Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Tesis de Licenciatura en Ciencias de la Computación

Sistemas de Archivos Distribuidos

Gateway NFS-CODA

Salomón Sergio Romano
LU 537/87

Paola Marcela Scherman
LU 56/88

Director: Claudio Righetti

1999

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Sistemas de Archivos Distribuidos, Gateway NFS-CODA

Resumen

Los *sistemas de archivos distribuidos* son componentes fundamentales de los sistemas de información en donde conviven múltiples equipamientos, con distintos sistemas operativos y manejadores de archivos heterogéneos. La heterogeneidad trae aparejada falta de compatibilidad y protocolos comunes para conformar un único *espacio de nombres*, impidiendo así que los clientes vean el sistema de información como un gran sistema de archivos homogéneo.

La información puede estar concentrada en un solo edificio o estar dispersa en zonas geográficas distantes. En este último caso se produce una situación conocida como *conectividad débil* donde los clientes suelen sufrir desconexiones intermitentes o anchos de banda variables. Esto deriva en la necesidad de *caches locales* para atenuar el impacto de las desconexiones conjuntamente con una problemática adicional relacionada con la *consistencia* de actualización de los datos.

En este trabajo se estudian diferentes sistemas de archivos distribuidos; populares como NFS o comercialmente incógnitos como CODA, fuertemente orientado a la computación móvil. Se analizan las características específicas de cada uno de ellos y se realizan comparativas desde distintos puntos de vista.

Finalmente, se propone e implementa una *interfase* entre NFS y CODA, como una solución al problema de la heterogeneidad entre los distintos sistemas de archivos. Esta interfase permite acceder desde un cliente NFS a datos que residen en un servidor CODA.

Tabla de contenidos

CAPITULO 1

Introducción

En este trabajo se analiza la problemática de los file systems distribuidos (DFSs), se diseña e implementa una interfase entre dos de ellos, con el fin de resolver algunos de los problemas inherentes a los file systems distribuidos.

El objetivo de la interfase es acceder a un servidor CODA [16] desde un cliente NFS [13]. Para lograr este objetivo, se estudian varios sistemas de archivos distribuidos, analizándolos desde distintos puntos de vista y resaltando las características sobresalientes de cada uno de ellos. Se revisa el comportamiento ante situaciones conocidas, como ser *espacio de nombres* y *control de concurrencia*. Luego se los compara siguiendo estos y otros conceptos [2].

En este capítulo se describe la situación existente y los problemas que trae aparejada la interacción entre distintos servers y clientes. Se introduce el concepto de cliente *desconectado* y se presentan los requerimientos de funcionalidad de la interfase. Finalmente, se da una reseña de la organización del resto del documento.

1.1 Problemática y Motivación

Como ya se sabe la arquitectura de los DFSs es siempre Cliente-Servidor. Este modelo ha probado ser especialmente conveniente en lo que respecta a escalabilidad [17], donde existen unos pocos servers y una gran cantidad de clientes. El modelo cliente/servidor descompone un gran sistema distribuido en un pequeño núcleo que cambia relativamente poco y un gran número de clientes periféricos menos estáticos [16].

Ante esta realidad se presentan dificultades de compatibilidad entre los servers, también entre los clientes y los servers y una problemática adicional relacionada con la movilidad de los clientes.

La incompatibilidad entre dos servers se nota cuando cada server utiliza un sistema operativo diferente y se quiere compartir información entre sendos file systems. En general el file system tiene que estar soportado en el kernel del sistema operativo. Sucede que a veces no está disponible el file system que utiliza uno en el sistema operativo del otro, imposibilitándose la interoperabilidad entre ambos.

Un caso muy similar se daría entre un cliente y los diversos servers del DFS, debiendo soportar el cliente los protocolos necesarios para acceder a cada uno de ellos.

En el marco de la investigación teórica sobre file systems distribuidos, revisamos los problemas abiertos existentes en el proyecto CODA del CMU (Carnegie Mellon University). Uno de ellos era la falta de portabilidad de CODA a sistemas operativos sin soporte de kernel. Nos pusimos en contacto con el Dr. Peter Braam, director del proyecto CODA, quien nos sugirió realizar una interfase entre NFS y CODA con lo cual contribuiríamos al proyecto.

