

SISTEMAS DE ARCHIVOS DISTRIBUIDOS

- 8.1. Introducción
- 8.2. Arquitectura del servicio de archivos
- 8.3. Sistema de archivos en red de Sun (NFS)
- 8.4. Sistema de archivos Andrew
- 8.5. Avances recientes
- 8.6. Resumen

Los sistemas de archivos distribuidos soportan la compartición de información en forma de archivos a través de Internet. Un servicio de archivos bien diseñado proporciona acceso a los archivos almacenados en un servidor con prestaciones y fiabilidad semejantes (y en algunos casos mejor) que la de los archivos almacenados en discos locales. Un sistema de archivos distribuidos permite a los programas almacenar y acceder a archivos remotos del mismo modo que si fueran locales, permitiendo a los usuarios que accedan a archivos desde cualquier computador en una intranet.

Describimos los diseños de dos sistemas de archivos distribuidos que han sido de uso extendido durante una década o más:

- Sistema de archivos en red de Sun, NFS.
- Sistema de archivos Andrew, AFS.

Estos casos de estudio ilustran un rango de soluciones de diseño para la emulación de una interfaz de un sistema de archivos UNIX con distintos grados de escalabilidad y tolerancia a fallos. Cada una exige alguna desviación de la emulación de la semántica de actualización de archivo de *una copia* de UNIX.

Recientes avances en el diseño de sistemas distribuidos han explotado la conectividad de mayor ancho de banda de las redes de área local conmutadas y los nuevos modos de organización de datos en disco para obtener prestaciones muy altas, tolerancia a fallos y sistemas de archivos altamente escalables.

8.1. INTRODUCCIÓN

En los Capítulos 1 y 2, identificamos el hecho de compartir recursos como un objetivo clave de los sistemas distribuidos. El compartir información almacenada es quizás el aspecto más importante de la compartición de recursos distribuidos. Se obtiene a gran escala en Internet principalmente por el uso de los servidores web, pero los requisitos para compartir en redes de área local e intranet conduce a una necesidad de un tipo de servicio diferente, uno que soporte el almacenamiento persistente de datos y programas de todos los tipos, en nombre de los clientes, y la consiguiente distribución de datos actualizados. El propósito de este capítulo es describir la arquitectura e implementación de sistemas de archivos distribuidos *básicos*. Utilizamos aquí la palabra «básico» para señalar sistemas de archivos distribuidos cuyo propósito principal es emular la funcionalidad de sistemas de archivos no distribuidos para programas cliente ejecutándose en múltiples computadores remotos. Éstos no mantienen múltiples réplicas persistentes de archivos, ni soportan el ancho de banda y las garantías de temporización requeridas para flujos de datos multimedia, cuyos requisitos se revisarán en capítulos posteriores. Los sistemas de archivos distribuidos básicos proporcionan un sustento esencial para la organización de la computación basada en intranets.

Comenzamos con un breve repaso del espectro de los sistemas de almacenamiento distribuidos y no distribuidos. Los sistemas de archivos fueron desarrollados originalmente para sistemas centralizados de computadores y computadores portátiles como una disponibilidad del sistema operativo que proporciona una interfaz de programación adecuada para el almacenamiento en disco. Así, adquirieron posteriormente características tales como control de acceso y mecanismos de bloqueo a archivos que les hicieron útiles para la compartición de datos y programas. Los sistemas de archivos distribuidos soportan la compartición de la información en forma de archivos y recursos hardware que dan soporte al almacenamiento persistente a través de una intranet. Un servicio de archivos bien diseñado proporciona acceso a los sistemas almacenados en un servidor con prestaciones y fiabilidad semejantes a, y en algunos casos mejor que, los archivos almacenados en discos locales. Su diseño se adapta para las características de prestaciones y fiabilidad de redes locales y por ello son más efectivos proporcionando almacenamiento persistente compartido para uso en intranets. Los primeros servidores de archivos fueron desarrollados por investigadores en la década de los setenta [Birrell y Needham 1980, Mitchell y Dion 1982, Leach y otros 1983] y el Sistema de Archivos en Red de Sun estuvo disponible en los primeros años ochenta [Sandberg y otros 1985, Callaghan 1999].

Un servicio de archivos permite a los programas almacenar y acceder a los archivos remotos del mismo modo que se hace con los locales, permitiendo a los usuarios acceder a sus archivos desde cualquier computador de una intranet. La concentración de almacenamiento persistente en unos pocos servidores reduce la necesidad de almacenamiento en disco local y (más importante) permite economizar la gestión y el archivo de datos persistentes pertenecientes a una organización. Otros servicios, como el servicio de nombres, el servicio de autenticación de usuarios y el servicio de impresión, pueden ser implementados más fácilmente si hacen llamadas al servicio de archivos, que satisface sus necesidades de almacenamiento permanente. Los servidores web dependen de los sistemas de archivos para el almacenamiento de las páginas web que sirven. En organizaciones que operan con servidores web para acceso externo e interno vía una intranet, los servidores web suelen almacenar y acceder al material desde un sistema de archivos local distribuido.

Con la llegada de la programación orientada a objetos distribuida, aparece la necesidad del almacenamiento persistente y la distribución para los objetos compartidos. Una forma de lograr esto es serializar los objetos (de la forma descrita en la Sección 4.3.2) y almacenar y recuperar estos objetos serializados sobre archivos. Pero este método para obtener persistencia y distribución llega a ser impráctico para objetos que cambian rápidamente, y se han desarrollado otras aproximaciones más directas. La invocación remota de objetos de Java y de los ORB de CORBA proporcio-

	Comparación	Persistencia	Caché/replicas distribuidas	Mantenimiento de consistencia	Ejemplo
Memoria principal	✗	✗	✗	1	RAM
Sistema de archivos	✗	✓	✗	1	Sistema de archivos UNIX
Sistema de archivos distribuido	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Servidor Web
Memoria compartida distribuida	✓	✗	✓	✓	Ivy (Cap. 16)
Objetos remotos (RMI/ORB)	✓	✗	✗	1	CORBA
Almacén de objetos persistentes	✓	✓	✗	1	Servicio de objetos persistentes de CORBA
Almacén de objetos persistentes distribuido	✓	✓	✓	✓	PerDis, Khazana

Figura 8.1. Sistemas de almacenamiento y sus propiedades.

nan acceso a objetos remotos, compartidos, pero ninguno de ellos asegura la persistencia de los objetos ni la replicación de los objetos distribuidos.

Los desarrollos recientes, sobre la distribución de la información almacenada, se orientan hacia los sistemas de memoria compartida distribuida (DSM) y los almacenes de objetos persistentes. DSM se describe con detalle en el Capítulo 16. Proporciona una emulación de una memoria compartida mediante la replicación de páginas o segmentos de memoria, en cada máquina. Los almacenes de objetos persistentes se presentaron en el Capítulo 5. Su finalidad es proporcionar persistencia para objetos compartidos distribuidos. Ejemplos de ello son el Servicio de Objetos Persistentes de CORBA (véase el Capítulo 17) y extensiones de persistencia para Java [Jordan 1996, java.sun.com IV]. Algunas investigaciones recientes han resultado en plataformas que soportan la replicación automática y el almacenamiento persistente de objetos (por ejemplo, PerDis [Ferreira y otros 2000] y Khazana [Carter y otros 1998]).

La Figura 8.1 proporciona un resumen de las propiedades de los distintos tipos de sistemas de almacenamiento que hemos mencionado. La columna *consistencia* indica qué mecanismos existen para el mantenimiento de la consistencia entre múltiples copias de datos cuando ocurren actualizaciones. Virtualmente todos los sistemas de almacenamiento confían en el uso de caché para optimizar las prestaciones de los programas. Las técnicas de caché se aplicaron en primer lugar a la memoria principal y a los sistemas de archivos no distribuidos, para los que la consistencia es estricta (señalado por un «1»), para consistencia de una copia en la Figura 8.1), los programas no contemplan discrepancias entre las copias en la caché y los datos almacenados después de una actualización. Cuando se utilizan réplicas distribuidas, la consistencia estricta es más difícil de obtener. Los sistemas de archivos distribuidos como Sun NFS y el sistema de archivos Andrew replican copias de porciones de archivos en los computadores de los clientes y adoptan mecanismos de consistencia específicos para mantener una aproximación a la consistencia estricta. Esto se indica mediante una marca (✓) en la columna de consistencia de la Figura 8.1. Discutiremos estos mecanismos y el grado en el que se desvían de la consistencia estricta en las Secciones 8.3 y 8.4.

La Web utiliza cachés extensivamente tanto en los computadores cliente como en los servidores proxy mantenidos por la organización de cada usuario. La consistencia entre las copias almacenadas en la caché de un proxy web y los clientes y en el servidor original sólo se consigue mediante acciones específicas del usuario. Cuando se actualiza una página almacenada en el servidor

original no se advierte a los clientes, y son ellos quienes deben realizar comprobaciones periódicas para mantener sus copias actualizadas. Esto sirve adecuadamente para el propósito de navegación web, pero no soporta el desarrollo de aplicaciones cooperativas como el de un tablero compartido distribuido. Los mecanismos de consistencia utilizados en los sistemas DSM se discutirán con detalle en el Capítulo 16. Los sistemas de objetos persistentes varían considerablemente en su aproximación al uso de caché y consistencia. Los esquemas CORBA y Persistent Java mantienen una única copia de cada objeto persistente y para acceder a ellos es preciso realizar una invocación remota, por lo que el único tema de consistencia está entre la copia persistente de un objeto en disco y la copia activa en memoria, que no es visible para los clientes remotos. Los proyectos PerDiS y Khazana que hemos mencionado anteriormente mantienen réplicas en la caché de los objetos y emplean mecanismos de consistencia bastante elaborados para producir formas de consistencia semejantes a las que se encuentran en los sistemas DSM.

Habiendo presentado algunos temas más profundos relacionados con el almacenamiento y distribución de datos persistentes y no persistentes, volvemos al tema principal de este capítulo, el diseño de sistemas básicos de archivos distribuidos. Describimos algunas características relevantes de los sistemas de archivos (no distribuidos) en la Sección 8.1.1, y los requisitos para sistemas de archivos distribuidos en la Sección 8.1.2. La Sección 8.1.3 presenta los casos de estudio que se utilizarán a lo largo del capítulo. En la Sección 8.2 definimos un modelo abstracto para un servicio básico de archivos distribuidos, incluyendo un conjunto de interfaces de programación. El sistema Sun NFS se describe en la Sección 8.3, comparte muchas de las características del modelo abstracto. En la Sección 8.4 describimos el Sistema de Archivos Andrew, un sistema ampliamente usado que emplea mecanismos de caché y consistencia sustancialmente diferentes. La Sección 8.5 revisa algunos de los desarrollos recientes en el diseño de servicios de archivos.

Los sistemas descritos en este capítulo no cubren el espectro total de los sistemas de archivos distribuidos y gestión de datos. Varios sistemas con características más avanzadas serán descritos posteriormente en el libro. El Capítulo 14 incluirá una descripción de Coda, un sistema de archivos distribuidos que mantiene réplicas persistentes de los archivos para la fiabilidad, disponibilidad y el trabajo desconectado. Bayou, un sistema de gestión de datos distribuidos que proporciona una forma de replicación débilmente consistente para alta disponibilidad se tratará también en el Capítulo 14. El Capítulo 15 discutirá el servidor de archivos de vídeo Tiger, que está diseñado para proporcionar la entrega oportuna de caudales de datos a gran número de clientes.

8.1.1. CARACTERÍSTICAS DE LOS SISTEMAS DE ARCHIVOS

Los sistemas de archivos son responsables de la organización, almacenamiento, recuperación, nomenclación, compartición y protección de los archivos. Proporcionan una interfaz de programación característica de la abstracción de archivo, liberando a los programadores de la preocupación por los detalles de la asignación y la disposición del almacenamiento. Los archivos se almacenan en discos y otros medios de almacenamiento no volátiles.

Los archivos contienen *datos* y *atributos*. Los datos consisten en una secuencia de elementos de datos (normalmente bytes de 8 bits), donde cualquier porción de ésta es accesible mediante operaciones de lectura y escritura. Los atributos se alojan como un único registro que contiene información como la longitud del archivo, marcas de tiempo, tipo del archivo, identidad del propietario y listas de control de acceso.

En la Figura 8.3 se muestra una estructura típica del registro de atributos. Los atributos resaltados son administrados por el sistema de archivos y no son modificables habitualmente desde los programas del usuario.

Los sistemas de archivos están diseñados para almacenar y gestionar gran número de archivos, con posibilidades de crear, nombrar y borrar archivos. La nomenclatura de los archivos está respal-

Módulo de directorio:	relaciona nombres de archivos con ID de archivos
Módulo de archivos:	relaciona ID de archivos con archivos concretos
Módulo de control de acceso:	comprueba los permisos para una operación solicitada
Módulo de acceso a archivos:	lee o escribe datos o atributos de un archivo
Módulo de bloques:	accede y asigna bloques de disco
Módulo de dispositivo:	E/S de disco y búferes

Figura 8.2. Módulos del sistema de archivos.

dada por la utilización de directorios. Un *directorio* es un archivo, a menudo de un tipo especial, que relaciona los nombres en texto con los identificadores internos de los archivos. Los directorios pueden incluir los nombres de otros directorios derivando en el esquema familiar de nomenclatura jerárquica de archivos y los *nombres de ruta*, tanto para los archivos en UNIX como para otros sistemas operativos. Los sistemas de archivo tienen también la responsabilidad del control de acceso a los archivos, restringiendo el acceso a los mismos de acuerdo con las autorizaciones de los usuarios y el tipo de acceso solicitado (lectura, actualización, ejecución y demás).

El término *metadato* se utiliza a menudo para referirse a toda la información extra almacenada por un sistema de archivos que es necesaria para la gestión de los mismos. Incluye los atributos de los archivos, los directorios y todas las demás informaciones persistentes empleadas por el sistema de archivos.

La Figura 8.2 muestra una estructura típica de niveles para la implementación de un sistema de archivos no distribuido en un sistema operativo convencional. Cada nivel depende sólo de los niveles que se encuentran debajo de él. La implementación de un servicio de archivos distribuidos requiere todos los componentes indicados, junto a componentes adicionales para ocuparse de la comunicación cliente-servidor y de la nomenclatura y ubicación de los archivos distribuidos.

Tamaño del archivo
Marca temporal de creación
Marca temporal de lectura
Marca temporal de escritura
Marca temporal de atributos
Contador de referencias
Propietario
Tipo de archivo
Lista de control de acceso

Figura 8.3. Estructuras del registro de atributos de un archivo.

<code>desarchivo = open(nombre, modo)</code>	Abre un archivo existente con un <i>nombre</i> determinado.
<code>desarchivo = creat(nombre, modo)</code>	Crea un archivo nuevo con un <i>nombre</i> determinado.
	Ambas operaciones devuelven un descriptor de archivo que referencia el archivo abierto. El <i>modo</i> es <i>read</i> , <i>write</i> u otro.
<code>estado = close(desarchivo)</code>	Cierra el archivo abierto <i>desarchivo</i> .
<code>recuento = read(desarchivo, búfer, n)</code>	Transfiere <i>n</i> bytes del archivo referenciado por <i>desarchivo</i> sobre <i>búfer</i> .
<code>recuento = write(desarchivo, búfer, n)</code>	Transfiere <i>n</i> bytes al archivo referenciado por <i>desarchivo</i> desde <i>búfer</i> .
	Ambas operaciones retornan el número de bytes transferidos realmente y adelanta el apuntador de lectura-escritura.
<code>pos = lseek(desarchivo, despl, desde)</code>	Desplaza el puntero de lectura-escritura hasta <i>despl</i> (relativo o absoluto, dependiendo del valor de <i>desde</i>).
<code>estado = unlink(nombre)</code>	Elimina el <i>nombre</i> de archivo de la estructura de directorios. Si el archivo no tuviera otros nombres, sería también eliminado.
<code>estado = link(nombre1, nombre2)</code>	Añade un nombre nuevo (<i>nombre2</i>) a un archivo (<i>nombre1</i>).
<code>estado = stat(nombre, búfer)</code>	Obtiene los atributos de archivo del archivo <i>nombre</i> sobre <i>búfer</i> .

Figura 8.4. Operaciones del sistema de archivos de UNIX.

◊ **Operaciones en el sistema de archivos.** La Figura 8.4 resume las principales operaciones sobre archivos que están disponibles para aplicaciones en sistemas UNIX. Ésas son las llamadas del sistema que implementa el núcleo, los programadores de aplicaciones normalmente acceden a ellas a través de bibliotecas de procedimientos, como la librería estándar de entrada-salida de C o las clases archivo de Java. Proporcionamos aquí las primitivas como una indicación de las operaciones que se espera que los servicios de archivo soporten y para poder comparar con las interfaces de servicios de archivo que veremos más adelante.

Las operaciones de UNIX se basan en un modelo de programación en el que se almacena cierta información del estado de un archivo, por el sistema de archivos, para cada programa que lo use. El sistema registra una lista de los archivos abiertos actualmente con un apuntador de lectura-escritura para cada uno, que proporciona la posición en el archivo en la que se aplicará la siguiente operación de lectura o escritura.

El sistema de archivos es responsable de aplicar el control de acceso para los archivos. En sistemas de archivos locales como en UNIX, esto se hace cada vez que se abre un archivo, comprobando los derechos permitidos para la identidad del usuario mediante la lista de control de acceso contra el modo de acceso solicitado en la llamada del sistema *open*. Si los derechos concuerdan con el modo, el archivo es abierto y el *modo* se almacena en la información del estado del archivo abierto.

8.1.2. REQUISITOS DEL SISTEMA DE ARCHIVOS DISTRIBUIDOS

Muchos de los requisitos y potenciales obstáculos en el diseño de servicios distribuidos fueron ya observados en los primeros desarrollos de sistemas de archivos distribuidos. Inicialmente ofrecían transparencia de acceso y transparencia de ubicación, los requisitos de prestaciones, escalabilidad, control de concurrencia, tolerancia a fallos y seguridad surgieron y se fueron satisfaciendo en fases posteriores del desarrollo. Discutimos estos requisitos, y otros relacionados, en las subsecciones siguientes.

