

4 Sistemas de ficheros distribuidos

4.1 Introducción

4.1.1 Propiedades de los sistemas de ficheros distribuidos

4.1.2 Caracterización del uso de los ficheros

4.2 Modelo

4.2.1 Estructura

4.2.2 Identificación de ficheros

4.3 Servidores de nombres

4.4 Servidores de ficheros

4.4.1 Semánticas de compartición

4.4.2 Tipos de servidores

4.4.3 Caching y gestión de la consistencia

4.5 Ejemplos

4.5.1 NFS (Network File System)

4.5.2 AFS (Andrew File System)

4.6 Ejercicios

4.1 Introducción

Los sistemas de ficheros y bases de datos son quizás los primeros recursos candidatos a distribuirse y compartirse en un sistema en red. De hecho, existen productos comerciales desde hace mucho tiempo para redes locales, como es el caso de NFS. Más actualmente se han desarrollado también sistemas de ficheros distribuidos sobre redes de área amplia.

La gestión de un sistema de ficheros distribuido se soporta mediante dos funciones a menudo bien diferenciadas: un **servicio de nombres** o directorios y un **servicio de ficheros**. Es fundamental proporcionar un rendimiento aceptable, por lo que hay que alcanzar un compromiso entre la disponibilidad local de la información para disminuir los costes de comunicación (*キャッシング* y **distribución**) y la consistencia, que se refleja en la semántica que muestran los accesos compartidos.

4.1.1 Propiedades de los sistemas de ficheros distribuidos

Un sistema de ficheros se caracteriza por un conjunto de propiedades generales:

- proporciona almacenamiento de información permanente;
- identifica los ficheros en un espacio de nombres (normalmente estructurado);
- es posible el acceso concurrente desde varios procesos;
- en sistemas multiusuario proporciona protección de accesos.

Un sistema de ficheros distribuido tiene también como objetivos las siguientes propiedades:

- Transparencia en la identificación. Espacio de nombres único e independiente del cliente.
- Transparencia en la ubicación. Para permitir la movilidad del fichero de una ubicación a otra, se requiere una correspondencia dinámica nombre-ubicación.
- Escalabilidad. Espacios de nombres estructurados, y replicación (caching) para evitar cuellos de botella.
- Robustez ante fallos. El servidor no debe verse afectado por los fallos de los clientes, lo que incumbe a la gestión del estado de los clientes en el servidor. Por otra parte, la interfaz ofrecida a los clientes debe proporcionar en lo posible operaciones *idempotentes*¹, que garanticen la corrección ante invocaciones repetidas (por sospecha de error) al servidor.

¹ Una función f es idempotente cuando $f(f(x)) = f(x)$.

- Disponibilidad y tolerancia a fallos. Implican alguna forma de replicación. Un aspecto de la disponibilidad es permitir el funcionamiento en *modo desconectado*, que requiere caching de ficheros enteros.
- Consistencia. El objetivo es mantener en lo posible la semántica de los sistemas centralizados, por ejemplo preservar la semántica UNIX en presencia de caching u otras formas de replicación.
- Seguridad. La necesidad de autenticación remota implica nuevos modelos de protección, basados en credenciales en lugar de listas de accesos.

4.1.2 Caracterización del uso de los ficheros

El diseño de un sistema de ficheros distribuido que proporcione un buen nivel de rendimiento deberá basarse en las características de uso de los ficheros por las aplicaciones. Aunque no es fácil generalizar, sí es posible determinar una serie de patrones de comportamiento (Satyanarayanan, 1981):

- La mayoría de los ficheros son de pequeño tamaño. Esto implica que el fichero puede ser la unidad de recuperación.
- La escritura es poco frecuente. Esto alienta el caching y la replicación.
- La compartición es poco frecuente. La mayoría de los ficheros se acceden por un lector y/o un escritor, algunos se acceden por n lectores y un escritor, y muy rara vez se acceden por n lectores y m escritores². Por lo tanto, puede ser rentable una gestión *optimista* del caching y la replicación, que presuponga que hay un único escritor y rectifique en caso de detectar a posteriori escrituras simultáneas.
- El ratio búsqueda/uso suele ser bajo³. Este hecho también favorece el caching.
- El acceso suele ser secuencial y existe un alto grado de localidad. Esto promueve el *buffering* para proporcionar anticipación en los accesos.
- La mayoría de los ficheros tienen una vida muy corta (por ejemplo, ficheros temporales). Hay que tender a gestionarlos localmente.
- Existen clases de ficheros, con propiedades diferenciadas (por ejemplo ejecutables, que rara vez se modifican).