Nuestro trabajo propone entonces una solución al problema de incompatibilidad entre clientes y servers mediante el uso de una interfase entre dos file systems distribuidos.

Elegimos, de acuerdo a la sugerencia del Dr. Braam, los file systems NFS y CODA. NFS está disponible prácticamente en todas las plataformas, por lo tanto lo modificamos utilizándolo como gateway para acceder a un servidor CODA. Este gateway NFS-CODA soluciona el problema que tendrían los clientes para soportar CODA.

Respecto a la movilidad de los clientes la necesidad es diferente. Mientras que las máquinas cliente están fuertemente conectadas mediante una red local a los servers, acceden a los mismos eficientemente. Pero, eventualmente, necesitarán alejarse de la LAN y a la vez tener la posibilidad de seguir accediendo a archivos que residen en los servers. CODA provee nativamente la característica de operar en forma desconectada. Con la misma interfase que implementamos aportamos una posible solución a este problema para los clientes NFS que quieran acceder a servers CODA.

Esta solución aporta las características de un file system a otro, mientras que también contribuye a la portabilidad, dado que el soporte de kernel será necesario sólo para uno de ellos.

Si bien esta es una solución específica para dos file systems determinados, consideramos a la interfase como un modelo de referencia para resolver problemas entre DFSs heterogéneos.

1.2 Requerimientos de funcionalidad

El objetivo de esta interfase es acceder a un file system CODA desde un cliente NFS. Para ello debemos contar con un cliente NFS que se comunique con un server NFS, y un cliente CODA que se comunique con un server CODA. La interfase está situada entre el server NFS y el cliente CODA. El único soporte de kernel que se necesita es para el cliente NFS, pues los demás módulos se ejecutan como procesos de usuario.

De los estudios teóricos que realizamos, se desprende que la interfase debe cumplir los siguientes requerimientos.

- Atención de las siguientes operaciones de file system:
  - Acceso inicial al file system (`mount()`, `getattr()`).
  - Listado de contenidos de un directorio (`readdir()`, `lookup()`, `lstat()`).
  - Lectura y escritura de archivos regulares (`read()`, `write()`).
  - Apertura y creación de archivos regulares (`open()`, `create()`).
  - Eliminación de archivos regulares (`remove()`).
  - Modificación de atributos de archivos y directorios existentes (`setattr()`).
  - Creación y eliminación de directorios (`mkdir()`, `rmdir()`).
  - Renombramiento de archivos regulares (`rename()`).

- La interfase no debe interferir con el normal funcionamiento del server NFS. A pesar que éste será modificado para servir archivos CODA, continuará brindando servicio de archivos NFS.

- El servidor CODA no debe ser modificado dado que seguirá siendo accedido solamente por clientes CODA.

- El cliente NFS tampoco debe modificarse ya que accederá sólo a servers NFS, independientemente del file system en donde se encuentre el archivo a acceder.