◊ **Transparencia.** El servicio de archivos es el servicio más fuertemente cargado en una intranet, por lo que su funcionalidad y prestaciones son críticas. El diseño de un servicio de archivos debe soportar muchos de los requisitos de transparencia para sistemas distribuidos identificados en

la Sección 1.4.7. El diseño debe balancear la flexibilidad y escalabilidad que se derivan de la transparencia frente a la complejidad del software y las prestaciones. Las siguientes formas de transparencia son parcial o totalmente tratadas por los actuales servicios de archivos:

Transparencia de acceso: los programas del cliente no deben preocuparse de la distribución de los archivos. Se proporciona un conjunto sencillo de operaciones para el acceso a archivos locales y remotos. Los programas escritos para trabajar sobre archivos locales serán capaces de acceder a los archivos remotos sin modificación.

Transparencia de ubicación: los programas del cliente deben ver un espacio de nombres de archivos uniforme. Los archivos o grupos de archivos pueden ser reubicados sin cambiar sus nombres de ruta, y los programas de usuario verán el mismo espacio de nombres en cualquier parte que sean ejecutados.

Transparencia de movilidad: ni los programas del cliente ni las tablas de administración de sistema en los nodos cliente necesitan ser cambiados cuando se mueven los archivos. Esta movilidad de archivos permite que archivos o, más comúnmente, conjuntos o volúmenes de archivos puedan ser movidos, ya sea por los administradores del sistema o automáticamente.

Transparencia de prestaciones: los programas cliente deben continuar funcionando satisfactoriamente mientras la carga en el servicio varíe dentro de un rango especificado.

Transparencia de escala: el servicio puede ser aumentado por un crecimiento incremental para tratar con un amplio rango de cargas y tamaños de redes.

◊ **Actualizaciones concurrentes de archivos.** Los cambios en un archivo por un cliente no deben interferir con la operación de otros clientes que acceden o cambian simultáneamente el mismo archivo. Esto es el tema bien conocido del control de concurrencia, discutido con detalle en el Capítulo 12. La necesidad de control de concurrencia para el acceso a datos compartidos en muchas aplicaciones está ampliamente aceptada y las técnicas para su implementación son conocidas, aunque muy costosas. La mayoría de los servicios de archivos actuales siguen los estándares de UNIX moderno proporcionando bloqueo consultivo u obligatorio a nivel de archivo o registro.

◊ **Replicación de archivos.** En un servicio de archivos que soporta replicación, un archivo puede estar representado por varias copias de su contenido en diferentes ubicaciones. Esto tiene dos beneficios, permite que múltiples servidores compartan la carga de proporcionar un servicio a los clientes que acceden al mismo conjunto de archivos, mejorando la escalabilidad del servicio y mejorando la tolerancia a fallos, permitiendo a los clientes localizar otro servidor que mantiene una copia del archivo cuando uno ha fallado. Muy pocos servicios de archivos soportan totalmente la replicación, pero la mayoría soportan la caché local de archivos o porciones de archivos, una forma limitada de replicación. La replicación de datos se discutirá en el Capítulo 14, que incluye una descripción del servicio de archivos replicado Coda.

◊ **Heterogeneidad del hardware y del sistema operativo.** Las interfaces del servicio deben estar definidas de modo que el software del cliente y el servidor pueden estar implementados por diferentes sistemas operativos y computadores. Este requisito es un aspecto importante de la extensibilidad.

◊ **Tolerancia a fallos.** El papel central de un servicio de archivos en los sistemas distribuidos hace que sea esencial que el servicio continúe funcionando aun en el caso de fallos del cliente y del servidor. Afortunadamente un diseño moderadamente tolerante a fallos es inmediato para servidores sencillos. Para manejar los fallos de comunicación transitorios, el diseño puede estar basado en la semántica de invocación de como *máximo una vez* (véase la Sección 5.2.4). O puede utilizarse la semántica más sencilla *al menos una vez* con un protocolo de servidor diseñado en términos de operaciones *idempotentes*, asegurando que solicitudes duplicadas no producen actualizaciones inválidas en los archivos. Los servidores pueden ser *sin estado*, por lo que pueden ser rearrancados

y el servicio restablecido después de un fallo sin necesidad de recuperar el estado previo. La tolerancia a la desconexión o fallos del servidor precisa replicación de los archivos, que es más difícil de alcanzar y será discutida en el Capítulo 14.

◊ **Consistencia.** Los sistemas de archivos convencionales, como los que se proporcionan en UNIX, ofrecen una *semántica de actualización de una copia*. Esto se refiere a un modelo para acceso concurrente a archivos en el que el contenido del archivo visto por todos los procesos que acceden o actualizan a un archivo dado es aquel que ellos verían si existiera un única copia del contenido del archivo. Cuando los archivos están replicados, o en la caché, en diferentes lugares, hay un retardo inevitable en la propagación de las modificaciones hechas en un lugar hacia los otros lugares que mantienen copias, y esto puede producir alguna desviación de la semántica de una copia.

◊ **Seguridad.** Virtualmente todos los sistemas de archivos proporcionan mecanismos de control de acceso basados en el uso de listas de control de acceso. En sistemas de archivos distribuidos, hay una necesidad de autenticar las solicitudes del cliente por lo que el control de acceso en el servidor está basado en identificar al usuario correcto y proteger el contenido de los mensajes de solicitud y respuesta con firmas digitales y (opcionalmente) encriptación de datos secretos. Discutiremos el impacto de estos requisitos en nuestras descripciones de casos de estudio.

◊ **Eficiencia.** Un servicio de archivos distribuidos debe ofrecer posibilidades con la misma potencia y generalidad que las que se encuentran en los sistemas de archivos convencionales y deben proporcionar un nivel de prestaciones comparable. Birrell y Needham [1980] expresaron sus objetivos de diseño para el Servidor de Archivos Cambridge (CFS) en estos términos:

Nosotros deseáramos tener un servidor de archivos sencillo, de bajo nivel, para compartir un recurso caro, por ejemplo un disco, mientras dejamos libertad para diseñar el sistema de archivos más apropiado para un cliente particular, pero deseáramos también tener disponible un sistema de alto nivel compartido entre clientes.

El cambio del coste de almacenamiento de disco ha reducido la importancia de su primer objetivo, pero su percepción de la necesidad de un rango de servicios que respondan a los requisitos de los clientes con diferentes objetivos, permanece y puede ser conseguido mejor por una arquitectura modular del tipo esbozado anteriormente.

Las técnicas utilizadas para la implementación de los servicios de archivo son una parte importante del diseño de sistemas distribuidos. Un sistema de archivos distribuidos debe proporcionar un servicio que sea comparable con, o mejor que, los sistemas de archivos locales en prestaciones y fiabilidad. Debe ser adecuado para administrar, proporcionando operaciones y herramientas que permitan a los administradores del sistema instalar y operar el sistema convenientemente.

8.1.3. CASOS DE ESTUDIO

Hemos construido un modelo abstracto para un servicio de archivos para actuar como un ejemplo introductorio separando las cuestiones de implementación y proporcionando un modelo simplificado. Describimos el Sistema de Archivos en Red (*Network File System*) de Sun (NFS) con algún detalle, basándonos en nuestro modelo abstracto más sencillo para aclarar su arquitectura. Posteriormente se describe el Sistema de Archivos Andrew (AFS), proporcionando una visión de un sistema de archivos distribuido que considera una aproximación diferente para la escalabilidad y el mantenimiento de la consistencia.

◊ **Arquitectura del servicio de archivos.** Éste es un modelo arquitectónico abstracto sobre el que se sustentan tanto NFS como AFS. Está basado en una división de las responsabilidades entre tres módulos, un módulo cliente que emula la interfaz de un sistema de archivos convencio-

nal para los programas de aplicación y módulos servidores, que realizan operaciones para los clientes en directorios y archivos. La arquitectura está diseñada para permitir una implementación *sin estado* del módulo del servidor.

◊ **NFS de SUN.** El sistema de archivos en red de SUN (NFS) ha sido adoptado ampliamente en la industria y en entornos académicos desde su introducción en 1985. El diseño y desarrollo de NFS fueron emprendidos por el personal de Sun Microsystems en 1984 [Sandberg y otros 1985; Sandberg 1987, Callaghan 1999]. Aunque ya habían sido desarrollados varios servicios de archivos distribuidos, y utilizados con éxito, en universidades y laboratorios de investigación, NFS fue el primer servicio de archivos que fue diseñado como un producto. El diseño e implementación de NFS ha obtenido un éxito considerable tanto técnica como comercialmente.

Para animar a su adopción como un estándar, las definiciones de las interfaces fundamentales fueron situadas en el dominio público [Sun 1989], permitiendo a otros vendedores producir implementaciones, y el código fuente fue puesto disponible para una implementación de referencia a otros vendedores de computadores bajo licencia. Actualmente está soportado por muchos vendedores y el protocolo NFS (versión 3) es un estándar de Internet, definido en RFC 1813 [Callaghan y otros 1995]. El libro del mismo Callaghan sobre NFS [Callaghan 1999] es una fuente excelente sobre el diseño y desarrollo de NFS y temas relacionados.

NFS proporciona acceso transparente a archivos remotos desde programas cliente ejecutándose sobre UNIX y otros sistemas. Normalmente, cada computador tiene un cliente NFS y módulos servidor instalados en el núcleo del sistema, al menos en el caso de los sistemas UNIX. La relación cliente-servidor es simétrica: Cada computador en una red NFS puede actuar tanto como cliente como servidor, y los archivos en cada máquina pueden hacerse disponibles para acceso remoto desde otras máquinas. Cualquier computador puede ser un servidor, exportando algunos de sus archivos, y un cliente, accediendo a archivos de otras máquinas. Pero es una práctica habitual configurar instalaciones grandes con algunas máquinas como servidores dedicados y otras como estaciones de trabajo.

Un objetivo importante de NFS es conseguir un elevado nivel de soporte para la heterogeneidad de hardware y el sistema operativo. El diseño es independiente del sistema operativo: Existen implementaciones de servidor y cliente para casi todos los sistemas operativos y plataformas actuales, incluyendo Windows 95, Windows NT, MacOS y VMS así como Linux y casi cualquier otra versión de UNIX. Se han desarrollado implementaciones de NFS en máquinas multiprocesador de altas prestaciones por varios vendedores y éstas se utilizan ampliamente para satisfacer los requisitos de almacenamiento en intranets con muchos usuarios concurrentes.

◊ **Andrew File System.** Andrew es un entorno de computación distribuida desarrollado en la Universidad Carnegie Mellon (CMU) para su utilización como un sistema de información y computación de campus [Morris y otros 1986]. El diseño de Andrew File System (a partir de ahora abreviado como AFS) refleja una intención de soportar la compartición de información en gran escala minimizando la comunicación cliente-servidor. Esto fue conseguido transfiriendo archivos completos (o para archivos grandes, trozos de 64-kbytes) entre los computadores del servidor y del cliente y haciendo caché de ellos en los clientes hasta que el servidor reciba una versión más actualizada. La red de computación de campus que servía Andrew se esperaba que creciera hasta incluir entre 5.000 y 10.000 estaciones de trabajo durante el tiempo de vida del sistema. Describiremos AFS-2, la primera implementación de producción, siguiendo las descripciones de Satyanarayanan [1989a, 1989b]. Se pueden encontrar descripciones más recientes en Campbell [1997] y [Linux AFS].

AFS fue implementado inicialmente sobre una red de estaciones de trabajo y servidores trabajando en BSD UNIX y el sistema operativo Mach en CMU y posteriormente estuvo disponible en versiones comerciales y de dominio público. Más recientemente, se ha hecho disponible una implementación de dominio público de AFS para el sistema operativo Linux [Linux AFS]. En 1991, AFS

estaba soportando aproximadamente 800 estaciones de trabajo servidas por aproximadamente 40 servidores en la CMU, con clientes y servidores adicionales en lugares remotos conectados a través de Internet. AFS fue adoptado como la base para el sistema de archivos DCE/DFS en el entorno de computación distribuido (DCE) de la Open Software Foundation [www.open-group.org]. El diseño de DCE/DFS fue más allá de AFS en varios aspectos importantes, que señalamos en la Sección 8.5.

8.2. ARQUITECTURA DEL SERVICIO DE ARCHIVOS

El alcance de la extensibilidad y configurabilidad se mejora si el servicio de archivos se estructura en tres componentes, un *servicio de archivos plano*, un *servicio de directorio* y un *módulo cliente*. Los módulos relevantes y sus relaciones se ven en la Figura 8.5. El servicio de archivos plano y el servicio de directorio exportan cada uno una interfaz, para su uso por los programas del cliente y sus interfaces RPC, consideradas conjuntamente, proporcionan un conjunto global de operaciones para acceso a archivos. El módulo cliente proporciona una interfaz de programación sencilla con operaciones sobre archivos semejantes a las encontradas en los sistemas de archivos convencionales. El diseño es *abierto* en el sentido que pueden utilizarse diferentes módulos cliente para implementar diferentes interfaces de programación, simulando las operaciones sobre archivos de una variedad de diferentes sistemas operativos y optimizando las prestaciones para diferentes configuraciones de hardware del cliente y el servidor.

La división de las responsabilidades entre los módulos puede definirse como sigue:

◊ **Servicio de archivos planos.** El servicio de archivos planos está relacionado con la implementación de operaciones en el contenido de los archivos. Se utilizan *identificadores únicos de archivos* (UFID) para referirse a los archivos en todas las solicitudes de operaciones del servicio de archivos plano. La división de responsabilidades entre el servicio de archivos y el servicio de directorio, está basada en la utilización de UFID. Cada UFID es una secuencia larga de bits elegidas de forma que cada archivo tiene un UFID que es único entre todos los archivos en un sistema distribuido. Cuando el servicio de archivos planos recibe una solicitud para crear un archivo, genera un nuevo UFID para él y lo devuelve al solicitante.

◊ **Servicio de directorio.** El servicio de directorio proporciona una transformación entre *nombres de texto* para los archivos y sus UFID. Los clientes pueden obtener el UFID de un archivo indicando su nombre de texto al servicio de directorio. El servicio de directorio proporciona las

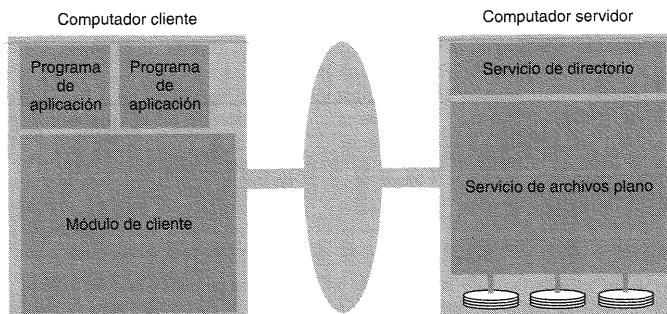


Figura 8.5. Arquitectura del servicio de archivos.

funciones necesarias para generar directorios, para añadir nuevos nombres de archivo a los directorios y para obtener UFID desde los directorios. Es un cliente del servicio de archivos plano, sus archivos de directorio están almacenados en archivos del servicio de archivos plano. Cuando se adopta un esquema jerárquico de nominación de archivos, como en UNIX, los directorios mantienen referencias a otros directorios.

◊ **Módulo cliente.** En cada computador cliente se ejecuta un módulo de cliente, que integra y extiende las operaciones del servicio de archivos plano y el servicio de directorio bajo una interfaz de programación de aplicaciones sencilla, que estará disponible para los programas a nivel de usuario en los computadores cliente. Por ejemplo, en máquinas UNIX, se debe proporcionar un módulo cliente que emula el conjunto total de operaciones UNIX sobre archivos, interpretando los nombres *compuestos* de archivos UNIX mediante reiteradas solicitudes al servicio de directorio. El módulo cliente mantiene también información sobre las ubicaciones en la red del proceso servidor de archivos planos y del proceso servidor de directorio. Finalmente el módulo cliente puede jugar un papel importante consiguiendo unas prestaciones satisfactorias mediante la implementación de una caché de los bloques de archivos utilizados recientemente en el cliente.

◊ **Interfaz del servicio de archivos plano.** La Figura 8.6 contiene una definición de la interfaz para un servicio de archivos planos. Es la interfaz RPC utilizada por los módulos cliente. No es utilizada directamente por los programas a nivel de usuario, normalmente. Un *IdArchivo* es inválido si el archivo a que se refiere no está presente en el servidor que procesa la solicitud o si sus permisos de acceso son inapropiados para la operación solicitada. Todos los procedimientos de la interfaz excepto *Crea* lanzan excepciones si el argumento *IdArchivo* contiene un UFID inválido o el usuario no tiene los derechos de acceso suficientes. Estas excepciones se han omitido por la claridad de la definición.

Las operaciones más importantes son las de lectura y escritura. Tanto la operación de lectura como la de escritura necesitan un parámetro *i* especificando una posición en el archivo. La operación de lectura copia la secuencia de *n* elementos de datos comenzando en el elemento *i* del archivo especificado en *Datos* que se devuelve entonces al cliente. La operación *Escribe* copia la secuencia de elementos de datos de *Datos* en el archivo especificado comenzando en el elemento *i*, reemplazando el contenido anterior del archivo en la posición correspondiente y extendiendo el archivo si es necesario.

Crea crea un archivo nuevo, vacío y devuelve el UFID que se ha generado. *Elimina* borra el archivo especificado.

DameAtributos y *PonAtributos* permiten a los usuarios acceder al registro de los atributos. *DameAtributos* está disponible generalmente para cualquier usuario que acceda al archivo. El acceso a la operación *PonAtributos* suele estar restringida normalmente al servicio de directorio que accede

<i>Lee(IdArchivo, i, n) → Datos</i> — lanza <i>MalaPosición</i>	Si $1 \leq i \leq \text{Tamaño}(\text{Archivo})$: Lee una secuencia de hasta <i>n</i> elementos del archivo, partiendo del elemento <i>i</i> y la devuelve en <i>Datos</i> .
<i>Escribe(IdArchivo, i, Datos)</i> — lanza <i>MalaPosición</i>	Si $1 \leq i \leq \text{Tamaño}(\text{Archivo}) + 1$: Escribe una secuencia de <i>Datos</i> sobre el archivo, partiendo del elemento <i>i</i> , y extendiendo el archivo si es preciso.
<i>Crea() → IdArchivo</i>	Crea un nuevo archivo de tamaño 0 y obtiene un UFID para él.
<i>Elimina(IdArchivo)</i>	Elimina el archivo del almacén de archivos.
<i>DameAtributos(IdArchivo) → Atrib</i>	Obtiene los atributos del archivo.
<i>PonAtributos(IdArchivo, Atrib)</i>	Pone los atributos del archivo (sólo aquellos que le están permitidos en la Figura 8.3)

Figura 8.6. Operaciones del servicio de archivos plano.

al archivo. Los valores de la longitud y de las partes de marcas de tiempo del registro de atributos no son afectadas por *PonAtributos*, son mantenidas por el servicio de archivos planos.

