Sistemas Distribuidos

Arquitectura de los Sistemas Distribuidos

Índice

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

Arquitectura de los SD

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

Grado de acoplamiento

- Sea cual sea el modelo, conlleva interacción entre entidades
- Interacción tradicional implica acoplamiento espacial y temporal
- Desacoplamiento espacial (de referencia)
  - Entidad inicia interacción **no** hace referencia directa a la otra entidad
    - No necesitan conocerse entre sí
- Desacoplamiento temporal (menos frecuente)
  - "Vidas" de entidades que interaccionan **no** tienen que coincidir
- Ej. 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."*

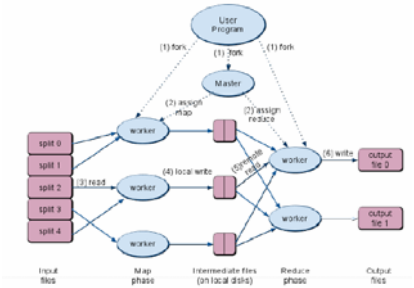
Arquitecturas para CD

- Maestro-trabajador *M/T* (*aka* maestro-esclavo)
  - *M* va repartiendo trabajos entre nodos trabajadores *T*
    - Si nº trabajos >> nº trabajadores → reparto automático de carga
  - Tolerancia a fallos:
    - Caída de *T*: *M* reasigna sus trabajos pendientes (¡efectos laterales!)
    - Caída de *M*: punto crítico de fallo
- Arquitecturas guiadas por "geometría" de los datos
  - Matrices multidimensionales, grafos, etc.
  - P.e. Matriz 2D
    - Cada nodo se encarga de sub-matriz
    - Comunicación más frecuente con nodos "vecinos cartesianos"
  - P.e. Grafo
    - Cada nodo se encarga de un sub-grafo
    - Comunicación siguiendo aristas

Arquit. de computación en Google

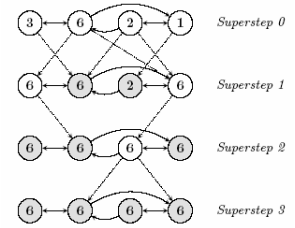
- *MapReduce*
  - Maestro-trabajador con dos fases: *Map* y *Reduce*
  - *Map*: *T* procesa su parte de datos de entrada y genera (*clave*, *valor*)
    - P.ej. Extrae de logs web → (página, usuario que la accede)
  - *Reduce*: *T* procesa valores asociados a una determinada clave
    - P.ej. Calcula nº accesos únicos a cada página → (página, nº accesos)
- *Pregel*
  - Modelo diseñado para procesar grafos de gran escala
  - Computación organizada en "*supersteps*" síncronos:
    - Cada vértice recibe datos de otros vértices por aristas de entrada
    - Cambia su estado y genera datos por vértices de salida
    - Incluso puede cambiar topología del grafo
  - Inspirado en modelo "*Bulk Synchronous Parallel*" de Valiant
  - Implementado como arquitectura maestro/trabajador
    - *M* reparte grafo entre *T* y controla sincronización de "*supersteps*"

Modelo de computación *Map-Reduce*



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

Modelo de computación *Pregel*



Pregel: A System for Large-Scale Graph Processing
Grzegorz Malewicz et al.; *SIGMOD '10*

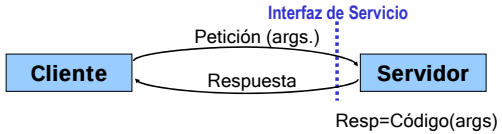
Arquitecturas en SD de propósito general

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

Modelo cliente/servidor

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

Esquema cliente/servidor



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

Reparto funcionalidad entre C y S

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

Posibles repartos entre C y S

- Arquitectura típica de aplicación basada en 3 capas:
  - Presentación (interfaz de usuario gráfica: GUI)
  - Aplicación: lógica de negocio
  - Acceso a datos
- ¿Qué labor de la aplicación se le asigna a máquina cliente?
- Alternativas de "grosor" creciente:
  - Nada: máquina cliente sólo incluye servidor gráfico (p.e. X11 o VNC)
    - Envía a app. eventos ratón/teclado y recibe de app. info. a dibujar en:
      - Píxeles (VNC) o Primitivas gráficas (X11)
  - Sólo GUI
  - GUI + parte de la lógica de negocio
  - GUI + lógica de negocio
  - GUI + lógica de negocio + parte de lógica de acceso

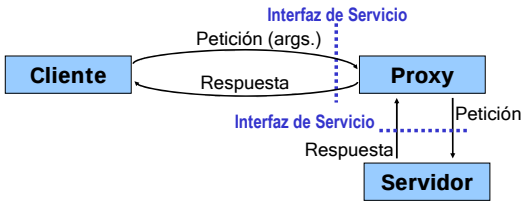
Cliente/servidor con caché

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

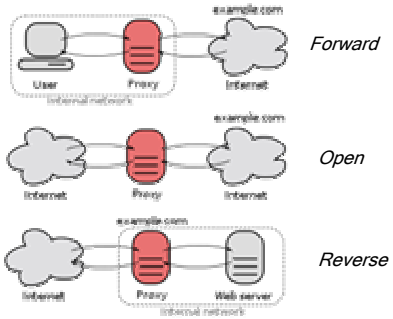
Cliente/servidor con *proxy*

- Componentes intermediarios entre cliente y servidor
- Actúan como "tuberías"
  - Pueden procesar/filtrar información y/o realizar labor de caché
    - Símil con clases *FilterInputStream*|*FilterOutputStream* de Java
- Diversos tipos: *forward proxy*, *reverse proxy*, *gateways*, ...
- 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