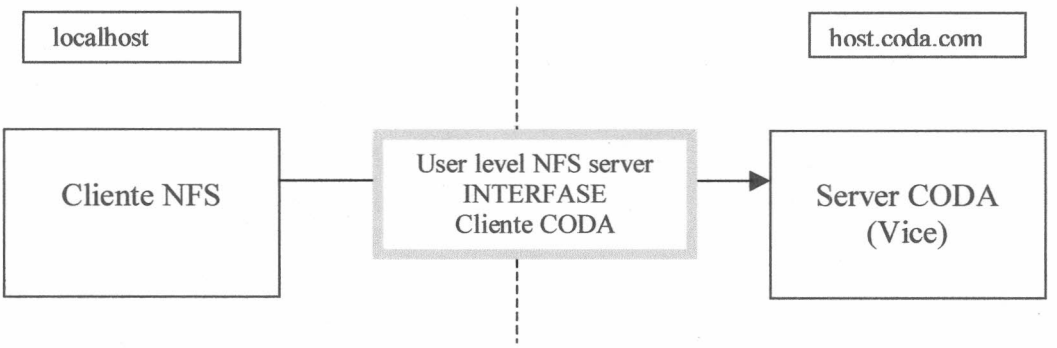

Fig. 1 - Acceso desde un cliente NFS a un server CODA

1.3 Organización del documento

El resto de este documento está dividido en siete capítulos. El capítulo 2 explica las características a través de las cuales se puede describir las capacidades de un file system distribuido y da una introducción a CODA y NFS que son los file systems que componen la interfase. Finalmente, se definen los requerimientos de funcionalidad de la interfase.

El capítulo 3 se dedica por entero a la implementación de la interfase entre los DFSs NFS y CODA. Se explica el funcionamiento de cada uno de los DFSs involucrados, describiendo el flujo de información dentro de cada uno de ellos y el método de acceso entre ambos. Seguidamente, se nombran y justifican las decisiones de diseño tomadas para la implementación. Por último se explica el funcionamiento de las rutinas más relevantes.

En el capítulo 4 se describe la arquitectura de los sistemas de archivos distribuidos, se explica las características fundamentales de los DFSs y se desarrolla en profundidad el concepto de *transparencia*.

En el capítulo 5 se realiza el estudio detallado de los principales DFSs que se encuentran disponibles en la actualidad: AFS, CIFS, CODA y NFS.

En el capítulo 6 se comparan los DFSs estudiados, a partir de las características descriptas en el capítulo 4.

En el capítulo 7 se exponen las conclusiones obtenidas del trabajo y se propone posibles continuaciones y mejoras a esta tesis.

CAPITULO 2

Introducción a los File systems distribuidos

Los file systems están diseñados para almacenar y manejar gran cantidad de archivos con facilidades para crear, nombrar y borrar los archivos. En un file system distribuido el componente equivalente es el *servicio de acceso a archivos*. El diseño de un servicio de archivos debe satisfacer varias *transparencias* [2] que derivarán en mayor flexibilidad y escalabilidad, pero aumentarán la complejidad del software y degradarán la performance.

Decidimos utilizar como casos de estudio CODA y NFS. Siendo NFS el estándar de facto en DFSs y CODA un proyecto del Carnegie Mellon University (CMU) que representa la tecnología emergente en materia de computación móvil. En ambos casos contamos con los fuentes para el sistema operativo Linux.

En este capítulo se dará una introducción a CODA y a NFS, se explicará brevemente cada una de las transparencias y en base a las mismas se planteará el comportamiento resultante del nuevo DFS NFS-CODA.

2.1 Transparencias en un DFS

Se define como transparencia, al acuerdo entre usuarios y programadores de aplicación, en separar los componentes a través del sistema distribuido, de tal manera que el sistema se perciba como un todo, en lugar de un conjunto de componentes independientes [2].

El siguiente cuadro resume las transparencias más relevantes.

Tipo de transparencia	Significado
Localización	Los usuarios no saben dónde están ubicados los recursos.
Migración	Los recursos pueden cambiarse de ubicación a voluntad sin cambiar sus nombres.
Replicación	Los usuarios no saben cuántas copias existen.
Concurrencia	Múltiples usuarios pueden compartir recursos automáticamente.
Acceso	Se refiere a que no es necesario modificar las aplicaciones para acceder a archivos en localizaciones remotas.
Performance	Permite que el sistema sea reconfigurado para mejorar la performance en la medida que varía la carga.
Escalabilidad	Permite al sistema y las aplicaciones expandir su escala sin cambiar la estructura del sistema o los algoritmos de la aplicación.
Fallas	Permite el pasar por alto las fallas, habilitando a los usuarios y programas de aplicación a finalizar las tareas y procesos a pesar de la falla de componentes de hardware y/o software.
Paralelismo	Actividades pueden ocurrir en paralelo sin que los usuarios lo noten.