Comparación con UNIX: Nuestra interfaz y las primitivas del sistema de archivos UNIX son funcionalmente equivalentes. No es complicado construir un módulo cliente que emule las llamadas al sistema UNIX en términos de las operaciones de nuestro servicio de archivos planos y el servicio de directorio descritas en la sección siguiente.

En comparación con la interfaz UNIX, nuestro servicio de archivos plano no tiene operaciones *open* y *close*, los archivos pueden ser accedidos inmediatamente proporcionando el UFID adecuado. Las solicitudes de *Lee* y *Escribe* de nuestra interfaz incluyen un parámetro que especifica el punto de partida para cada transferencia desde el archivo, mientras que las operaciones UNIX equivalentes no. En UNIX cada operación *read* y *write* comienza en la posición actual del apuntador de *lectura escritura*, y dicho apuntador se desplaza en una cantidad correspondiente al número de bytes transferidos después de cada lectura y escritura. Se proporciona una operación *seek* para permitir que dicho apuntador de *lectura escritura* sea posicionado explícitamente.

La interfaz de nuestro servicio de archivos planos difiere de la del sistema de archivos de UNIX fundamentalmente por razones de tolerancia a fallos:

Operaciones repetibles: con la excepción de *Crea*, las operaciones son *idempotentes*, permitiendo la utilización de la semántica de *al menos una vez* de RPC, los clientes pueden repetir las llamadas de las que no han recibido contestación. La ejecución repetida de *Crea* produce un archivo diferente en cada llamada.

Servidores sin estado: la interfaz es adecuada para su implementación por servidores sin estado (*stateless*). Los servidores sin estado pueden ser rearrancados después de un fallo y reanudar la operación sin necesitar que ni los clientes ni el servidor restablezcan su estado.

Las operaciones sobre archivos de UNIX no son ni idempotentes ni consistentes con los requisitos para una implementación sin estado. Cuando se abre un archivo se genera un apuntador de *lectura escritura* desde el sistema de archivos de UNIX, y se mantiene, junto con las comprobaciones de los resultados de control de acceso, hasta que se cierra el archivo. Las operaciones de lectura y escritura de UNIX no son idempotentes, si una operación se repite accidentalmente, el avance automático del apuntador de *lectura escritura* produce el acceso a una porción diferente del archivo en la operación repetida. El apuntador del *lectura escritura* es una variable de estado oculta, relacionada con el cliente. Para imitar su comportamiento en un servidor de archivos, se necesitan las operaciones de *open* y *close*, y hay que mantener el valor del apuntador de *lectura escritura* en tanto en cuanto el archivo considerado esté abierto. Eliminando el apuntador de *lectura escritura*, hemos eliminado la necesidad de que un servidor de archivos retenga información sobre el estado en nombre de clientes específicos.

◊ **Control de acceso.** En el sistema de archivos UNIX, se comprueban los derechos de acceso del usuario frente al *modo* de acceso (lectura o escritura) solicitado en la operación de llamada (la Figura 8.4 muestra la interfaz de aplicaciones de UNIX) y el archivo es abierto sólo si el usuario tiene los derechos de acceso necesarios. La identidad del usuario (UID) utilizada en la comprobación de los derechos de acceso es el resultado del *login* autenticado anterior del usuario y no puede ser alterado en las implementaciones no distribuidas. Los derechos de acceso resultantes se retienen hasta que el archivo se cierra y no se necesitan otras comprobaciones en las operaciones posteriores solicitadas sobre el mismo archivo.

En las implementaciones distribuidas, la comprobación de los derechos de acceso se ha de realizar en el servidor porque la interfaz del servidor RPC es un punto desprotegido de acceso para los archivos. La identidad del usuario ha de pasarse con cada solicitud, y el servidor se hace vulnerable a identidades antiguas. Además si se retuvieran en el servidor los resultados de una comprobación de derechos de acceso, y se usaran para otros acceso, el servidor ya no sería sin estado. Se pueden adoptar dos aproximaciones alternativas a este último problema:

ción de derechos de acceso, y se usarán para otros acceso, el servidor ya no sería sin estado. Se pueden adoptar dos aproximaciones alternativas a este último problema:

- Se realiza una comprobación de acceso cuando se convierte un nombre de archivo en un UFID, y los resultados se codifican en forma de una habilitación (ver Sección 7.2.4), que se devuelve al cliente para su envío con las solicitudes posteriores.
- Se envía la identidad del usuario con cada solicitud del cliente, y las comprobaciones de acceso se realizan en el servidor para cada operación sobre el archivo.

Ambos métodos permiten una implementación de servidor sin estado, y ambas se han utilizado en los sistemas de archivos distribuidos. El segundo es más común, y se utiliza tanto en NFS como en AFS. Ninguna de estas aproximaciones resuelve el problema de seguridad relacionado con las identidades olvidadas del usuario. Hemos visto en el Capítulo 7 que este problema puede abordarse mediante el uso de firmas digitales. Kerberos es un esquema de autenticación efectivo que ha sido aplicado tanto en NFS como en AFS.

En nuestro modelo abstracto, no hacemos suposiciones sobre el método con el que debe estar implementado el control de acceso. La identidad del usuario se pasa como un parámetro implícito y puede utilizarse cuando sea preciso.

◊ **Interfaz del servicio de directorio.** La Figura 8.7 contiene una definición de la interfaz RPC para un servicio de directorio. El propósito principal del servicio de directorio es proporcionar un servicio para trasladar nombres a UFID. Con este fin, se mantienen archivos de directorios que contienen las equivalencias entre los nombres de los archivos y cada UFID. Cada directorio se almacena en un archivo convencional con una UFID, de forma que el servicio de directorio es un cliente del servicio de archivos.

Definimos únicamente las operaciones sobre directorios individuales. Para cada operación, se precisa una UFID para el archivo que contiene el directorio (en el parámetro *Dir*). La operación *Busca* en el servicio básico de directorios realiza una traducción sencilla *Nombre* → *UFID*. Es un bloque básico para su uso en otros servicios o para realizar traducciones más complejas en el módulo del cliente, como la interpretación jerárquica de nombres encontrada en UNIX. Como antes, las excepciones producidas por derechos de acceso no adecuados se omiten de las definiciones.

Hay dos operaciones para modificar directorios: *AñadeNombre* y *DesNombre*. *AñadeNombre* añade una entrada al directorio e incrementa el campo de contador de referencias en el registro de atributos del archivo.

DesNombre elimina una entrada de un directorio y decremente el contador de referencias. Si esto produce que el contador de referencias llegue a cero, se elimina el archivo. *DameNombres* se proporciona para permitir a los clientes examinar el contenido de los directorios y para implementar operaciones de concordancia de patrones de nombres de archivos como las que se encuentran en el intérprete de órdenes de UNIX. Devuelven todos o un subconjunto de los nombres almacenados.

<i>Busca</i> (<i>Dir</i> , <i>Nombre</i>) → <i>IdArchivo</i>	Busca el texto <i>Nombre</i> en el directorio y devuelve el UFID relevante. Si <i>Nombre</i> no está en el directorio, lanza una excepción.
— lanza <i>NoEncontrado</i>	
<i>AñadeNombre</i> (<i>Dir</i> , <i>Nombre</i> , <i>Archivo</i>)	Si <i>Nombre</i> no está en el directorio, añade (<i>Nombre</i> , <i>Archivo</i>) al directorio y actualiza el registro de atributos de archivo.
— lanza <i>NombreDuplicado</i>	
<i>DesNombre</i> (<i>Dir</i> , <i>Nombre</i>)	Si <i>Nombre</i> está en el directorio: la entrada que contiene <i>Nombre</i> se elimina del directorio.
— lanza <i>NoEncontrado</i>	Si <i>Nombre</i> no está en el directorio: lanza una excepción.
<i>DameNombres</i> (<i>Dir</i> , <i>Patrón</i>) → <i>SecNombres</i>	Devuelve todos los nombres del directorio que concuerdan con la expresión regular <i>Patrón</i> .

Figura 8.7. Operaciones del servicio de directorio.

dos en un directorio dado. Los nombres son seleccionados mediante comparación de patrones frente a una expresión regular proporcionada por el cliente.

La disponibilidad de reconocimiento de patrones en las operación *DameNombres* permite a los usuarios determinar los nombres de uno o más archivos al proporcionar una especificación incompleta de los caracteres de los nombres. Una expresión regular es una especificación de una clase de cadenas de caracteres en forma de una expresión que contiene una combinación de subcadenas literales y símbolos que representan caracteres variables u ocurrencias repetidas de caracteres o subcadenas.

◊ **Sistema de archivos jerárquico.** Un servicio de archivos jerárquico, como uno de los que proporciona UNIX, consiste en un número de directorios organizados en una estructura de árbol. Cada directorio contiene los nombres de los archivos y otros directorios que son accesibles desde él. Cualquier archivo o directorio puede ser referenciado con un nombre de ruta (*path*), un nombre compuesto de varias partes que representa una ruta a través del árbol. La raíz tiene un nombre que le distingue a cada archivo o directorio tiene un nombre en un directorio. El esquema de nominación de archivos de UNIX no es una jerarquía estricta, los archivos pueden tener varios nombres y pueden estar en varios directorios. Esto se implementa mediante una operación de enlace (*link*), que añade un nuevo nombre para un archivo en un directorio especificado.

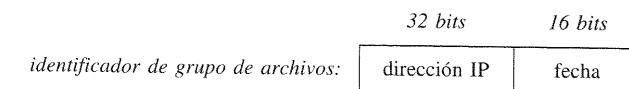
Un sistema de nominación de archivos como el de UNIX puede ser implementado desde el módulo cliente utilizando los servicios de directorio y archivos planos que hemos definido. Se construye una red de directorios estructurados en forma de árbol con los archivos en las hojas y los directorios en los otros nodos del árbol. La raíz del árbol es un directorio con una UFID *bien conocida*. Se da soporte a varios nombres un archivo utilizando la operación *AñadeNombre* y el campo de contador del registro de atributos.

Se puede proporcionar una función en el módulo del cliente que consiga la UFID de un archivo dado un nombre de ruta. La función interpreta el nombre de ruta comenzando desde la raíz, utilizando *Busca* para obtener la UFID de cada directorio en la ruta.

En un servicio jerárquico de archivos, los atributos asociados con los archivos deben incluir un tipo que distingue a los archivos ordinarios de los directorios. Esto se utiliza cuando se sigue una ruta para garantizar que cada parte del nombre, excepto la última, se refiere a un directorio.

◊ **Agrupación de archivos.** Un *grupo de archivos* es una colección de archivos ubicada en un servidor dado. Un servidor puede mantener varios grupos de archivos y los grupos pueden ser reubicados entre servidores, pero un archivo no puede cambiar el grupo al que pertenece. Una construcción similar (llamada *filesystem-volumen*) se utiliza en UNIX y en la mayoría de los sistemas operativos. Los grupos de archivos se introdujeron inicialmente para trasladar colecciones de archivos almacenados en cartuchos de disco extraíbles entre computadores. En un servicio de archivos distribuidos, los grupos de archivos permiten la asignación de archivos a servidores de archivos almacenados en varios servidores. En un sistema de archivos distribuidos que soporta grupos de archivos, la representación de UFID incluye un componente identificador de grupo de archivos, permitiendo al módulo cliente en cada computador cliente tomar la responsabilidad de enviar las solicitudes al servidor que mantiene el grupo de archivos relevante.

Los identificadores de grupo de archivos deben ser únicos a lo largo de un sistema distribuido. Puesto que se pueden trasladar grupos de archivos, y los sistemas distribuidos que están separados inicialmente se pueden mezclar para formar un sistema único, la única forma de asegurar que los identificadores de grupo serán siempre distintos en un sistema dado es generarlos con un algoritmo que garantice la unicidad global. Por ejemplo, cuando se crea un nuevo grupo de archivos, se puede generar un identificador único de archivos concatenando la dirección IP de 32 bits de la máquina que crea el nuevo grupo con un entero de 16 bits derivado de la fecha, produciendo un entero único de 48 bits:



Observe que la dirección IP no debe utilizarse para el propósito de localizar el grupo de archivos, puesto que puede ser trasladado a otro servidor. En lugar de eso, hay que mantener en el servicio de archivos una traducción entre identificadores de grupo y servidores.

8.3. SISTEMA DE ARCHIVOS EN RED DE SUN (NFS)

La Figura 8.8 representa la arquitectura de NFS de Sun. Sigue el modelo abstracto definido en la sección precedente. Todas las implementaciones de NFS soportan el protocolo de NFS: un conjunto de llamadas a procedimientos remotos que proporcionan el medio para que los clientes realicen operaciones en un almacén de archivos remotos. El protocolo NFS es independiente del sistema operativo pero fue desarrollado originalmente para su utilización en redes de sistemas UNIX, y nosotros describiremos la implementación del protocolo NFS en UNIX (versión 3).

El módulo *servidor NFS* reside en el núcleo de cada computador que actúa como un servidor NFS. Las solicitudes que se refieren a archivos en un sistema de archivos remoto se traducen en el módulo cliente a operaciones del protocolo NFS y después se trasladan al módulo servidor NFS en el computador que mantiene el sistema de archivos relevante.

Los módulos cliente y servidor NFS se comunican utilizando llamadas a procedimientos remotos. El sistema RPC de SUN, descrito en la Sección 5.3.1, se desarrolló para su uso en NFS. Puede configurarse para utilizar UDP o TCP, y el protocolo NFS es compatible con ambos. Se incluye un servicio de enlace que permite a los clientes encontrar los servicios de una máquina a partir del nombre. La interfaz RPC para el servidor NFS es abierta: cualquier proceso puede enviar solicitudes a un servidor NFS. Si las solicitudes son válidas e incluyen credenciales válidas del usuario, serán ejecutadas. El envío de credenciales firmadas del usuario puede requerirse como una característica adicional de seguridad, como lo puede ser la encriptación de los datos para privacidad e integridad.

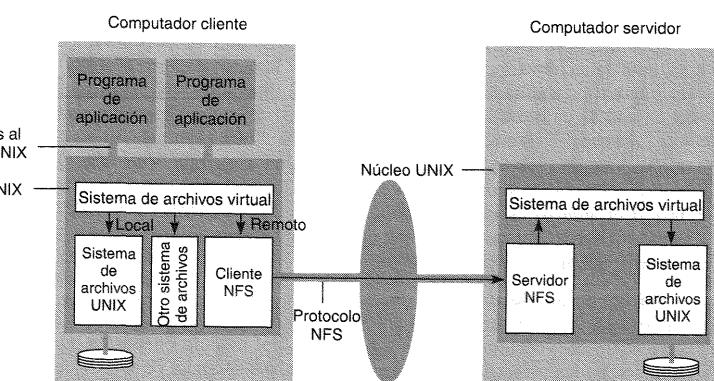


Figura 8.8. Arquitectura NFS.

◊ **Sistema de archivos virtuales.** La Figura 8.8 expone claramente que NFS proporciona acceso transparente: los programas del usuario pueden realizar operaciones sobre los archivos locales o remotos sin distinción. Otros sistemas de archivos distribuidos pueden considerar que permiten las llamadas al sistema de UNIX, y si lo hacen, podrían integrarse de la misma forma.

La integración se obtiene mediante un módulo de sistema de archivos virtual (VFS), que ha sido añadido al núcleo de UNIX para distinguir entre archivos remotos y locales y para traducir los identificadores de archivo, independientes de UNIX, utilizados por NFS en los identificadores de archivo internos utilizados en UNIX y otros sistemas de archivos. En resumen, VFS mantiene la pista de los sistemas de archivos que están actualmente disponibles tanto local como remotamente, pasa cada solicitud al módulo del sistema local apropiado (el sistema de archivos UNIX, el módulo cliente NFS o el módulo de servicio de otro sistema de archivos).

Los identificadores de archivo utilizados en NFS se llaman *apuntadores de archivo* (*file handles*). Un apuntador de archivo es opaco para los clientes y contiene toda la información que necesita el servidor para distinguir un archivo individual. En las implementaciones UNIX de NFS, el apuntador de archivo se deriva del *número de i-nodo* del archivo, añadiendo dos campos extra como sigue (el número de i-nodo de un archivo UNIX es un número que sirve para identificar y localizar el archivo en el sistema de archivos en el que está almacenado):

Apuntador de archivo:	Identificador del filesystem	Número de i-nodo del archivo	Número de generación de i-nodos
-----------------------	------------------------------	------------------------------	---------------------------------

NFS adopta el *filesystem* (volumen) montable de UNIX como la unidad de agrupación de archivos definida en la sección precedente. (Nota de terminología: el término *filesystem* —*volumen*— se refiere al conjunto de archivos mantenidos en un dispositivo de almacenamiento o partición, mientras que las palabras sistema de archivos se refieren al componente software que proporciona el acceso a los mismos). El campo *identificador del volumen* es un número único que se reserva para cada volumen cuando se crea (y en la implementación UNIX se almacena en el superbloque del sistema de archivos). La *generación del número de i-nodo* es necesaria porque en el sistema de archivos normal de UNIX los números de i-nodo se reutilizan después de la eliminación del archivo. En las extensiones VFS al sistema de archivos UNIX, se almacena un número de generación con cada archivo y se incrementa cada vez que se reutiliza el número de i-nodo (por ejemplo, en una llamada *creat* del sistema UNIX). El cliente obtiene el primer apuntador de archivo para un sistema de archivos remotos cuando lo monta. Los apuntadores de archivos se pasan del servidor al cliente en los resultados de las operaciones *lookup*, *create* y *mkdir* (consultar la Figura 8.9) y del cliente al servidor en la lista de argumentos de todas las operaciones del servidor.

La capa del sistema de archivos virtual tiene una estructura VFS por cada sistema de archivos montado y un *v-nodo* por archivo abierto. Una estructura VFS relaciona un sistema de archivos remoto con el directorio local en el que está montado. El *v-nodo* contiene un indicador para mostrar si el archivo es local o remoto. Si el archivo es local, el *v-nodo* contiene una referencia al índice del archivo local (un i-nodo en una implementación UNIX). Si el archivo es remoto, contiene un apuntador al archivo remoto.

