Sistemas Distribuidos

Telecomunicaciones y Sistemas Distribuidos
Licenciatura en Ciencias de la Computación

Marcelo Arroyo
Departamento de Computación
Universidad Nacional de Río Cuarto

2017

Contenidos

Sistemas Distribuidos

Definición

Conjunto de procesos que resuelven un problema en forma cooperativa.

Características

- No tienen un reloj físico común.
- No comparten memoria.
- Pueden estar dispersos geográficamente.
- Los sistemas pueden ser heterogénos.
- Los sistemas son autónomos.
- Comunicación: mediante mensajes

Objetivos

- Solución de problemas inherentemente distribuidos (ej: transacciones bancarias).
- Compartir recursos.
- Mejorar la confiabilidad.
- Escalabilidad.
- Modularidad y extensibilidad.
- Aprovechamiento de recursos en sistemas con múltiples unidades de ejecución (multiprocesadores, GPUs, . . .)

Hardware: Sistemas paralelos

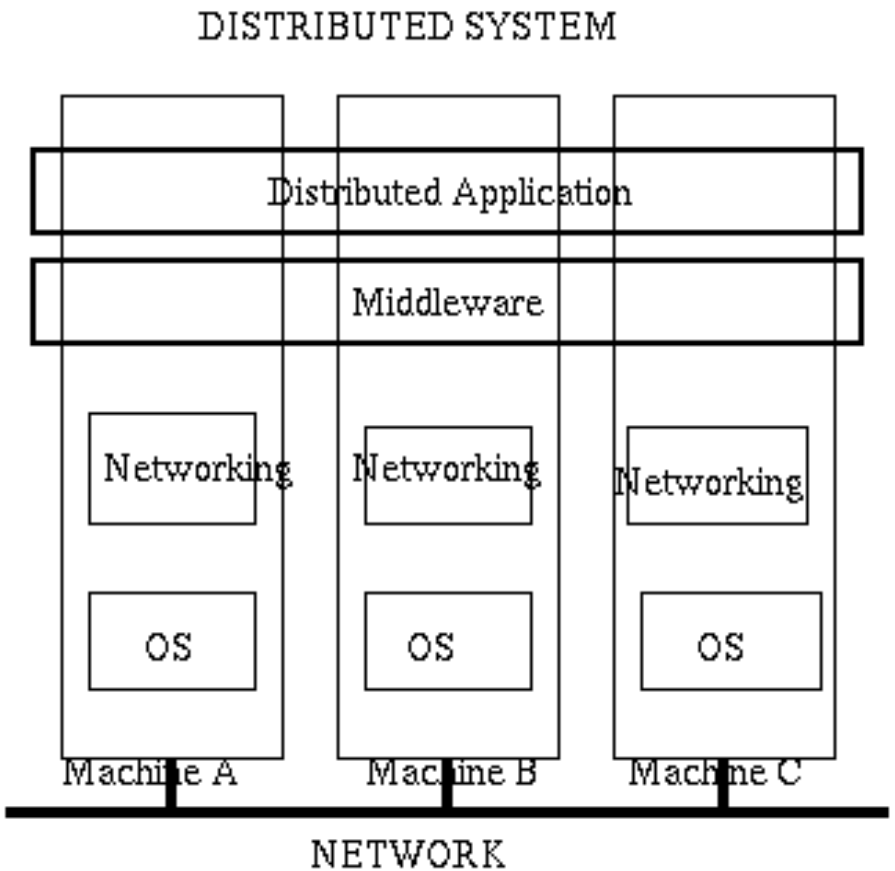
Clasificación (Flynn)

- Single Instruction, Multiple Data (SIMD)
  - Ej: Array processors (como GPUs)
  - Componentes fuertemente acoplados (buses de alta velocidad)
- Multiple Instruction, Multiple Data (MIMD)
  - Sistemas multiprocesadores y multicomputadoras.

Interconexión

- Memoria compartida:
  - Jerarquías de memoria. Acceso: Uniforme (UMA) vs no uniforme (NUMA)
- Sistema de pasaje de mensajes.

Arquitectura



Middleware

Definición

Capara software (herramientas, bibliotecas, . . .) que permiten homogeneizar las comunicaciones y provee una visión uniforme de sistema.