Las siguientes formas de transparencia son parcial o completamente abordadas por la mayoría de los servicios de archivos: acceso, localización, concurrencia, fallas y performance. Sin embargo, existen otros requerimientos importantes que afectan la utilidad de un servicio de archivos distribuido [2]:

- hardware y sistemas operativos heterogéneos
- escalabilidad
- transparencia de replicación
- transparencia de migración

2.2 Introducción a CODA

CODA, como descendiente directo de AFS, hereda muchos de sus aspectos. Provee gran disponibilidad a través de la replicación de volúmenes y la posibilidad de operar en forma desconectada [16].

Los archivos están organizados en volúmenes para facilitar su administración. Todos los clientes montan el volumen raíz en el directorio `/coda` reservado para este fin, con lo cual todos perciben un único espacio de nombres. Las aplicaciones que corren en los clientes CODA utilizan la interfase standard del file system UNIX, que les permite continuar ejecutándose sin modificaciones.

El cache en los clientes trabaja con archivos completos y es manejado por un proceso llamado Venus. Las modificaciones se reintegran al server al cerrarse el archivo, resultando en una semántica de sesión, es en este momento donde se detectan posibles conflictos de actualización. Es posible detectar actualizaciones realizadas por otros clientes mediante el mecanismo de callback que consiste en obtener una promesa de aviso en caso que se produzcan cambios en el archivo cacheado [17].

Cuando el cliente se desconecta del server, Venus continuará dando servicios de lectura y escritura solamente sobre los archivos que se encuentren en el cache. También en este caso se llevará a cabo el proceso de reintegración en cuanto se restaure la conexión con el server [31].

Los volúmenes pueden ser replicados en otros servers para lectura o lectura/escritura lo cual permite balancear la carga, contribuyendo a la escalabilidad del sistema.

La posibilidad de replicar datos en el server, la capacidad de los clientes para acceder a los archivos en su cache local, junto con la operación desconectada, hacen que el sistema sea robusto y tolerante a fallas (de los servers y de comunicación).

2.3 Introducción a NFS

NFS [27] sigue el modelo cliente-servidor, donde cada cliente arma su espacio de nombres montando en un directorio local (determinado por el cliente) el file system remoto, exportado por el server. El cliente es el encargado de convertir el acceso a archivos que provee el server en un método local logrando así, transparencia de acceso para las aplicaciones.

El server no conserva el estado de los clientes, todas las operaciones son repetibles o idempotentes. Estas dos características confieren robustez al sistema de archivos y transparencia ante fallas de comunicación y caídas del server.

En lo que al cache se refiere, el server utiliza los recursos del sistema operativo en el cual se está ejecutando. En cambio, los clientes cachean los resultados de los system calls agregándoles un timestamp para validar los bloques cacheados al momento de ser usados. Para actualizar el server, el cliente utiliza una ventana de tiempo ajustable. Esta ventana se constituye en una potencial fuente de inconsistencias, aunque esporádicas.

El protocolo NFS no soporta replicación de archivos, lo cual limita el crecimiento en cantidad de clientes ya que los servers se convierten en el cuello de botella del sistema

Para lockear archivos se utiliza un protocolo separado llamado NLM (Network Lock Manager) que mantiene el registro de los archivos lockeados.

2.4 Comportamiento del "DFS" NFS-CODA resultante

Para la visión del usuario, al utilizar un cliente NFS, la interfase se presentará como un file system NFS. Sin embargo, no será un NFS puro, y el usuario deberá estar advertido de las diferencias entre esta interfase y un verdadero NFS, pues se presentarán situaciones que pueden llevar a resultados que no sean los esperados. Algunas características se heredarán de NFS, otras de CODA y algunas tendrán un nuevo comportamiento. A continuación detallaremos las características que poseerá la interfase en los distintos aspectos anteriormente presentados.

Cabe destacar que será posible utilizar la interfase en dos modalidades distintas: una es teniendo el gateway en una máquina remota, atendiendo a varios clientes NFS simultáneamente; la otra es que cada cliente NFS posea el gateway en su PC sirviendo, tanto archivos NFS como archivos CODA a este único cliente. Llamaremos a la primera "gateway local" y a la segunda "gateway remoto". Algunas características de funcionamiento variarán según nos encontremos en la primera o segunda modalidad.

