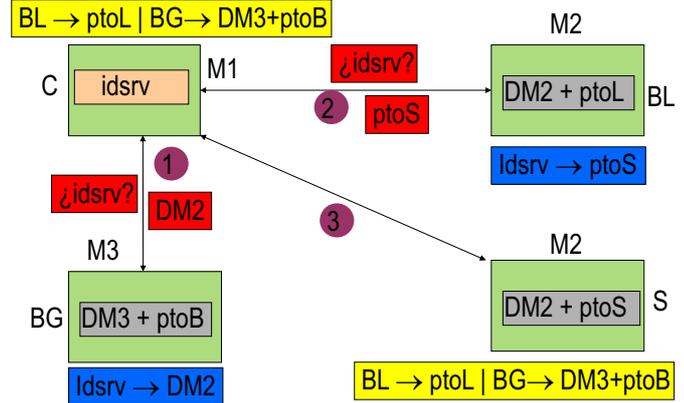
### (3a) ID servicio = [idsrv]; Sólo BG: Consulta



### (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 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
    - Evento: actividad asociada con 1 petición (p.e. llegada datos de cliente)
  - Implementación en UNIX de espera simultánea; alternativas:
    - *select/poll/epoll*; uso de señales de *t.real*; operaciones asincrónicas (*aió*)
  - 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"

## 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 → gasto innecesario
  - Mucha carga → insuficientes
- Esquema híbrido con mínimo *m* y máximo *M*
  - *m* pre-arrancados;  $m \leq T/P \leq M$
  - Si petición, ninguno libre y  $n^{\circ} < M \rightarrow$  se crea
  - Si inactivo tiempo prefijado y  $n^{\circ} > m \rightarrow$  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!)
  - Protocolos de transporte orientados a C/S (*T/TCP*, *TCP Fast Open*)
- Conexiones persistentes: *N* peticiones cliente misma conexión
  - Más complejo pero menor sobrecarga
  - Esquema usado en HTTP/1.1 (además, *pipeline* de peticiones)
  - Dado que servidor admite  $n^{\circ}$  limitado de conexiones
    - Dificulta reparto de servicio entre clientes
  - Implica que servidor mantiene un estado
  - Dificulta reparto de carga en esquema con escalado horizontal
  - Facilita *server push*

## Evolución de HTTP

- Cliente necesita pedir varios objetos al mismo servidor
- HTTP/1.0
  - Connect | GET | Resp | Close | Connect | GET | Resp | Close | ...
  - Sobrecarga conexiones + latencia de conexiones y peticiones
- HTTP/1.0 + conexiones persistentes
  - Connect | GET | Resp | GET | Resp | ... | Close
  - Latencia de peticiones
- HTTP/1.1 (conexiones persistentes + *pipeline* de peticiones)
  - Connect | GET | GET | ... | Resp | Resp | ... | Close
  - No latencia acumulada
  - Servicio paralelo de peticiones
    - aunque respuestas deben llegar en orden

## HTTP: conexiones simultáneas

- Paralelismo también mediante conexiones simultáneas
- HTTP/1.0
  - Connect | GET | Resp | Close
  - .....
  - Connect | GET | Resp | Close
- HTTP/1.0 + conexiones persistentes
  - Connect | GET | Resp | GET | Resp | ... | Close
  - .....
  - Connect | GET | Resp | GET | Resp | ... | Close
- HTTP/1.1 (conexiones persistentes + *pipeline* de peticiones)
  - Connect | GET | GET | ... | Resp | Resp | ... | Close
  - .....
  - Connect | GET | GET | ... | Resp | Resp | ... | Close

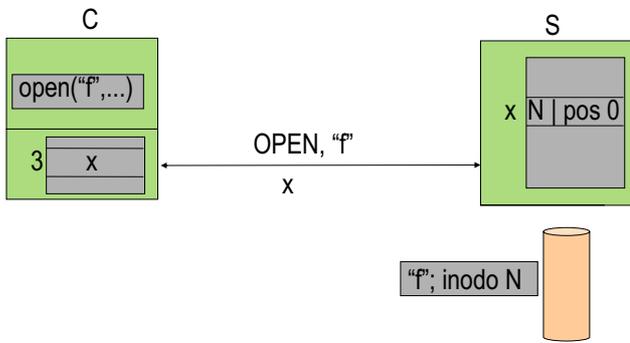
## Client Pull vs Server Push

- 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

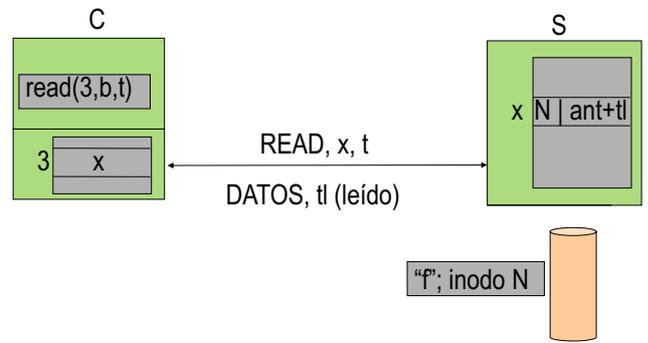
## Servicio con/sin estado

- ¿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 rearme del servidor)
    - Peticiones autocontenidas.
  - Reduce  $n^{\circ}$  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*)

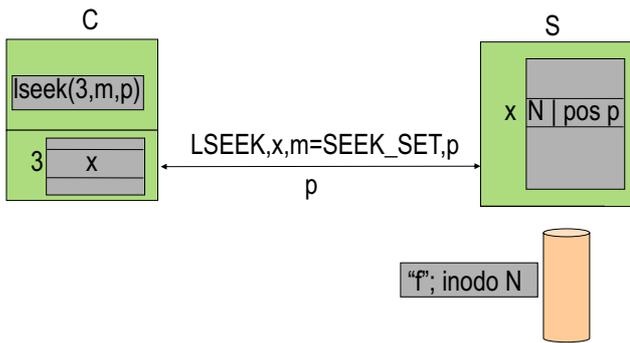
### Servicio de ficheros con estado: OPEN



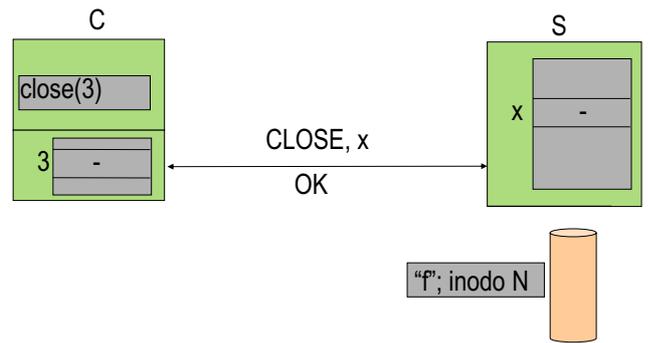
### Servicio de ficheros con estado: READ



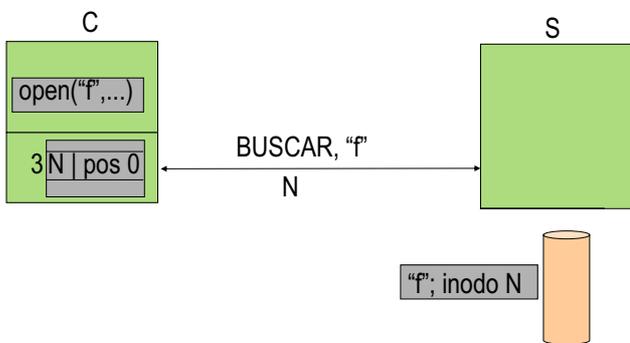
### Servicio de ficheros con estado: LSEEK



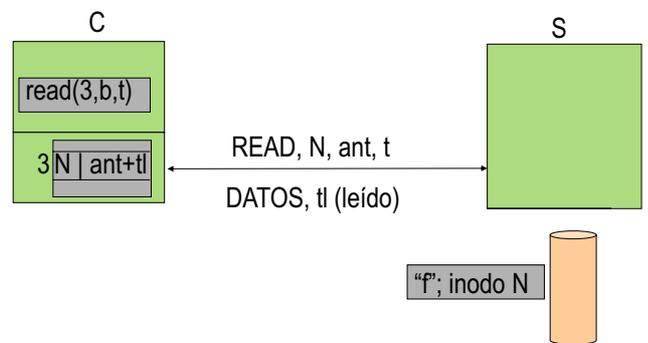
### Servicio de ficheros con estado: CLOSE



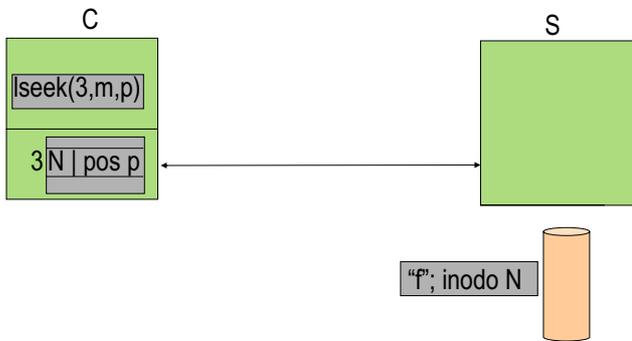
### Servicio de ficheros sin estado: OPEN



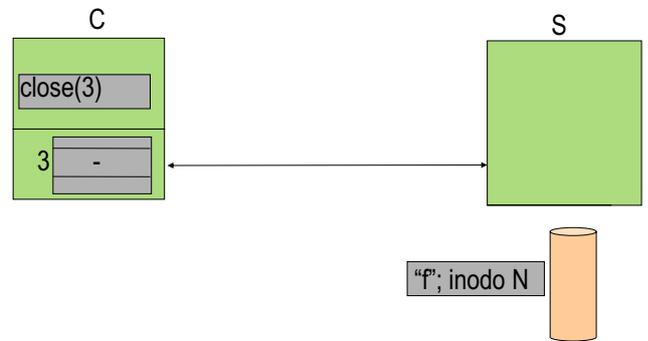
### Servicio de ficheros sin estado: READ



## Servicio de ficheros sin estado: LSEEK



## Servicio de ficheros sin estado: CLOSE



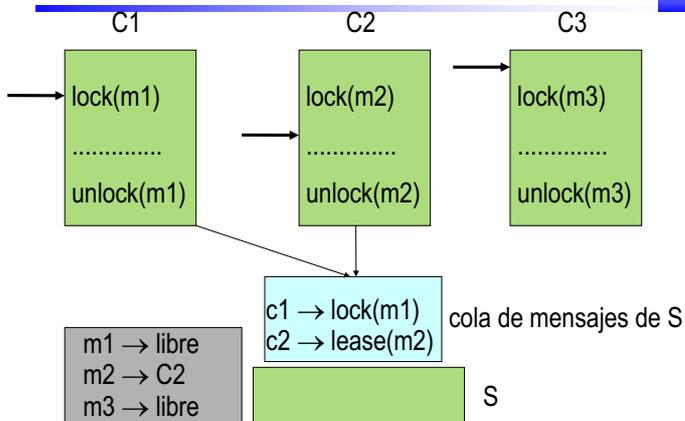
## Leases

- 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 ( $\neq$  *lease*)