² De hecho ni siquiera la semántica UNIX en sistemas centralizados gestiona la sincronización entre varios escritores, pasando la responsabilidad a las aplicaciones.

³ Esta relación es una medida de la tasa de accesos al fichero en proporción a la cantidad de proceso que se realiza sobre el elemento accedido. En las bases de datos este ratio es elevado porque las aplicaciones suelen acceder a una gran cantidad de elementos y en la mayoría de los casos para una simple consulta. En cambio, en ficheros es habitual realizar un proceso más o menos costoso para cada elemento accedido.

4.2 Modelo

El modelo de sistema de ficheros distribuido que vamos a definir aquí [COU05 §8] considera una estructura cliente-servidor que incluye servicios diferenciados de nombres y de ficheros, y un mecanismo de identificación única de los ficheros por los clientes.

4.2.1 Estructura

La estructura del modelo de sistema de ficheros distribuido que estamos presentando consta de tres módulos:

Cliente

Es la interfaz local con la aplicación. Interpreta las llamadas al sistema sobre ficheros y genera las peticiones (habitualmente RPCs) para los accesos remotos. Conoce la ubicación de los servicios de nombres y de ficheros y gestiona el almacenamiento local (caching).

Servicio de ficheros

Mantiene el contenido de los ficheros (y directorios) y los atributos de los ficheros: tiempos de creación, último acceso y última modificación; longitud; cuenta de referencias. Un fichero se identifica en el servicio de ficheros mediante un identificador único de fichero, **UFID**. Las operaciones sobre un fichero se refieren explícitamente a su UFID. La interfaz del servicio de ficheros con el cliente ofrece operaciones como *leer*, *escribir*, *crear*, *borrar*, *obtener_atributos* y *modificar_atributos*.

Servicio de nombres (directorios)

Es el encargado de proporcionar transparencia en la ubicación. En general, es una base de datos con elementos (nombre, UFID), donde se crean, se modifican y se buscan entradas. El nombre viene especificado por el string de caracteres que describe el path. La interfaz del servicio de nombres ofrece al cliente operaciones de *buscar_nombre*, *añadir_nombre* y *borrar_nombre*. Algunos atributos del fichero se mantienen por el servicio de nombres: tipo de fichero (ordinario o directorio); identificador del usuario propietario del fichero; derechos de acceso.

4.2.2 Identificación de ficheros

El cliente especifica un fichero al servidor de ficheros mediante el identificador único de ficheros, **UFID**, expedido por el servidor de ficheros cuando se crea un fichero. En principio, el UFID requiere identificar:

- El servidor del fichero, *S*.
- El fichero dentro del servidor, *F*.
- Los derechos de acceso sobre el fichero, *D*.

Los UFIDs deben protegerse de la manipulación por el cliente, ya que éste podría generar un UFID con derechos de acceso falsos. En un sistema centralizado tipo UNIX el identificador del proceso que accede al fichero determina directamente el identificador del usuario (UID), almacenado en el inodo del fichero, lo que autentifica al cliente. En cambio, en sistemas distribuidos se requiere un mecanismo adicional de autenticación que garantice la aplicación segura de los derechos de acceso. Un modelo de autenticación es el que vamos a describir a continuación, similar al de Amoeba [TAN95 §7].

El UFID es una credencial (*capability*) para el acceso al fichero. Cuando se crea un fichero, el servidor de ficheros genera un número aleatorio, R , como componente adicional del UFID del fichero que se almacena asociado al nombre del fichero para su utilización por el servidor de nombres. El UFID expedido por el servidor de nombres para un fichero solicitado por un cliente (en una operación de *buscar_nombre*) incluye un campo C que es una codificación de R y los derechos de acceso para ese cliente (estos se mantienen también sin codificar en un campo del UFID, D , para uso del propio cliente). La codificación se realiza mediante una función unidireccional, f_c :

$$C = f_c(R, D)$$

En las operaciones sobre el fichero, para comprobar la validez de un acceso, el servidor del fichero aplicará la función f_c sobre el valor R almacenado y el campo D del UFID del cliente, y comparará el resultado con el campo C del UFID del cliente:

$$\text{¿ } f_c(R, D) = C \text{ ?}$$

Un cliente que reclame unos derechos de acceso distintos a los reconocidos no superará esta comprobación. El esquema de este modelo de UFID se muestra en la Figura 1.