Esquema con *proxy*



Wikipedia: *Proxy server*



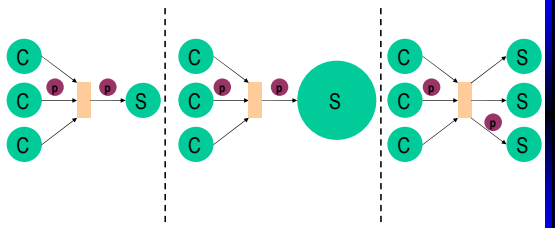
Cliente/servidor jerárquico

- 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

Esquema multi-servidor

- 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 ni tolerancia a fallos
- Uso de múltiples servidores (interacción M-N)
  - Escalado horizontal (*scale-out*)
  - Mejora latencia, escalabilidad y tolerancia a fallos
  - Requiere esquema de reparto de carga
- Si servicio requiere datos replicados (p.e. DNS, Google FS)
  - Necesidad (y sobrecarga) de mantener coherencia entre las réplicas
  - Esquema simétrico: Actualización simultánea en todas las réplicas
  - Esquema asimétrico: Actualizar en primario y propagar a secundarios
    - *Push*: primario envía cambios a secundarios
    - *Pull*: secundario solicita cambios a primario

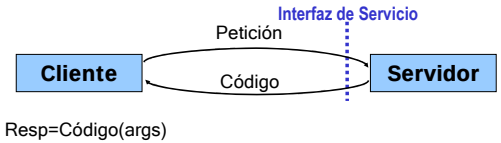
Scale-up vs Scale-out



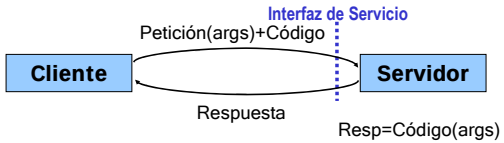
Código móvil

- 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



Aspectos de diseño de cliente/servidor

Se van a considerar 5 aspectos específicos:

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

Localización del servidor

- Servidor en máquina con dirección *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

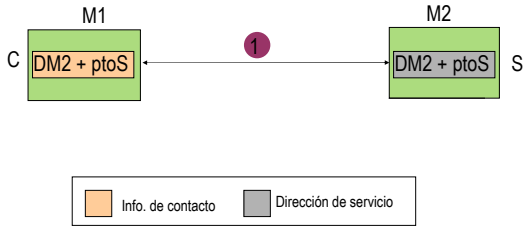
Alternativas en la ID del servicio

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

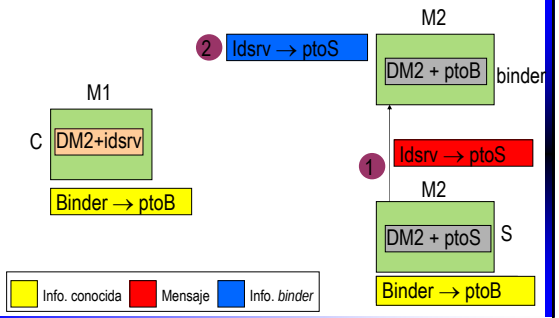
Símil basado en "hechos reales"

- Suponiendo nº tfno persona = (nº tfno empresa + extensión)
- Quiero contactar con persona P (TP = TE + EP)
- Alternativas:
  1. Conozco TP: lo uso
  2. Conozco TE y extensión de información de la empresa
     - Llamo a servicio información de la empresa y obtengo TP
  3. (a) Conozco teléfono general de información de personas
     - Llamo a servicio información general y obtengo TP
  3. (b) Conozco tfno info. empresas y extensión info. de toda empresa
     - Llamo a servicio información general de empresas y obtengo TE
     - Llamo a servicio información de la empresa y obtengo TP

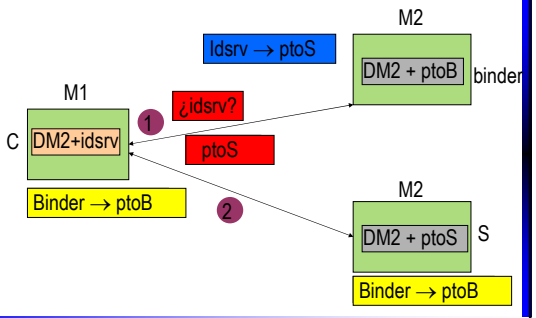
(1) ID servicio = [DM+pto]