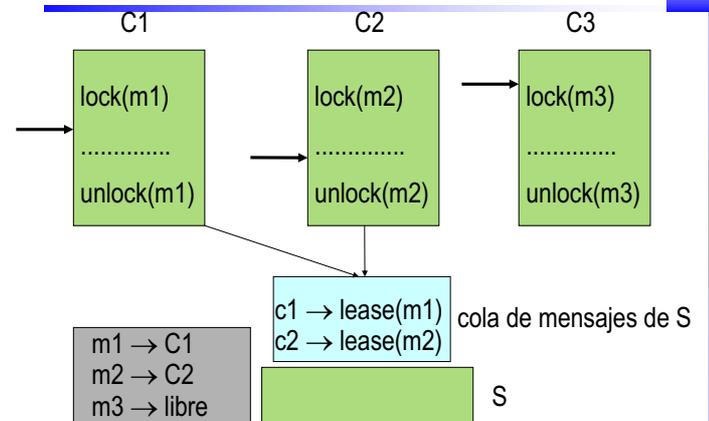
## Aplicaciones de *leases*

- Aparecerán a menudo:
  - Binding*, caídas del cliente, suscripción en Ed/Su, caché de SFD, etc.
- 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 re-arranque de S
  - Caída de cliente: falta de renovación
    - Revocación automática de cerrojos de ese cliente

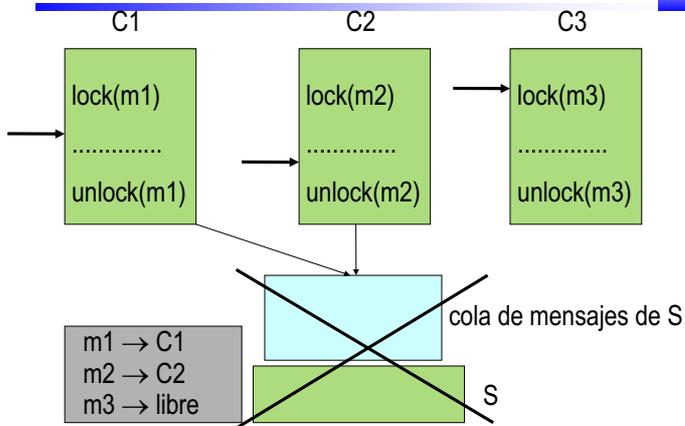
## Serv. cerrojos con estado: *leases* (1)



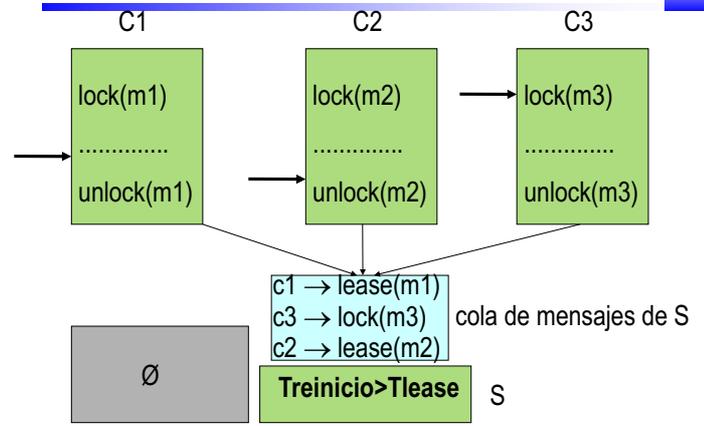
## Serv. cerrojos con estado: *leases* (2)



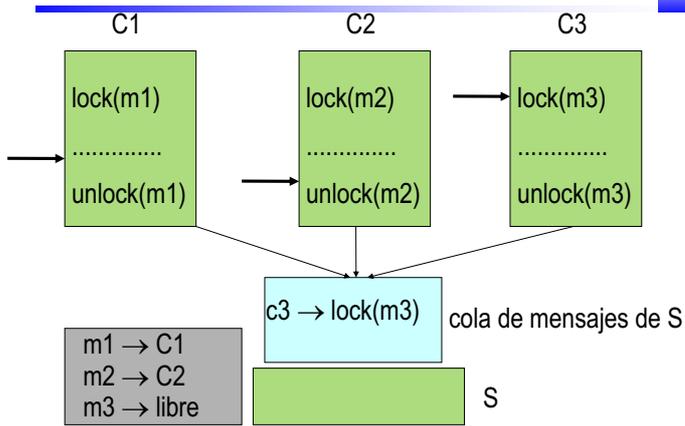
### Serv. cerrojos con estado: leases (3)



### 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 (**SI**)
- Operación idempotente + al menos 1 vez  $\approx$  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
  - N° 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 n° 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 n° de época
    - En rearranque de cliente C: transmite (++n° 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
- Abortar un servicio no es trivial
  - Puede dejar incoherente el estado del servidor (p.e. cerrojos)
  - En ocasiones puede ser mejor no abortar

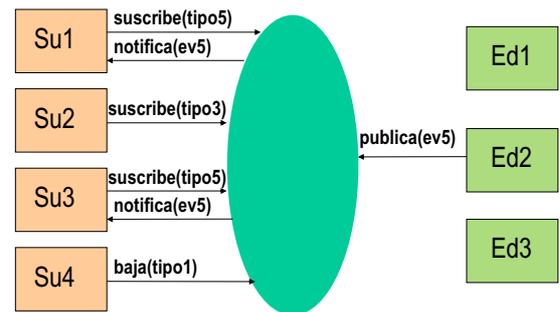
## Modelo editor/subscriptor

- Sistema de eventos distribuidos
- Subscriptor *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*: subscriptor recibe evento
  - Alternativa, *pull*: subscriptor 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

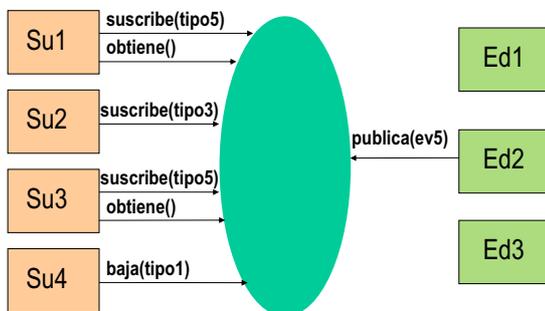
- Estructura típica del evento: [*atrib1=val1; atrib2=val2; ...*]
  - Un atributo puede ser el **tema** del evento
- *suscribe(tipo)* [*S*→]: interés por cierto tipo de eventos
  - Posible uso de *leases* en suscripción
- *baja(tipo)* [*S*→]: cese del interés
- *publica(evento)* [*E*→]: generación de evento
- *notifica(evento)* [*E*→*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)* [*E*→*S*]: aviso de nuevo tipo de eventos

## Modelo editor/subscriptor (*push*)



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

## Modelo editor/subscriptor (*pull*)



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

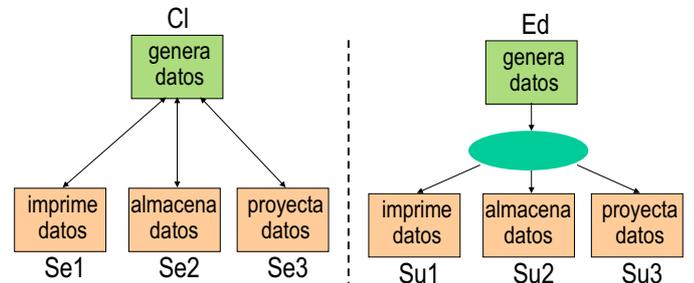
## Filtro de eventos por tema

- *S* se suscribe 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 suscribirse a todas las bolsas españolas y
    - descartar todos los eventos no inmobiliarios

## Filtro de eventos por contenido

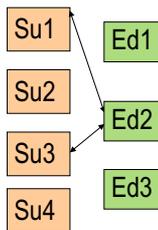
- Debe cumplirse condición sobre atributos del evento
  - Extensión del esquema previo: tema es un atributo del evento
- Uso de lenguaje para expresión de la condición ( $\approx$  SQL)
- Filtrado de grano más fino y dinámico que usando temas
  - Ej. Interés en valores inmobiliarios de cualquier bolsa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

## Cliente/servidor vs. Editor/suscriptor



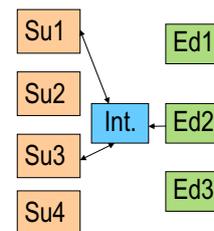
¿En cuál es más fácil añadir nuevo consumidor de datos?  
¿Y si queremos que generador sepa cuándo ha procesado datos cada consumidor?

## Implementación ed/su sin intermediario



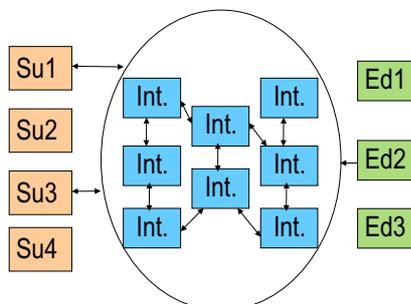
Contacto directo ed./ suscr.  
↓ Acoplamiento espacial

## Implementación ed/su con intermediario



Proceso intermediario  
↑ Desacoplamiento espacial  
↓ Cuello botella y punto fallo

## Implementación ed/su con red intern.



Red de intermediarios  
↑ Desacoplamiento espacial  
↑ Escalabilidad y fiabilidad

## Paradigma de paso de mensajes

- HW de paso de mensajes  $\rightarrow$  API de paso de mensajes
- Sist. de paso de mensajes: Capa sobre protocolo de transporte
- Alternativas de diseño en aspectos como:
  - API de comunicación ofrecido
  - Direccionamiento: ¿cómo especifica origen/destino de comunicación?
  - Especificación del mensaje
  - Optimización de transferencias (*zero-copy*)
  - Integridad de los mensajes
  - *Serialización* de los datos
  - Grado de sincronía (y *buffering*)
- Ejemplos con distintos niveles de funcionalidad (sockets, MPI)

## API de paso de mensajes

- **MPI** (qué envío; a quién; qué recibo; de quién)
 

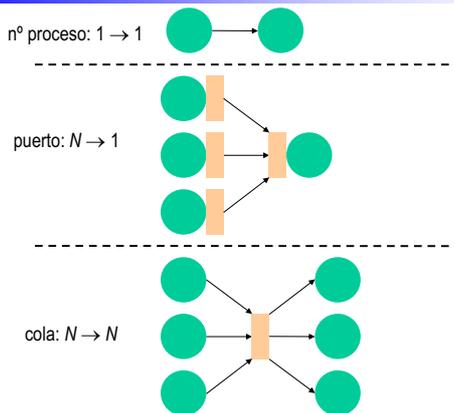
```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```
- **Sockets datagrama** (qué envío; a quién; qué recibo; de quién)
 