◊ **Integración del cliente.** El cliente NFS juega el papel descrito para el módulo cliente en nuestro modelo arquitectónico, proporcionando una interfaz idónea para su uso por programas de aplicación convencionales. Pero a diferencia del módulo cliente de nuestro modelo, emula la semántica de las primitivas del sistema de archivos estándar de UNIX de forma precisa y está integrado con el núcleo UNIX. Está integrado con el núcleo y no proporcionado como una biblioteca para cargar en los procesos clientes de modo que:

- Los programas de usuario pueden acceder a los archivos mediante llamadas del sistema de UNIX sin recompilación o recarga.
- Un único módulo cliente sirve a todos los procesos del nivel del usuario, con una caché compartida de los bloques utilizados recientemente (se describirán posteriormente).
- La clave de encriptación utilizada para autenticar las ID del usuario pasadas al servidor (ver más adelante) puede retenerse en el núcleo, previniendo la impersonación por clientes a nivel de usuario.

El módulo cliente de NFS coopera con el sistema de archivos virtual en cada máquina cliente. Funciona de una manera semejante al sistema de archivos convencional de UNIX, transfiriendo bloques de archivos hacia y desde el servidor y haciendo *caching* de los bloques en la memoria local cuando es posible. Comparte el mismo búfer de caché que el utilizado por el sistema de entrada-salida local. Pero puesto que varios clientes en diferentes máquinas pueden acceder simultáneamente al mismo archivo remoto, se plantea un nuevo y significativo problema de consistencia de caché.

◊ **Control de acceso y autenticación.** A diferencia del sistema de archivos convencional UNIX, el servidor NFS es sin estado y no mantiene archivos abiertos en nombre de sus clientes. Por lo tanto el servidor debe comprobar la identidad del usuario frente a los atributos de acceso del archivo en cada solicitud, para ver si el usuario tiene permiso de acceso al archivo de la forma solicitada. El protocolo Sun RPC requiere que los clientes envíen la información de autenticación del usuario (por ejemplo, los convencionales 16 bits de la ID del usuario y del grupo de UNIX) con cada solicitud y ésta se comprueba frente a los permisos de acceso en los atributos del archivo. Estos parámetros adicionales no se muestran en nuestra revisión del protocolo NFS de la Figura 8.9, vienen dados automáticamente por el sistema RPC.

En la forma más simple, hay una laguna de seguridad en este mecanismo de control de acceso. Un servidor NFS proporciona una interfaz RPC convencional en un puerto bien conocido en cada máquina y cualquier proceso puede comportarse como un cliente, enviando solicitudes al servidor para acceder o actualizar un archivo. El cliente puede modificar las llamadas RPC para incluir la ID de cualquier usuario, haciéndose pasar por el usuario sin su conocimiento o permiso. Esta laguna de seguridad ha sido cerrada con el uso de una opción en el protocolo RPC para la encriptación de la información de autenticación del usuario. Más recientemente, se ha integrado Kerberos en Sun NFS para proporcionar una solución más fuerte y completa a los problemas de autenticación y seguridad del usuario, que nosotros describimos más adelante.

◊ **Interfaz del servidor NFS.** En la Figura 8.9 se ve una representación simplificada de la interfaz RPC proporcionada por el servidor NFS (definida en RFC 1813 [Callaghan y otros 1995]). Las operaciones NFS de acceso al archivo *read*, *write*, *getattr* y *setattr* son casi idénticas a las operaciones *Lee*, *Escribe*, *DameAtributos* y *PonAtributos* definidas en nuestro modelo de servicio de archivos planos (Figura 8.6). La operación *lookup* y la mayoría de las demás operaciones de directorio definidas en la Figura 8.9 son semejantes a las de nuestro modelo de servicios de directorio (Figura 8.7).

Las operaciones de archivo y directorio están integradas en un único servicio, la creación e inserción de nombres de archivo en directorios se realiza en una única operación *create* que toma el nombre del nuevo archivo y el apuntador de archivo del directorio destino (*adir*) como argumentos. Las otras operaciones NFS sobre directorios son *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* y *statfs*. Se parecen a sus equivalentes UNIX con la excepción de *readdir*, que proporciona un método independiente de la representación para leer el contenido de directorios, y *statfs*, que proporciona la información del estado en los sistemas de archivos remotos.

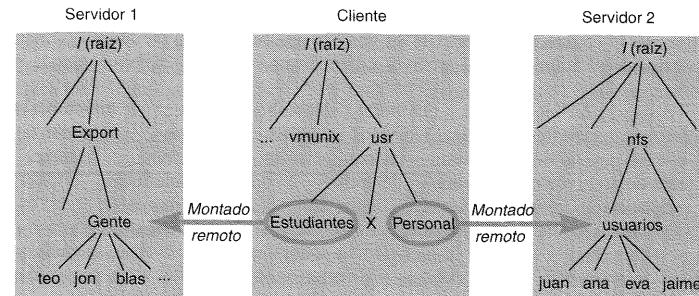
◊ **Servicio de montado.** El montado de los sub-árboles de los sistemas de archivos remotos por los clientes está soportado por un proceso de *servicio de montado* separado que se ejecuta a

<i>lookup(aadir, nombre) → aa, atrib</i>	Devuelve el apuntador del archivo y los atributos para el archivo <i>nombre</i> en el directorio <i>aadir</i> .
<i>create(aadir, nombre, atrib) → aanuevo, atrib</i>	Crea un nuevo archivo <i>nombre</i> en el directorio <i>aadir</i> con los atributos <i>atrib</i> y devuelve el nuevo apuntador de archivo y sus atributos.
<i>remove(aadir, nombre) → estado</i>	Elimina el archivo <i>nombre</i> del directorio <i>aadir</i> .
<i>getattr(aa) → atrib</i>	Devuelve los atributos del archivo <i>aa</i> . (Igual que la llamada UNIX <i>stat</i> .)
<i>setattr(aa) → atrib</i>	Establece los atributos (modo, ID usuario, ID grupo, tamaño, tiempo de acceso y tiempo de modificación de un archivo). Si se pone el tamaño a 0 se trunca el archivo.
<i>read(aa, despl, conteo) → atrib, datos</i>	Devuelve hasta <i>conteo</i> bytes de datos desde el archivo, comenzando en <i>despl</i> . También devuelve los atributos más recientes del archivo.
<i>write(aa, despl, conteo, datos) → atrib</i>	Escribe <i>conteo</i> bytes de datos sobre el archivo, comenzando en <i>despl</i> . También devuelve los atributos del archivo tras la operación de escritura.
<i>rename(aadir, nombre, destadir, destnombre) → estado</i>	Cambia el nombre del archivo <i>nombre</i> del directorio <i>aadir</i> en <i>destnombre</i> del directorio <i>destadir</i> .
<i>link(nuevoaadir, nuevonombre, aadir, nombre) → estado</i>	Crea una entrada <i>nuevonombre</i> en el directorio <i>nuevoaadir</i> que se refiere al archivo <i>nombre</i> en el directorio <i>aadir</i> .
<i>symlink(nuevoaadir, nuevonombre, texto) → estado</i>	Crea una entrada <i>nuevonombre</i> en el directorio <i>nuevoaadir</i> , con un <i>enlace simbólico</i> con el valor <i>texto</i> . El servidor no interpreta <i>texto</i> pero crea un archivo de enlace simbólico para alojarlo.
<i>readlink(aa) → texto</i>	Devuelve el <i>texto</i> asociado con el archivo de enlace simbólico identificado por <i>aa</i> .
<i>mkdir(aadir, nombre, atrib) → nuevoaa, atrib</i>	Crea un nuevo directorio <i>nombre</i> con los atributos <i>atrib</i> y devuelve el nuevo apuntador de archivo (<i>nuevoaa</i>) y sus atributos.
<i>rmdir(aadir, nombre) → estado</i>	Elimina el directorio vacío <i>nombre</i> del directorio padre <i>aadir</i> . Falla si el directorio no está vacío.
<i>readdir(aadir, cookie, conteo) → entradas</i>	Devuelve <i>conteo</i> bytes de entradas de directorio desde el directorio <i>aadir</i> . Cada entrada contiene un nombre de archivo, un apuntador a archivo, y un puntero opaco a la siguiente entrada del directorio, denominado <i>cookie</i> (galletita). <i>cookie</i> se emplea en las siguientes llamadas a <i>readdir</i> para comenzar la lectura desde la siguiente entrada. Si se pone el valor de <i>cookie</i> a 0, lee desde la primera entrada.
<i>statfs(aa) → estadosf</i>	Devuelve información sobre el sistema de archivos (tal como tamaño de bloque, número de bloques libres y demás) para el sistema de archivos que contiene el archivo <i>aa</i> .

Figura 8.9. Operaciones del servidor NFS (simplificado).

nivel de usuario en cada computador servidor NFS. En cada servidor, hay un archivo con un nombre bien conocido (*/etc/exports*) conteniendo los nombres de los sistemas de archivos locales que están disponibles para montado remoto. Se asocia una lista de acceso con cada nombre del sistema de archivos indicando qué máquinas están autorizadas para montar el sistema de archivos.

Los clientes utilizan una versión modificada del mandato *mount* de UNIX para solicitar el montado de un sistema de archivos remoto, especificando el nombre de la máquina remota, el nombre de la ruta de un directorio en el sistema de archivos remoto y el nombre local con el que va a ser montado. El directorio remoto puede ser cualquier sub-árbol del sistema de archivos remoto solicitado, permitiendo a los clientes montar cualquier parte del sistema de archivos remoto.



Nota: El sistema montado en */usr/estudiantes* en el cliente, en realidad es un sub-árbol ubicado en */export/gente* en el Servidor 1; el sistema de archivos montado en */usr/personal* en el cliente es realmente el sub-árbol */nfs/users* ubicado en el Servidor 2.

Figura 8.10. Sistemas de archivos locales y remotos accesibles desde un cliente NFS.

mandato *mount* modificado comunica con el proceso del servicio de montado en la máquina remota, utilizando un *protocolo de montado*. Éste es un protocolo RPC e incluye una operación que toma un nombre de ruta de directorio y devuelve el asidero de archivo del directorio especificado si el cliente tiene permiso de acceso para el sistema de archivos relevante. La ubicación (dirección IP y número de puerto) del servidor y el asidero de archivo para el directorio remoto se pasan a la capa VFS y al cliente NFS.

La Figura 8.10 muestra un *Cliente* con dos almacenes de archivos montados remotamente. Los nodos *gente* y *usuarios* en los sistemas de archivos en el *Servidor 1* y *Servidor 2* están montados sobre los nodos *estudiantes* y *personal* en el almacén de archivos local del *Cliente*. El significado de esto es que programas que se ejecuten en *Cliente* pueden acceder a archivos en *Servidor 1* y *Servidor 2* utilizando nombres de ruta como */usr/estudiantes/jon* y */usr/personal/ana*.

Los volúmenes remotos pueden tener un *montado rígido* (*hard-mounted*) o *flexible* (*soft-mounted*) en el computador del cliente. Cuando un proceso a nivel de usuario accede a un archivo en un volumen montado rígidamente, el proceso se suspende hasta que se completa la solicitud y si la máquina remota no está disponible por alguna razón el módulo cliente NFS continúa reintentando la solicitud hasta que se satisface. Por tanto en el caso de un fallo del servidor, los procesos a nivel de usuario se suspenden hasta que el servidor rearropa y entonces continúan justo como si no hubiera habido fallo. Pero si el volumen relevante tiene un montado flexible, el módulo cliente NFS devuelve una indicación de fallo a los procesos a nivel de usuario después de un pequeño número de reintentos. Los programas construidos adecuadamente detectarán el fallo y tomarán las acciones de recuperación e información adecuadas. Pero muchas utilidades y aplicaciones UNIX no comprueban el fallo de las operaciones de accesos a archivos y éstas se comportan de forma impredecible en el caso de fallo de un volumen con un montado flexible. Por esta razón, muchas instalaciones utilizan exclusivamente montado rígido, con la consecuencia que los programas son incapaces de recuperarse adecuadamente cuando un servidor NFS está indisponible durante un período de tiempo significativo.

◊ **Traducción de un nombre de ruta.** Los sistemas de archivos UNIX traducen nombres de ruta de archivo compuestos a referencias de i-node, en un proceso paso a paso cuando se utilizan las llamadas del sistema *open*, *creat* o *stat*. En NFS, los nombres de ruta no pueden traducirse en el servidor, porque el nombre puede cruzar un «punto de montado» en el cliente, los directorios que mantienen diferentes partes de un nombre compuesto pueden residir en volúmenes en diferentes

servidores. Por lo tanto los nombres de ruta se analizan y su traducción se realiza de manera interactiva por el cliente. Cada parte de un nombre que se refiere a un directorio montado remotamente se traducido al apuntador del archivo utilizando una solicitud *lookup* separada para el servidor remoto.

La operación *lookup* busca una parte del nombre de ruta en un directorio dado y devuelve el asidero correspondiente del archivo y los atributos del mismo. El apuntador devuelto del archivo en el paso previo se utiliza como un parámetro en el paso *lookup* siguiente, el identificado del sistema de archivos en el apuntador de archivo se compara inicialmente con las entradas en la tabla de montado remoto mantenida en el cliente para ver si otro almacén de archivos montados remotamente debiera ser accedido. Manteniendo en caché los resultados de cada paso en las traducciones de nombres de ruta se alivia la ineficiencia aparente de este proceso, sacando partido de la proximidad de referencia de los archivos y directorios, los usuarios y los programas acceden normalmente a los archivos de solo uno o un pequeño número de directorios.

◊ **Automontador.** El automontador se añadió a la implementación UNIX de NFS para poder montar un directorio remoto dinámicamente cuando un punto de montado «vacío» es referenciado por un cliente. La implementación original del automontador se ejecutaba como un proceso UNIX a nivel de usuario en cada computador cliente. Las versiones posteriores (llamadas *autofs*, se implementaron en el núcleo de Solaris y Linux. Aquí describimos la versión original.

El automontador mantiene una tabla de puntos de montado (nombres de ruta) con una referencia a uno o más servidores NFS por cada punto. Se comporta igual que un servidor local NFS en la máquina del cliente. Cuando el módulo cliente NFS intenta resolver un nombre de ruta que incluye uno de estos puntos de montado, pasa una solicitud *lookup* al automontador local que localiza el volumen solicitado en su tabla y envía una solicitud de «prueba» a cada servidor de la lista. Se monta el volumen del primer servidor que responda, utilizando el servicio normal de montado. El volumen montado se enlaza con un punto de montado utilizando un enlace simbólico, por lo que el acceso a él no precisará de otras solicitudes al automontador. El acceso al archivo se realiza de la forma normal sin más referencias al automontador a menos que no haya referencias al enlace simbólico durante varios minutos. En este caso, el automontador desmonta el volumen remoto.

Las últimas implementaciones del núcleo reemplazan los enlaces simbólicos con montados reales, impidiendo algunos problemas que se presentan con las aplicaciones que hacen *caching* de los nombres de ruta temporales utilizados en los automontadores a nivel de usuario [Callaghan 1999].

Puede conseguirse una forma sencilla de replicación de sólo lectura listando los distintos servidores que contienen copias idénticas de un volumen o un sub-árbol de archivos frente a un nombre en la tabla de automontador. Esto se utiliza en sistemas de archivos utilizados frecuentemente que cambian con poca frecuencia, como los programas binarios de UNIX. Por ejemplo, se pueden mantener copias del directorio */usr/lib* y su subárbol en más de un servidor. En la primera ocasión que se abre un archivo de */usr/lib* en un cliente, se enviarán mensajes de prueba a todos los servidores, y el primero que responda será montado en el cliente. Esto proporciona un grado limitado de tolerancia a fallos y balance de carga, puesto que el primer servidor que responda será uno que no ha fallado y es probable que sea uno que no está fuertemente ocupado sirviendo otras solicitudes.

◊ **Caché en el servidor.** El mantenimiento de caché tanto en el computador del cliente como en el servidor son características indispensables de las implementaciones NFS con el fin de obtener las prestaciones adecuadas.

En los sistemas UNIX convencionales, las páginas de archivo, los directorios y los atributos de archivo que han sido leídas del disco son retenidas en un *cache búfer* en la memoria principal hasta que se precisa el espacio del búfer para otras páginas. Si un proceso efectúa entonces una solicitud de lectura o escritura de una página que ya está en la caché, podrá satisfacerse sin hacer otro acceso a disco. El modo de *lectura anticipada* (*read-ahead*) se adelanta a los accesos de lectura y busca las páginas que siguen a las que han sido leídas más recientemente, y el modo de *escritura retardada* (*delayed-write*) optimiza las escrituras: cuando una página ha sido alterada (por una solicitud de escritura), su nuevo contenido se escribe en disco sólo cuando se necesita el búfer para otra página. Para salvaguardar los datos frente a pérdidas por una caída del sistema, la operación *sync* de UNIX pasa las páginas actualizadas a disco cada 30 segundos. Las técnicas de caché funcionan en un entorno UNIX tradicional porque todas las solicitudes de lectura y escritura realizadas por los procesos a nivel de usuario pasan a través de una única caché que está implementada en el espacio del núcleo de UNIX. La caché se mantiene actualizada y los accesos a los archivos no pueden evitar la caché.

Los servidores NFS utilizan la caché en la máquina de servidor como se utiliza para otros accesos a archivos. La utilización de la caché del servidor para mantener los bloques de disco leídos recientemente no plantea ningún problema de consistencia, pero cuando un servidor realiza operaciones de escritura, se necesitan medidas extra para garantizar que los clientes pueden estar seguros que los resultados de las operaciones de escritura son persistentes, incluso cuando ocurren caídas del servidor. En la versión 3 del protocolo NFS, la operación *write* ofrece para esto dos opciones (no representadas en la Figura 8.9):

1. Los datos recibidos de los clientes en las operaciones de *escritura* se almacenan en la memoria caché en el servidor y se escriben en disco antes de que se envíe una respuesta al cliente. Esto se llama *escritura a través* (*write-through*) de la caché. El cliente puede estar seguro de que los datos son almacenados persistentemente en el momento en el que se ha recibido la respuesta.
2. Los datos en las operaciones de *escritura* sólo se almacenan en la memoria caché. Éstos serán escritos en disco cuando se reciba una operación de *consumación* (*commit*) para el archivo relevante. El cliente puede estar seguro que los datos están almacenados persistentemente sólo cuando se ha recibido una respuesta a una operación de *consumación* para el archivo relevante. Los clientes de NFS estándar utilizan este modo de operación, emitiendo una *consumación* cuando se cierra un archivo que fue abierto para escritura.