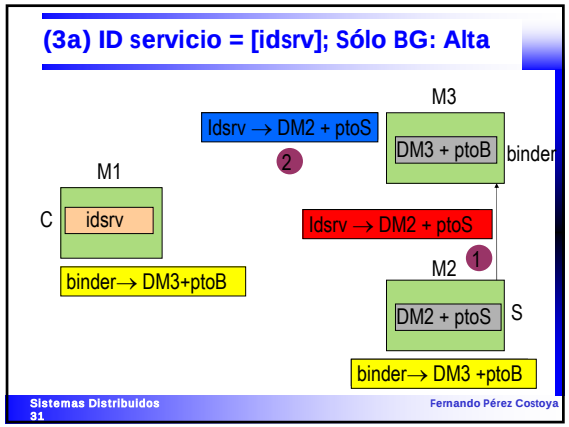
(2) ID servicio = [DM+idsrv]: Alta



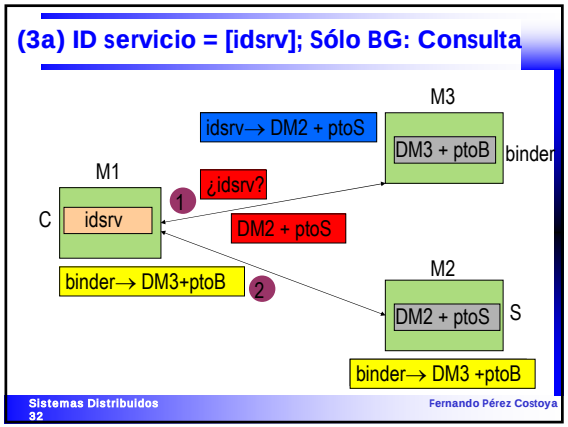
(2) ID servicio = [DM+idsrv]: Consulta



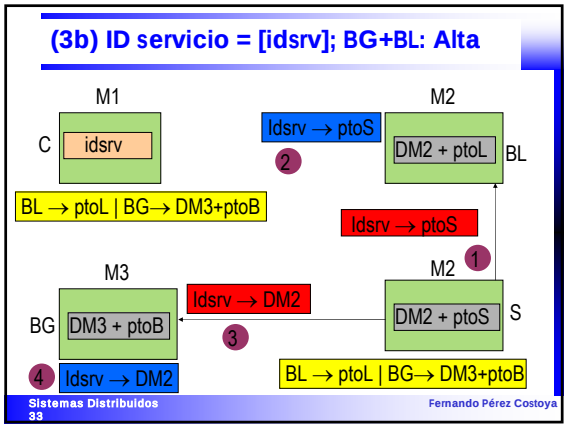
(3a) ID servicio = [idsrv]; Sólo BG: Alta



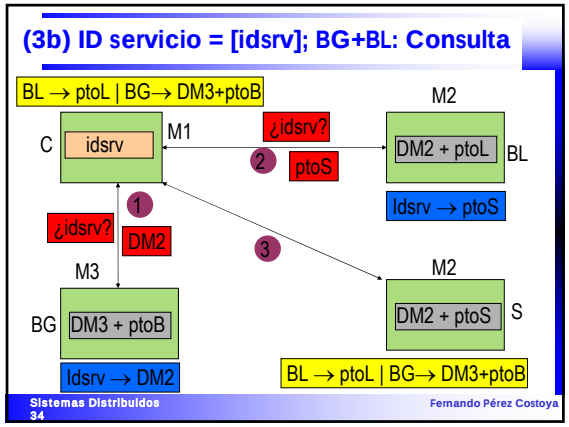
(3a) ID servicio = [idsrv]; Sólo BG: Consulta



(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 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”

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 nº < *M* → se crea
  - Si inactivo tiempo prefijado y nº > *m* → 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 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
  - 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

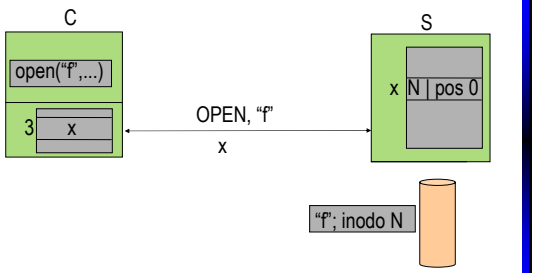
Client Pull vs Server Push

- 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

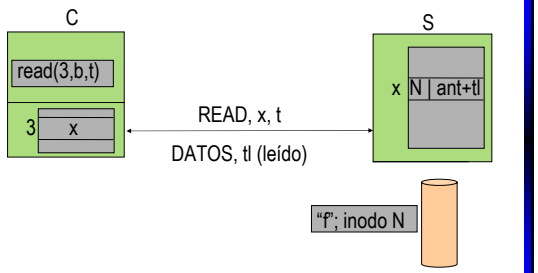
- ¿Servidor mantiene información de clientes?
- Ventajas de servicio con estado (*aka* con sesión remota):
  - Mensajes de petición más cortos
  - Mejor rendimiento (se mantiene información en memoria)
  - Favorece optimización de servicio: estrategias predictivas
- Ventajas de servicio sin estado:
  - Más tolerantes a fallos (ante 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