```
ssize_t sendto(int socket, const void *buffer, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t dest_len);
ssize_t recvfrom(int socket, void * buffer, size_t length, int flags,
                 struct sockaddr *address, socklen_t * address_len);
```
- **Esquemas con conexión**
  - Existen además primitivas para conectar y desconectar
  - Operaciones de envío y recepción no incluyen direcciones

## Esquemas de direccionamiento

- Usando **número de proceso (MPI)**:
  - En envío: n° proceso destinatario
  - En recepción: n° proceso origen; sólo interacción 1 → 1
    - O cualquiera (*MPI\_ANY\_SOURCE*): interacción N → 1
  - Difícil asignar n° proceso único en entorno de propósito general
    - Pero no en aplicación ejecutada en entorno de computación paralela
- Usando **puertos**: buzón asociado a una máquina (sockets)
  - Comunicación entre puertos
  - Proceso reserva uso de un puerto de su máquina (*bind* de sockets)
  - Envío: desde puerto origen local a puerto destino especificados
  - Recepción: de puerto local; interacción N → 1
  - Sockets INET: ID puerto = dir. IP + n° puerto + protocolo (TCP/UDP)
- Usando **colas**: buzón de carácter global; interacción N → N
  - Sistemas de colas de mensajes; desacoplamiento espacial+temporal
  - Posibilita reparto de carga

## Modos de interacción punto-a-punto



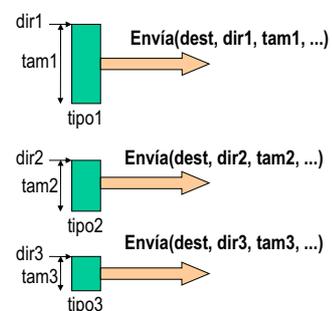
## Especificación del mensaje

- Con o sin información de tipos (MPI vs. sockets)
  - Sin ella: aplicación debe gestionar heterogeneidad
  - Con ella: sistema de comunicaciones gestiona heterogeneidad
- **Clases de mensajes (etiquetas)**
  - Sistema de comunicación puede gestionar clases de mensajes
    - En envío: especifica clase de mensaje enviado
    - Recepción: especifica clase de mensaje que se quiere recibir
      - o usa comodín (*MPI\_ANY\_TAG*)
  - Múltiples canales sobre una misma comunicación
  - Diversas aplicaciones como por ejemplo:
    - Establecer prioridades entre mensajes
    - En cliente-servidor puede identificar operación a realizar
    - En editor-subscriptor basado en temas como identificador de tema
  - Disponible en MPI como parámetro de primitivas (*tag*)
  - No soportado en sockets, aunque sí mensajes urgentes (OOB)

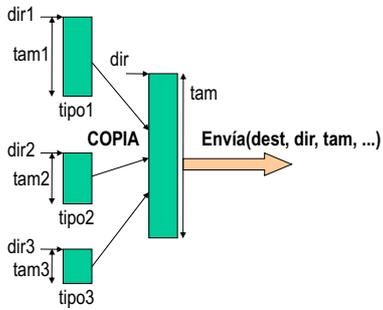
## Zero-Copy

- Reducir al mínimo ( $\approx$  a cero) copias entre zonas de memoria
- **Escenario 1**: envío de *N* datos dispersos de emisor a receptor
  - *N* envíos: sobrecarga de llamada de as + fragmentación de mensajes
  - Reserva de *buffer* y 1 envío: sobrecarga de copias
  - Funciones *scatter/gather*: minimizar copias y llamadas
    - UNIX: *readv, writev, sendmsg, recvmsg*
- **Escenario 2**: envío de un fichero
  - Uso de operaciones convencionales de lectura y envío
    - Dos copias de memoria: de *buffer* de sistema a de usuario y viceversa
  - Uso de proyección de ficheros (*mmap*) y envío
    - Una copia de memoria a *buffer* de sistema en envío
  - Uso de operaciones de transferencia directa entre descriptores
    - No requiere copias entres *buffers*; reduce n° llamadas al sistema
    - Linux: *sendfile, splice*

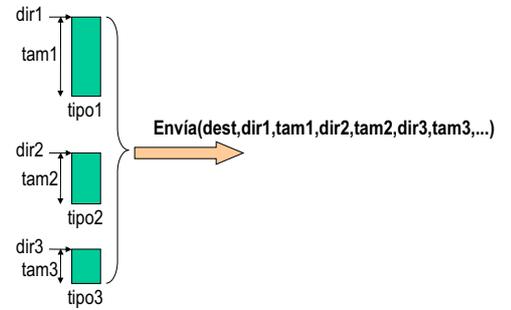
## Datos dispersos: Envío múltiple



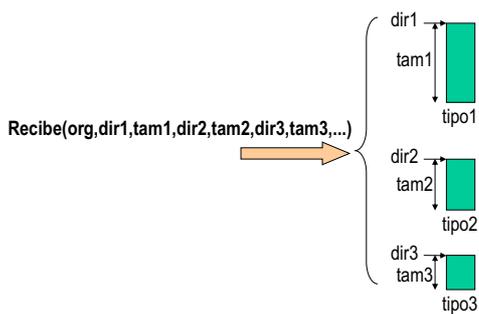
## Datos dispersos: Envío con copia



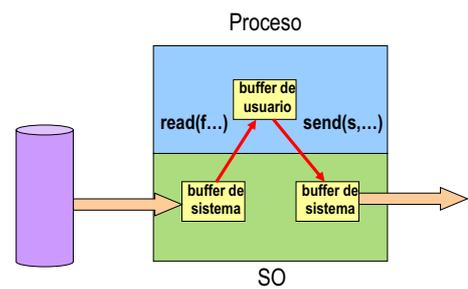
## Datos dispersos: Envío gather



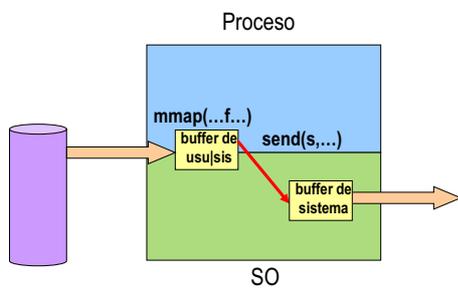
## Datos dispersos: Recepción scatter



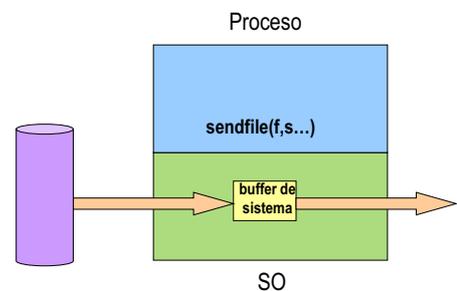
## Envío convencional de fichero



## Envío con proyección de fichero



## Envío zero-copy de fichero



## Serialización de datos

- Emisor y receptor misma interpretación de información
  - Misma cuestión, y soluciones, para lector y escritor de un fichero
- Procesadores, lenguajes, compiladores difieren en:
  - Orden de bytes en tipos numéricos (*endian*)
  - Tamaño de datos numéricos (en C: ¿tamaño de int, long,...?)
  - Strings (con longitud vs. carácter terminador (¿qué carácter?))
  - Formatos de texto (ISO-8859-1, UTF-8,...)
  - Organización estructuras datos (compactación, alineamientos,...),...
- Se necesitan “**serializar**” los datos para enviar/almacenar
  - Asegurando misma interpretación en sistema heterogéneo
  - Eficientemente (en *serialización*, en uso de red/disco,...)
  - Facilitando la programación de la *serialización*
  - Admitiendo cambios incrementales en protocolo
    - P.e. protocolo con nuevo campo opcional pero cliente antiguo sigue OK

## Marshalling

- Operación para *serializar* información en emisor
  - Y la operación inversa (*unmarshalling*) en receptor
- Con paso de mensajes puede ser:
  - Responsabilidad del programador (sockets)
    - Sockets tampoco ofrece funciones *serialización* (excepto para int: *htonl*...)
  - Automático (MPI)
- RPC/RMI lo realizan automáticamente
- Alternativas:
  - S. de comunicación en emisor convierte a formato de receptor
    - transformar a formato de cualquier receptor
  - S. de comunicación en receptor convierte a su formato
    - transformar desde formato de cualquier emisor
  - S. de comunicación en emisor convierte a formato externo
    - Sólo transformar de nativo a externo y viceversa
    - Ineficiente si formato de emisor = receptor pero ≠ de externo

## Formato de serialización de datos

- Define cómo se transmiten/almacenan datos (*wire protocol*)
  - Secuencia de bits que representan cada dato
- Alternativas:
  - Formato propio vs. Estándar (mejor)
  - Texto vs. binario: menos compacto pero interpretable por usuarios
  - Información de tipos implícita o explícita:
    - Implícita: emisor y receptor conocen tipos de parámetros
      - no viaja info. de tipos con datos
    - Explícita: disponible información explícita de tipos
      - Viaja mezclada con datos o como referencia a un esquema
    - Explícita más flexible (permite reflexión) pero menos compacto
  - Información de nombres de campos implícita vs. explícita:
    - Explícita: viaja nombre de campo con datos
      - Puede facilitar cambios incrementales de un protocolo
  - Información explícita de campos y tipos
    - Función de *deserialización* genérica

## Componentes de un s. de serialización

- No sólo define un *wire protocol*, además puede incluir:
  - IDL (*Interface Definition Language*) y API para la *serialización*
- Lenguaje IDL para especificar datos a transmitir/almacenar
  - Permite definir datos con independencia de la plataforma
    - Usando, habitualmente, lenguaje específico (véase sección sobre RPC)
  - *Compilador IDL*: a partir de especificación genera tipos/clases nativos
  - Aplicación usa tipos nativos generados y API para (de)serializar
- API hipotético para *serialización*
  - *Encode(dato, tipo) → buffer*
  - *Decode(buffer, tipo) → dato*
    - Si información de tipos/campos explícita: *Decode(buffer)*
- Puede estar integrado en entorno de RPC/RMI
  - ¡Buenas noticias!: Programador no realiza *serialización*
  - Código generado usa automáticamente API de *serialización*

## Ejemplos de formatos de serialización

- XDR (RFC 1832): binario, info. implícita campos y tipos
- Soluciones basadas XML: texto, inf. explícita campos y tipos
  - Info de tipos mediante referencia a XML Schema
- JSON: texto, info. explícita campos y tipos
- Protocol Buffers (Google): binario, no explícita campos y tipos
  - Pero sí viaja ID único y longitud de cada campo con datos
  - Facilita cambios incrementales en protocolo
- Java Serialization: binario, info. explícita campos y tipos
  - Info de campos y tipos mezclada con datos; no requiere IDL
- Muchos otros: ASN.1, Apache Thrift, Apache Avro, BSON,...
- Wikipedia: Comparison data serialization formats
- Ejemplos: <http://laurel.datsi.fi.upm.es/~ssoo/SD.dir/serializacion>

## XDR