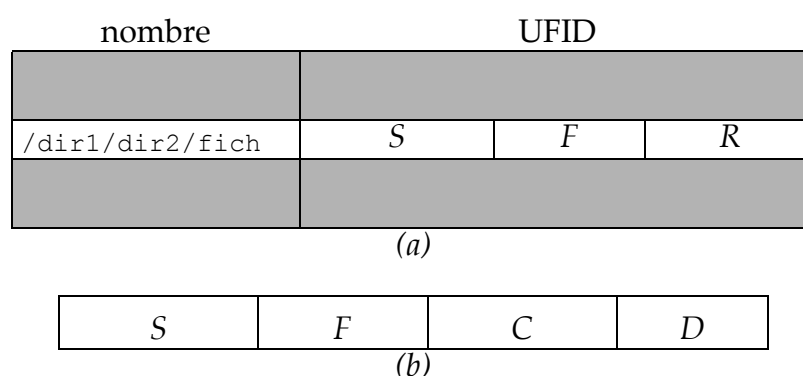


Figura 1. El UFID de un fichero `/dir1/dir2/fich` (a) almacenado en el servidor de nombres, (b) como se especifica en la petición de un cliente.

4.3 Servidores de nombres

Se requiere un servicio que permita al módulo cliente determinar la ubicación de un fichero (o más específicamente su UFID) a partir del nombre simbólico

del fichero (path). Se trata estrictamente de un servicio de directorios, aunque, por generalidad, este tipo de servicio se suele denominar servicio de nombres, y se utiliza también para otros tipos de aplicaciones distribuidas.

Como ya se sabe, los directorios poseen una estructura arborescente. Desde el punto de vista de la resolución de nombres, la estructura se divide en **dominios**, de forma que un dominio puede asociarse a una parte del path. Un servidor de nombres gestiona uno o más dominios. El cliente mantiene una tabla de entradas (dominio, servidor), de forma que, dado un path, busca en la tabla mediante comparación de strings a qué dominio pertenece.

Un servidor de nombres puede resolver el path como asociado a un dominio mantenido por otro servicio de nombres⁴, lo que ocurre en particular en redes de área amplia. En este caso, la resolución del nombre simbólico es indirecta y encadenará una sucesión de peticiones a diferentes servidores, lo que en general se conoce como **navegación**.

Podemos distinguir diferentes esquemas de navegación:

- **Iterativa.** El servidor de un dominio resuelve su parte del path y responde al cliente indicando el nuevo servidor para resolver el resto del path. La comunicación se resuelve mediante RPC convencional. Esta alternativa está limitada a un *dominio administrativo* único, ya que el cliente puede no tener acceso fuera de él. Un ejemplo es el servidor NIS⁵, utilizado por NFS. Una variante es la navegación **iterativa controlada por el servidor**. En este caso, el servidor invocado por el cliente es el encargado de realizar (iterativamente) la secuencia de peticiones para la resolución del path, respondiendo al cliente cuando éste ha sido resuelto completamente.
- **Recursiva.** El servidor de un dominio resuelve su parte del path y cursa una petición a un nuevo servidor para resolver el resto del path. El servidor que resuelve el último elemento del path es el que responde al cliente, por lo que este esquema no admite RPC convencional⁶. Por el contrario, requiere menos comunicación que el esquema anterior. Un ejemplo es el servidor DNS, que se utiliza en Internet⁷.

Para disminuir latencias, los servicios de nombres hacen un extenso uso del caching, lo que conduce a situaciones de inconsistencia entre los servidores de nombres. Sin embargo, ya que la migración de dominios entre servidores es infrecuente, no suele ser un objetivo prioritario del servicio de nombres el prevenir las posibles situaciones de identificación errónea de dominios debidas al caching.

⁴ En general desconocido por el cliente, ya que en caso contrario cabe esperar que tuviera registrada localmente dicha asociación.

⁵ Si bien hay que recalcar que NIS no es un servicio de directorios que se ajuste al modelo definido, ya que no gestiona UFIDs, sino un servicio de nombres más general que resuelve la navegación.

⁶ En este caso (respuesta directa al cliente), se dice que la búsqueda es transitiva. Una alternativa con mayor latencia es que el retorno siga el camino inverso nodo a nodo, que tiene la ventaja de que los nodos pueden hacer caching de los nombres resueltos.

⁷ DNS también soporta navegación iterativa.