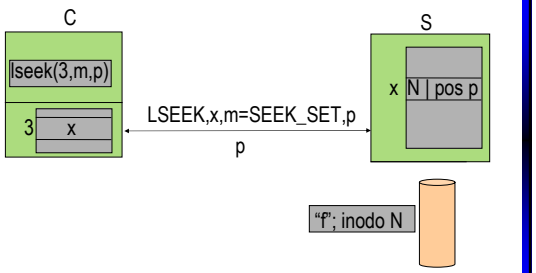
Sistemas Distribuidos 43 Fernando Pérez Costoya

Servicio de ficheros con estado: READ



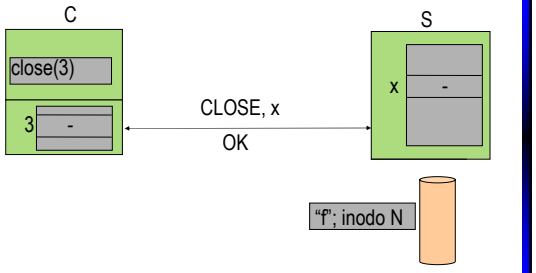
Sistemas Distribuidos 44 Fernando Pérez Costoya

Servicio de ficheros con estado: LSEEK



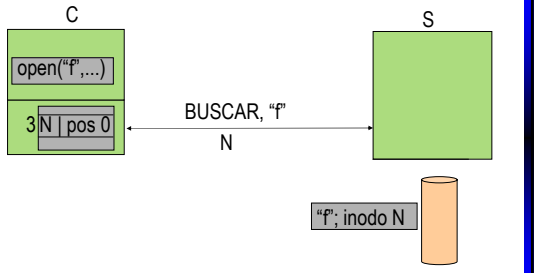
Sistemas Distribuidos 45 Fernando Pérez Costoya

Servicio de ficheros con estado: CLOSE



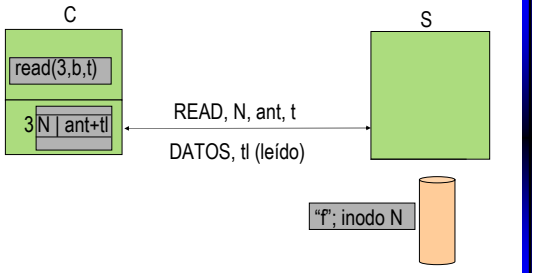
Sistemas Distribuidos 46 Fernando Pérez Costoya

Servicio de ficheros sin estado: OPEN



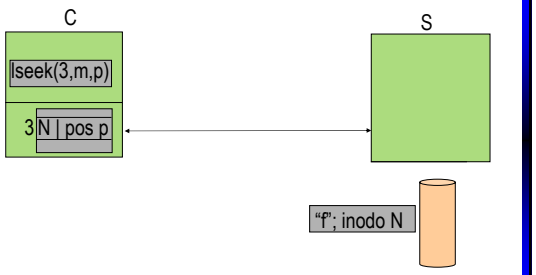
Sistemas Distribuidos 47 Fernando Pérez Costoya

Servicio de ficheros sin estado: READ

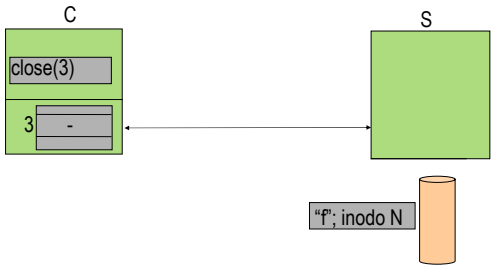


Sistemas Distribuidos 48 Fernando Pérez Costoya

Servicio de ficheros sin estado: LSEEK



Servicio de ficheros sin estado: CLOSE



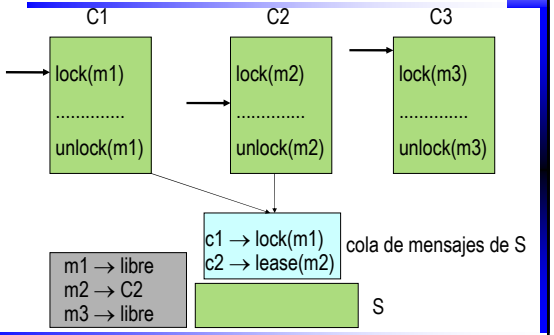
Leases

- 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*)

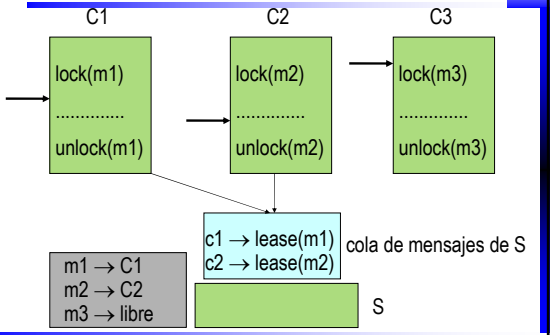
Aplicaciones de *leases*

- Aparecerán a menudo:
  - *Binding*, caídas del cliente, suscripción en Ed/Su, caché de SFD, etc.
- *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