Espacio de nombres y Transparencia de localización

Como el acceso se realizará a través de un mount de NFS, el espacio de nombres no será uniforme entre las estaciones de trabajo. Con respecto a la transparencia de localización, se deberá conocer al momento del montado, el nombre del server con la interfase y el del server en donde corre el servidor de CODA. Sin embargo, una vez montado, dentro del server CODA se gozará de transparencia de localización y de un espacio de nombres uniforme.

Transparencia de acceso (acceso local vs remoto)

Se contará con total transparencia de acceso, con las restricciones impuestas por NFS en el montado soft. De alguna forma hay que tener en cuenta que se podrán producir errores mientras se tenga el archivo abierto. Estos errores se producirán debido a actualizaciones remotas que invaliden los datos cacheados.

Transparencia a la replicación

Se tendrá transparencia a la replicación, como consecuencia de que el acceso final es a un server CODA que proveerá transparentemente réplicas de lectura/escritura.

Transparencia a la migración

Se preservará la transparencia a la migración que tiene CODA pues la mínima unidad a mover es un volumen, el cual estará referenciado desde el server de CODA.

Transparencia de escalabilidad

En la modalidad de "gateway remoto" la transparencia a la escalabilidad estará dada por NFS que, como se sabe, no es muy efectiva. En cambio, en "gateway local" se tendrán las posibilidades ofrecidas por CODA.

Transparencia ante fallas

En la modalidad de "gateway local" la transparencia ante fallas será la misma que tiene CODA. En "gateway remoto" la transparencia que el usuario percibirá será la que ofrece NFS.

Transparencia de concurrencia y consistencia

En la modalidad de "gateway local" la semántica que se impondrá en el cliente es la semántica de sesión con detección de conflictos de actualización, que se hereda de CODA.

En "gateway remoto" existirán dos semánticas superpuestas: una entre los clientes NFS que comparten el gateway, la cual será idéntica a la de NFS; la otra es la que surge de la concurrencia entre el gateway y los demás clientes que el server CODA pudiese atender, que es la misma que tiene CODA.

De esta superposición de semánticas resulta que, la que el cliente NFS finalmente verá, es la de CODA (dentro del mismo Venus, es UNIX; entre distintos Venus es detección de conflictos). Sin embargo existen ciertas particularidades, dadas por el hecho de que el server NFS cierra los archivos luego de una determinada cantidad de tiempo de inactividad. Esto va a resultar en la siguiente situación: supongamos que dos clientes NFS, compartiendo el mismo gateway, tienen abierto un archivo. El primero lo modifica y lo cierra. NFS al cabo de un tiempo, si el segundo no lo accedió, lo cierra. A su vez, pasado un lapso de tiempo, Venus lo reintegrará al server CODA pudiendo generar o no conflictos. En caso de generarse un

conflicto, la próxima vez que el segundo usuario quiera acceder el archivo, NFS se lo pedirá a Venus, que al verificar que el de su cache es inválido lo pedirá nuevamente al server. El server detectará que hubo un conflicto con dicho archivo y retornará un error a Venus. Este devolverá el error a NFS, resultando finalmente en que el segundo usuario no podrá grabar sus modificaciones.

Una situación similar se daría si se reintegra el archivo sin conflictos, otro cliente Venus lee, modifica y reintegra ese archivo. Cuando el segundo usuario quiere grabar el archivo, el NFS se lo pide a Venus, que a su vez se lo pide al server CODA. El archivo es transferido al cache de Venus pero no va a coincidir con el file handle con el que el cliente NFS lo referenciaba, por lo tanto el cliente NFS recibirá un error y no podrá grabar el archivo. En ambos casos el error es devuelto directamente al usuario.

Coherencia de caches

En la modalidad "gateway local" el mecanismo de sincronización de caches estará basado en callbacks, al igual que en CODA.