4.4 Servidores de ficheros

El objetivo en el diseño de los servidores de ficheros distribuidos es el proporcionar una semántica lo más cercana posible a la que ofrecen los sistemas centralizados sin incurrir en una fuerte penalización en el rendimiento (fundamentalmente la latencia). Se utiliza extensamente el caching y se utilizan mecanismos de gestión de las copias que permiten compromisos razonables entre semántica y rendimiento.

4.4.1 Semánticas de compartición

Como ya hemos visto, es necesario establecer un compromiso entre el rendimiento deseado y la semántica que queremos garantizar en el sistema de ficheros distribuido cuando se producen accesos concurrentes. Podemos distinguir varios tipos de semánticas:

- **Semántica UNIX.** Denominada así porque es la que caracteriza el acceso a los ficheros de los sistemas UNIX clásicos. Las operaciones sobre un fichero se ordenan totalmente en el tiempo, lo que implica que una lectura devuelve la última actualización del fichero. Como veremos, es complejo para un sistema distribuido proporcionar semántica UNIX.
- **Semántica de sesión.** En una sesión de uso del fichero (desde que éste se abre hasta que se cierra) el proceso ve una copia privada del fichero; es decir, no se comparte el estado del fichero (por ejemplo el apuntador a la posición actual). En un sistema distribuido la semántica de sesión equivale a trabajar sobre una copia local que se carga cuando el fichero se abre y se actualiza en el servidor cuando se cierra. Se producen condiciones de carrera cuando se escribe en el servidor la copia actualizada, cuya resolución compete al usuario o a la aplicación.
- **Ficheros inmutables.** Los ficheros no se modifican, sino que se reemplazan atómicamente (los directorios sí se modifican) por nuevas versiones. Es posible la compartición concurrente para lectura, pero si se pretende acceder un fichero abierto por otro proceso para escritura existen dos alternativas: (a) considerar error la operación de abrir, y (b) obtener la versión anterior del fichero. Esta semántica es adecuada en servicios particulares, como los de back-up y los repositorios de información con gestión de versiones (por ejemplo, *subversion*⁸).
- **Semántica de transacciones.** Como se vio en el capítulo precedente, el uso de transacciones permite definir explícitamente secuencias de operaciones sobre ficheros, garantizando la semántica transaccional definida por las propiedades *ACID*.

⁸ <http://subversion.apache.org/>

4.4.2 Tipos de servidores

Dependiendo de la información que almacena el servidor acerca del fichero que está siendo accedido por un cliente, se pueden distinguir dos categorías de servidores, que determinarán estrechamente las características del servicio.

- **Servidor sin estado.** El servidor no almacena información del cliente en una sesión sobre un fichero. Por esta razón, las interfaces cliente-servidor que ofrece este tipo de sistema no incluyen primitivas específicas de abrir/cerrar. El cliente suministra en cada llamada toda la información necesaria para realizar la operación, incluido el puntero a la posición de acceso actual. Una de las principales ventajas de este enfoque es que el fallo de un cliente no afecta en absoluto al servidor. Por contra, hace difícil el proporcionar semántica UNIX (por ejemplo, los derechos de acceso al fichero se comprueban en cada acceso). El ejemplo más notable de servidor sin estado es NFS.
- **Servidor con estado.** El servidor crea una entrada para el fichero ante una invocación de abrir fichero de un cliente. Las sucesivas invocaciones de acceso al fichero requieren mensajes más cortos, ya que el servidor almacena el apuntador a la última posición accedida y otra información, que puede incluir hasta bloques del fichero, lo que posibilita lectura anticipada en el servidor. Este enfoque limita de forma inherente el número de ficheros abiertos simultáneamente. El principal inconveniente es que el servidor tiene que gestionar el posible fallo de los clientes, para liberar la información de estado de los ficheros abiertos por el cliente que falla, evitando así la degradación del servidor. Un ejemplo de servidor con estado es RFS (*Remote File System*), actualmente muy poco utilizado.

Por su propia naturaleza, las operaciones de la interfaz cliente-servidor en un sistema de ficheros sin estado, al tener que especificar en la invocación todos los parámetros de la operación, tienden a ser idempotentes, lo que permite una gestión más sencilla de las reinvocaciones ante sospechas de fallo en la transmisión de la petición. Por ejemplo, para un servidor sin estado una operación de lectura se especificaría como:

```
status= leer_fich (ufid, buffer, longitud)
```

mientras que la operación análoga para un servidor con estado sería:

```
status= leer_fich (ufid, buffer, posición, longitud)
```