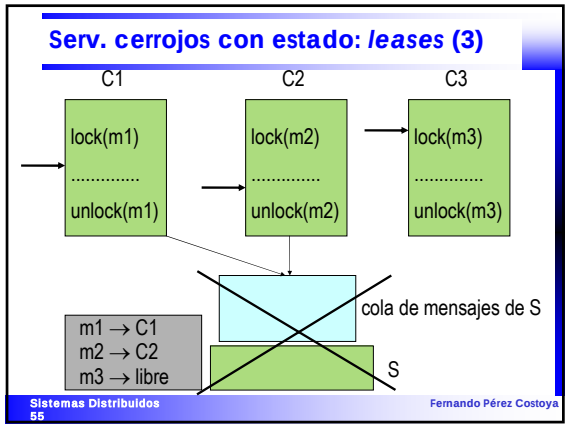
Serv. cerrojos con estado: *leases* (1)



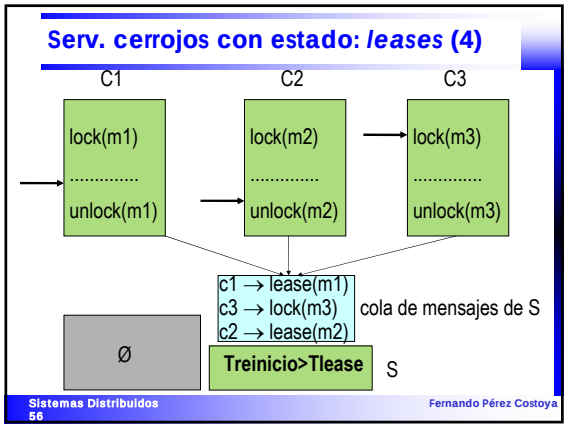
Serv. cerrojos con estado: *leases* (2)



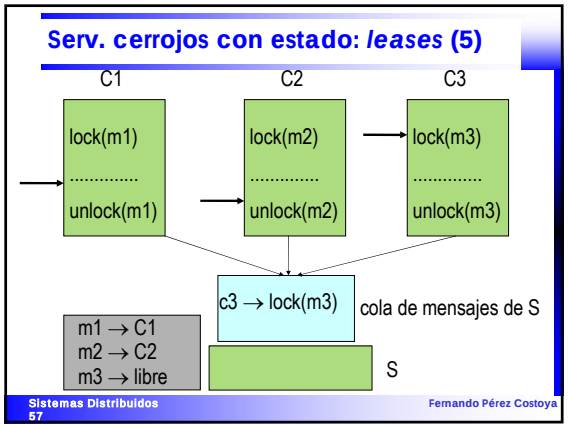
Serv. cerrojos con estado: *leases* (3)



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 (**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

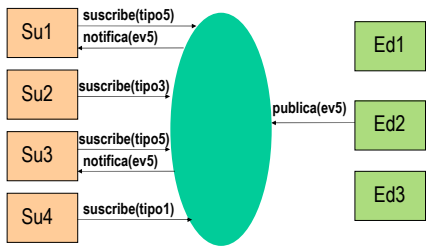
Modelo editor/subscriptor

- Sistema de eventos distribuidos (Jini, s. eventos/notifi. CORBA)
- 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
  - Híbrido: suscriptor recibe notificación pero tiene que solicitar evento
  - *Pull* e híbrido requieren que se almacenen eventos (+ complejo)
    - Posibilitan 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,…

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)* [→S]: envío de evento (*push*)
- Extensión de modelo: creación dinámica de tipos de eventos
  - *anuncia(tipo)* [E→]: se crea un nuevo tipo de evento
  - *baja_tipo(tipo)* [E→]: se elimina tipo de evento
  - *notifica_tipo(tipo)* [→S]: aviso de nuevo tipo de eventos
- Ejemplos de aplicación
  - Mercado bursátil, subastas, *chat*, aplicación domótica, etc.

Modelo editor/subscriptor



Posible extensión: anuncio de nuevo tipo de evento (E→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 la suscripción
- Filtrados adicionales deben hacerse en aplicación
  - En ocasiones filtro por tema resulta un grano demasiado grueso
    - Ej. Interesado en todos los valores de un ámbito de negocio

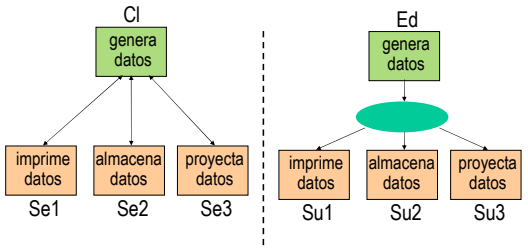
Filtro de eventos por contenido

- 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
  - Ej. Interés en valores de mercado en alza
- 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*"

Filtro de eventos por tipo

- Aplicable sólo a sistemas distribuidos basados en objetos
- Interés en un eventos de una determinada clase
  - Como es habitual, esa clase pertenece a una jerarquía de herencia
- Permite ambos esquemas previos
- Filtro por temas jerárquicos:
  - Clase = tema
    - Jerarquía de clases = Jerarquía de temas
- Filtro por contenido
  - En suscripción, se especifica función de filtrado
- Ventaja: las que proporciona todo el sistema de tipado
  - Pero requiere un sistema que use un esquema de objetos homogéneo

Cliente/servidor vs. Editor/suscriptor

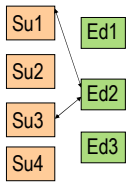


¿En cuál es más fácil añadir nuevo consumidor de datos?