Tipos

- Frameworks como Common Object Request Broker Architecture (CORBA), .NET, J2EE, . . .
- Mecanismo de Remote Procedure Call (RPC): Sun-RPC, RMI, DCOM, D-Bus, XML-RPC, SOAP, . . .
- Servicios de pasajes de mensajes y topologías virtuales: TCP/IP, Message Passing Interface (MPI), . . .

Problemas

- Algoritmos sobre grafos distribuidos (dinámicos, ruteo, . . .)
- Obtención de estado global
- Mecanismos de sincronización y coordinación:
  - Sincronización de relojes físicos (ej: NTP)
  - Elección de coordinador (leader)
  - Exclusión mutua.
  - Detección de deadlock y terminación.
  - Recolección de basura.
- Replicación y distribución de datos. Modelos de consistencia.
- Sistemas tolerantes a fallas.

Relojes lógicos

Causalidad: Relación de precedencia entre eventos en un sistema

- En un sistema con reloj (físico) común es simple.
- En un sistema distribuido se deberá obtener una aproximación.

Sistema de relojes lógicos

- T: dominio de tiempo. H: conjunto de eventos.
  Reloj $C : H \to T$.
- $C(e)$ (timestamp de e): Deberá cumplir que
  1. Consistencia: para cada e_i, e_j,
     $e_i \to e_j \implies C(e_i) < C(e_j)$
  2. $e_i \to e_j$ significa que e_i ocurrión antes que e_j

Relojes lógicos (Lamport clocks)

Relojes escalares

- Cada proceso p_i tiene un reloj $C_i : int$.
- Algoritmo: Cada proceso p_i hace:
  1. Antes de cada evento: $C_i += +d$ $(d > 0)$.
  2. En cada mensaje enviado m_{ij} (a p_j), se anexa C_i (timestamp).
  3. En cada mensaje $m_{ji} = (msg, C_j)$ (recibido de p_j),
     $C_i = max(C_i, C_j)$.
     $C_i += C_i$

Relojes lógicos (Lamport clocks)

Relojes escalares

- Cada proceso p_i tiene un reloj $C_i : int$.
- Algoritmo: Cada proceso p_i hace:
  1. Antes de cada evento: $C_i += +d$ $(d > 0)$.
  2. En cada mensaje enviado m_{ij} (a p_j), se anexa C_i (timestamp).
  3. En cada mensaje $m_{ji} = (msg, C_j)$ (recibido de p_j),
     $C_i = max(C_i, C_j)$.
     $C_i += C_i$

Causalidad fuerte

- Notar que no es posible lograr un orden total de eventos.
- Para dos eventos e_i, e_j, $C(e_i) < C(e_j) \implies e_i \rightarrow e_j$

Relojes lógicos

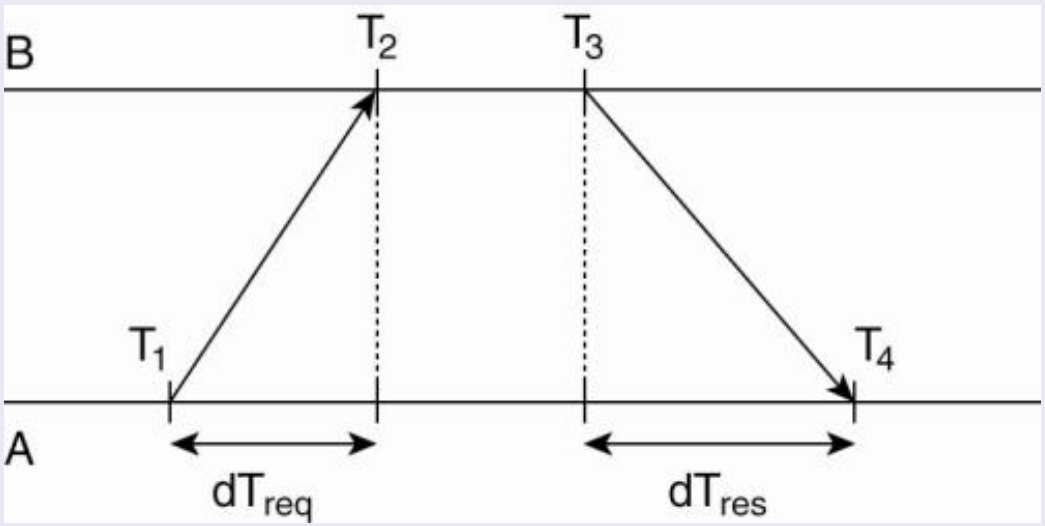
Relojes vectoriales (vector clocks)