El parámetro `posición` es un puntero al siguiente byte a leer en el fichero. Obsérvese que en un servidor con estado este parámetro lo gestiona el servidor mediante la tabla de ficheros abiertos, donde se habrá reservado una entrada como consecuencia de la operación previa de abrir el fichero. Esta entrada incluirá el puntero a la posición del fichero que el servidor actualizará tras cada acceso. Por el contrario, un servidor sin estado no gestiona los ficheros abiertos por el cliente, por lo que este deberá gestionar la posición de acceso y especificarla en cada petición. Si, tras expirar el `time-out` de espera de una respuesta, el cliente reenvía la invocación con los mismos parámetros, el comportamiento difiere en ambos tipos de servidores. En un servidor sin estado

la operación siempre se comportará como idempotente gracias a que el servidor, en cada lectura, abrirá el fichero, posicionará en posición y realizará el acceso. Sin embargo, en un servidor con estado, si el primer mensaje de petición terminó llegando al servidor, este habrá actualizado el puntero de acceso en la tabla de ficheros abiertos, por lo que la reinvocación se tratará como una nueva operación.

4.4.3 Caching y gestión de la consistencia

Se refiere al almacenamiento temporal de (trozos de) ficheros en el nodo cliente⁹, con el objetivo de minimizar los costes de comunicación que el acceso remoto lleva asociados.

Para ello se mantienen copias locales de (parte de) los ficheros en los nodos clientes, utilizando como criterio de gestión del espacio la localidad temporal. La unidad de gestión, cantidad de información que se transmite en cada petición, puede ser el fichero completo o un bloque¹⁰. De hecho, el transmitir cantidades grandes de información proporciona **lectura anticipada** (*buffering*), que potencia también la localidad espacial.

El caching se puede soportar bien en el disco local, de forma **persistente**, bien en memoria, **no persistente**. En este último caso puede usarse memoria del núcleo, que optimiza el espacio de almacenamiento, o memoria de usuario. Los micronúcleos, como Mach 3.0, permiten la utilización de un gestor de memoria (definido fuera del micronúcleo, en espacio de usuario) que mapea los bloques almacenados en su espacio de direcciones en los espacios de direcciones de los procesos que solicitan dichos bloques, de forma que existe una copia única de cada bloque y a su vez es visible en el espacio de cada proceso que la usa.

Una generalización del almacenamiento temporal, en redes WAN, es el **caching estructurado**, almacenamiento temporal en nodos intermedios a diferentes niveles. Este servicio es el que ofrecen los nodos *proxy* en Internet.

El caching, como toda forma de replicación, lleva asociada la necesidad de gestionar la consistencia. La política de gestión de la consistencia condiciona la semántica de compartición. Las políticas básicas para gestionar la consistencia son las siguientes:

- **Write-through.** Cuando un elemento se modifica, se copia en el servidor. Se suelen introducir adaptaciones para mejorar el rendimiento, como la **escritura retardada** (acumulando modificaciones), y no copiar en el servidor los ficheros temporales.
- **Write-on-close.** El fichero se escribe en el servidor cuando se cierra. Determina semántica de sesión. Una mejora consiste en retardar la escritura (es frecuente que un fichero se borre después de cerrar).

⁹ El servidor puede proporcionar también *buffer cache* de bloques como en cualquier sistema de ficheros centralizado.

¹⁰ El tamaño del bloque usado como unidad de gestión del caching al que nos referimos aquí no tiene por qué coincidir con el definido para el sistema de ficheros local de cada nodo.

- **Gestión centralizada.** El servidor de ficheros gestiona la apertura/cierre de ficheros mediante un algoritmo de sincronización lectores-escritores. Proporciona semántica UNIX, pero es poco escalable y no tolerante a fallos.

Salvo en el caso de la gestión centralizada, las políticas de gestión de la consistencia descritas no son suficientes para proporcionar la semántica definida (en particular UNIX), requiriendo un mecanismo adicional de **validación** para permitir a un nodo conocer cuándo la réplica de un fichero que está se usando en su cache ha quedado obsoleta por la actualización de otra réplica del mismo fichero en otro nodo (no necesariamente el servidor). Existen dos políticas básicas de validación, dependiendo de dónde parta la iniciativa:

- (a) Desde los clientes, accediendo periódicamente a los atributos del fichero en el servidor, para ver si se ha modificado. El periodo entre validaciones es un parámetro crítico: preservar la semántica requiere periodos cortos, a costa de sobrecargar la red.
- (b) Desde el servidor, notificando a los clientes cuando una copia ha quedado obsoleta (*callback*). Requiere almacenar algo de estado en el servidor.