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"
- Posible uso de comunicación de grupo en cualquiera de ellas
  - Ej. Comunicación directa + comunicación de grupo
    - Ed/Su basado en temas: tema = dirección de grupo

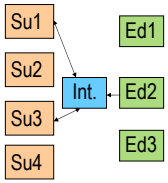
Implementación ed/su sin intermediario



Contacto directo ed./ suscr.

↓ Acoplamiento espacial

Implementación ed/su con intermediario

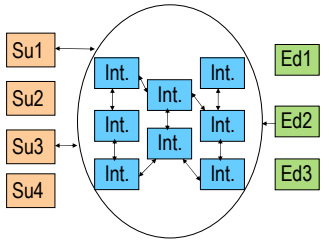


Proceso intermediario

↑ Desacoplamiento espacial

↓ Cuello botella y punto fallo

Implementación ed/su con red interm.



Red de intermediarios

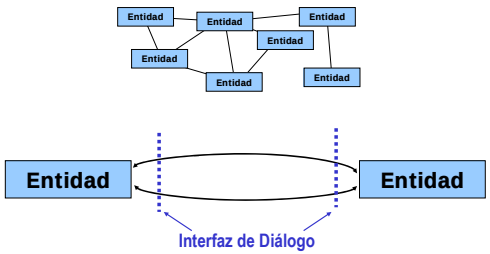
↑ Desacoplamiento espacial

↑ Escalabilidad y fiabilidad

Modelo *Peer-to-Peer (P2P)*

- Todos los nodos tienen mismo rol y funcionalidad (*servents*)
  - No hay cuellos de botella ni puntos críticos de fallo
  - Se aprovechan recursos de todas las máquinas
- Se suelen caracterizar además por:
  - Volatilidad: Nodos entran y salen del SD; variabilidad en conectividad
  - Capacidad de autogestión sin una autoridad global centralizada
- Uso de red superpuesta (*overlay*):
  - Red lógica sobre la física
  - Nodos de proceso como nodos de red
  - Gestionan esquemas de encaminamiento y localización de recursos
- Desventajas de arquitectura P2P
  - Difícil administración conjunta y mayores problemas de seguridad

Esquema *Peer-to-Peer (P2P)*



Aplicaciones de *P2P*

- **Compartir contenidos** (*P2P content-sharing*)
  - Diversos aspectos no tratados: *copyright*, anonimato, *free-riding*, …
- Distribución de contenidos (1 editor; múltiples descargadores)
- Almacenamiento distribuido
  - Sistemas de ficheros P2P
  - Cachés P2P
- Comunicación
  - Mensajería instantánea
  - VoIP
  - Videoconferencia
- *Streaming* (P2PTV)
- Computación distribuida …

Operaciones en *P2P content-sharing*

- **Alta (Baja)** de nodo en la red P2P
- **Publicar (Eliminar)** contenido
  - ¿Cómo identificar (ID) un determinado recurso?
    - Nombre de fichero usando alguna convención (p.ej. *artista + canción*)
    - *hash* basado en contenido para detectar recursos repetidos
  - Publicación: [ID + contenido + metadatos (p.e. estilo musical)]
- **Buscar** contenido
  - Por ID o por metadatos (con posible uso de comodines)
- **Descargar** contenido
  - De un único nodo o de múltiples

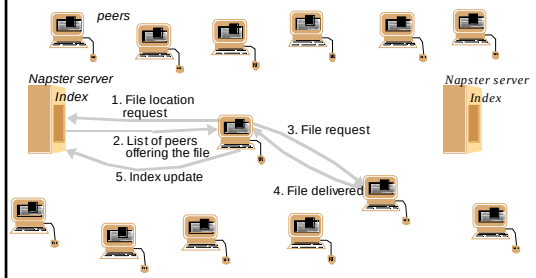
Tipos de sistemas P2P

- Desestructurados:
  - Topología de conexión lógica arbitraria
  - Ubicación de recursos impredecible e independiente de la topología
    - Cada nodo posee un conjunto de recursos
  - Corresponden a sistemas +volátiles con nodos más autónomos
  - Tipo de sistemas P2P desestructurados:
    - Centralizados, Descentralizados y Parcialmente centralizados
- Estructurados:
  - Topología de conexión prefijada (p.e. anillo en protocolo *Chord*)
  - Ubicación de recursos predecible y dependiente de la topología
    - *lookup*(ID recurso) → máquina que posee recurso
  - Corresponden a sistemas -volátiles con nodos más cooperativos
    - Posesión de recursos cambia según sistema evoluciona