En "gateway remoto" los buffers de lectura en los clientes estarán regidos por el timestamp que les impone NFS unido a los callbacks que utiliza CODA. Cada vez que se busque un nuevo buffer del server NFS, éste se fijará si el archivo aun permanece abierto en el server; si es así traerá el bloque del cache local de Venus, caso contrario, se solicitará a Venus la reapertura del archivo con el consiguiente chequeo del callback que, en el caso de estar invalidado, procederá a la retransmisión del archivo actualizado desde el server CODA. Todo esto resultará en que los callbacks se chequeen con mayor frecuencia que si se estuviera en un CODA puro y se podrán suscitar los mismos conflictos descriptos en la sección de concurrencia y consistencia.

En el caso de escrituras, entre los clientes NFS que se manejan con delayed writes, se transferirán los buffers al server NFS cuando se realice un close() o un sync(). Si NFS tenía el archivo abierto, grabará las modificaciones, cerrará el archivo y notificará el close a Venus. Venus lo marcará para reintegración. Cuando se reintegre el archivo, el server CODA romperá los callbacks a los demás clientes. Si NFS ya había cerrado el archivo se lo pedirá nuevamente a Venus y se producirá la misma situación ya descripta en la sección de concurrencia.

Locks

En el caso de "gateway local" el cliente NFS podrá loquear bloques entre procesos en la misma workstation.

En la modalidad "gateway remoto" con NLM instalado en el gateway, se proveería de locks entre los clientes NFS, pero ninguno de estos locks se propagaría hacia el server CODA.

Disponibilidad

En "gateway local" se contará con la disponibilidad ofrecida por CODA a través del cache de Venus (posibilidad de operación desconectada) y de las réplicas de lectura/escritura.

En "gateway remoto" entre el cliente NFS y el gateway, existirá la escasa disponibilidad de NFS. Entre el gateway y el server CODA, encontraremos la gran disponibilidad de CODA. Sin embargo, como la cadena se rompe por el eslabón más débil, en este caso la disponibilidad de NFS restringirá a la de CODA.

Seguridad

La seguridad dentro de la interfase (entre NFS server y Venus) no será un objetivo primordial del desarrollo. En la versión de NFS utilizada (NFS 2) la seguridad precaria impone límites en la autenticación entre el cliente NFS y el server CODA. Si esta interfase se reescribiese utilizando NFS versión 3 o superior, habría que encontrar el medio de enviar el token de autenticación Kerberos que solicita el cliente NFS al iniciar la sesión al server CODA para su verificación y otorgamiento de acceso.

Plataformas soportadas

La interfase se ejecutará como un proceso a nivel de usuario y no necesitará integración con el kernel más allá del que NFS requiere, con lo cual estará disponible en sistemas operativos donde sea posible compilarla.

Adicionalmente, cualquier sistema operativo con soporte para cliente NFS podrá ser utilizado para acceder a un server CODA a través de la interfase.

2.5 Resumen

En este capítulo se han nombrado las características fundamentales en el diseño de un sistema de archivos distribuido. Hemos visto el concepto de *transparencia* como el acuerdo entre usuarios y desarrolladores en percibir el sistema como un todo en lugar de componentes aislados. Cada una de estas transparencias dan una idea de las características deseables que un file system distribuido debe cumplir y a su vez concretan la visión del usuario de contar con un único file system homogéneo.

Se comentaron las características básicas que poseen los dos file systems de trabajo y se expuso el comportamiento que tendrá el nuevo DFS, resultante de unir NFS y CODA mediante la interfase propuesta.

CAPITULO 3

Implementación de la interfase

El objetivo de esta interfase es acceder a un file system CODA desde un cliente NFS. Para ello debemos contar con un cliente NFS que se comunique con un server NFS, y un cliente CODA que se comunique con un server CODA. La interfase está situada entre el server NFS y el cliente CODA. El único soporte de kernel que se necesita es para el cliente NFS, pues los demás módulos se ejecutan como procesos de usuario. Los programas fuente se encuentran disponibles para cada uno de estos módulos en Linux, por lo cual la implementación la realizamos en esta plataforma.