- Objetivo: lograr causalidad fuerte.
- $C_i = vc_i[1..n]$. $vc_i[j]$ = último valor conocido (en p_i) del reloj de p_j.
- Algoritmo: Cada proceso p_i hace:
  1. Antes de cada evento interno $vc_i[i] += d$ $(d > 0)$.
  2. En cada mensaje enviado m_{ij} (a p_j), se anexa vc_i (timestamp).
  3. Por cada mensaje $m_{ji} = (msg, vc_j)$ (recibido de p_j), $vc_i[k] = max(vc_i[k], vc_j[k])$, $\forall k$ $(1 \leq k \leq n)$, luego $vc_i[i] += d$

Sincronización de relojes físicos

Network Time Protocol (NTP)

- Un nodo (cliente) `A` consulta la hora a un servidor `B`.
- Problema: La respuesta no será exacta por la demora en la comunicación
- Solución: Estimar esa demora y ajustar



Sincronización de relojes físicos

Network Time Protocol (NTP)

- Objetivo: actualizar periódicamente relojes de nodos de una red.
- NTP usa el método *Offset-Delay Estimation*
  1. Jerarquía de servidores (top: sincroniza con UTC)
  2. La hora local de A: $A(t) = B(t) + \theta$
  3. θ (offset) y δ (delay) de `A` con `B`:
     ($T_i, 1leqi \leq 4$: *timestamps* de mensajes)

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

  4. Cada servidor mantiene los últimos pares (θ, δ) con cada peer.
  5. Se elige θ con menor δ

Exclusión mutua

Acceso exclusivo a recursos: Mecanismos

- **Región crítica (RC)**: Trozo de programa delimitado por *ENTER*(*cs*) y *RELEASE*(*cs*) con acceso exclusivo a recursos.
- En modelos de memoria compartida: locks, semáforos, regiones críticas condicionales, monitores, ...

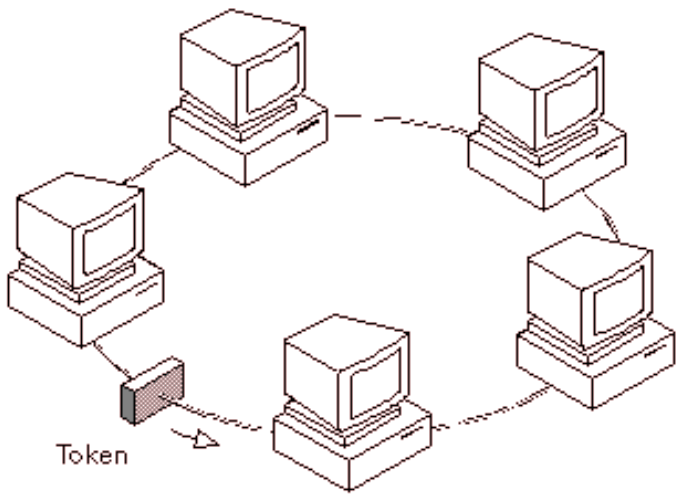
Propiedades de un algoritmo de exclusión mutua

- `Safety`: Sólo un proceso puede estar ejecutando dentro de la RC en un momento dado.
- `Liveness`: Ausencia de deadlock o inanición (un proceso no debería esperar por un mensaje que nunca será enviado).
- `Fairness`: Cada proceso tendrá la chance de ejecutar la RC.

Exclusión mutua

Algoritmo basado en token (topología: anillo)

- Cada proceso que require acceder a un conjunto de recursos (región crítica), debe esperar el *token*.
- Cuando le llega el token, le toca ingresar a la región crítica.
- Al salir de la región crítica (libera los recursos), reenvía el token.