La *consumación* es una operación adicional proporcionada en la versión 3 del protocolo NFS, se añadió para superar un cuello de botella en las prestaciones producido por el modo de operación de *escritura a través* en los servidores que reciben un gran número de operaciones *write*.

El requisito de *escritura a través* en los sistemas de archivos distribuidos es una instancia de los modos independientes de fallos discutidos en el Capítulo 1, los clientes continúan trabajando cuando un servidor falla, y los programas de aplicación pueden realizar acciones en la suposición de que los resultados de las escrituras previas han sido consumados en el almacenamiento del disco. Esto es improbable que ocurra en el caso de actualizaciones de archivos locales, dado que el fallo de un sistema de archivos locales es casi seguro produzca el fallo de todos los procesos de aplicación ejecutándose en el mismo computador.

◊ **Caché en el cliente.** El módulo cliente NFS emplea caché para los resultados de las operaciones *read*, *write*, *getattr*, *lookup* y *readdir*, con el fin de reducir el número de solicitudes transmitidas a los servidores. La caché del cliente introduce el potencial para que existan diferentes versiones de archivos o porciones de archivos en diferentes nodos cliente, porque las escrituras por un cliente no producen la actualización inmediata de las copias en la caché del mismo archivo en otros clientes. En su lugar, los clientes son responsables de sondear al servidor para comprobar la actualidad de los datos en la caché que ellos mantienen.

Para validar los bloques en la caché antes de que sean utilizados se emplea un método basado en marcas de tiempo. Cada elemento de datos o metadatos de la caché se etiqueta con dos marcas de tiempo:

Tc es el tiempo en el que la entrada en la caché fue validada últimamente.

Tm es el tiempo en el que el bloque fue modificado por última vez en el servidor.

Una entrada en la caché es válida en el tiempo T si $T - T_c$ es menor que un intervalo de refresco t , o si el valor registrado para T_m en el cliente concuerda con el valor T_m en el servidor (esto es, los datos no han sido modificados en el servidor desde que se realizó la entrada en la caché). Formalmente la condición de validez es:

$$(T - T_c < t) \vee (T_m_{cliente} = T_m_{servidor})$$

La selección de un valor para t viene del compromiso entre consistencia y eficiencia. Un intervalo de refresco muy corto conlleva una aproximación muy próxima a la consistencia de una copia, al coste de una carga relativamente pesada de llamadas al servidor para comprobar el valor de $T_m_{servidor}$. En los clientes Solaris de Sun, t se coloca de forma adaptativa para los archivos individuales a un valor en el rango de 3 a 30 segundos dependiendo de la frecuencia de actualizaciones en el archivo. Para directorios el rango es de 30 a 60 segundos reflejando el riesgo más bajo de actualizaciones concurrentes.

Puesto que los clientes NFS no pueden determinar si un archivo está siendo compartido o no, el procedimiento de validación debe ser utilizado para todos los accesos al archivo. Cuando se utiliza una entrada en la caché se realiza una comprobación de validez. La primera mitad de la condición de validez puede evaluarse sin acceder al servidor. Si es verdadera, entonces la segunda mitad no necesita ser evaluada; si es falsa, se obtiene el valor actual de $T_m_{servidor}$ (por medio de una llamada *getattr* al servidor) y se compara con el valor local de $T_m_{cliente}$. Si son iguales, entonces la entrada a la caché se considera que es válida y el valor de T_c para la entrada en la caché es actualizada al tiempo actual. Si difieren, entonces los datos en la caché han sido actualizado en el servidor y la entrada en la caché es invalidada, resultando en una solicitud para el servidor por los datos relevantes.

Se utilizan varias medidas para reducir el tráfico de las llamadas *getattr* al servidor:

- Cuando se recibe un nuevo valor de $T_m_{servidor}$ en un cliente, se aplica para todas las entradas en la caché derivadas del archivo relevante.
- Los valores actuales del atributo son enviados se acarrean con los resultados de cada operación en un archivo, y si el valor de $T_m_{servidor}$ ha cambiado el cliente lo utiliza para actualizar las entradas en la caché relativas al archivo.
- El algoritmo adaptativo para fijar el intervalo de refresco t indicado anteriormente reduce considerablemente el tráfico para la mayoría de los archivos.

El procedimiento de validación no garantiza el mismo nivel de consistencia de los archivos que es proporcionado en los sistemas convencionales UNIX, puesto que las actualizaciones recientes no siempre son visibles para los clientes que comparten un archivo; hay dos fuentes de retardo temporal, el retardo después de la escritura antes de que los datos dejen la caché en el núcleo de actualización del cliente y la «ventana» de tres segundos para la validación de la caché. Afortunadamente, la mayoría de las aplicaciones UNIX no dependen de forma crítica de la sincronización de las actualizaciones de archivo, y se han informado pocas dificultades con este origen.

Las escrituras se mantienen de forma diferente. Cuando se modifica una página en la caché se marca como sucia y se planifica para ser volcada al servidor asíncronamente. Las páginas modificadas se vuelcan cuando se cierra el archivo u ocurre un *sync* en el cliente, y esto ocurre más frecuentemente si hay bio-demonios en uso (ver a continuación). Esto no proporciona la misma garantía de persistencia que en la caché del servidor, pero emula el comportamiento para escrituras locales.

Para implementar lectura de la entrada y *escritura retardada*, el cliente NFS necesita realizar asíncronamente algunas lecturas y escrituras. Esto se consigue en las implementaciones UNIX de NFS mediante la inclusión de uno o más procesos *bio-demonio* en cada cliente (*Bio* indica *block input-output —entrada-salida de bloques*—, el término demonio se utiliza a menudo para referirse

a procesos a nivel de usuario que realizan tareas del sistema). El papel de los bio-demonios es realizar operaciones de lectura adelantada y escritura retrasada. Después de cada solicitud se avisa al bio-demonio, y éste solicita la transferencia de el siguiente bloque del archivo desde el servidor a la caché del cliente. En el caso de escritura el bio-demonio enviará un bloque al servidor cuando se haya llenado un bloque una operación del cliente. Los bloques de directorio se envían cuando se ha producido una modificación.

Los procesos bio-demonio mejoran las prestaciones, asegurando que el módulo cliente no se bloquea esperando la vuelta de las lecturas o la consumación de escrituras en el servidor. No hay un requisito lógico, puesto que en ausencia de lectura adelantada, una operación *read* en un proceso de usuario disparará una solicitud síncrona para el servidor relevante, y los resultados de *write* en los procesos de usuario serán transferidos al servidor cuando el archivo relevante se cierra o cuando el sistema de archivos virtual en el cliente lanza una operación *sync*.

◊ **Otras optimizaciones.** El sistema de archivos de Sun está basado en el Sistema Rápido de Archivos (*Fast File System*, FFS) de UNIX BSD que utiliza bloques de disco de ocho kbytes, lo que resulta en menos llamadas al sistema de archivos para acceso a archivos secuenciales que en los sistemas UNIX anteriores. Los paquetes UDP utilizados para la implementación de Sun RPC se extienden a nueve kilobytes, permitiendo que una llamada RPC conteniendo un bloque entero como argumento sea transferida en un único paquete y minimice el efecto de la latencia de red cuando se lean archivos secuenciales. En NFS versión 3 no hay límite en el tamaño máximo de los bloques de archivo que pueden ser mantenidos en las operaciones *read* y *write*, los clientes y los servidores pueden negociar tamaños más grandes que ocho kbytes si son capaces de mantenerlos.

Como se ha mencionado anteriormente la información del estado del archivo en la caché en los clientes debe actualizarse al menos cada tres segundos para los archivos activos. Para reducir la consecuente carga del servidor resultante de solicitudes *getattr*, todas las operaciones que se refieren a archivos o directorios son tomadas como solicitudes *getattr* implícitas, y los valores de los atributos actuales son acarreados con el resto de los resultados de la operación.

◊ **Haciendo seguro NFS con Kerberos.** En la Sección 7.6.2 se ha descrito el sistema de autenticación desarrollado en el MIT Kerberos, que ha llegado a ser un estándar industrial para servidores de intranet seguros contra accesos no autorizados y ataques de impostores. La seguridad de las implementaciones de NFS ha sido incrementada con el uso del esquema Kerberos para autenticar clientes. En esta subsección, describimos la «Kerberización» de NFS tal y como fue realizada por los diseñadores de Kerberos.

En la implementación estándar original de NFS, la identidad del usuario se incluye en cada solicitud en forma de un identificador numérico no encriptado (el identificador fue encriptado en versiones posteriores de NFS). NFS no realiza ningún otro paso para comprobar la autenticidad del identificador proporcionado. Esto implica un alto grado de confianza en la integridad del computador cliente y en su software para NFS, mientras que el propósito de Kerberos y otros sistemas de seguridad basados en autenticación es reducir a un mínimo el rango de componentes sobre los que se supone dicha confianza. Esencialmente, al utilizar NFS en un entorno «Kerberizado» sólo se aceptarán solicitudes de clientes cuya identidad pueda ser comprobada que ha sido autenticada por Kerberos.

Una solución obvia, considerada por los desarrolladores de Kerberos, fue cambiar la naturaleza de las credenciales solicitadas por NFS para ser un sistema Kerberos basado en tickets y un autenticador de pleno derecho. Pero como NFS está implementado como un servidor sin estado, cada solicitud de acceso a un archivo individual se despacha por su validez intrínseca y entonces los datos de autenticación deberían incluirse en cada solicitud. Esto fue considerado inaceptablemente costoso dado el tiempo requerido para realizar las encriptaciones necesarias y que debieran llevarse a cabo añadiendo la biblioteca cliente Kerberos al núcleo de todas las estaciones de trabajo.

En lugar de eso, se adoptó una aproximación híbrida en la que se proporciona el servidor de montado NFS con todos los datos de autenticación de Kerberos para el usuario cuando se montan sus volúmenes raíz e inicial (*home*). Los resultados de la autenticación, incluyendo el identificador numérico convencional del usuario y la dirección del computador del cliente, se retienen en el servidor con la información de montado de cada volumen (aunque el servidor NFS no retiene el estado relativo de los procesos individuales cliente, retiene los montados actuales en cada computador cliente).

En cada solicitud de acceso al archivo, el servidor NFS comprueba el identificador del usuario y las direcciones del remitente y proporciona el acceso sólo si concuerdan con los almacenados en el servidor para el cliente relevante en el tiempo de montado. Esta aproximación híbrida implica un coste mínimo adicional y es segura contra la mayoría de las formas de ataque, proporciona que sólo un usuario en el tiempo puede entrar en cada computador del cliente. En el MIT, el sistema está configurado de esta forma. Implementaciones recientes de NFS incluyen la autenticación Kerberos como una de las varias opciones de autenticación, y se aconseja que los sitios que también ejecutan servidores Kerberos empleen esta opción.

◊ **Prestaciones.** Las primeras cifras sobre prestaciones comunicadas por Sandberg [1987] mostraron que el uso de NFS no imponía usualmente una penalización de las prestaciones, en comparación con el acceso a los archivos almacenados en los archivos locales. Sandberg identificó dos áreas de problemas restantes:

- El uso frecuente de la llamada *getattr* con el fin de buscar marcas de tiempo de los servidores para la validación de la caché.
- Las prestaciones relativamente pobres de la operación *write* porque se utilizaba la *escritura a través* en el servidor.

También señaló que las escrituras son relativamente infrecuentes en las cargas de trabajo UNIX típicas (alrededor del 5 % de todas las llamadas al servidor), y el coste de la *escritura a través* es, por tanto, tolerable excepto cuando se escriben grandes archivos en el servidor. La versión de NFS que él comprobó no incluía el mecanismo *commit* esbozado anteriormente, que produce una mejora sustancial en las prestaciones de escritura en las versiones actuales. Sus resultados también muestran que la operación *lookup* supone el 50 % de las llamadas al servidor. Esto es consecuencia del método de traducción de nombres de ruta paso a paso requerido por la semántica de nombrado de archivos UNIX.

Hoy en día se realizan medidas regularmente, por Sun y otros implementadores de NFS, utilizando una versión actualizada de un conjunto exhaustivo de programas de prueba conocidos como LADDIS [Keith y Wittle 1993]. Los resultados actuales y pasados están disponibles en [\[www.spec.org\]](http://www.spec.org). Las prestaciones se resumen aquí para implementaciones del servidor NFS de diferentes vendedores y variadas configuraciones hardware. Resultados recientes incluyen un rendimiento de 5.011 operaciones del servidor por segundo, con una latencia promedio de 3,71 milisegundos y una latencia máxima de 8 ms (1 × 450 MHz Pentium III CPU, 34 discos en un controlador Ethernet de un Gbps y sistema operativo de tiempo real dedicado) y un rendimiento de 29.083 operaciones del servidor por segundo con (latencia promedio)/máxima de 4,25/7,8 ms (24 × 450 MHz IBM RS64-III CPU, 289 discos en cuatro controladores, 5 redes de 1 Gbps, AIX UNIX). Estas figuras indican que es probable que NFS ofrezca una solución muy efectiva para las necesidades de almacenamiento distribuido en las intranets de la mayoría de los tamaños y tipos de uso, oscilando por ejemplo de una carga tradicional UNIX de desarrollo por varios cientos de ingenieros de software a una batería de servidores web accediendo y sirviendo material de un servidor NFS.

◊ **Resumen NFS.** NFS de Sun sigue fuertemente nuestro modelo abstracto. El diseño resultante proporciona buena transparencia de ubicación y acceso si el servicio de montado NFS se utiliza

adecuadamente, para producir espacios de nombre semejantes en todos los clientes. NFS soporta hardware y sistemas operativos heterogéneos. La implementación del servidor NFS es sin estado, permitiendo a los clientes y servidores recuperar la ejecución después de un fallo sin necesidad de ningún procedimiento de recuperación. La migración de archivos o sistemas de archivos no está soportada, excepto en el nivel de intervención manual para reconfigurar las directivas de montado después de un movimiento de un sistema de archivos a una nueva ubicación.

Las prestaciones de NFS mejoran mucho gracias a la caché de bloques de archivo en cada computador cliente. Esto es importante para la obtención de prestaciones satisfactorias pero produce alguna desviación de la semántica de actualización de una copia de archivo estricta UNIX.

Los otros objetivos de diseño de NFS y las extensiones a ellos que han sido conseguidas, se discuten a continuación.

Transparencia de acceso: el módulo cliente NFS proporciona una interfaz de programación de aplicación para los procesos locales que es idéntica a la interfaz del sistema operativo local. Por tanto en un cliente UNIX, los accesos a archivos remotos se realizan utilizando llamadas al sistema normales de UNIX. No se precisa modificar los programas existentes para poder trabajar correctamente con archivos remotos.

Transparencia de ubicación: cada cliente establece un espacio de nombres de archivo añadiendo directorios montados en volúmenes remotos sobre su espacio local de nombres. Los sistemas de archivos han de ser *exportados* por el nodo que los mantiene y *montados remotamente* por un cliente antes que ellos puedan ser accedidos por procesos ejecutándose en el cliente (véase la Figura 8.10). El punto de la jerarquía de nombres del cliente donde aparece montado remotamente un sistema de archivos lo determina el cliente, por tanto NFS no fuerza un espacio de nombres de archivo único a través de la red; cada cliente ve un conjunto de sistemas de archivos remotos que es determinado localmente, y los archivos remotos pueden tener diferentes nombres de ruta en diferentes clientes, pero se puede establecer un espacio de nombres uniforme en cada cliente mediante las tablas de configuración apropiadas, logrando el objetivo de transparencia de ubicación.

Transparencia de movilidad: los volúmenes (en el sentido UNIX, esto es, subárboles de archivos) pueden ser reubicados entre servidores, pero las tablas de montado remoto en cada cliente deben ser actualizadas separadamente para permitir a los clientes el acceso al sistema de archivos en su nueva ubicación, por lo que la transparencia de migración no está totalmente conseguida en NFS.

Escalabilidad: las cifras de prestaciones publicadas muestran que se pueden construir servidores NFS que mantengan cargas reales muy grandes, de una manera eficiente y beneficiosa. Se pueden incrementar las prestaciones de un único servidor mediante la adición de procesadores, discos y controladores. Cuando los límites de esos procesos son alcanzados, hay que instalar servidores adicionales y los sistemas de archivos deben ser reubicados entre ellos. La efectividad de esa estrategia está limitada por la existencia de archivos de uso muy frecuente, son archivos sencillos que son accedidos tan frecuentemente que el servidor llega al límite de prestaciones. Cuando las cargas exceden el máximo de prestaciones disponible con esa estrategia, una solución mejor es emplear un sistema de archivos distribuidos que soporte la replicación de los archivos actualizables (como Coda, descrito en el Capítulo 14), o uno como AFS que reduce el tráfico del protocolo haciendo caché de los archivos completos. En la Sección 8.5 discutiremos otras aproximaciones a la escalabilidad.

Replicación de archivos: los almacenes de archivos de *sólo lectura* pueden ser replicados en varios servidores NFS, pero NFS no soporta la replicación de archivos actualizables. El Servicio de Información de Red de Sun (*Network Information Service*, NIS) es un servicio separado disponible para su uso con NFS que soporta la replicación de bases de datos sencillas organiza-

das como pares clave-valor (por ejemplo los archivos del sistema UNIX */etc/passwd* y */etc/hosts*). Éste gestiona la distribución de actualizaciones y accesos a los archivos replicados basada en un modelo de replicación sencilla maestro-esclavo (también conocido como modelo de *copia primaria*, que se discutirá en el Capítulo 14) que provee la replicación de parte o toda la base de datos en cada sitio. NIS proporcionan un almacén compartido para información del sistema que cambia infrecuentemente y no requiere que las actualizaciones ocurran simultáneamente en todos los sitios.

Heterogeneidad del hardware y del sistema operativo: NFS ha sido implementado para casi todos los sistemas operativos y plataformas hardware conocidos y está soportado por una variedad de sistemas de archivos.

Tolerancia a fallos: la naturaleza sin estado e idempotente del protocolo de acceso a archivos NFS asegura que los modos de fallo observados por los clientes cuando acceden a archivos remotos son similares a aquéllos de acceso a archivos locales. Cuando un servidor falla, el servicio que proporciona se suspende hasta que se rearanca el servidor, pero una vez que ha sido reiniciado los procesos cliente a nivel de usuario proceden desde el punto en el que fue interrumpido el servicio sin darse cuenta del fallo (excepto en el caso de acceso a sistemas de archivos remotos *con montado flexible*). En la práctica el montado rígido se ha utilizado en la mayoría de los casos, y esto propende a impedir que los programas de aplicación manejen los fallos del servidor de modo elegante.