3.1 Componentes de los file systems involucrados

El cliente CODA está compuesto por dos módulos. Uno es el que permite al kernel dar soporte de CODA, o sea poder montar un file system CODA y acceder a los archivos; en Linux se llama "coda.o" y es uno de los módulos dinámicos instalables en el kernel. El otro es Venus, un proceso de usuario que se encarga de la comunicación con el server CODA vía UDP y el manejo del cache local.

El server CODA, **Vice**, está compuesto por varios procesos de usuario que administran el volumen exportado a los clientes. A título informativo, el volumen exportado es una partición con formato propietario que sólo es interpretada por los procesos del server CODA, existen módulos para autenticación de usuarios, actualización de datos y reintegración de caches.

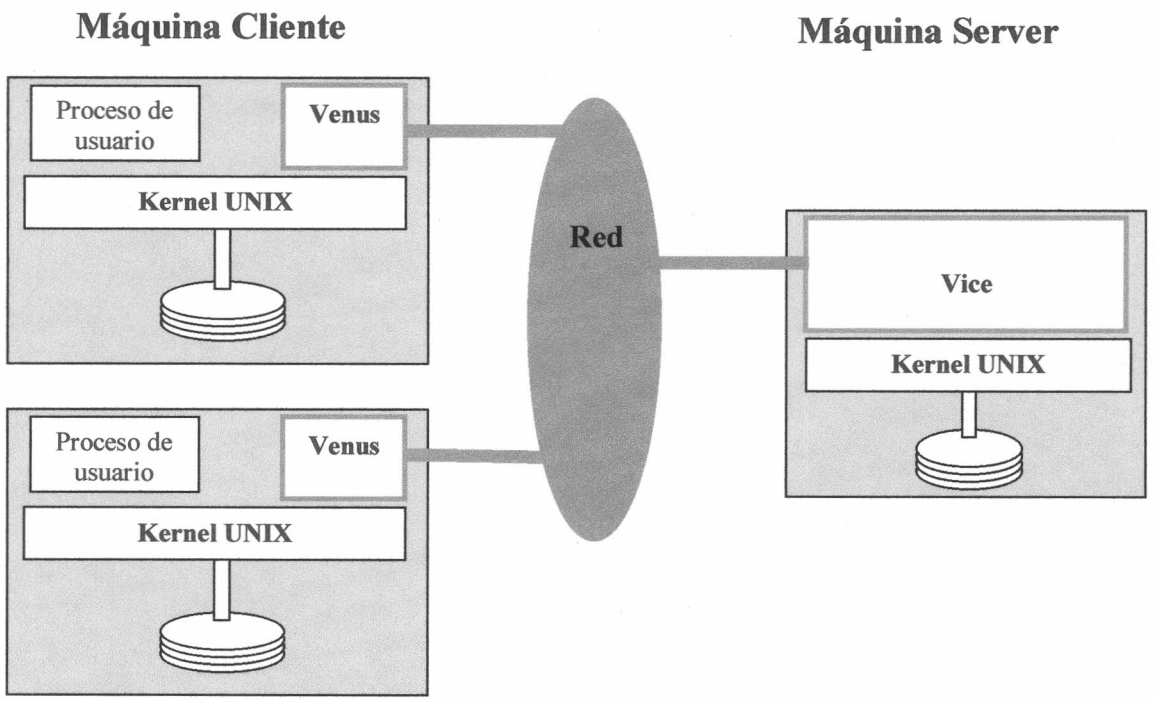

Fig. 2 – Arquitectura cliente/servidor del file system CODA

El cliente NFS en Linux es un módulo standard que se ejecuta en el kernel, se lo puede ver como "nfs.o" utilizando el comando *lsmod* una vez montado un file system remoto.