Exclusión mutua

Algoritmo de Lamport

- Cada proceso p_i que mantiene una cola $requests_i$.
  1. Cuando p_i requiere entrar en la región crítica (RC):
     1. Broadcast y encola $(ENTER, i, ts_i)$ (ts_i: timestamp).
     2. p_j recibe $(ENTER, i, ts_i)$, lo encola y responde $(REPLY, j, ts_j)$ a p_i.
  2. p_i puede entrar a la RC cuando:
     1. p_i ha recibido mensajes con $timestamp > ts_i$ de todos los otros procesos.
     2. p_i tiene $(ENTER, i, ts_i)$ primero en la cola.
  3. p_i al salir de la RC hace:
     1. Saca $(ENTER, i, ts_i)$ de la cola y envía (broadcast) $RELEASE(i, ts_i)$
     2. Cuando p_j recibe $RELEASE(i, ts_i)$ remueve $(ENTER, i, ts)$ de su cola.

Exclusión mutua (cont.)

Algoritmo de Ricart-Agrawala

- Cada proceso p_i tiene un arreglo *bool* $RD_i[N]$ (N procesos)
  1. Cuando p_i requiere entrar en la región crítica (RC):
     - Broadcast $(REQUEST, i, ts_i)$ (ts_i: timestamp).
     - p_j recibe $(REQUEST, i, ts_i)$:
       - Responde $(REPLY, j, ts_j)$ a p_i si no está en la RC o si p_j ha hecho $(REQUEST, j, ts'_j)$ y $ts_i < ts'_j$.
       - Sino, $RD_j[i] := 1$ (se difiere la respuesta)
  2. p_i puede entrar a la RC cuando:
     1. p_i ha recibido $(REPLY, j, ts_j)$, $\forall 1 \leq j \leq n \wedge j \neq i$
  3. p_i al salir de la RC hace:
     1. Envía $(REPLY, j, ts_i)$ $(\forall j \mid RD_i[j] = 1)$; $RD_i = [0]$

Exclusión mutua (cont.)

Algoritmo de Ricart-Agrawala

- Cada proceso p_i tiene un arreglo *bool* $RD_i[N]$ (N procesos)
  1. Cuando p_i requiere entrar en la región crítica (RC):
     - Broadcast $(REQUEST, i, ts_i)$ (ts_i: timestamp).
     - p_j recibe $(REQUEST, i, ts_i)$:
       - Responde $(REPLY, j, ts_j)$ a p_i si no está en la RC o si p_j ha hecho $(REQUEST, j, ts'_j)$ y $ts_i < ts'_j$.
       - Sino, $RD_j[i] := 1$ (se difiere la respuesta)
  2. p_i puede entrar a la RC cuando:
     1. p_i ha recibido $(REPLY, j, ts_j)$, $\forall 1 \leq j \leq n \wedge j \neq i$
  3. p_i al salir de la RC hace:
     1. Envía $(REPLY, j, ts_i)$ $(\forall j \mid RD_i[j] = 1)$; $RD_i = [0]$

Comparación con el algoritmo de Lamport

$2(N-1)$ mensajes vs $3(N-1)$ mensajes (Lamport)

Obtención de estado global

Ejemplo: transacción bancaria distribuida

1. Suponga dos cuentas bancarias (A=600 y B=200) y los siguientes eventos:
2. Un proceso en A transfiere 50 de A a B: $A = A - 50$, $sendto(B, (CREDIT, 50))$.
3. Antes que el crédito arribe a B, éste transfiere a A 80 por el canal C_{BA}.
4. A recibe el crédito de B: A=630, B=120.
5. B recibe el crédito de A: A=630, B=170.

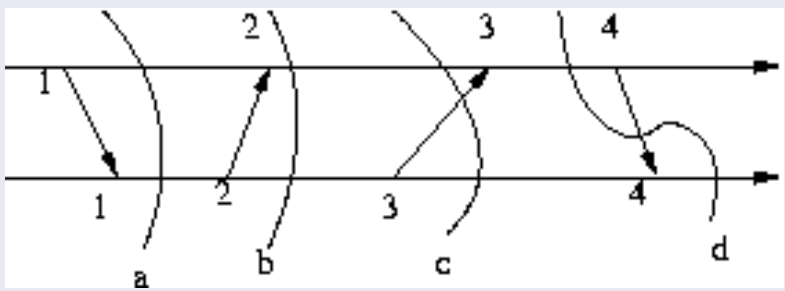
Calculando estado global

- El estado global desde estados locales en los tiempos 1 y 3: $A_{t_1} + B_{t_3} + C_{ABt_3} + C_{BAt_3} = 850$!!!inconsistente

Estado global consistente

Cortes

- Un *corte* es una partición de eventos: $cut = PAST \cup FUTURE$, $PAST \cap FUTURE = \emptyset$.
- Un *corte es consistente* si cada $recv(m) \in PAST$ y $send(m) \in PAST$ de ese corte. (Ej: a y b de la figura).
- Todos los eventos tales que $send(m) \in PAST$ y $recv(m) \in FUTURE$, se consideran en el canal correspondiente (Ej: corte c).