4.5 Ejemplos

4.5.1 NFS (Network File System)

Introducido por Sun Microsystems en 1985, fue desarrollado originalmente para UNIX. Se concibió como sistema abierto, lo que le ha permitido ser adoptado por todas las familias UNIX y por otros sistemas operativos (VMS, Windows), convirtiéndose en un estándar de facto en LANs. NFS ha evolucionado mucho y la Versión 4 actual poco tiene que ver con las anteriores, ya que incluye estado y la posibilidad de implementación en WAN. Sin embargo, aquí nos referiremos a las características de las versiones anteriores. Una descripción general de NFS puede encontrarse en [TAN95 §5] [VAH96 §10] [BRO94 §3] y [COU05 §8].

4.5.1.1 Características generales

Los servidores *exportan* directorios. Para hacer exportable un directorio se incluye el path en un determinado fichero de configuración. Los clientes *montan* los directorios exportados, y estos se ven en el cliente completamente integrados en el sistema de ficheros. El montaje se ejecuta en el booting del sistema operativo, o por demanda cuando se abre un fichero mediante un servicio adicional de NFS, el *automounter*. Las operaciones sobre ficheros y las peticiones de montar son atendidas por sendos procesos daemon en el servidor (*nfsd* y *mountd* respectivamente).

Los servidores NFS son sin estado, lo que evita el tener que tratar en el servidor los fallos de los clientes. Gracias a que la mayoría de las operaciones son idempotentes, la gestión de errores de comunicación en el cliente se simplifica.

La semántica de compartición intenta ser UNIX, aunque con alguna limitación, debido fundamentalmente a la gestión del caching y a su condición de servidor sin estado. Ofrece el mismo modelo de protección de UNIX, aunque, debido a la ausencia de estado en el servidor, los derechos de acceso se comprueban en cada operación de acceso al fichero en vez de sólo al abrir. Inicialmente NFS no adoptaba ningún mecanismo de autenticación. La interfaz del cliente incluía en las RPCs el identificador de usuario UNIX, que se comprobaba en el servidor, lo que no impedía la posibilidad de suplantar la identidad de un usuario construyendo una RPC al margen de la ofrecida por la interfaz. Actualmente suele combinarse con sistemas de autenticación como Kerberos.

NFS utiliza clásicamente el servicio NIS (*Network Information Server*) para centralizar la información sobre ubicación de los servidores.

4.5.1.2 Interfaces

Las aplicaciones usan la interfaz de UNIX. NFS define una interfaz para comunicación cliente-servidor, que consta de tres protocolos:

- El protocolo RPC de Sun define el formato de la comunicación cliente-servidor. Los datos se serializan de acuerdo al formato XDR. La comunicación se basa en UDP. A partir de la Versión 3 también se soporta TCP para comunicación en WAN.
- Protocolo para operaciones de montar/desmontar directorios [VAH96 §10].
- Protocolo NFS. Procedimientos para operaciones sobre ficheros (búsqueda, crear, leer, escribir, borrar, obtener atributos...). Consúltese [VAH96 §10] o [COU05, Fig. 8.9] para una descripción del protocolo.

4.5.1.3 Implementación

Identificación de ficheros

Los clientes especifican como identificador único del fichero un *file handle*, proporcionado por la operación de *lookup* (búsqueda del nombre del fichero). Es una estructura de datos que contiene información para la identificación del fichero en el servidor, fundamentalmente el identificador del sistema de ficheros y el número de i-nodo. El *file handle* es opaco para el cliente (no maneja su contenido). La resolución del path se hace iterativamente en el cliente, requiriendo una operación de *lookup* por cada componente del path.

Sistema de ficheros virtual, VFS

Sobre UNIX, NFS utiliza la interfaz del *Virtual File System (VFS)*¹¹ para el acceso transparente a ficheros locales y remotos. mantiene un *v-node* (i-nodo virtual)

¹¹ VFS proporciona a las aplicaciones UNIX una interfaz independiente del sistema de ficheros, lo que permite soportar tanto NFS como cualquier otro sistema de ficheros. Consúltese, por ejemplo, [VAH96 §8].

por cada fichero abierto. Si el fichero al que representa es local, el v-node apunta directamente al i-node correspondiente en el sistema de ficheros local. Si el fichero es remoto, el v-node apunta a un *r-node* (i-nodo remoto) que almacena el file handle que usará el cliente NFS en la RPC. La Figura 2 representa esta arquitectura.