```
struct dato {int id; string nombre<>;}; // especificación (fichero .x)
```

```
struct dato d; XDR x; char buf[TAM];
d.id=1; d.nombre="yo";
xdrmem_create(&x, buf, TAM, XDR_ENCODE);
xdr_dato (&x, &d); // serializa
write(1, buf, xdr_getpos(&x));
```

```
XDR x; int tam; char buf[TAM];
struct dato d = {0, NULL};
tam=read(0, buf, TAM);
xdrmem_create(&x, buf, tam, XDR_DECODE);
xdr_dato (&x, &d); // deserializa
printf("id %d nombre %s\n", d.id, d.nombre);
```

Contenido = 12 bytes: 00 00 00 01 00 00 00 02 'y' 'o' 00 00

## JSON (wikipedia)

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```

## JSON (con Javascript)

```
<!DOCTYPE html><html><body><script>
var pers = new Object();
pers.nombre="yo";
pers.tfno=666;
var buf = JSON.stringify(pers); // Serializa a {"nombre":"yo","tfno":666}
alert(buf);

var p = JSON.parse(buf); // Deserialización genérica
alert(p.nombre + " " + p.tfno);
</script></body></html>
```

## Protocol Buffers (con C++)

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

Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");

fstream out("person.pb", ios::out | ios::binary | ios::trunc);
person.SerializeToStream(&out);
out.close();

Person person;
fstream in("person.pb", ios::in | ios::binary);
if (!person.ParseFromStream(&in)) {
  cerr << "Failed to parse person.pb." << endl;
  exit(1);
}

cout << "ID: " << person.id() << endl;
cout << "name: " << person.name() << endl;
if (person.has_email()) {
  cout << "e-mail: " << person.email() << endl;
}
```

Especificación (archivo .proto)

Serialización a un fichero (C++)

Deserialización desde un fichero (C++)

## Java Serialization

```
public class Dato implements Serializable {
  int id; String nombre; public Dato(int i, String n) {id = i; nombre = n; }

class Encode {
  static public void main (String args[]) {
    Dato d = new Dato(1, "yo");
    try { ObjectOutputStream o = new ObjectOutputStream(System.out);
        /* serialización */ o.writeObject(d); o.close(); }
    catch (java.io.IOException e) { System.err.println("Error serializando");}}

class Decode {
  static public void main (String args[]) {
    try { ObjectInputStream i = new ObjectInputStream(System.in);
        /* deserialización genérica */ Dato d = (Dato) i.readObject(); i.close();
        System.out.println(d.id + " " + d.nombre); }
    catch (Exception e) { System.err.println("Error deserializando");}}

Contenido = ¡69 BI! http://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html
```

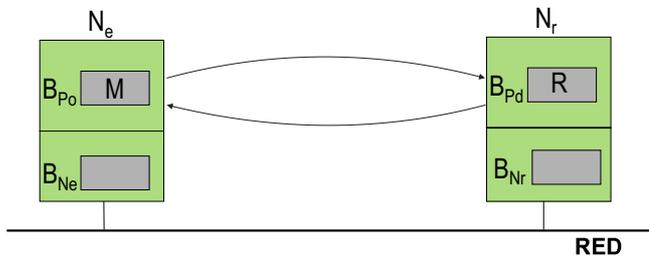
## El precio de un entero (α=150)

Definición	Contenido	Formato
<pre>message Test1 {   required int32 a = 1; }</pre>	08 96 01	<a href="https://developers.google.com/protocol-buffers/docs/encoding">https://developers.google.com/protocol-buffers/docs/encoding</a>
struct dato {int a;};	00 00 96 00	XDR
var d=new Object(); d.a=150;	{"a":150}	JSON
public class D implements Serializable {int a;}	30B	Java

## Grado de sincronía y buffering

- $P_o$  envía  $M$  a  $P_d$ : copia entre buffers de procesos:  $B_{P_o} \rightarrow B_{P_d}$ 
  - Además puede haber buffers en nodo emisor  $B_{N_e}$  y/o receptor  $B_{N_r}$ 
    - Minimizar copias entre buffers (ideal: zero copy)
- De menor a mayor grado de sincronía
  1. Envío devuelve control inmediatamente
    - No requiere  $B_{N_e}$  pero  $P_o$  no puede reutilizar  $B_{P_o}$  hasta que sea seguro
      - Fin de operación o mensaje copiado en algún buffer ( $B_{N_e}$  o  $B_{N_r}$ )
    - Requiere operación para comprobar si ya se puede reutilizar
  2. Envío devuelve control después de  $B_{P_o} \rightarrow B_{N_e}$ 
    - $P_o$  puede reutilizar  $B_{P_o}$ , pero posible bloqueo si  $B_{N_e}$  lleno
  3. Envío devuelve control cuando  $M$  llega a nodo receptor ( $B_{N_r}$ )
    - No requiere  $B_{N_e}$ ; ACK de  $N_r$  a  $N_e$
  4. Envío devuelve control cuando  $M$  llega a  $P_d$  ( $B_{P_d}$ )
    - No requiere  $B_{N_e}$  ni  $B_{N_r}$ ; ACK de  $N_r$  a  $N_e$
  5. Envío devuelve control cuando  $P_d$  tiene respuesta
    - No requiere  $B_{N_e}$  ni  $B_{N_r}$ ;  $B_{P_o} \leftrightarrow B_{P_d}$ ; respuesta sirve de ACK

## Posibles buffers en comunicación



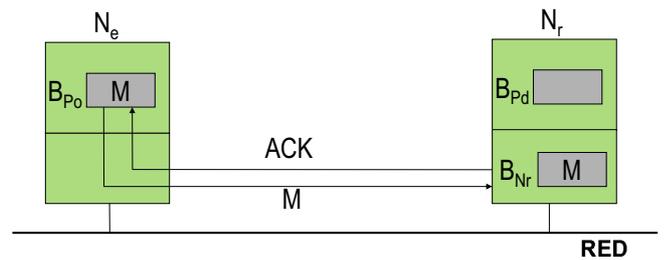
## Retorno inmediato



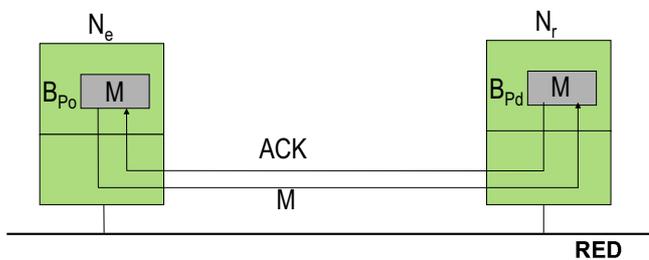
## Retorno después de copia local



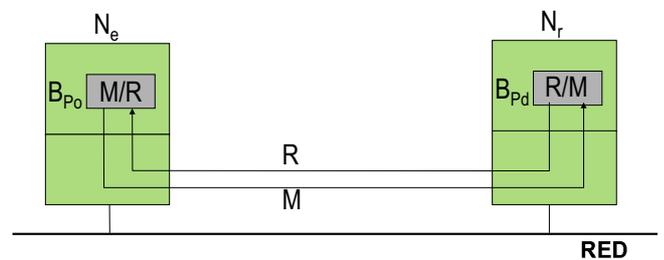
## Retorno después de llegada



## Retorno después de recepción



## Retorno después de respuesta



## Modo de operación en recepción

- Recepción generalmente bloqueante
- Opción no bloqueante: retorna si no hay datos
- Opción asíncrona:
  - Especifica *buffer* donde se almacenará el mensaje y
  - Retorna inmediatamente
  - S. comunicaciones realiza recepción mientras proceso ejecuta
- Espera temporizada: se bloquea un tiempo máximo
- Espera múltiple: espera por varias fuentes de datos

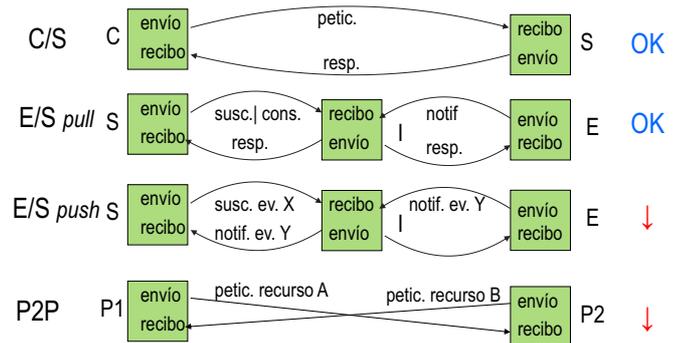
## Sockets: grado de sincronía y buffering

- Modo de operación de envío tipo 2
  - Retorno después de copia local con bloqueo si *buffer* local lleno
  - *Buffer* reservado por SO
- Si aplicación no quiere bloquearse en envío:
  - Usar modo no bloqueante en descriptor socket: error si *buffer* lleno
  - Usar *select/poll/epoll* para comprobar que envío no bloquea
  - Usar E/S asíncrona (*aio\_write*): modo de envío tipo 1
- Modo de operación de recepción bloqueante
- Si aplicación no quiere bloquearse en recepción:
  - Usar modo no bloqueante en descriptor socket: error si *buffer* vacío
  - Usar *select/poll/epoll* para comprobar que hay datos que recibir
  - Usar E/S asíncrona (*aio\_read*)
- Espera múltiple temporizada mediante *select/poll/epoll*

## Adecuación a arquitecturas del SD

- Paso de mensajes adecuado para cualquier arquitectura
  - Pero cuidado con su asimetría: uno envía y otro recibe
- Cliente/servidor: su asimetría encaja con la del paso mensajes
  - Cliente: envía petición y recibe respuesta
  - Servidor: recibe petición y envía respuesta
- Editor/subscriptor: su asimetría no siempre encaja con p. mens.
  - Si *pull* con intermediario I: buen encaje
    - Su|Ed envían ops. a I y reciben respuestas de I → I siempre pasivo
  - Si *push* con intermediario I: encaje problemático
    - Su envía suscripción(evento X) y espera confirmación de I
    - Pero justo antes I envía notificación de evento Y a Su
    - Soluciones: uso de múltiples puertos y concurrencia en subscriptor
- P2P: arquitectura simétrica
  - ¿Quién envía y quién recibe?