Estado global consistente

Toma de instantáneas

- ¿Cómo distiguir qué mensajes deben incluirse en la instantánea global (estado de canales + estado del proceso)?
  - Cada mensaje enviado por un proceso antes de tomar su instantánea local
- ¿Cuándo un proceso debe tomar su instantánea (local)?
  - p_j debe tomar su instantánea antes de procesar un mensaje m_{ij} que fue enviado por p_i luego de tomar su instantánea.

Estado global consistente

- El estado global del sistema con n procesos: $GS = \cup LS_i$, $(1 \leq i \leq n)$
- Un *estado global es consistente* si:
  1. $send(m_{ij}) \in LS_i \Rightarrow$

Algoritmos de tomas de instantáneas (snapshot)

Algoritmo de Chandy-Lamport (canales FIFO)

- Iniciador de toma de instantánea (proceso p).
  1. Registra su estado y el de cada canal c saliente.
  2. $send(c, MARK)$, para cada canal c de salida.
  3. La toma termina al recibir $MARK$ en cada canal de p
- Proceso q recibe $MARK$ por el canal c_i
  - Si q no ha registrado su estado:
    1. Registra estado; $state(c_i) = \emptyset$ ($\forall c_i$ saliente)
    2. $send(c, MARK)$ por cada canal saliente c
  - Sino:
    1. $state(c_i) = \{m \mid timestamp(m) > time(last\_snapshot)\}$
- Cada proceso (q) puede enviar su estado local al iniciador (p)
- Varios procesos pueden iniciar la toma concurrentemente.

Detección de terminación

Consideraciones

- Un proceso puede estar *idle* o *active*.
- Un proceso *idle* se torna activo al recibir un mensaje.
- Terminación: sea $p_i(t) = \{idle, active\}$, $C_{ij}(t) = \{m_{ij}\}$ mensajes en tránsito de p_i a p_j en el tiempo t.
  $terminated = (\forall i \mid p_i(t) = idle) \wedge \forall i, j \mid C_{ij}(t) = \emptyset$
- Terminación es una propiedad *estable*.

Detección de terminación

Algoritmo basado en token-ring (Dijkstra)

- Cada proceso p_i tiene un color c_i, inicialmente *white* y un contador m_i.
- Cuando p_i envía un mensaje: $m_i = m_i + 1$. Cuando recibe: $m_i = m_i - 1$
- Token=$(color, m)$. $color : \{white, black\}$, $m = 0..n$.
- Cuando p_i se torna *activo*, $c_i = black$.
- p_0 (*idle*) inicia una ronda enviando $(white, 0)$ a p_{n-1}.
- p_i recibe el token (c, m). Luego, al hacerse *idle*: Si $c_i = black$ reenvía $(black, m + 1)$, $c_i = white$. Sino reenvía $(c, m + 1)$
- Cuando p_0 recibe (c, m). Envía *FINISH* si:
  $p_0 = idle \wedge c_0 = white \wedge c = white \wedge m = 0$
  Sino, se deberá luego iniciar otra ronda.

Detección de terminación

Algoritmo basado en instantáneas

1. Cada p_i mantiene c_i (reloj) y $k_i = max(k', c)$ de los mensajes $R(k', c)$ enviados o recibidos.
2. Un proceso p_i que recibe $M(msg, c_j)$, hace $c_i = c_j + 1$ y $state(p_i) = active$
3. Cuando p_i se torna *idle*:
   $c_i = c_i + 1$, $k_i = (i, c_i)$ y envía (broadcast) $R(i, c_i)$ y recolecta instantáneas.
4. Cuando p_i recibe $R(k', c_j)$:
   - Si $(p_i = idle) \wedge ((k', c_j) > k_i) \rightarrow k_i := (k', c_j)$ y envía instantánea.
   - Si $p_i = active \rightarrow c_i = max(c_i, c_j)$