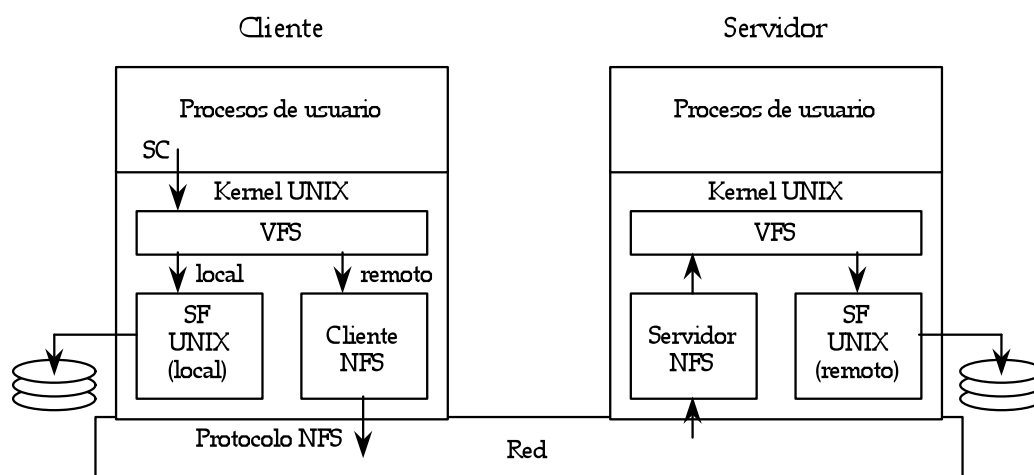


Figura 2. Estructura del servicio NFS

Caching

Se trabaja con bloques de gran tamaño (típicamente 8 Kbytes), lo que proporciona lectura anticipada. La política de escritura es *write-through* retardada. Sólo se envían al servidor bloques completos.

Periódicamente el cliente valida un bloque cargado en su cache comprobando en el servidor si los atributos han cambiado (mediante una RPC de obtener atributos, *getattr*) y actualizando la copia en este caso. El periodo de validación suele ser de 30 segundos.

4.5.1.4 Limitaciones de NFS

El mantenimiento de la consistencia UNIX resulta problemático. La disminución del periodo de validación para mejorar la consistencia produce una sobrecarga por la gran cantidad de operaciones *getattr* que se realizan.

En principio, el montaje de sistemas de ficheros remotos no era transparente (hay que identificar al servidor). El *automounter*, una utilidad que permite el montaje dinámico de sistemas de ficheros por demanda, mejora este aspecto.

Debido a la falta de estado, bloquear el acceso a ficheros remotos requiere un mecanismo de exclusión mutua independiente. En UNIX se utiliza un servidor específico, *lockd*.

No está diseñado para soportar replicación de servidores. Para incrementar la disponibilidad, las partes del sistema de ficheros que tengan que soportar una tasa muy alta de accesos pueden replicarse en un conjunto de servidores,

siempre que sean para lectura. Esto se hace con el NIS, de forma que cada réplica está accesible para lectura, pero la escritura se hace siempre sobre la copia master y manualmente se actualizan las réplicas.

En principio, NFS se concibió para redes locales de unas decenas de nodos, aunque las mejoras en las tecnologías LAN y las optimizaciones introducidas en las últimas versiones de NFS permiten soportar un número mucho mayor de clientes. A partir de la Versión 3 permite configuraciones en redes WAN.

4.5.2 AFS (Andrew File System)

Andrew es el nombre de una familia de sistemas de ficheros distribuidos para UNIX desarrollados en la Universidad de Carnegie Mellon a partir de 1983. Los componentes de la familia son:

- AFS-1 (1983). Prototipo no optimizado.
- AFS-2 (1985)
- AFS-3 (1988)
- Coda (1987). Proporciona funcionamiento en modo desconectado.

AFS puede definirse en general como un sistema de ficheros distribuido sin estado que proporciona semántica de sesión. Aquí presentaremos las características generales. Para más detalle, pueden consultarse las siguientes referencias: [MUL93 §14] [TAN93 §13] [COU05 §8] [SAT90].

4.5.2.1 Arquitectura Andrew

La arquitectura de AFS consta de dos componentes, uno en el servidor y otro en el cliente:

- *Vice*: Código de los servidores. Desde el punto de vista del cliente, Vice es un conjunto de servidores de ficheros interconectados en red.
- *Venus*: Código cliente que se ejecuta sobre el sistema operativo en los nodos conectados a Vice.