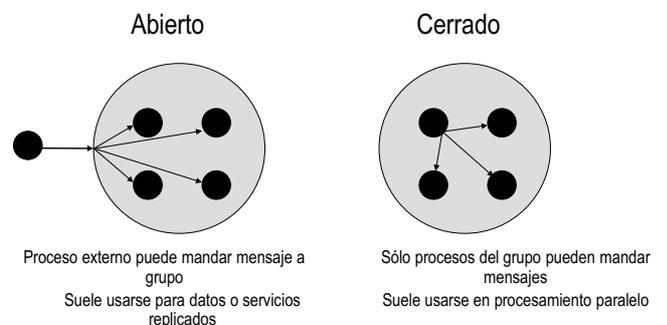
## Paso de mensajes y arquitecturas



## Multidifusión: comunicación de grupo

- Destino de mensaje → grupo de procesos
- Envío/recepción especifican dirección de grupos de procesos
  - Desacoplamiento espacial
- Trabajo seminal: ISIS (posteriores Horus, Ensemble, JGroups)
- Adecuación a arquitectura del SD:
- Cliente-servidor replicado
    - Facilita actualizaciones múltiples
  - Modelo editor/subscriptor
    - Envío de notificaciones (p.e. 1 grupo/tema)
- Aplicable a replicación de datos y de servicios
- Implementación depende de si red tiene *multicast* (IP-multicast)
- Si no, se implementa enviando *N* mensajes
- Un proceso puede pertenecer a varios grupos (grupos solapados)

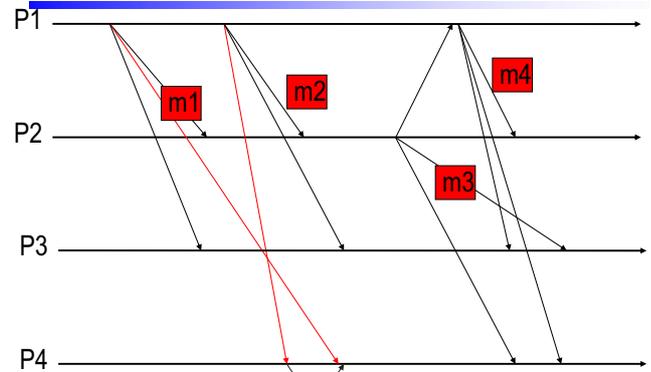
## Grupo abierto versus cerrado



## Aspectos de diseño de com. de grupo

- Atomicidad: o reciben el mensaje o ninguno
  - Con unidifusión fiable (TCP): en medio, se puede caer emisor
  - Con multicast IP: pérdida de mensajes
- Orden de recepción de los mensajes
  - **FIFO**: mensajes de misma fuente llegan en orden de envío
    - No garantía sobre mensajes de distintos emisores
  - **Causal**: entrega respeta relación "causa-efecto"
    - Si no hay relación, no garantiza ningún orden de entrega
  - **Total**: Todos los mensajes recibidos en mismo orden por todos
- El grupo suele tener carácter dinámico
  - Se pueden incorporar y retirar procesos del grupo
  - **Vista**: conjunto de procesos en el grupo en un instante dado
  - Procesos son notificados de los cambios de vista
  - Pertenencia debe coordinarse con comunicación → **sincronía virtual**

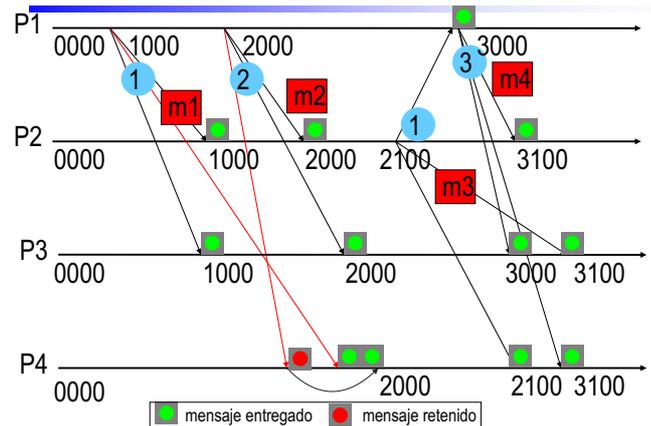
## Orden FIFO



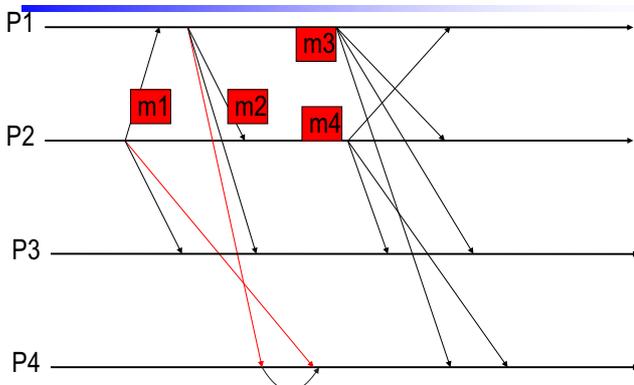
## Mantenimiento de orden FIFO

- Nodo  $N_i$  almacena vector de contadores  $V_i$  (1 posición/nodo)
  - $V_i[i]$ : nº último mensaje enviado por  $N_i$
  - $V_i[j]$  ( $j \neq i$ ): nº último mensaje de  $N_j$  recibido por  $N_i$
- $N_i$  envía mensaje  $M$  que incluye contador  $C_m$ :
  - $V_i[i]++$ ;  $C_m = V_i[i]$
- $N_j$  recibe mensaje  $M$  de  $N_i$ 
  - No se entrega  $M$  si  $C_m > V_j[j] + 1$ 
    - No han llegado todavía mensajes previos de  $N_i$ 
      - Retenido hasta que lleguen mensajes de  $N_i$  que faltan [ $V_j[j] + 1, C_m$ ]
  - En entrega:  $V_j[j] = C_m$

## Mantenimiento de orden FIFO



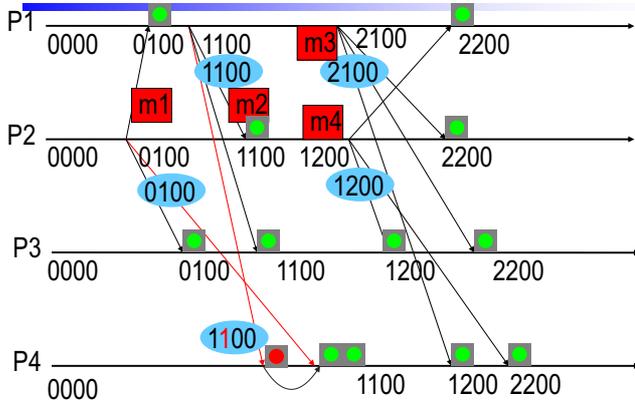
## Orden causal



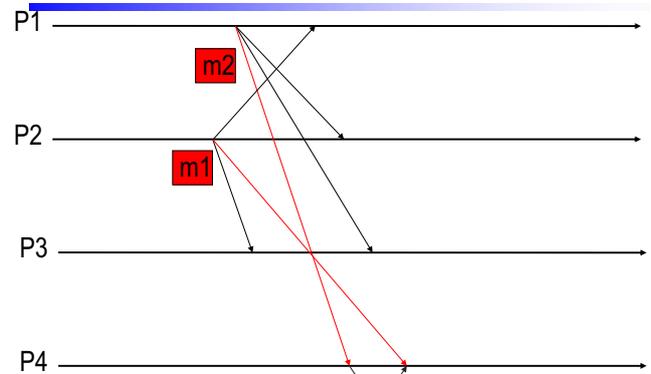
## Mantenimiento de orden causal

- Nodo  $N_i$  almacena vector de contadores  $V_i$  (1 posición/nodo)
  - $V_i[i]$ : nº último mensaje enviado por  $N_i$
  - $V_i[j]$  ( $j \neq i$ ): nº último mensaje de  $N_j$  recibido por  $N_i$
- $N_i$  envía mensaje  $M$  que incluye vector  $V_m$ :
  - $V_i[i]++$ ;  $V_m = V_i$
- $N_j$  recibe mensaje  $M$  de  $N_i$ : No se entrega  $M$  a  $N_j$  si
  - O bien  $V_m[i] > V_j[j] + 1$ 
    - No han llegado todavía mensajes previos de  $N_i$  ( $FIFO \subset causal$ )
    - Retenido hasta que lleguen mensajes de  $N_i$  que faltan [ $V_j[j] + 1, V_m[i]$ ]
  - O bien  $\exists k (k \neq i)$  tal que  $V_m[k] > V_j[k]$ 
    - No han llegado todavía mensajes de  $N_k$  que ya ha recibido  $N_i$
    - Retenido hasta que lleguen mensajes de  $N_k$  que faltan [ $V_j[k] + 1, V_m[k]$ ]
  - En entrega:  $V_j[j] = V_m[j]$
- Basado en vectores de relojes lógicos (tema "sincronización")

## Mantenimiento de orden causal



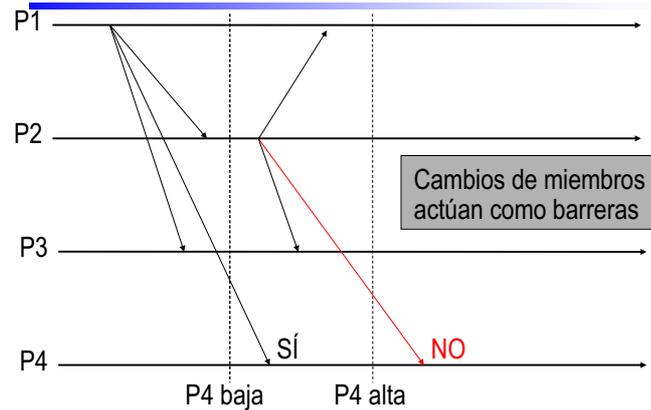
## Orden total



## Mantenimiento de orden total

- Por simplicidad, sólo solución basada en secuenciador S
  - S "cuello de botella" y punto único de fallo
- Proceso S en el sistema asigna número único a cada mensaje
  - S gestiona contador creciente Cs para cada grupo
- Ni envía mensaje de datos M: a todos miembros grupo G + S
  - Mensaje de datos M no incluye contador; sólo algún tipo de ID de M
- Recepción de M en S:
  - Cs++; asigna Cs a M
  - Envía a miembros G mensaje especial Ms = {ID de M, Cs}
- Nodo Ni incluye Ci: n° próximo mensaje Ms que espera recibir
- Recepción de M en Ni: siempre se retiene
- Recepción de Ms = {ID de M, Cs} en Ni:
  - Ms retenido hasta que recibido M y Cs == Ci → entrega de M

## Sincronía virtual



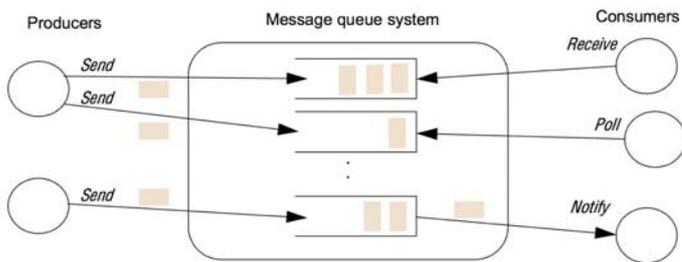
## Ejemplo de uso de Jgroups

- Cliente de *chat* (<http://www.jgroups.org/tutorial/html/ch02.html>):
  - Sin servidor central (OK escalabilidad y tolerancia a fallos)
  - Clientes son informados cuando otro cliente entra/sale del sistema
    - Como cambios de vista
  - Nuevo cliente obtiene estado del sistema: histórico de mensajes
    - Adecuado para implementar replicación de servicios
- Cliente debe implementar *callbacks* para:
  - Recibir los mensajes (*receive*): otro cliente ha escrito en el *chat*
  - Recibir cambios de vista (*viewAccepted*): alta/baja de cliente
  - Obtener estado del sistema (*getState*): nuevo cliente le pide el estado
  - Recibir estado del sistema (*setState*): nuevo cliente recibe el estado
- Código del cliente de *chat*:
  - <http://www.jgroups.org/tutorial/code/SimpleChat.java>
- ¿Uso de orden FIFO/causal/total?
  - Se establece en la configuración del canal