El fallo de un computador cliente o de un proceso a nivel de usuario en un cliente no tiene efecto sobre ningún servidor que el pueda estar utilizando, puesto que los servidores no mantienen el estado en nombre de sus clientes.

Consistencia: hemos descrito el comportamiento de las actualizaciones con cierto detalle. NFS proporciona una cercana aproximación a la semántica de *una copia* y coincide con las necesidades de la gran mayoría de las aplicaciones, pero no podemos recomendar el uso de archivos compartidos vía NFS para la comunicación o coordinación fuerte entre procesos en diferentes computadores.

Seguridad: las necesidades de seguridad en NFS sólo emergen al conectar la mayoría de las intranets con Internet. La integración de Kerberos con NFS fue un salto fundamental hacia delante. Otros desarrollos recientes incluyen la opción de utilizar una implementación RPC segura (RPCSEC- GSS, documentada en RFC 2203 [Eisler y otros 1997]) para la autenticación y la privacidad y seguridad de los datos transmitidos con las operaciones de lectura y escritura. Abundan las instalaciones que todavía no han desplegado estos mecanismos, y son inseguras.

Eficiencia: las prestaciones medidas de varias implementaciones de NFS y su amplia adopción para uso en situaciones que generan cargas muy pesadas son indicaciones claras de la eficiencia con la que puede ser implementado el protocolo NFS.

8.4. SISTEMA DE ARCHIVOS ANDREW

Como NFS, AFS proporciona acceso transparente a archivos compartidos remotos para los programas UNIX que se ejecutan en estaciones de trabajo. El acceso a los archivos AFS se hace mediante las primitivas usuales de los archivos UNIX, permitiendo a los programas UNIX existentes acceder a archivos AFS sin modificación o recompilación. AFS es compatible con NFS. Los servidores AFS mantienen los archivos UNIX «locales», pero el sistema de archivos en los servidores está basado en NFS, por lo que los archivos se refieren mediante apuntadores de archivo de estilo NFS más que mediante números de i-nodo, y los archivos pueden ser accedidos remotamente a través de NFS.

AFS difiere notoriamente de NFS en su diseño e implementación. Las diferencias son atribuibles principalmente a la identificación de la escalabilidad como el objetivo de diseño más importante. AFS está diseñado para trabajar bien con gran número de usuarios activos más que otros sistemas de archivos distribuidos. La estrategia clave para alcanzar la escalabilidad es la caché de los archivos completos en los nodos cliente. AFS tiene dos características de diseño inusuales:

Servicio de archivo completo: el contenido entero de los directorios y los archivos es transmitido a los computadores cliente por los servidores AFS (en AFS-3, los archivos más grandes de 64 kbytes son transferidos en trozos de 64 kbytes).

Caché de archivo completo: una vez que una copia de un archivo, o un trozo, ha sido transferido a un computador cliente es almacenado en una caché en el disco local. La caché contiene varios cientos de archivos, los utilizados más recientemente en ese computador. La caché es permanente sobreviviendo a los rearranques del computador cliente. Las copias locales de los archivos son utilizadas para satisfacer las solicitudes *open* de los clientes con preferencias sobre las copias remotas cuando es posible.

◊ **Escenario.** Aquí presentamos un escenario simple que ilustra el funcionamiento de AFS:

- Cuando un proceso de usuario en un computador cliente emite una llamada del sistema *open* para un archivo en un espacio de archivos compartido y no hay una copia actual del archivo en la caché local, el servidor que mantiene el archivo se localiza y se envía una solicitud para una copia del archivo.
- La copia es almacenada en el sistema de archivos UNIX local en el computador del cliente; se *abre* la copia y el descriptor UNIX resultante del archivo es devuelto al cliente.
- Las operaciones posteriores: *read*, *write* y demás, sobre el archivo por los procesos en el computador cliente, se aplican a la copia local.
- Cuando el proceso en el cliente emite una llamada del sistema *close*, si la copia local ha sido actualizada su contenido es enviado de vuelta al servidor. El servidor actualiza el contenido del archivo y las marcas de tiempo del mismo. La copia en el disco local del cliente es retenida en el caso de que se necesite de nuevo por un proceso a nivel de usuario en la misma estación de trabajo.

Discutiremos las prestaciones de AFS observadas más adelante, pero podemos hacer aquí algunas observaciones y predicciones generales basadas en las características de diseño descritas anteriormente:

- Para archivos compartidos que son actualizados infrecuentemente (aquellos que contienen el código de los comandos y librerías de UNIX) y para archivos que son accedidos normalmente por un único usuario (como la mayoría de los archivos en un directorio *home* del usuario y su sub-árbol), las copias en la caché localmente probablemente permanezcan válidas durante períodos largos, en el primer caso porque no se actualizan y en el segundo porque si se actualizan, la copia actualizada estará en la caché de la estación de trabajo del propietario. Esta clase de archivo es la abrumadora mayoría de los accesos.
- La caché local puede reservar una proporción sustancial del espacio de disco en cada estación de trabajo, pongamos 100 megabytes. Esto es normalmente suficiente para el establecimiento de un conjunto de trabajo de los archivos utilizados por un usuario. La provisión de almacenamiento caché suficiente para el establecimiento de un conjunto de trabajo asegura que los archivos de uso regular en una estación de trabajo dada son normalmente retenidos en la caché hasta que son necesarios de nuevo.
- La estrategia de diseño está basada en algunas suposiciones sobre los tamaños promedio y máximo de los archivos y la proximidad de referencia de los mismos en sistemas UNIX. Estas suposiciones se derivan de las observaciones de las cargas de trabajo UNIX típicas en entornos académicos y otros [Satyanarayanan 1981; Ousterhout y otros 1985; Floyd 1986].

Las observaciones más importantes son:

- Los archivos son pequeños; la mayoría son menores de 10 kilobytes de tamaño.
- Las operaciones de lectura en los archivos son mucho más comunes que la escritura (unas seis veces más habituales).
- El acceso secuencial es lo habitual, y el acceso aleatorio es inusual.
- La mayoría de los archivos son leídos y escritos por sólo un usuario. Cuando se comparte un archivo normalmente sólo lo modifica un usuario.
- Los archivos son referenciados a ráfagas. Si un archivo ha sido referenciado recientemente hay una alta probabilidad que sea referenciado en el futuro próximo.

Estas observaciones fueron utilizadas para guiar el diseño y optimización de AFS, *no* para restringir la funcionalidad vista por los usuarios.

- AFS trabaja mejor con las clases de archivo identificadas en el primer punto anterior. Hay un tipo importante de archivo que no concuerda con ninguna de estas clases, las bases de datos son compartidas normalmente por muchos usuarios y generalmente son actualizadas muy a menudo. Los diseñadores de AFS han excluido explícitamente la provisión de facilidades de almacenamiento para bases de datos de sus objetivos de diseño, afirmando que las restricciones impuestas por las diferentes estructuras de nombrado (esto es, acceso basado en contenido) y la necesidad para acceso a datos con gran detalle, control de concurrencia y atomicidad de las actualizaciones hace difícil el diseño de un sistema de base de datos distribuida que es también un sistema de archivos distribuido. Ellos arguyen que la provisión de funcionalidades de bases de datos distribuidas debiera ser considerada separadamente [Satyanarayanan 1989a].

8.4.1. IMPLEMENTACIÓN

El escenario anterior ilustra la operación de AFS pero deja muchas preguntas no contestadas sobre su implementación. Entre las más importantes están:

- ¿Cómo consigue el control AFS cuando una llamada del sistema *open* o *close* refiriéndose a un archivo en el espacio de archivos compartidos es emitida por un cliente?
- ¿Cómo está manteniendo el servidor el archivo localizado requerido?
- ¿Qué espacio está reservado para los archivos en la caché en las estaciones de trabajo?
- ¿Cómo asegura AFS que las copias en la caché de los archivos están actualizadas cuando los archivos pueden ser actualizados por varios clientes?

A continuación respondemos a estas preguntas.

AFS está implementado como dos componentes software que existen como procesos UNIX llamados *Vice* y *Venus*. La Figura 8.11 muestra la distribución de los procesos Vice y Venus. Vice es el nombre dado al servidor software que se ejecuta como un proceso UNIX a nivel de usuario en cada computador servidor, y Venus es un proceso a nivel de usuario que se ejecuta en cada computador cliente y corresponde al módulo cliente en nuestro modelo abstracto.

Los archivos disponibles para los procesos de usuario que se ejecutan en estaciones de trabajo son o *locales* o *compartidos*. Los archivos locales se manejan como archivos normales UNIX. Están almacenados en el disco de la estación de trabajo y están disponibles sólo para los procesos de usuario locales. Los archivos compartidos están almacenados en los servidores y las copias de ellos son introducidas en la caché en los discos locales de las estaciones de trabajo. El espacio de nombres visto por los procesos del usuario es el ilustrado en la Figura 8.12. Es una jerarquía convencional de directorios UNIX, con un subárbol específico (llamado *cmu*) conteniendo todos los archivos compartidos. Este desdoblamiento del espacio de nombres de archivos en archivos loca-

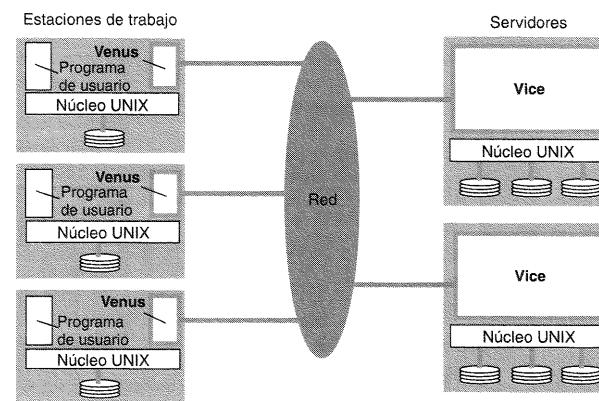


Figura 8.11. Distribución de procesos en el Sistema de Archivos Andrew.

les y compartidos conduce a alguna pérdida de la transparencia de ubicación, pero esto apenas evide para usuarios distintos de los administradores del sistema. Los archivos locales son utilizados sólo para archivos temporales (*/tmp*) y procesos que son esenciales para el arranque de la estación de trabajo. Otros archivos estándar UNIX (como aquellos encontrados normalmente en */bin*, */lib*, etc.) están implementados como enlaces simbólicos desde los directorios locales a los archivos mantenidos en el espacio compartido. Los directorios de los usuarios están en el espacio compartido permitiendo a los usuarios acceder a sus archivos desde cualquier estación de trabajo.

El núcleo UNIX de cada estación de trabajo y servidor es una versión modificada de UNIX BSD. Las modificaciones están diseñadas para interceptar *open*, *close* y algunas otras llamadas del sistema para archivos cuando ellas se refieren a archivos en el espacio de nombres compartido y pasan entonces al proceso Venus en el computador cliente (como se ve en la Figura 8.13). Se incluye otra modificación del núcleo por razones de prestaciones y se describe más adelante.

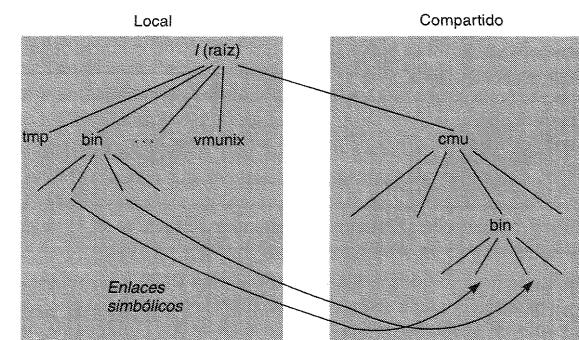


Figura 8.12. Espacio de nombres visto por los clientes de AFS.

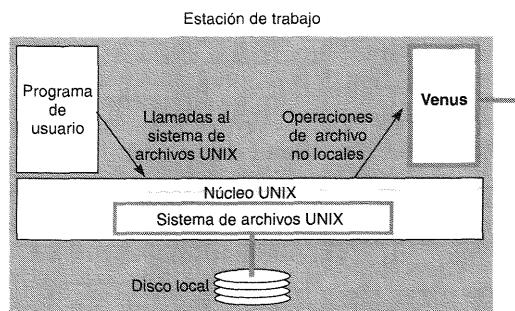


Figura 8.13. Interpretación de llamadas al sistema en AFS.

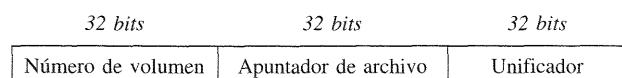
Una de las particiones de archivos en el disco local de cada estación de trabajo se usa como una caché, que mantiene las copias cacheadas de los archivos del espacio compartido. Venus administra la caché, eliminando los archivos menos recientemente utilizados cuando se adquiere un nuevo archivo desde un servidor para conseguir el espacio requerido si la partición está llena. La caché de la estación de trabajo es generalmente lo suficientemente grande para acomodar varios cientos de archivos de tamaño promedio en gran parte que la estación de trabajo sea independiente de los servidores Vice una vez que se ha introducido en la caché un espacio de trabajo de los archivos actuales del usuario y de los archivos del sistema utilizados frecuentemente.

AFS se parece al modelo abstracto de archivos descrito en la Sección 8.2 en estos aspectos:

- Los servidores Vice implementan un servicio de archivos planos, y la estructura jerárquica de directorios requerida por los programas de usuario UNIX es implementada en el conjunto de procesos Venus en las estaciones de trabajo.
- Cada archivo y directorio en el espacio de archivos compartido se reconoce por un identificador de archivo único de 96 bits (*ida*) semejante a una UFID. Los procesos Venus trasladan los nombres de rutas emitidos por los clientes a *ida*.

Los archivos se agrupan en *volumenes* para facilitar la localización y el movimiento. Estos volúmenes son generalmente más pequeños que los volúmenes UNIX, que son la unidad de agrupamiento de archivos en NFS. Por ejemplo, los archivos personales de cada usuario están ubicados generalmente en un volumen separado. Se reservan otros volúmenes para los binarios del sistema, la documentación y el código de las bibliotecas.

La representación de cada *ida* incluye el número de volumen para el volumen que contiene el archivo (como el *identificador de grupo del archivo* en cada UFID) y un *identificador único* para asegurar que los identificadores de archivo no son reutilizados:



Los programas de usuario utilizan los nombres de ruta UNIX para referirse a los archivos, pero AFS utiliza los *idas* en la comunicación entre los procesos Venus y Vice. Los servidores Vice aceptan solicitudes sólo en términos de *idas*. Venus traduce los nombres de ruta proporcionados por los clientes en *idas* utilizando una búsqueda paso a paso para obtener la información de los archivos y directorios mantenidos en los servidores Vice.

Proceso de usuario	Núcleo UNIX	Venus	Red	Vice
<i>open(NombreArchivo, modo)</i>	Si <i>NombreArchivo</i> se refiere a un archivo del espacio de archivos compartido, pasa la petición a Venus.	Comprueba la lista de archivos en la caché local. Si no está presente o no hay una <i>promesa de devolución de llamada</i> , envía una petición de archivo al servidor Vice que es el custodio del volumen que contiene el archivo.	→	Transfiere una copia del archivo y una promesa de <i>devolución de llamada</i> a la estación de trabajo. Registra la promesa de devolución de llamada.
	Abre el archivo local y devuelve el descriptor de archivo a la aplicación.	Sitúa la copia del archivo en el sistema de archivos local, inserta su nombre local en la caché de lista local y devuelve el nombre local a UNIX.	←	
<i>read(DescriptorArchivo, Búfer, tamaño)</i>	Realiza una operación de lectura UNIX normal sobre la copia local.		→	
<i>write(DescriptorArchivo, Búfer, tamaño)</i>	Realiza una operación de escritura UNIX normal sobre la copia local.		→	
<i>close(DescriptorArchivo)</i>	Cierra la copia local y notifica a Venus que el archivo ha sido cerrado.	Si la copia local ha sido cambiada, envía una copia al servidor Vice que es el custodio del archivo.	→	Reemplaza los contenidos del archivo y envía una <i>devolución de llamada</i> a todos los otros clientes que posean <i>promesas de devolución de llamada</i> sobre el archivo.

Figura 8.14. Implementación de las llamadas al sistema en AFS.

La Figura 8.14 describe las acciones tomadas por Vice, Venus y el núcleo UNIX cuando un proceso de usuario emite cada llamada del sistema mencionada en nuestro bosquejo de escenario anterior. La promesa de *devolución de llamada* (*callback*) mencionada aquí es un mecanismo para asegurar que las copias en la caché de los archivos son actualizadas cuando otro cliente cierra el mismo archivo después de actualizarlo. Este mecanismo se discute en la sección siguiente.

8.4.2. CONSISTENCIA DE LA CACHÉ

Cuando Vice proporciona una copia de un archivo al proceso Venus también le proporciona una *promesa de devolución de llamada*, un testigo emitido por el servidor Vice que es el custodio del

archivo, que garantiza que notificará al proceso Venus cuando otro cliente modifica el archivo. Las promesas de devolución de llamada se almacenan con los archivos en la caché sobre los discos de las estaciones de trabajo y tienen dos estados: *válida* o *cancelada*. Cuando un servidor realiza una solicitud para actualizar un archivo notifica a todos los procesos Venus a los que ha emitido promesas de devolución de llamada enviando *una devolución de llamada* a cada uno, una promesa de devolución de llamada es una llamada a un procedimiento remoto desde el servidor al proceso Venus. Cuando el proceso Venus recibe una devolución de llamada, coloca el testigo de la *promesa de devolución de llamada* para el archivo relevante a *cancelada*.

Cuando Venus usa *open* en nombre de un cliente, comprueba la caché. Si el archivo requerido se encuentra en la caché, se comprueba el testigo. Si su valor es *cancelada*, entonces debe buscarse una copia reciente del archivo en el servidor Vice, pero si el testigo es *válida*, entonces puede ser abierta y utilizada la copia de la caché sin referenciar a Vice.