Los ficheros de Vice se ven como integrados en el sistema local de cada puesto cliente.

Soporta replicación de subconjuntos del sistema de ficheros (*volumes*) de actualización poco frecuente (AFS-2). Esta técnica también se usa para back-ups (mediante copias de sólo lectura de un *volume*).

4.5.2.2 Implementación

AFS-1, AFS-2 y Coda trabajan con ficheros enteros; AFS-3 con bloques de 64 Kb. El caching se implementa en el disco del cliente.

La política de escritura es *write-on-close*. La semántica de sesión se intenta proporcionar mediante *callbacks* (a partir de AFS-2). Cuando Vice envía un fichero a un cliente le adjunta una *promesa de callback* y toma nota de ello¹². Cuando un cliente cierra un fichero modificado, Vice comunica a los clientes para quienes mantiene una promesa de callback de ese fichero que cancelen la promesa. El código Venus de un cliente que acceda a la copia local de un fichero con la promesa cancelada se encargará de recargar la nueva versión.

AFS-3 introduce importantes optimizaciones con respecto a AFS-2. Inserta Venus en el núcleo (utilizando la interfaz VFS, como NFS) y define *cells* de servidores para escalar el sistema a WAN.

4.5.2.3 Seguridad

Define dominios de acceso como UNIX. Los derechos de acceso se definen de modo compatible con UNIX.

Proporciona autenticación por medio de un servidor de autenticación que expide *tokens* (fichas) ante la presentación de la clave del usuario en el login para acceder al sistema de ficheros durante un plazo preestablecido (típicamente 24 horas). Alguna versión de AFS-3 ha adoptado Kerberos.

4.5.2.4 Disponibilidad: Coda

Coda es una versión de AFS pensada para proporcionar disponibilidad en entornos sujetos a fallos, tanto en la red como en los servidores, por lo que resulta adecuado para dispositivos móviles (sujetos a desconexiones frecuentes) y en general en sistemas replicados que requieran tolerancia a fallos.

Gestiona réplicas de *volumes* siguiendo una estrategia optimista. Para ello se basa en dos mecanismos:

- Utiliza números de versión y vectores de tiempos para la actualización consistente de las réplicas (a veces requiere intervención manual).
- Opera sobre la cache local cuando pierde la conexión hasta que consigue conectar a otro servidor (funcionamiento en modo desconectado).

Las características básicas de Coda son las de AFS-2. Para más información puede consultarse [COU05 §15] y [SAT90].

4.6 Ejercicios

1 Dos procesos redirigen sus salidas hacia un fichero común *x.out*. Ambos procesos han abierto el fichero en modo *append*. Explicar las diferencias de comportamiento según el sistema de ficheros soporte semántica UNIX, semántica de sesión, o semántica de ficheros inmutables.

¹² Esto es lo único del estado del cliente que AFS gestiona en el servidor.

2 Del manual del programador de un sistema de ficheros distribuido hemos extraído las siguientes especificaciones de RPCs del servidor de ficheros:

```
status= read_remote_file (ufid, buffer, longitud)
```

Lee *longitud* bytes del fichero identificado por *ufid* en *buffer*. Devuelve número de bytes leídos o *ERROR*.

```
status= write_remote_file (ufid, buffer, longitud)
```

Escribe *longitud* bytes de *buffer* en el fichero identificado por *ufid*. Devuelve número de bytes escritos o *ERROR*.

- (a) Discutir si se trata de un servidor de ficheros con estado o sin estado.
- (b) ¿Qué se puede decir acerca de la idempotencia o no de estas operaciones?

3 Dada la lista de RPCs para comunicación cliente-servidor de NFS ([COU05 Fig. 8.9]), ¿cuáles son idempotentes y cuáles no?

4 Explicar el efecto semántico que tiene la gestión de la consistencia en NFS (se comprueba periódicamente la validez de los ficheros), así como la escritura retardada. ¿Hasta qué punto se mantiene la semántica UNIX?

5 En un sistema de ficheros UNIX sin estado (como NFS), una llamada al sistema *unlink* puede no respetar la semántica UNIX. Explica el porqué. Pista: cuando el *unlink* de UNIX provoca que la cuenta de referencias llega a cero, puede haber procesos accediendo al fichero, por lo que no libera el i-nodo hasta que el último de los procesos cierra el descriptor.

6 El sistema de ficheros de UNIX permite que un proceso pueda bloquear el acceso a un fichero (flag *O_EXCL* en *open*). Discutir si NFS puede proporcionar esta semántica.