## MOM – Sistemas de colas de mensajes

- *Message-oriented middleware*: RabbitMQ, ZeroMQ, Ap. Kafka
  - AMQP protocolo estándar para MOM
- Envío/recepción mensajes a colas con comunic. "persistente":
  - Comunicación "convencional"
    - Destinatario debe estar presente cuando se recibe mensaje
  - Comunicación "persistente"
    - No es necesario que proceso receptor esté presente
    - Sistema de comunicación (p.e. red de intermediarios) guarda mensaje
- Comunicación desacoplada en espacio y tiempo
- API típico:
  - SEND: envía mensaje a cola
  - RECEIVE: recibe mensaje de cola (bloqueante)
  - POLL: recibe mensaje de cola (no bloqueante)
  - NOTIFY: proceso pide ser notificado cuando llegue mensaje a cola

## Sistema de colas de mensajes

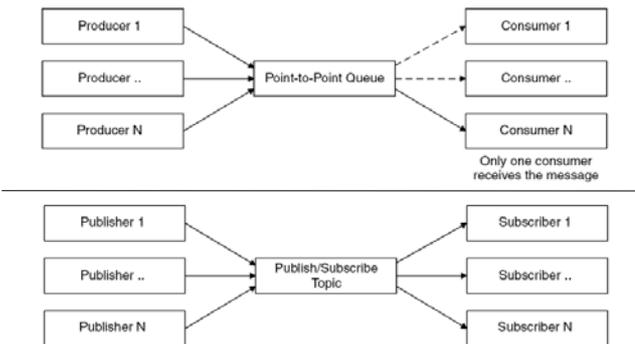


Distributed Systems: Concepts and Design. G. Coulouris et al.

## MOM – Sistemas de colas de mensajes

- 2 modelos de comunicación habituales:
  - Basado en colas punto-a-punto
  - Basado en temas editor/subscriptor
- Adecuación a arquitecturas de SD
  - C/S: punto-a-punto → multi-servidor con reparto automático de carga
  - Ed/Su: modelo basado en temas; NOTIFY permite esquema *push*
- Características avanzadas habituales:
  - Filtrado de mensajes: receptor selecciona en cuáles está interesado
    - por propiedades, por contenido,...
    - Puede usarse como filtro por contenido en archit. Ed./Su.
  - Mensajería con transacciones
  - Transformaciones de mensajes
- Apropiado para integración de aplicaciones de empresa (EAI)

## Punto-a-punto vs. editor/subscriptor



Message-Oriented Middleware. Edward Curry

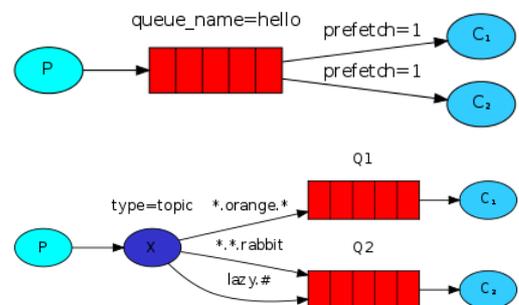
## Rabbit MQ. Ejemplo: Work Queues

- Implementa estándar AMQP; usa único intermediario (*broker*)
- Mensajes y colas pueden configurarse como persistentes
- Permite entrega de mensajes a consumidores tipo *pull* y *push*
- Múltiples consumidores de una cola: reparto *round-robin*
- Mensaje no se elimina de cola hasta reconocimiento:
  - Automático al entregarlo/recogerlo un consumidor
  - Explícito cuando consumidor lo ha procesado
- Ejemplo: servicio con reparto de carga
  - Pero sin respuesta del servidor
  - No se envía petición a un servidor hasta que no ha procesado previa
- <https://www.rabbitmq.com/tutorials/tutorial-two-java.html>

## Rabbit MQ. Ejemplo: Editor/subscriptor

- Rabbit MQ distingue entre *exchanges* y *queues*
  - Productor envía mensaje a *exchange* (buzón de entrada)
  - Sistema encamina mensaje desde *exchange* a cola(s) asociada(s)
  - Por defecto, cada cola tiene un *exchange* anónimo ("" ) asociado
    - Usado en el ejemplo anterior
- Distintos tipos de *exchanges* (*direct*, *fanout*, ***topic***,...)
- *Topic*: secuencia de palabras separadas por .
  - Ejemplo: *quick.orange.rabbit*
  - Productor envía mensaje a *exchange* especificando *topic* y mensaje
  - Consumidor crea cola y la asocia a *exchange* especificando un *topic*
    - Puede usar comodines \* y #: *\*.orange.rabbit* *#.rabbit*
  - Sistema encamina mensaje a todas las colas con *topics* que encajan
- Ejemplo: <https://www.rabbitmq.com/tutorials/tutorial-five-java.html>

## Work Queue vs. editor/subscriptor



<http://www.rabbitmq.com/getstarted.html>

# Sistemas Distribuidos: Ejercicio de los temas arquitectura y comunicación en los sistemas distribuidos

Marzo del 2016.

## Grupo de mañana.

La empresa *SoftStore* proporciona un servicio de distribución de aplicaciones destinado a dos tipos de personas: usuarios de las aplicaciones y desarrolladores de las mismas, organizados en equipos de desarrollo. Los usuarios instalan un programa U que proporciona las siguientes operaciones: (OP1) Búsqueda de aplicaciones basada en los criterios especificados por el usuario (nombre de la aplicación, equipo de desarrollo, temática, precio,...) mostrándose los resultados de la consulta paginados con botones para acceder a la página siguiente y anterior de los mismos; (OP2) Descarga e instalación de la aplicación seleccionada, tal que una vez instalada se muestra una ventana *popup* cada vez que hay una nueva versión de dicha aplicación por si el usuario quiere actualizarla; (OP3) Desinstalación de una aplicación; (OP4) Activación de un servicio de alertas que permite al usuario que lo activa conocer cuándo el equipo de desarrollo seleccionado publica una nueva aplicación (nueva, es decir, la primera versión de la aplicación) mostrándose una ventana *popup* para indicarlo y permitir descargarla e instalarla si lo considera oportuno. Los desarrolladores instalan un programa D que proporciona las siguientes operaciones: (OP5) Crear una nueva aplicación; (OP6) Actualizar una aplicación (es decir, crear una nueva versión de la misma). En la empresa está instalado el módulo software S. El servicio está implementado mediante un esquema cliente/servidor para las búsquedas, cargas y descargas, y dos esquemas editor/subscriptor independientes para el control de actualizaciones y las alertas de nuevas aplicaciones.

1. ¿Qué módulos realizan el papel de subscriptores en el control de actualizaciones?
  - a. U
  - b. D
  - c. U y D
  - d. S

### Explicación

El módulo U realiza el rol de subscriptor del servicio de control de actualizaciones puesto que está interesado en ser notificado cada vez que la aplicación instalada es actualizada.

2. ¿Qué módulos realizan el papel de editores en el control de actualizaciones?
  - a. D
  - b. U
  - c. U y D
  - d. S

### Explicación

El módulo D desempeña el papel de editor del servicio de control de actualizaciones ya que cada vez que un desarrollador crea una nueva versión de una aplicación deben ser informados todos los usuarios que la tienen instalada.

3. ¿Qué módulos realizan el papel de subscriptores en el servicio de alertas?
  - a. U
  - b. D
  - c. U y D
  - d. S

### Explicación

El módulo U realiza el rol de subscriptor del servicio de alertas puesto que está interesado en ser notificado cada vez que un determinado equipo de desarrollo crea una nueva aplicación.

4. ¿Qué módulos realizan el papel de editores en el servicio de alertas?
  - a. D
  - b. U
  - c. U y D
  - d. S

### Explicación

El módulo D desempeña el papel de editor del servicio de alertas ya que cada vez que un desarrollador de un determinado equipo crea una nueva aplicación deben ser informados todos los usuarios que están interesados en las aplicaciones de ese equipo de desarrollo.

5. ¿Qué acción implica OP2?
  - a. **subscripción**

- b. baja
- c. publicación
- d. notificación

### Explicación

La instalación de una aplicación conlleva la subscripción al tema asociado a dicha aplicación para poder recibir avisos cuando se actualiza.

6. ¿Qué acción implica OP3?
  - a. **baja**
  - b. subscripción
  - c. publicación
  - d. notificación

### Explicación

La desinstalación de una aplicación implica la baja de la subscripción al tema asociado a dicha aplicación puesto que ya no se está interesado en las actualizaciones de la misma.

7. ¿Qué acción implica OP4?
  - a. **subscripción**
  - b. baja
  - c. publicación
  - d. notificación

### Explicación

La activación del servicio de alertas conlleva la subscripción al tema asociado al equipo desarrollador correspondiente para poder recibir alertas cuando éste crea una nueva aplicación.

8. ¿Qué acción implica OP5?
  - a. **publicación**
  - b. baja
  - c. subscripción
  - d. notificación

**Explicación** La operación de crear una nueva aplicación por parte de un desarrollador de un determinado equipo deberá ser comunicada a todos los usuarios que han activado una alerta asociada a dicho equipo. Se trata, por tanto, de una operación de publicación.

9. ¿Qué acción implica OP6?
  - a. **publicación**
  - b. baja
  - c. subscripción
  - d. notificación

### Explicación

La operación de crear una nueva versión de una aplicación deberá ser comunicada a todos los usuarios que la tienen instalada. Se trata, por tanto, de una operación de publicación.

10. ¿Qué acción implica el *popup* de actualizaciones?
  - a. **notificación**
  - b. baja
  - c. subscripción
  - d. publicación

### Explicación

El *popup* de actualizaciones se muestra cuando llega al módulo U la notificación de la actualización.

11. ¿Qué acción implica el *popup* de alertas?
  - a. **notificación**
  - b. baja
  - c. subscripción
  - d. publicación

### Explicación

El *popup* de alerta se muestra cuando llega al módulo U la notificación de la nueva aplicación.

12. ¿Cuántos temas existen en el esquema editor/subscriptor que gestiona las actualizaciones?

- a. **Uno por cada aplicación**
- b. Uno por cada usuario
- c. Uno por cada equipo de desarrollo
- d. Uno por cada actualización

#### Explicación

El servicio de control de actualizaciones permite ser informado cada vez que una aplicación se actualiza. Por tanto, hay un tema por cada aplicación existente.