Cuando una estación de trabajo se rearranca después de un fallo o una parada, Venus intenta retener tanto archivos de la caché en el disco local como sea posible, pero no puede presuponer que los testigos de promesa de devolución de llamada sean correctos, puesto que algunas devoluciones de llamada pueden haberse perdido. Por cada archivo con un testigo válido, Venus debe enviar una solicitud de validación de la caché contenido la marca de tiempo de modificación del archivo al servidor que es el custodio del archivo. Si la marca de tiempo es actual, el servidor responde con *válida* y el testigo es rehabilitado. Si la marca de tiempo muestra que el archivo está caducado, el servidor responde con *cancelada* y el testigo es colocado a *cancelada*. Las promesas de devolución de llamada deben ser renovadas antes de un *open* si ha transcurrido un tiempo T (normalmente del orden de unos pocos minutos) desde que el archivo fue introducido en la caché sin comunicación con el servidor. Esto se hace para tratar los posibles fallos de comunicación, que producen la pérdida de mensajes de devolución de llamada.

Este mecanismo basado en devoluciones de llamada para mantener la consistencia de la caché se adoptó porque ofrecía la aproximación más escalable, siguiendo la evaluación en el prototipo (AFS-1) de un mecanismo basado en marca de tiempo semejante al utilizado en NFS. En AFS-1, un proceso Venus que mantiene una copia en la caché de un archivo interroga al proceso Vice en cada *open* para determinar si la marca de tiempo en la copia local concuerda con la del servidor. La aproximación basada en devoluciones de llamada es más escalable porque produce comunicación entre el cliente y el servidor y actividad en el servidor sólo cuando el archivo ha sido actualizado, mientras que la aproximación por marca de tiempo produce una interacción cliente-servidor en cada *open*, incluso cuando hay una copia válida en la caché. Puesto que la mayoría de los archivos no son accedidos concurrentemente, y las operaciones *read* predominan sobre las *write*, el mecanismo de *devolución de llamada* redundante en una reducción dramática del número de interacciones cliente-servidor.

El mecanismo de devolución de llamada utilizado en AFS-2 y versiones posteriores de AFS requiere que los servidores Vice mantengan algo del estado en nombre de sus clientes Venus a diferencia de AFS-1, NFS y nuestro modelo de servicio de archivos. El estado precisado dependiente del cliente precisado consiste en una lista de los procesos Venus para los que se han planteado promesas de devolución de llamada para cada archivo. Estas listas de devoluciones de llamada deben retenerse aun en el caso de fallos en el servidor, mantenerse en los discos del servidor y ser actualizadas utilizando operaciones atómicas.

La Figura 8.15 muestra las llamadas RPC proporcionadas por los servidores AFS para operaciones sobre archivos (esto es, la interfaz proporcionada por los servidores AFS para los procesos Venus).

◊ **Semántica de actualización.** El objetivo de este mecanismo de coherencia de caché es conseguir la mejor aproximación a la semántica de *una copia* de archivo que sea practicable sin una degradación considerable de las prestaciones. Una implementación estricta de la semántica de

<i>Fetch(ida) → atrib, datos</i>	Devuelve los atributos (estado) y, opcionalmente, los contenidos del archivo identificado por el <i>ida</i> y registra una promesa de devolución de llamada sobre él.
<i>Store(ida, atrib, datos)</i>	Actualiza los atributos y (opcionalmente) los contenidos de un archivo especificado.
<i>Create() → ida</i>	Crea un nuevo archivo y registra una promesa de devolución de llamada sobre él.
<i>Remove(ida)</i>	Elimina el archivo especificado.
<i>SetLock(ida, modo)</i>	Establece un bloqueo sobre el archivo o directorio especificado. El modo del bloqueo puede ser compartido o exclusivo. Los bloqueos no eliminados expiran a los 30 minutos.
<i>ReleaseLock(ida)</i>	Desbloquea el archivo o directorio especificado.
<i>RemoveCallback(ida)</i>	Informa al servidor de que un proceso Venus ha volcado un archivo desde su caché.
<i>BreakCallback(ida)</i>	Esta llamada se realiza desde un servidor Vice a un proceso Venus. Cancela la promesa de devolución de llamada del archivo en cuestión.

Nota: No se muestran las operaciones de directorio ni las administrativas (*Rename, Link, RemoveDir, GetTime, CheckToken*, y demás).

Figura 8.15. Componentes principales de la interfaz de servicio Vice.

una copia para las primitivas de acceso a archivos UNIX requeriría que los resultados de cada *write* a un archivo fueran distribuidos a todos los sitios en los que se mantenga el archivo en la caché antes de que puedan ocurrir otros accesos. Esto no es practicable en los sistemas de gran escala, en su lugar, el mecanismo de promesa de devolución de llamada mantiene una aproximación bien definida a la semántica de una copia.

Para AFS-1 la semántica de actualización puede ser establecida formalmente en muy pocos términos. Para un cliente *C* operando sobre un archivo *F* cuyo custodio es un servidor *S*, se mantienen las siguientes garantías de actualidad de las copias de *F*:

después de un <i>open</i> con éxito:	<i>latest(F, S)</i> -el último
después de un <i>open</i> fallido:	<i>failure(S)</i> -fallo
después de un <i>close</i> con éxito:	<i>updated(F, S)</i> -actualizado
después de un <i>close</i> fallido:	<i>failure(S)</i> -fallo

Donde *latest(F, S)* indica una garantía de que el valor actual de *F* en *C* es el mismo que el valor en *S*, *failure(S)* indica que la operación *open* o *close* no ha sido realizada en *S* (y el fallo puede ser detectado por *C*), y *updated(F, S)* indica que el valor de *F* en *C* ha sido propagado con éxito hacia *S*.

Para AFS-2, la garantía actual es ligeramente más débil, y la declaración formal correspondiente de la garantía es más compleja. Esto es porque un cliente puede abrir una copia antigua de un archivo después de que haya sido actualizado por otro cliente. Esto ocurre si se pierde un mensaje de *devolución de llamada*, por ejemplo como resultado de un fallo de la red. Pero hay un máximo de tiempo *T* para el que un cliente puede permanecer sin darse cuenta de una versión más nueva de un archivo. Aquí tenemos la garantía siguiente:

después de un <i>open</i> con éxito:	<i>latest(F, S, 0)</i> o <i>(lostCallback(S, T) e inCache(F) y latest(F, S, T))</i>
--------------------------------------	--

Donde *latest(F, S, T)* indica que la copia de *F* vista por el cliente está caducada no más allá de *T* segundos, *lostCallback(S, T)* indica que se ha perdido un mensaje de devolución de llamada desde *S* hasta *C* en algún instante durante los últimos *T* segundos, y *inCache(F)* que el archivo *F* estaba en la caché en *C* antes que fuera intentada la operación *open* intentada. La declaración formal anterior expresa el hecho que la copia en la caché de *F* en *C* después de una operación *open* es la

versión más reciente en el sistema o que un mensaje de devolución de llamada se ha perdido (debido a un fallo en la comunicación) y la versión que ya estaba en la caché ha sido utilizada, la versión en la caché no estará caducada más que en T segundos. (T es una constante del sistema que representa el intervalo al que deben ser renovadas las promesas de devolución de llamada. En la mayoría de las instalaciones el valor de T se pone en diez minutos aproximadamente). En línea con su objetivo, proporcionar a gran escala, un servicio de archivos distribuidos compatible UNIX, AFS no proporciona ningún otro mecanismo para el control de las actualizaciones concurrentes. El algoritmo de consistencia de la caché descrito anteriormente entra en acción sólo en las operaciones *open* y *close*. Una vez un archivo ha sido abierto, el cliente puede acceder y actualizar la copia local de cualquier forma que él elija sin el conocimiento de ninguno de los procesos en las otras estaciones de trabajo. Cuando se cierra un archivo, se devuelve una copia al servidor reemplazando la versión actual.

Si los clientes en diferentes estaciones de trabajo hacen *open*, *write* y *close* sobre el mismo archivo concurrentemente, todas las actualizaciones menos la resultante del último *close* se perderán silenciosamente (no se da ningún informe de error). Los clientes deben implementar el control de concurrencia independientemente si lo precisan. Por otro lado, cuando dos procesos cliente en la misma estación de trabajo abren un archivo, comparten la misma copia en la caché y las actualizaciones son realizadas en la forma normal UNIX, bloque a bloque.

Aunque las semánticas de actualización difieren, dependiendo de las ubicaciones de los procesos concurrentes que acceden a un archivo, y no son precisamente las mismas que las proporcionadas por el sistema de archivos estándar UNIX, son lo suficientemente próximas para que la gran mayoría de programas UNIX existentes trabajen correctamente.

8.4.3. OTROS ASPECTOS

◊ **Modificaciones al núcleo UNIX.** Hemos indicado que el servidor Vice es un proceso a nivel de usuario ejecutándose en el computador del servidor y la máquina del servidor está dedicada a la provisión de un servicio AFS. El núcleo UNIX en las máquinas AFS está alterado para que Vice pueda realizar operaciones en archivos en términos de apuntadores de archivo en lugar de los convencionales descriptores de archivo UNIX. Ésta es la única modificación al núcleo requerida por AFS y es necesaria si Vice no mantiene ningún estado del cliente (como descriptores de archivo).

◊ **Base de datos de ubicaciones.** Cada servidor contiene una copia de una base de datos de ubicaciones totalmente replicada proporcionando una correspondencia de los nombres de volúmenes para los servidores. Pueden ocurrir inexactitudes temporales en esta base de datos cuando se recoloca un volumen, pero son inocuas porque la información emitida se deja en el servidor desde el que se mueve el volumen.

◊ **Hilos.** Las implementaciones de Vice y Venus hacen uso de un paquete de hilos sin desalojo para permitir que las solicitudes sean procesadas concurrentemente, tanto en el cliente (donde varios procesos de usuario pueden tener solicitudes de acceso al archivo en progreso concurrentemente) como en el servidor. En el cliente, las tablas que describen los contenidos de la caché y la base de datos de volúmenes son mantenidas en memoria, que es compartida entre los hilos de Venus.

◊ **Rélicas de sólo lectura.** Los volúmenes conteniendo archivos que son leídos frecuentemente pero modificados raramente, como los directorios */bin* y */usr/bin* de comandos del sistema UNIX y el directorio */man* de páginas del manual, pueden estar replicados como volúmenes de sólo lectura en varios servidores. Cuando se hace esto, sólo hay una réplica de *lectura escritura* y todas las actualizaciones se dirigen a ella. La propagación de los cambios en la réplica de sólo

lectura, se realiza después de la actualización por un procedimiento operativo explícito. Las entradas en la base de datos de ubicaciones para volúmenes que están replicados de esta forma son *una a muchas*, y el servidor para cada solicitud del cliente se selecciona en base a las cargas y accesibilidad de los servidores.

◊ **Transferencias al por mayor.** AFS transfiere archivos entre los clientes y los servidores en trozos de 64-kilobytes. El uso de paquetes de tan gran tamaño es una ayuda importante para las prestaciones, minimizando el efecto de latencia de la red. Por tanto el diseño de AFS permite optimizar el uso de la red.

◊ **Cacheado parcial de archivos.** La necesidad de transferir el contenido completo de los archivos a los clientes incluso cuando el requisito de la aplicación es para leer tan sólo una pequeña porción del archivo, es una fuente de inefficiencia obvia. La versión 3 de AFS elimina este requisito permitiendo que los datos del archivo sean transferidos e introducidos en la caché en bloques de 64-kbytes mientras se mantiene todavía la semántica de consistencia y otras características del protocolo AFS.

◊ **Prestaciones.** El objetivo principal de AFS es la escalabilidad, por lo que sus prestaciones con un gran número de usuarios son de particular interés. Howard y otros [1988] dan detalles de una extensa comparativa de medidas de prestaciones, que fueron acometidas utilizando AFS *benchmark* desarrollado con este fin, y que ha sido utilizado posteriormente ampliamente para la evaluación de sistemas de archivos distribuidos. Como era de esperar, el cacheado de los archivos completos y el protocolo de devolución de llamadas derivaron una drástica reducción de las cargas en los servidores. Satyanarayanan [1989a] sostiene que se midió una carga del servidor del 40% con 18 nodos cliente ejecutando una prueba estándar, frente a una carga del 100% para NFS evaluando la misma prueba. Satyanarayanan atribuye muchas de las ventajas de prestaciones de AFS a la reducción en la carga del servidor derivada del uso de devoluciones de llamada para notificar a los clientes de las actualizaciones en los archivos, comparado con el mecanismo de timeout utilizado en NFS para la comprobación de la validez de las páginas en las cachés de los clientes.

◊ **Soporte de área amplia.** La versión 3 de AFS soporta múltiples celdas administrativas, cada una con sus propios servidores, clientes, administradores de sistema y usuarios. Cada celda es un entorno completamente autónomo, pero una federación de celdas puede cooperar presentando a los usuarios un espacio de nombres de archivo uniforme de una pieza. El sistema resultante fue ampliamente utilizado por Transarc Corporation y fue publicado un resumen detallado de los patrones de uso de las prestaciones resultantes [Spasojevic y Satyanarayanan 1996]. El sistema se instaló en unos 1.000 servidores en 150 sitios. El resumen mostró proporciones de éxito de la caché en el rango de 96-98 % para accesos a una muestra de 32.000 volúmenes de archivos manteniendo 200 Gbytes de datos.

8.5. AVANCES RECIENTES

Desde la aparición de NFS y AFS se han realizado varios avances en el diseño de sistemas de archivos distribuidos. En esta sección, describimos los avances que mejoran las prestaciones, disponibilidad y escalabilidad de los sistemas de archivos distribuidos convencionales. Avances más radicales se describen en otras partes del libro, incluyendo el mantenimiento de la consistencia en sistemas de archivos de *lectura escritura* replicados, para soportar la operación de modo desconectado y alta disponibilidad en los sistemas Bayou y Coda (véase las Secciones 14.4.2 y 14.4.3), y una arquitectura altamente escalable para la entrega de secuencias de datos en tiempo real con calidad garantizada en el servidor de archivos de video Tiger (véase la Sección 15.6).

◊ **Mejoras NFS.** Varios proyectos de investigación han tratado la cuestión de la semántica de actualización de una copia extendiendo el protocolo NFS para incluir las operaciones *open* y *close* y añadir un mecanismo de devolución de llamada para permitir al servidor notificar a los clientes la necesidad de invalidar entradas en la caché. Aquí describimos dos de tales esfuerzos, sus resultados parecen indicar que estas mejoras pueden ser acomodadas sin complejidad excesiva o costes de comunicación extras.

Agunos esfuerzos recientes de Sun y otros desarrolladores NFS se han dirigido a hacer más accesibles y útiles los servidores NFS en redes de área ancha. Mientras que el protocolo HTTP soportado por los servidores web ofrece un método efectivo y altamente escalable para hacer disponibles los archivos completos a los clientes a través de Internet, es menos útil para los programas de aplicación que requieren el acceso a porciones de grandes archivos o aquellos que actualizan porciones de archivos. El desarrollo WebNFS (describo posteriormente) hace posible que los programas de aplicación lleguen a ser clientes de los servidores NFS en cualquier sitio de Internet (utilizando el protocolo NFS directamente en lugar de hacerlo indirectamente a través de un módulo del núcleo). Esto, junto con las bibliotecas apropiadas para Java y otros lenguajes de programación de red, deben ofrecer la posibilidad de implementar aplicaciones de Internet para compartir datos directamente, como juegos multiusuario o clientes de grandes bases de datos dinámicas.

Obtención de la semántica de actualización de una copia: La arquitectura de servidor sin estado de NFS proporciona grandes ventajas en robustez y facilidad de implementación para NFS, pero evita la consecución de una semántica de actualización de *una copia* requerida (los efectos de las escrituras concurrentes por diferentes clientes en el mismo archivo no está garantizado que sean las mismas que serían en un sistema único UNIX cuando múltiples procesos escriben en un archivo local). También impide la utilización de devoluciones de llamada para notificar a los clientes los cambios en los archivos, y esto deviene en solicitudes frecuentes de *getattr* de los clientes para comprobar la modificación de archivos.

Se han desarrollado dos sistemas de investigación que tratan con estas desventajas. Spritely NFS [Srinivasan y Mogul 1989, Mogul 1994] es un desarrollo de un sistema de archivos desarrollado para el sistema operativo distribuido Sprite en Berkeley [Nelson y otros 1988]. Spritely NFS es una implementación del protocolo NFS con la adición de llamadas *open* y *close*. Los módulos de los clientes deben enviar una operación *open* cuando un proceso local a nivel de usuario abre un archivo que está en el servidor. Los parámetros de una operación *open* en Sprite especifican un modo (lectura, escritura o ambas) e incluyen contadores del número de procesos locales que tienen el archivo abierto para lectura y para escritura. De modo semejante, cuando un proceso local cierra un archivo remoto se envía una operación *close* al servidor con los contadores actualizados de los lectores y los escritores. El servidor registra esos números en una *tabla de archivos abiertos* con la dirección IP y el número de puerto del cliente.

Cuando un servidor recibe un *open*, comprueba la *tabla de archivos abiertos* para otros clientes que tengan el mismo archivo abierto y envía mensajes de devolución de llamada a dichos clientes instruyéndoles para modificar su estrategia de caché. Si *open* especifica el modo de escritura, entonces fallará si cualquier otro cliente tiene el archivo abierto para escritura. Los otros clientes que tengan el archivo abierto para lectura serán instruidos para invalidar cualquier porción del archivo en la caché local.

Para operaciones *open* que especifican modo de lectura, el servidor envía un mensaje de devolución de llamada a cualquier cliente que esté escribiendo, indicándole que detenga la introducción en la caché (*caching*) (es decir, que utilice estrictamente el modo de escritura a través), e instruye a todo los clientes que están leyendo dejen de introducir en la caché el archivo (por lo que todas las llamadas de lectura locales producen una solicitud al servidor).

Estas medidas conducen a un servicio de archivos que mantienen la semántica de actualización de una copia UNIX con el coste de llevar algo del estado relacionado con el cliente al servidor.

También permiten ganar alguna eficiencia en el mantenimiento de las escrituras en la caché. Si el estado relacionado con el cliente se mantiene en memoria volátil en el servidor se vuelve vulnerable a los fallos del servidor. Spritely NFS implementa un protocolo de recuperación que interroga a una lista de clientes con archivos abiertos recientemente en el servidor para recuperar la *tabla de archivos abiertos* completa. La lista de clientes se almacena en el disco, se actualiza relativamente con poca frecuencia y es «pesimista». Puede incluir de forma segura más clientes que los que tienen archivos abiertos en el momento de una caída. Los clientes fallidos pueden producir también entradas en exceso en la *tabla de archivos abiertos*, pero se eliminarán cuando el cliente rearanque.