$$(k, c) > (k', c') \text{ si y sólo si } c > c' \vee (c = c' \wedge k > k')$$

Elección

Algoritmo basado en anillo

- Usos: Elección de coordinador
- Se elige el proceso con identificador mayor

1. p_i (el *iniciador*) envía *election* i a $(p_i + 1)$ *mod* n
2. Cuando p_j recibe el mensaje *election* i:
   - si $i > j$, reenvía el mensaje a su vecino
   - si $i < j$, envía *election* j
   - si $i = j$, el proceso p_j es el elegido y envía *elected* i
3. Cada proceso que recibe *elected* i define a p_i como el coordinador y reenvía el mensaje a su vecino
4. Cuando el proceso p_k recibe *elected* k lo descarta

- Requiere $3N - 1$ mensajes

Elección (cont.)

Algoritmo *bully* (García-Molina, 1982)

Cada proceso conoce a los otros con mayor identificador.

1. p_i (iniciador) envía *election* a cada p_j, con $j > i$
2. Cuando p_j recibe el mensaje *election*, inicia la elección
3. p_i espera por mensajes de respuesta *answer*
4. Si ninguno arriba en un tiempo T, p_i se asume como el elejido y envía *coordinator* i a los procesos p_k con $k < i$
5. En otro caso, espera un tiempo T adicional a la espera de mensajes *coordinator*. Si no recibe ninguno, re-inicia la elección.
6. Si un proceso p_i recibe un mensaje *coordinator* j, define a p_j como coordinador y lo reenvía a los procesos p_k con $k < i$

Transacciones distribuidas

Transacción atómica

- Una transacción es una *secuencia de operaciones* que debe cumplir con:
  1. **Atomicidad**: Todo o nada (*commit/rollback*)
  2. **C***onsistencia*: El sistema debe quedar siempre en un estado consistente
  3. *AI**slamiento***: No puede tener interferencias de otras operaciones
  4. **D***urabilidad*: Los cambios deben ser persistentes
- Requiere los siguientes mecanismos:
  1. Tener una memoria *transaccional* (*log* o *journal*)
  2. Un *commit* mueve las operaciones del *log* a la memoria no volátil