13. ¿Cuántos temas existen en el esquema editor/subscriptor que gestiona las alertas?
  - a. **Uno por cada equipo de desarrollo**
  - b. Uno por cada usuario
  - c. Uno por cada aplicación
  - d. Uno por cada actualización

#### Explicación

El servicio de alertas permite ser informado cada vez que un equipo de desarrollo crea una nueva aplicación. Por tanto, hay un tema por cada equipo de desarrollo existente.

14. Se están valorando dos implementaciones del mecanismo de paginación de las respuestas: cuando se pulsa el botón siguiente o anterior, se envía en el mensaje (1) un parámetro que indica si se pide la siguiente o la anterior, respectivamente, o bien (2) el número absoluto de la página requerida. ¿Qué solución (i) proporciona mayor tolerancia a los reinicios del servidor y cuál (ii) facilita el reparto de carga entre servidores si se usa un esquema de múltiples servidores en S?
  - a. **(2)(2)**
  - b. (1)(1)
  - c. (1)(2)
  - d. (2)(1)

#### Explicación

La solución (1) requiere estado en el servidor puesto que éste tiene que almacenar el último número de página de resultados que ha presentado cada usuario para poder interpretar adecuadamente la petición de la página siguiente o previa. La solución (2) no requiere estado en el servidor al especificarse en la petición el número de página requerido. Por tanto, la solución (2), al no requerir estado en el servidor, proporciona mayor tolerancia a los reinicios del servidor y facilita el reparto de carga entre servidores si se usa un esquema de múltiples servidores.

15. La descarga de una aplicación toma un tiempo considerable y durante la misma puede perderse la conectividad con la empresa, sobretodo si el usuario utiliza un dispositivo móvil. Ante este problema, cuando se recupera la conectividad, se usa una petición GET de HTTP con rangos que permite solicitar un determinado intervalo de bytes de un recurso web. ¿Se trata de una operación (i) idempotente, (ii) que requiere estado en el servidor?
  - a. **sí; no**
  - b. no; sí
  - c. no; no
  - d. sí; sí

#### Explicación

La operación GET con rangos es idempotente puesto que, aunque se repita varias veces, el resultado final es el mismo. Además, esta operación no requiere estado en el servidor al contener toda la información necesaria para su procesamiento (la identificación del recurso y el rango de bytes solicitado).

16. Suponiendo que se usa un esquema de *leasing* para mejorar la tolerancia a fallos del esquema editor/subscriptor de alertas, ¿qué módulos deben renovar el *lease*?
  - a. **U**
  - b. D
  - c. U y D
  - d. S

#### Explicación

Cuando se aplica un mecanismo de *leasing* en un esquema editor-subscriptor, son los subscriptores los encargados de enviar el mensaje de renovación. Por tanto, se trata del módulo U.

17. Se plantea usar un esquema con un filtro de eventos por contenido en vez de un filtro por temas para el sistema de alertas. ¿Para cuál de estos casos ese cambio sería más ventajoso en el sentido de reducir el número de mensajes recibidos pero no deseados?
  - a. **Interés en cuando se publica una aplicación de una determinada temática.**
  - b. Interés en cualquier nueva aplicación.
  - c. Interés en cuando un determinado equipo de desarrollo crea aplicaciones de un conjunto de temáticas.
  - d. Interés en cuando cualquiera de un conjunto de equipos de desarrollo crea aplicaciones de una determinada temática.

#### Explicación

Veamos cada caso planteado analizando en cuál el uso de un filtro por contenido sería más efectivo (es decir, descartaría más eventos no deseados).

- En el caso de estar interesado en cualquier nueva aplicación, el subscriptor tendría que suscribirse a todos los equipos de desarrollo y no descartaría ningún evento.
- En cuanto al caso de estar interesado en ser notificado cuando un cierto equipo de desarrollo crea aplicaciones de un conjunto de temáticas, el subscriptor tendría que suscribirse al tema asociado a ese equipo y descartar todos los eventos que no correspondan a aplicaciones de esas temáticas.
- Con respecto al caso de estar interesado en ser notificado cuando cualquiera de un conjunto de equipos de desarrollo crea aplicaciones de una determinada temática, el subscriptor tendría que suscribirse a los temas correspondientes a esos equipos y descartar todos los eventos que no correspondan a aplicaciones de esa temática.
- Por lo que se refiere al caso de estar interesado en ser notificado cuando se publica una aplicación de una determinada temática, el subscriptor tendría que suscribirse a todos equipos y descartar todos los eventos que no correspondan a esa temática. Nótese que en este caso, con un filtro por temas, el subscriptor tendría que recibir todos los eventos del sistema.

18. Suponga un sistema de comunicación en grupo causal con 4 procesos (de P1 a P4) donde P2 tiene un vector (2,2,2,2) y recibe un mensaje M de P3 con un vector (4,0,4,1). ¿Cuántos mensajes adicionales tiene que recibir antes de entregar M?
  - a. **3**
  - b. 2
  - c. 4
  - d. 5

#### Explicación

Dado que el mensaje proviene de P3, la tercera componente del vector recibido está indicando que es el cuarto mensaje de P3 pero, sin embargo, la tercera componente del vector de P2 indica que hasta el momento sólo se ha recibido dos, por lo que falta el tercer mensaje para satisfacer la política FIFO que debe cumplirse también en un sistema causal. En cuanto a la primera componente del mensaje recibido, indica que el proceso P3 ya ha recibido cuatro mensajes de P1 pero, sin embargo, la primera componente del vector de P2 indica que éste hasta el momento sólo ha recibido dos de P1, por lo que faltan dos mensajes de dicho proceso. Por lo que se refiere a la cuarta componente, indica que P3 ha recibido un único mensaje de P4 mientras que esa misma componente del vector de P2 refleja que éste ya ha recibido dos mensajes de P4, por lo que no tiene que esperar ninguno adicional. Recapitulando, faltan 3 mensajes.

19. ¿Qué es cierto con respecto a un sistema de serialización que proporciona un esquema de *unmarshalling* genérico: (i) consume menos ancho de banda; (ii) es más eficiente?
  - a. **Ninguna.**
  - b. (i)
  - c. (ii)
  - d. Ambas.

#### Explicación

Un esquema de *unmarshalling* genérico requiere que, junto con los datos que se pretenden transmitir o almacenar, se incluya información de cuál es el tipo de los mismos, lo que implica un mayor consumo de ancho de banda y un procesamiento menos eficiente frente a un esquema no genérico donde sólo se envían los datos y el receptor conoce implícitamente cuál es el tipo de los mismos.

20. ¿Cuántas de las siguientes afirmaciones sobre qué situación se produce cuando se completa una llamada *send* sobre un socket son ciertas: (i) el programa puede reutilizar el buffer del mensaje inmediatamente; (ii) el mensaje ha llegado al nodo receptor pero todavía no ha sido entregado al proceso destinatario; (iii) el mensaje ha sido entregado al proceso destinatario?
  - a. **1**
  - b. 2
  - c. 3
  - d. 0

#### Explicación

Cuando devuelve el control una llamada *send* sobre un socket, el mensaje se ha copiado a un buffer local, por lo que el programa puede reutilizar inmediatamente el buffer del mensaje, pero el mensaje con ha llegado al nodo remoto.

# Sistemas Distribuidos: Ejercicio de los temas arquitectura y comunicación en los sistemas distribuidos

Marzo del 2016.

## Grupo de tarde.

La empresa *GameOver* proporciona la infraestructura para el despliegue de juegos *online* multijugador de gran escala (*Massively Multiplayer Online Game*) basados en un universo de *habitaciones* conectadas por *puertas*. Esta infraestructura está implementada en la empresa (módulo G) mediante una solución que combina un esquema cliente/servidor, para la descarga de información, con un editor/subscriptor, para la actualización automática de la posición de los jugadores. En este sistema hay dos tipos de usuarios: los jugadores, que deambulan por el universo virtual, y los árbitros, que contemplan las acciones de los diversos jugadores, sin intervenir en el universo virtual. La empresa proporciona una aplicación (C) que permite a un usuario crear un nuevo juego definiendo cada una de las habitaciones del universo y su conectividad. Una vez completada la fase de definición, toda esa información, de gran volumen, se carga en la infraestructura de la empresa. Finalizada esa etapa, el proceso creador se convierte en el árbitro inicial del sistema, pudiéndose incorporar progresiva y dinámicamente más árbitros a lo largo de una partida. Un árbitro contemplará en su pantalla todo el universo y conocerá en todo momento qué jugadores están ubicados en todas y cada una de las habitaciones. La aplicación que corresponde a un árbitro (A) dispone de dos operaciones: (OP1) inicio de la supervisión, que, entre otras acciones, descarga del servidor G todas las imágenes del universo virtual y descarga desde cualquier otro árbitro la información sobre qué jugadores están ubicados en todas y cada una de las habitaciones (el árbitro inicial no necesita descargar esa información de ubicación) mostrando en la pantalla toda esta información, momento a partir del cual esta aplicación irá recibiendo automáticamente todas las actualizaciones de la ubicación de los jugadores, que reflejará en la pantalla; (OP2) fin de la supervisión. Un jugador ve en su pantalla sólo la habitación que ocupa en ese instante y, por tanto, sólo debe conocer en cada momento qué otros jugadores están en esa misma habitación. La aplicación que corresponde a un jugador (J) dispone, entre otras, de las siguientes operaciones: (OP3) *login* y (OP4) *logout*, que contactan con el servidor de la empresa para proporcionar información del jugador y para finalizar el juego, respectivamente; (OP5) entrar en una determinada habitación, que, entre otras acciones, descarga desde el servidor G la imagen de la habitación y desde cualquier árbitro la información sobre qué jugadores están ubicados en esa habitación en ese momento mostrando en la pantalla esa información (esta entrada se reflejará automáticamente en las pantallas de todos los árbitros y de los otros jugadores que están en esa nueva habitación), momento a partir del cual esta aplicación irá recibiendo automáticamente todas las actualizaciones que corresponden a otros jugadores que entran y salen de la habitación, que reflejará en su pantalla; (OP6) salir de la habitación que ocupa el jugador, que elimina de la pantalla la representación de la habitación, y que se reflejará automáticamente en las pantallas de todos los árbitros y de los otros jugadores que estaban en esa habitación. Nótese que la acción de atravesar la puerta entre dos habitaciones se interpreta como una operación de salir seguida por una de entrar.

1. ¿Qué módulos realizan el papel de subscriptores?
  - a. **J y A**
  - b. J
  - c. A
  - d. G

### Explicación

El módulo J correspondiente a un determinado jugador realiza el rol de subscriptor puesto que está interesado en ser notificado cada vez que otro jugador entra o sale de la habitación que este primero ocupa actualmente. El módulo A también hace ese mismo papel de subscriptor ya que necesita conocer en todo momento las ubicaciones de todos los jugadores.