Cuando se evaluó Spritely NFS frente a NFS versión 2, mostró una modesta mejora de prestaciones. Esto se debió a la introducción de una mejorada caché de escritura. Los cambios en NFS versión 3 resultarían probablemente en una mejora al menos tan grande, pero los resultados del proyecto Spritely NFS indican claramente que es posible conseguir la semántica de actualización de *una copia* sin pérdida sustancial de prestaciones, aunque a costa de alguna complejidad de implementación extra en los módulos cliente y servidor y la necesidad de un mecanismo de recuperación, para restablecer el estado después de una caída del servidor.

NQNFS: El proyecto NQNFS (Not Quite NFS) [Macklem 1994] tenía objetivos similares a Spritely NFS, añadir consistencia de caché más precisa al protocolo NFS y mejorar las prestaciones a través de un mejor uso de la caché. Un servidor NQNFS mantiene un estado relativo al cliente relacionado con los archivos abiertos similar, pero utiliza concesiones (véase la Sección 5.2.6) para ayudar a la recuperación después de una caída del servidor. El servidor coloca un límite superior en el tiempo para que el cliente puede mantener una concesión en un archivo abierto. Si el cliente desea continuar más allá de ese tiempo, debe renovar la concesión. Las devoluciones de llamada se utilizaban de una manera similar a Spritely NFS para solicitar a los clientes que vuelquen sus cachés cuando ocurre una solicitud de escritura, pero si los clientes no contestan, el servidor simplemente espera hasta que sus concesiones expiren antes de responder a la nueva solicitud de escritura.

WebNFS: La llegada del Web y los applets Java condujeron al reconocimiento, por el equipo de desarrollo de NFS y otros, de que algunas aplicaciones Internet podrían beneficiarse del acceso directo a los servidores NFS sin muchas de las sobrecargas asociadas mediante la emulación de las operaciones de archivos UNIX incluida en los clientes estándar de NFS.

El objetivo de WebNFS (describo en las RFC 2055 y 2056 [Callaghan 1996a, 1996b]) es permitir a los navegadores web, programas Java y otras aplicaciones interaccionar con un servidor NFS directamente para acceder a archivos que se encuentran «publicados» utilizando un apuntador de archivos público para acceder a los archivos relativos a un directorio raíz público. Este modo de utilización evita el servicio de montado y el servicio de enlace de puertos (*portmapper*, descrito en el Capítulo 5). Los clientes WebNFS interaccionan con un servidor NFS en un número de puerto bien conocido (2049). Para acceder a los archivos por nombre de ruta ellos lanzan solicitudes *lookup* utilizando un apuntador público de archivo. El apuntador público de archivo tiene un valor bien conocido que es interpretado especialmente por el sistema de archivos virtual en el servidor. Debido a la alta latencia de las redes de área amplia, se utiliza una variante *multicomponente* de la operación *lookup* para buscar un nombre de ruta compuesta en una única solicitud.

Por tanto WebNFS permite escribir clientes que accedan a porciones de archivos almacenados en servidores NFS en sitios remotos con mínimas sobrecargas de inicialización. Se proporciona control de acceso y autenticación, pero en muchos casos el cliente sólo requerirá acceso de lectura a archivos públicos y en ese caso la opción de autenticación puede ser deshabilitada. Para leer una porción de un único archivo localizado en un servidor NFS que soporta WebNFS se precisa el establecimiento de una conexión TCP y dos llamadas RPC, un *lookup* compuesto y una operación *read*. El tamaño del bloque leído no está limitado por el protocolo NFS.

Por ejemplo, un servicio meteorológico debería publicar un archivo en su servidor NFS que contiene una gran base de datos meteorológicos actualizados frecuentemente con un URL como:

```
nfs://data.weather.gov/weatherdata/global.data
```

Un cliente interactivo WeatherMap, que muestra mapas del tiempo, podría estar construido en Java u otro lenguaje que soporte una biblioteca de procedimientos de WebNFS. El cliente lee sólo aquellas porciones del archivo /weatherdata/global.data que fueran necesarias para construir los mapas particulares solicitados por un usuario, mientras que una aplicación semejante que utilizara HTTP para acceder a los datos meteorológicos debiera haber transferido la base de datos completa al cliente o debiera solicitar el soporte de un programa del servidor de propósito especial para proporcionarle los datos que precisa.

NFS versión 4: Se encuentra en desarrollo una nueva versión del protocolo NFS en el momento de publicación de este libro. Los objetivos de NFS versión 4 están descritos en la RFC 2624 [Shepler 1999] y en el libro de Ben Callaghan [Callaghan 1999]. Como WebNFS, pretende convertirlo y hacerlo práctico para utilizar NFS en redes de área extendida y aplicaciones Internet. Incluye las características de WebNFS, pero la introducción de un nuevo protocolo también ofrece una oportunidad para hacer mejoras más radicales. (WebNFS estaba restingido a cambios en el servidor que no implicuen la adición de nuevas operaciones al protocolo).

El grupo de trabajo que desarrolla NFS versión 4 intenta explotar los resultados que han emergido de la investigación en el diseño de servidores de archivos en la década pasada, como el uso de devoluciones de llamada o contratos para mantener la consistencia. NFS soportará la recuperación al vuelo de los fallos en el servidor permitiendo a los sistemas de archivos sean trasladados a nuevos servidores transparentemente. La escalabilidad será mejorada utilizando servidores proxy de una forma análoga a su uso en la Web.

◊ **Mejoras en AFS.** Hemos mencionado que DCE/DFS, el sistema de archivos distribuidos incluido en el Entorno de Computación Distribuido (DCE) de la Open Software Foundation [www.opengroup.org], estaba basado en el Andrew File System. El diseño de DCE/DFS va más allá que AFS, particularmente en su aproximación a la consistencia de caché. En AFS, las devoluciones de llamada se generan sólo cuando el servidor recibe una operación *close* para un archivo que ha sido actualizado. DFS adoptó una estrategia semejante a la de Spritley NFS o NQNFS para generar devoluciones de llamada tan pronto como ha sido actualizado el archivo. Para actualizar un archivo, el cliente debe obtener un testigo *write* del servidor, especificando el rango de bytes del archivo en el que se permite al cliente actualizar el archivo. Cuando se solicita un testigo *write*, los clientes que mantienen copias del mismo archivo para lectura reciben devoluciones de llamada de revocación. Los testigos de otros tipos son utilizados para conseguir consistencia para los atributos y otros metadatos del archivo en la caché. Todos los testigos tienen un tiempo de vida asociado, y los clientes los deben renovar después de que dicho tiempo ha transcurrido.

◊ **Mejoras en la organización del almacenamiento.** Ha habido un progreso considerable en la organización de los archivos de datos almacenados en discos. El ímpetu por mucho de este trabajo surgió de las cargas incrementadas y de la mayor fiabilidad que los sistemas de archivos distribuidos necesitan soportar, y ha producido sistemas de archivos con prestaciones mejoradas sustancialmente. Los principales resultados de este trabajo son:

Redundant Arrays of Inexpensive Disks (*Cadenas Redundantes de Discos Baratos*, RAID): éste es un modo de almacenamiento [Patterson y otros 1988, Chen y otros 1994] en el que los bloques de datos están segmentados en trozos de tamaño fijo y almacenados en *riscas* entre varios discos, junto con códigos correctores de errores que permiten que los bloques de datos sean

reconstruidos completamente y la operación continúe normalmente en el caso de fallos de disco. RAID produce también mejores prestaciones que un único disco, porque las listas que representan un bloque son leídas y escritas concurrentemente.

Log-structured file storage (*Almacen de archivo estructurado en histórico*, LFS): como Spritley NFS, esta técnica originada en el proyecto de sistema operativo Sprite en Berkeley [Rosenblum y Ousterhout 1992]. Los autores observaron que a medida que cantidades más grandes de memoria principal llegaban a estar disponibles para caché en los servidores de archivos, un incremento en el nivel de éxitos en la caché producía excelentes prestaciones en la lectura, pero las prestaciones en la escritura seguían siendo mediocres. Esto se debía a las altas latencias asociadas con las escrituras de bloques de datos individuales a disco y las actualizaciones asociadas con los datos de los metabloques (esto es, los bloques conocidos como *i-nodos* que mantienen los atributos de los archivos y un vector de apuntadores a los bloques en un archivo).

La solución LFS es acumular un conjunto de escrituras en memoria y entonces realizarlas a disco en segmentos de tamaño fijo, grandes, contiguos. Éstos se llaman *segmentos log* porque los bloques de datos y metadatos son almacenados estrictamente en el orden en el que fueron actualizados. Las copias frescas de los bloques de datos y los metadatos actualizados se escriben siempre, precisando el mantenimiento de un mapa dinámico (en memoria con una copia de respaldo persistente) apuntando a los bloques de *i-nodos*. Se precisa también la recolección de basura de los bloques rancios, con la compactación de los bloques *vivos* para dejar áreas continuas de almacenamiento libres para el almacenamiento de los *segmentos log*. El último es un proceso bastante complejo, es realizado como una actividad en segundo plano de un componente llamado el *limpiador*. Se han desarrollado algunos algoritmos sofisticados para el limpiador basados en los resultados de simulaciones.

A pesar de estos costes extra, la ganancia en prestaciones globales es excelente. Rosenblum y Ousterhout midieron un rendimiento de escritura tan alto como el 70 % de ancho de banda disponible del disco, comparado con menos del 10 % para un sistema de archivos UNIX convencional. La estructura log simplifica también la recuperación después de caídas del servidor. El sistema de archivos Zebra [Hartman y Ousterhout 1995], desarrollado como una continuación del trabajo original LFS, combina escrituras estructuradas en históricos con una aproximación RAID distribuida, los *segmentos log* se subdividen en secciones con corrección de error en los datos y escrituras en los discos en nodos de red separados. Se indican prestaciones de cuatro a cinco veces las de NFS, para las escrituras de archivos grandes, con ganancias más pequeñas para archivos pequeños.

◊ **Nuevas aproximaciones de diseño.** La disponibilidad de redes conmutadas de altas prestaciones (como ATM o Ethernet de alta velocidad conmutado) han provocado varios esfuerzos para proporcionar sistemas de almacenamiento persistente que distribuyan los archivos de datos de una manera altamente escalable y tolerante a fallos entre muchos nodos en una intranet, separando las responsabilidades de lectura y escritura de los datos de las de gestión de los metadatos y solicitudes de servicio de los clientes. A continuación, esbozamos dos de tales desarrollos.

Estas aproximaciones escalan mejor que las de los servidores más centralizados que han sido descritas en secciones precedentes. Demandan generalmente un alto nivel de confianza entre los computadores que cooperan para proporcionar el servicio, porque incluyen un protocolo de bastante bajo nivel para la comunicación con los nodos que mantienen los datos (algo análogo a una API de *disco virtual*). Su alcance aquí está probablemente limitado a una única red local.

xFS: Un grupo de la Universidad de Berkeley en California, propuso una arquitectura de sistema de archivos en red sin servidor y desarrolló un prototipo de implementación, xFS [Anderson y otros 1996]. Su aproximación estuvo motivada por tres factores:

1. La oportunidad proporcionada por las redes de área local (LAN) conmutadas rápidamente para múltiples servidores de archivos en una red de área local para transferir volúmenes de datos a los clientes concurrentemente.
2. La expansión de las demandas para acceso a datos compartidos.
3. Las limitaciones fundamentales de los sistemas basados en servidores de archivos centrales.

En lo concerniente a (3), se refieren al hecho que la construcción de servidores NFS con altas prestaciones precisa un hardware relativamente costoso con múltiples CPU, discos y controladores de red, y que hay límites para el proceso de dividir el espacio de archivos, colocando los archivos compartidos en sistemas de archivos separados montados en diferentes servidores. También apuntan el hecho de que un servidor central representa un punto de fallo único.

xFS es *sin servidor* en el sentido que distribuye las responsabilidades de proceso del servidor de archivos entre un conjunto de computadores disponibles en una red local con la granularidad de archivos individuales. Las responsabilidades de almacenamiento están distribuidas independientemente de la gestión y otras responsabilidades de servicio: xFS implementa un software de sistema de almacenamiento RAID, reparto de los archivos de datos entre los discos de múltiples computadores (desde este punto de vista es un precursor del sistema de archivos de vídeo Tiger que se describirá en el Capítulo 15) junto con una técnica de *estructuración en históricos* de una manera similar a las del sistema de archivos Zebra.

La responsabilidad de la gestión de cada archivo puede reservarse a cualquiera de los computadores que soportan el servicio xFS. Esto se consigue mediante una estructura de metadatos llamada *gestor de correspondencias* (*map manager*), que se replica en todos los clientes y servidores. Los identificadores de archivo incluyen un campo que actúa como un índice en el gestor de correspondencias, y cada entrada en la relación identifica el computador que es responsable actualmente de la gestión del archivo correspondiente. Otras varias estructuras de metadatos, semejantes a las encontradas en otros sistemas de almacenamiento *estructuración en históricos* y RAID, son utilizadas para la gestión del almacenamiento de archivos *estructurada en históricos* y el almacenamiento en disco en *rísticas*.

Se construyó un prototipo preliminar de xFS y se evaluaron sus prestaciones. El prototipo estaba incompleto en el momento de realización de la evaluación, la implementación de recuperación de fallos estaba indefinida y el esquema de almacenamiento *estructurado en históricos* carecía de un componente limpiador para recuperar el espacio ocupado por los históricos de estado y compactar los archivos.

Las evaluaciones de prestaciones llevadas a cabo con este prototipo preliminar utilizaron 32 máquinas de un solo procesador y SPARCStations de Sun con dos procesadores conectados a una red de alta velocidad. Las evaluaciones compararon el servicio de archivos xFS corriendo en hasta 32 estaciones de trabajo con NFS y AFS, ejecutándose en una única SPARCStation de Sun con dos procesadores. Los anchos de Banda de lectura y escritura obtenidos con xFS con 32 servidores excedieron a los de NFS y AFS con servidores de dos procesadores en aproximadamente un factor de 10. La diferencia en prestaciones fue mucho menos marcada cuando xFS fue comparado con NFS y AFS utilizando las pruebas estándar de AFS. Pero en conjunto los resultados indican que el proceso altamente distribuido y la arquitectura de almacenamiento de xFS ofrecen una dirección prometedora para conseguir una mejor escalabilidad en los sistemas de archivos distribuidos.

Frangipani: Frangipani es un sistema de archivos distribuidos desarrollado y utilizado en el Centro de Investigación de Digital Systems (ahora Centro de Investigación de Compaq Systems) [Thekkath y otros 1997]. Sus objetivos son muy semejantes a los de xFS, y como xFS, los aproxima con un diseño que separa las responsabilidades de almacenamiento persistente de otras acciones del servicio de archivos. Pero el servicio de Frangipani está estructurado en dos niveles totalmente independientes. El nivel más bajo está proporcionado por el sistema de discos distribuido Petal [Lee y Thekkath 1996].

Petal proporciona una abstracción de discos virtuales distribuidos entre muchos discos ubicados en múltiples servidores en una red local conmutada. La abstracción de disco virtual tolera la mayoría de los fallos de hardware y software con la ayuda de réplicas de los datos almacenados y balancea automáticamente la carga en los servidores para reubicar datos. Los discos virtuales de Petal son accedidos mediante un manejador de disco UNIX utilizando operaciones estándar de entrada/salida de bloques, por lo que pueden ser utilizados para soportar la mayoría de los sistemas de archivos. Petal añade entre el 10 y 100 % a la latencia de los accesos a disco, pero la estrategia de caché produce rendimientos en las lecturas y escrituras al menos tan buenos como los manejadores de disco subyacentes.

Los módulos de servidor Frangipani se ejecutan en el núcleo del sistema operativo. Como en xFS, la responsabilidad de gestionar archivos y las tareas asociadas (incluyendo la provisión de un servicio de bloqueo de archivos para los clientes) se ha asignado dinámicamente a las máquinas, y todas las máquinas ven un nombre unificado de archivo con accesos coherentes (con la semántica similar a *una copia*) para los archivos compartidos actualizables. Los datos se almacenan en un formato *estructurado en históricos* y en *rísticas* en el depósito de discos virtuales Petal. El uso de Petal libera a Frangipani de la necesidad de administrar el espacio físico de disco, implementando un sistema de archivos distribuidos mucho más sencillo, Frangipani puede emular las interfaces de servicio de varios servicios de archivos existentes, incluyendo NFS y DCE/DFS. Las prestaciones de Frangipani son al menos tan buenas como las de la implementación de Digital del sistema de archivos UNIX.

8.6. RESUMEN

Las características fundamentales de diseño para sistemas de archivos distribuidos son:

- La utilización efectiva de la memoria caché en el cliente para conseguir iguales prestaciones o mejores que las de los sistemas de archivos locales.
- El mantenimiento de la consistencia entre múltiples copias de archivos en las cachés de los clientes cuando son actualizadas.
- La recuperación después de un fallo en el servidor o en el cliente.
- El alto rendimiento en la lectura y escritura de archivos de todos los tamaños.
- La escalabilidad.

Los sistemas de archivos distribuidos son empleados intensamente en la computación de las organizaciones, y sus prestaciones han estado sujetas a muchos ajustes. NFS tiene un protocolo sin estado sencillo, que ha mantenido su posición inicial como la tecnología dominante de los sistemas de archivos distribuidos con la ayuda de mejoras relativamente menores al protocolo, implementaciones muy ajustadas y soporte de hardware de altas prestaciones.

AFS demostró la viabilidad de una arquitectura relativamente sencilla utilizando el estado del servidor para reducir el coste del mantenimiento de la coherencia de las cachés en los clientes. AFS sobrepasa a NFS en muchas situaciones. Los avances recientes han empleado reparto de los datos entre múltiples discos y escritura *estructurada en históricos* para mejorar más las prestaciones y la escalabilidad.

En el estado actual del arte, los sistemas de archivos distribuidos son altamente escalables, proporcionan buenas prestaciones tanto entre las redes de área local como las de áreas grandes, mantienen la semántica de actualización de *una copia* y toleran y se recuperan de los fallos. Los requisitos futuros incluyen soporte para usuarios móviles con operación desconectada y garantías de reintegración automática y calidad del servicio para satisfacer la necesidad de un almacenamiento persistente y la entrega de secuencias de multimedia y otros datos dependientes del tiempo. Los soluciones a estos requisitos se discutirán en los Capítulos 14 y 15.