Commit protocols

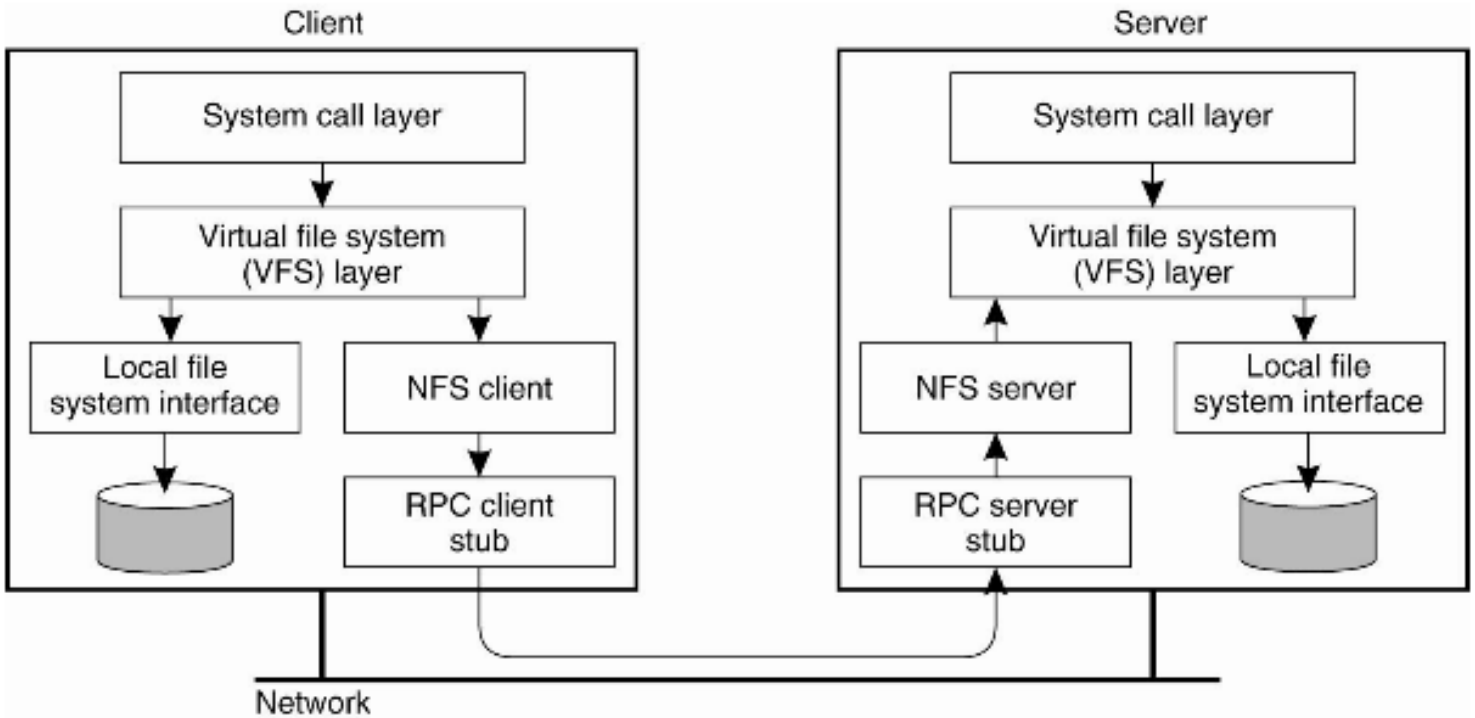
Two-Phase Commit

- Existe un *cliente*, *coordinador* y *participantes*
- Los participantes ejecutan las operaciones
- El *cliente consulta* al coordinador por *commit*

1. Fase de votación:
   1. El *coordinador* envía a los *participantes canCommit*
   2. Los *participantes* envían *yes* o *no*
2. Comunicación:
   1. Si todas las respuestas son *yes*, envía a los *participantes commit*, sino envía *rollback*
   2. Cuando los *participantes* reciben *commit*, confirman sus operaciones y envían *commited* al coordinador, sino deshacen sus operaciones (*roolback*)

Network File System (NFS)

1. Desarrollado por Sun Microsystems en 1985.
2. Es un estándar de Internet (RFC 1813)



NFS: Características

- *Virtual File System (VFS)*: Permite que sea implementable en cualquier plataforma (UNIX, MS-Windows, . . .)
- *Transparencia* de localización (*mounting service*) y de *acceso*
- *Replicación*: Pueden ser fácilmente replicados archivos de sólo lectura
- *Tolerancia a fallas*: Al ser *stateless* soporta fallas
- *Consistencia*: Soporta la semántica *one-copy*
- *Seguridad*: Puede usarse con *Kerberos*
- *Eficiencia*: Puede usarse razonablemente en Internet

Peer to Peer

Características

- Cada nodo comparte recursos con las mismas capacidades y responsabilidades
- No dependen de un servicio centralizado
- Proveen algún grado de anonimato a los participantes
- Acceso transparente a recursos distribuidos

Peer to Peer

Características

- Cada nodo comparte recursos con las mismas capacidades y responsabilidades
- No dependen de un servicio centralizado
- Proveen algún grado de anonimato a los participantes
- Acceso transparente a recursos distribuidos

Algunos ejemplos. . .

- Napster: Sistema de archivos distribuido de música (*índice centralizado*)
- BitTorrent: Sistema de archivos compartido (distribuidos en chunks). Uso de *trackers* con *metadata* de archivos.
- Skype: Video-conferencia/chat/file-sharing

BitTorrent: Arquitectura

- Un archivo `.torrent` contiene información de:
  1. Los archivos que contiene
  2. La *dirección* de al menos un *tracker*
- *Tracker*: Servidor que mantiene una lista de nodos participantes
- Cada nodo (*peer*), se registra en uno o más *trackers* anunciando sus recursos (*chunks* o partes de archivos)
- Cada *peer* solicita a un tracker las IPs de otros *peers* para descargar *chunks* de archivos en forma aleatoria y concurrente.
- Un *seed* es un *peer* que contiene un archivo completo. Este es necesario para comenzar a replicar un archivo.
- Un *peer* que descargó un archivo completo se convierte en *seed*

Skype: Arquitectura