2. ¿Qué módulos realizan el papel de editores?
  - a. **J**
  - b. J y A
  - c. A
  - d. G

### Explicación

El módulo J realiza el rol de editor puesto que los cambios de ubicación del jugador asociado a ese módulo deben ser notificados a todos los árbitros así como a todos los jugadores que estén en la misma habitación que este jugador.

3. ¿Qué acción implica OP1?
  - a. **subscripción**
  - b. baja
  - c. publicación
  - d. notificación

### Explicación

El inicio de la supervisión por parte de un árbitro conlleva la subscripción a todos los temas asociados con todas y cada una de las habitaciones puesto que necesita conocer en todo momento el estado del sistema.

4. ¿Qué acción implica OP2?

- a. **baja**
- b. subscripción
- c. publicación
- d. notificación

### Explicación

El fin de la supervisión por parte de un árbitro conlleva la baja de la subscripción a todos los temas del sistema puesto que ya necesita conocer ninguna información acerca del mismo.

5. ¿Las acciones OP5 y OP6 provocan una subscripción?
  - a. **OP5 sí**
  - b. OP6 sí
  - c. Ambas
  - d. Ninguna

### Explicación

Cuando un jugador entra en una habitación, debe subscribirse al tema asociado a la misma para ser notificado a partir de ese momento de que otros jugadores entran y salen de esa habitación.

6. ¿Las acciones OP5 y OP6 causan baja de una subscripción?
  - a. **OP6 sí**
  - b. OP5 sí
  - c. Ambas
  - d. Ninguna

### Explicación

Cuando un jugador sale de una habitación, debe darse de baja del tema asociado a la misma para dejar de ser notificado cuando otros jugadores entran y salen de la misma.

7. ¿Las acciones OP5 y OP6 provocan publicar un evento?
  - a. **Ambas**
  - b. OP5 sí
  - c. OP6 sí
  - d. Ninguna

### Explicación

Tanto la operación de entrar en una habitación como la de salir de la misma deberán ser comunicadas a todos los árbitros así como a los usuarios que comparten esa habitación para que puedan reflejar en sus respectivas pantallas ese hecho. Ambas acciones, por tanto, corresponden a una operación de publicación.

8. ¿Qué mensaje recibido por un jugador provoca que se elimine a otro jugador de su pantalla?
  - a. **notificación**
  - b. baja
  - c. subscripción
  - d. publicación

### Explicación

El borrado en la pantalla de un jugador de la imagen de otro se lleva a cabo cuando se recibe la notificación de que ese segundo jugador ha salido de la habitación que compartían hasta ese momento.

9. ¿Qué mensaje recibido por un jugador provoca que se añada otro jugador en su pantalla?
  - a. **notificación**
  - b. baja
  - c. subscripción
  - d. publicación

### Explicación

El dibujar en la pantalla de un jugador la imagen de otro se lleva a cabo cuando se recibe la notificación de que ese segundo jugador ha entrado en la habitación donde está ubicado el primero.

10. Teniendo en cuenta la funcionalidad de J, ¿Cuántos temas existen en el esquema editor/subscriptor?
  - a. **Uno por habitación**
  - b. Sólo uno
  - c. Uno por árbitro
  - d. Uno por jugador

### Explicación

Un jugador sólo debe ser informado cada vez que otro jugador entra o sale de la habitación donde está ubicado el primero. Por tanto, hay tantos temas como habitaciones.

11. Se requiere saber cuánto tiempo participa en el juego cada jugador. Para ello, se calcula la diferencia del tiempo tomado en el servidor en OP3 y OP4. Se están valorando dos implementaciones: (1) en OP3 G envía el valor de tiempo a J y éste presenta ese valor original a G en OP4 para calcular la diferencia con el tiempo actual; (2) en OP3 G almacena el tiempo en ese momento y le devuelve a J un identificador de sesión que éste presenta a G en OP4 lo que le permite recuperar el valor original y restarlo del tiempo actual. ¿Qué solución (i) proporciona mayor tolerancia a los reinicios del servidor y cuál (ii) facilita el reparto de carga entre servidores si se usa un esquema de múltiples servidores en G?
- (1)(1)
  - (2)(2)
  - (1)(2)
  - (2)(1)

#### Explicación

La solución (2) requiere estado en el servidor puesto que éste tiene que almacenar el momento en el que el usuario ha hecho el *login*. La solución (1) no requiere estado en el servidor ya que en la operación de *logout* éste recibe la marca de tiempo inicial. Por tanto, la solución (1), al no requerir estado en el servidor, proporciona mayor tolerancia a los reinicios del servidor y facilita el reparto de carga entre servidores si se usa un esquema de múltiples servidores.

12. La carga al servidor de las imágenes del universo desde C toma un tiempo considerable y durante la misma puede perderse la conectividad con la empresa, sobretodo si el usuario creador utiliza un dispositivo móvil. Ante este problema, cuando se recupera la conectividad, se usa una petición PUT de HTTP con rangos que permite enviar un determinado intervalo de bytes de un recurso web. ¿Se trata de una operación (i) idempotente, (ii) que requiere estado en el servidor?
- sí; no
  - no; sí
  - no; no
  - sí; sí

#### Explicación

La operación PUT con rangos es idempotente puesto que, aunque se repita varias veces, el resultado final es el mismo. Además, esta operación no requiere estado en el servidor al contener toda la información necesaria para su procesamiento (la identificación del recurso y el rango de bytes escritos).

13. Suponiendo que se usa un esquema de *leasing* para mejorar la tolerancia a fallos del esquema editor/subscriber, ¿qué módulos deben renovar el *lease*?
- J y A
  - J
  - A
  - G

#### Explicación

Cuando se aplica un mecanismo de *leasing* en un esquema editor-subscriber, son los subscribers los encargados de enviar el mensaje de renovación. Por tanto, se trata de los módulos J y A.

14. ¿Qué módulo se beneficiaría más de una operación de suscripción múltiple con comodines?
- A
  - J
  - G
  - C

#### Explicación

Un jugador sólo está suscrito a un tema/habitación en cada momento. Un árbitro, sin embargo, necesita estar suscrito a todos los temas/habitaciones por lo que se beneficiaría del uso de una operación de suscripción múltiple.

15. Suponga que se usa un sistema de comunicación de grupo para implementar un esquema editor/subscriber basado en temas, ¿cuántas direcciones de grupo deben existir en el sistema?
- Una por cada tema
  - Una por subscriber
  - Una por editor
  - Una por evento

#### Explicación

Habría una dirección por cada tema ya que esto permitiría que la operación de publicación de un evento se llevara a cabo realizando un envío a la dirección asociada al tema correspondiente.

16. Si se usa un esquema de *binding*, ¿qué módulos deben darse de alta en el mismo?
- G y A
  - J y A
  - G y J
  - J

#### Explicación

Todo módulo que sea el destinatario de una petición debe darse de alta en el servicio de binding. El módulo G lo requiere tanto en su rol de proceso intermediario en el esquema editor/subscriber como en su papel de servidor a la hora de descargar las imágenes del universo del juego. El módulo A también lo requiere puesto que actúa como servidor al que se le solicita el estado de ocupación de una o más habitaciones.

17. ¿Para el seguimiento de cuál de las siguientes informaciones de estado sería más ventajoso usar un filtro de eventos por contenido en vez de un filtro por temas en el sentido de reducir el número de notificaciones no deseadas?
- Ubicación en cada momento de un jugador dado.
  - Qué jugadores hay en cada instante en un determinado conjunto de habitaciones.
  - Conocer todo el estado del sistema.
  - Interés en cuando cualquiera de un conjunto de jugadores entra en una determinada habitación.

#### Explicación

Veamos cada caso planteado analizando en cuál el uso de un filtro por contenido sería más efectivo (es decir, descartaría más eventos no deseados).

- En cuanto al caso de estar interesado en saber qué jugadores hay en cada instante en un determinado conjunto de habitaciones, el subscriber tendría que suscribirse a los temas asociados a esas habitaciones y no descartar ningún evento.
  - En el caso de estar interesado en conocer todo el estado del sistema, el subscriber tendría que suscribirse a todas las habitaciones y no descartaría ningún evento.
  - Con respecto al caso de estar interesado en ser notificado cuando cualquiera de un conjunto de jugadores entra en una determinada habitación, el subscriber tendría que suscribirse al tema correspondiente a esa habitación y descartar todos los eventos que no estén vinculados con la entrada de uno de esos jugadores.
  - Por lo que se refiere al caso de estar interesado en conocer la ubicación en cada momento de un jugador dado, el subscriber tendría que suscribirse a todas las habitaciones y descartar todos los eventos que no correspondan a ese jugador. Nótese que en este caso, con un filtro por temas, el subscriber tendría que recibir todos los eventos del sistema.
18. Suponga un sistema de comunicación en grupo causal con 4 procesos (de P1 a P4) donde P4 tiene un vector (2,2,2,2) y recibe un mensaje M de P1 con un vector (5,3,3,0). ¿Cuántos mensajes adicionales tiene que recibir antes de entregar M?
- 4
  - 2
  - 3
  - 5

#### Explicación

Dado que el mensaje proviene de P1, la primera componente del vector recibido está indicando que es el quinto mensaje de P1 pero, sin embargo, la primera componente del vector de P4 indica que hasta el momento sólo se ha recibido dos, por lo que faltan el tercer y cuarto mensaje para satisfacer la política FIFO que debe cumplirse también en un sistema causal. En cuanto a la segunda componente del mensaje recibido, indica que el proceso P1 ya ha recibido tres mensajes de P2 pero, sin embargo, la segunda componente del vector de P4 indica que éste hasta el momento sólo ha recibido dos de P2, por lo que falta un mensaje de dicho proceso. Con la tercera componente ocurre exactamente lo mismo. Recapitulando, faltan 4 mensajes.

19. ¿Cuántos de estos sistemas de serialización incluyen información sobre la identificación de cada campo y sobre el tipo del mismo: XDR, Protocol Buffers y Java Serialization?
- 1
  - 2
  - 3
  - 0

#### Explicación

XDR sólo transmite, o almacena, los datos que se pretenden compartir, no incluyendo información identificativa de los campos que componen la información ni del tipo de los mismos. Protocol Buffers no incluye tampoco información de tipos pero sí añade información identificativa de los campos. Java Serialization incluye ambos tipos de informaciones.

20. ¿Qué es cierto con respecto a un `send` sobre un socket: (1) cuando retorna, el programa no puede reutilizar el buffer inmediatamente; (2) no es *zero-copy*; (3) nunca se bloquea; (4) se bloquea hasta que el destinatario recibe el mensaje?
- 2
  - 1
  - 3
  - 4

#### Explicación

Cuando devuelve el control una llamada `send` sobre un socket, el mensaje se ha copiado a un buffer local, rompiendo el criterio *zero-copy*, por lo que el programa puede reutilizar inmediatamente el buffer del mensaje, pero el mensaje con ha llegado al nodo remoto.