Sistemas P2P centralizados

- P2P + Cliente/servidor
  - Uso de servidor centralizado que conoce nodos y recursos
  - Simple y fácil de gestionar
  - Cuello de botella y punto único de fallo (puede estar replicado)
- Uso para compartir contenidos (p.e. Napster: 1999-2001)
  - **Alta**: cliente C contacta con servidor S
  - **Publicar**: en alta C informa a S de su lista de ficheros publicados
  - **Buscar**: C envía petición a S y recibe lista de clientes con ese fichero
    - En Napster: (*artista + canción*) que se busca en nombre de fichero
  - **Descargar**: C elige cliente con mejor t. respuesta y pide descarga
    - C informa a S de que ya posee copia de fichero
      - Se lo podrán pedir otros clientes a partir de ahora
    - No hay descarga en paralelo de múltiples clientes

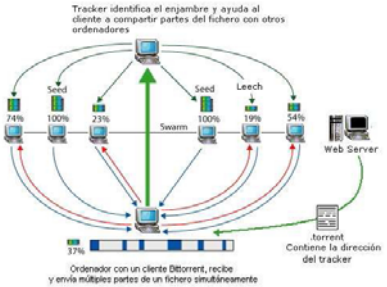
Napster (Libro de Couloris et al.)



Protocolo *BitTorrent* (2001)

- Distribución de contenidos mediante P2P centralizado
  - Se centra en descarga; búsqueda externa al protocolo (p.e. Google)
  - Descarga paralela de trozos del fichero de clientes que los poseen
  - Servidor (*tracker*) conoce qué clientes tienen trozos del fichero
  - Fichero *.torrent*: información del fichero y del *tracker*
- Operaciones:
  - **Publicar**: Cliente inicial (*initial seeder*) crea *.torrent*
    - Puede especificar un *tracker* propio o público
  - **Buscar**: Cliente C encuentra de alguna forma *.torrent* del recurso
  - **Alta**: C contacta con *tracker* → lista de algunos clientes con trozos
    - C contacta con esos clientes que le dicen qué trozos tienen
  - **Descargar**: C descarga en paralelo trozos de clientes
    - Primero los "más raros"
    - Cuando completa descarga de un trozo, lo notifica a otros clientes
    - C envía trozos a clientes que se los solicitan

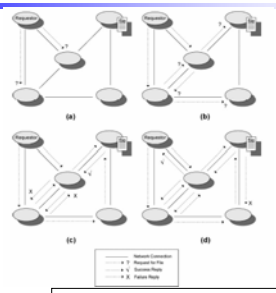
Protocolo *BitTorrent (wikipedia)*



Sistemas P2P descentralizados

- Sistemas P2P puros: no punto único de fallo ni cuello botella
  - Todos los nodos con la misma funcionalidad
- Red superpuesta de topología arbitraria
  - Nodos intercambian conocimiento de otros nodos "vecinos"
  - Localización de recursos por "inundación"
- Uso para compartir contenidos (p.e. Gnutella: 2000)
  - **Alta**: nodo N obtiene de alguna forma direcciones de nodos de red
    - A partir de entonces intercambian periódicamente info. de "vecinos"
    - Permite mantenimiento de red lógica aunque implica sobrecarga
  - **Publicar**: NOP
  - **Buscar**: N propaga petición a vecinos y éstos a los suyos…
    - Uso de TTL para limitar propagación pero impide encontrar recursos
  - **Descargar**: N pide recurso al nodo que lo posee
- Problemas de rendimiento y escalabilidad

Búsqueda en P2P descentralizados

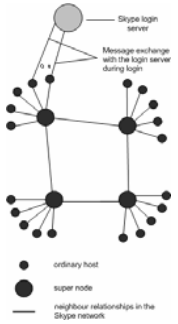


A Survey of Peer-to-Peer Content Distribution Technologies
S. Androutsellis-Theotokis y D. Spinellis; ACM Computing Surveys, 2004

Sistemas P2P parcialmente centralizados

- Sistema jerárquico: nodos ordinarios ON y supernodos SN
  - Nodo ordinario debe estar asociado a un supernodo
  - Supernodo tiene información de grupo de nodos asociados
  - Supernodos dinámicamente elegidos por potencia, conectividad,…
- Mezcla de descentralizado y centralizado
  - Red descentralizada de supernodos (como Gnutella)
  - Cada SN servidor de ONs asociados al mismo (como Napster)
- Uso para compartir contenidos (p.e. Kazaa/FastTrack: 2001)
  - **Alta**: nodo N obtiene dirección de algún supernodo SN
    - En algún momento N puede llegar a ser SN
  - **Publicar**: en alta N informa a SN de su lista de ficheros publicados
  - **Buscar**: N petición a su SN y éste la propaga sólo por SNs (con TTL)
    - N recibe lista de nodos con ese fichero
  - **Descargar**: N pide recurso a nodo que lo posee
    - Extensión: descarga simultánea desde múltiples nodos

Skype



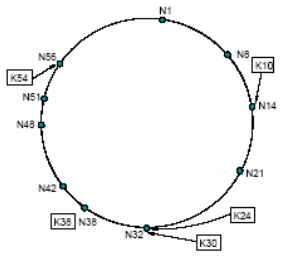
Sistemas P2P estructurados

- Sistema descentralizado con búsquedas O(log n) en vez O(n)
  - Protocolos *Chord* (Stoica et al. MIT 2001), *CAN*, *Tapestry* y *Pastry*
  - Red superpuesta de topología prefijada
- ¿En qué nodo se almacena un recurso? → función *hash*
  - "Tabla *hash* distribuida" (DHT: *Distributed Hash Table*)
    - lookup(ID recurso) → ID nodo que posee recurso
  - Esquema estable y escalable ante nº enorme y dinámico de nodos
- Uso para compartir contenidos
  - **Alta**: nodo N obtiene dirección de algún nodo y se incorpora a red
  - **Publicar**: N aplica la función de *lookup* para ubicarlo
  - **Buscar**: N aplica la función de *lookup* para encontrarlo
  - **Descargar**: No definida por el protocolo
    - Nodo "poseedor" puede tener el recurso o una referencia al mismo

Protocolo *Chord*

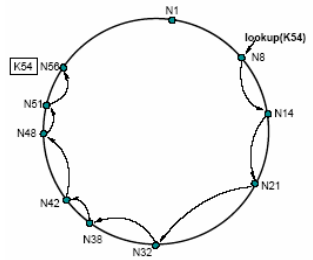
- *Hashing* "consistente" asigna ID a clave recurso y a IP de nodo
  - ID con *m* bits tal que nº recursos (K) + nº nodos (N) << 2^m
  - IDs organizados en anillo módulo 2^m
  - Proporciona distribución uniforme
- Asignación de recurso *K* a nodo *N*
  - 1er nodo ID(*N*) ≥ ID(*K*) → sucesor(*K*) (NOTA: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

Anillo 2^6 con 10 nodos y 5 recursos



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
Ion Stoica et al.; ACM SIGCOMM'01

Búsqueda simple lineal

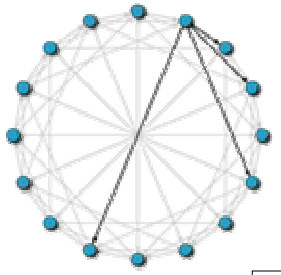


Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
Ion Stoica et al.; ACM SIGCOMM'01

Búsqueda basada en *fingers*

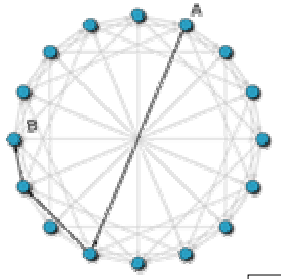
- Cada nodo con ID *n* incluye tabla *fingers TF* con *m* entradas:
  - Entrada *i*: primer nodo a una distancia ≥ 2^{i-1}
    - *sucesor*$(n + 2^{i-1})$ tal que $1 \le i \le m$
    - Entrada 0: sucesor directo
- Búsqueda de *K* desde nodo *n*:
  - Si $ID \in (n, s$: sucesor directo de $n] \rightarrow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

Fingers en anillo 2^4



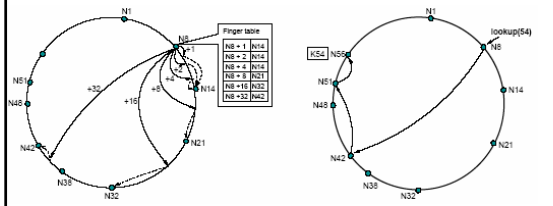
Wikipedia Chord

Búsqueda en anillo 2^4



Wikipedia Chord

Búsqueda con *fingers*



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
Ion Stoica et al.; ACM SIGCOMM'01

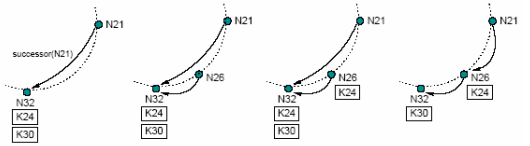
Mantenimiento del anillo

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

Alta de un nodo

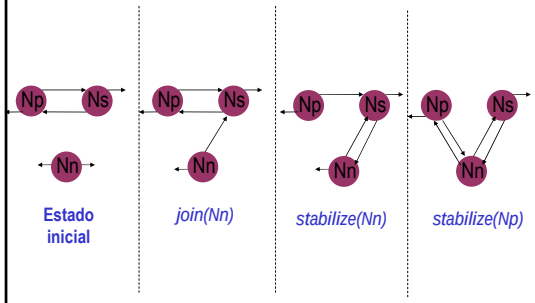
- Operación *join* de nodo *N*:
  - Conoce de alguna forma dir. de cualquier nodo existente *N'*
  - *N* calcula su ID y pide a *N'* búsqueda de su sucesor
  - *N* anota su sucesor (por ahora predecesor = NULO)
- Operación periódica en cada nodo *stabilize*:
  - Pregunta a su sucesor *S* por su predecesor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

Alta de un nodo



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
Ion Stoica et al.; ACM SIGCOMM'01

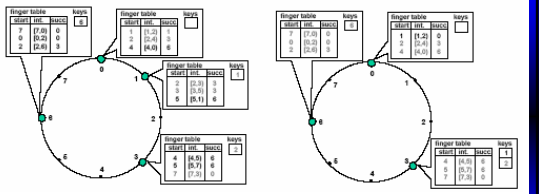
Alta de un nodo: paso a paso



Baja de un nodo

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

Tablas *finger* después de alta y baja



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
Ion Stoica et al.; ACM SIGCOMM'01