El server NFS en Linux esta compuesto por dos módulos que se ejecutan en el espacio de usuario. El "rpc.mountd" es uno de ellos y es el encargado de realizar las validaciones adecuadas para una petición de montado. El otro, el "rpc.nfsd" es el encargado de procesar los requerimientos de los clientes. A este módulo llegan las peticiones de read(), create(), write(), etc., que se resuelven usando el file system local y se responden al cliente.

La interfase recibe los requerimientos del rpc.nfsd y los pasa a Venus luego de hacer las conversiones necesarias. Esto se logra modificando Venus para recibir pedidos desde un socket UDP en vez de recibirlos desde el coda.o en el kernel. El rpc.nfsd y rpc.mountd también están modificados para enviar los pedidos a través de un socket UDP en lugar de resolverlos en forma local. Con esta modificación, ya no es necesario contar con ningún módulo de CODA dentro del kernel.

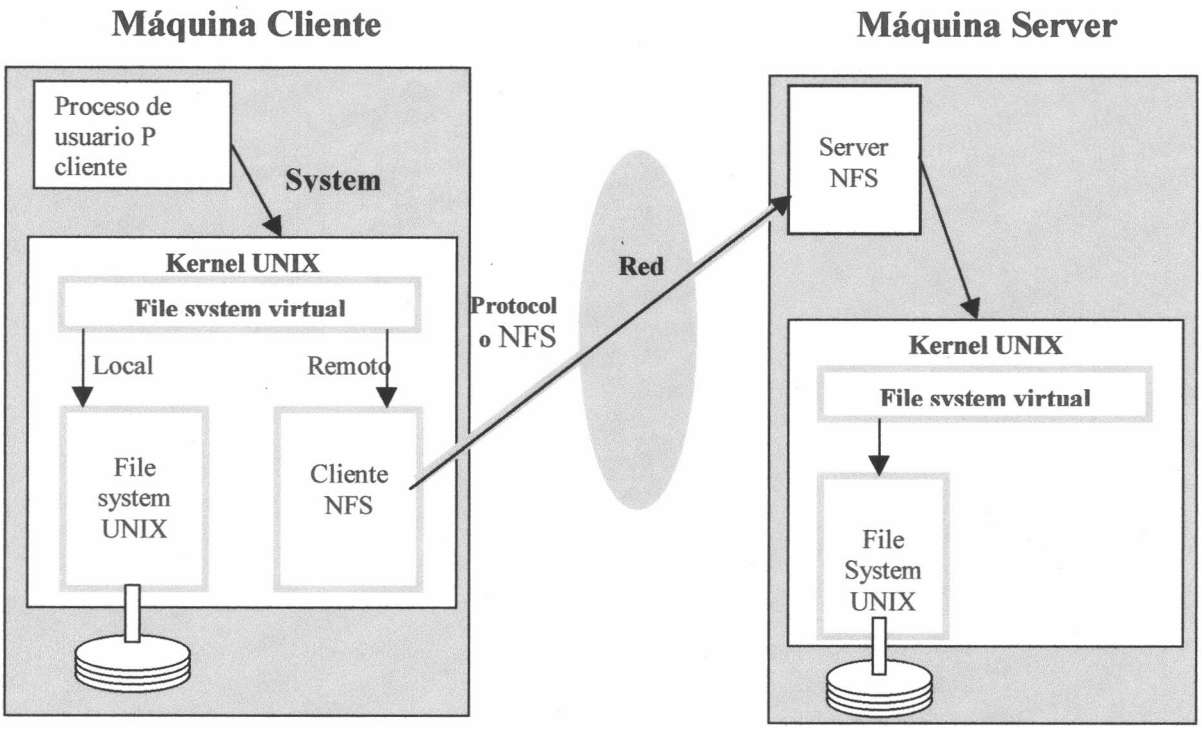

Fig. 3 – Arquitectura cliente/servidor del software NFS

3.2 Atención de system calls

La atención de un requerimiento que se origina en un proceso de usuario **P** para acceder a un file system comienza con un system call que llega al kernel del sistema operativo. Por ejemplo read(), write(), open(), create(), mkdir(), rmdir(), chmod() en el contexto UNIX.

Generalmente el sistema operativo maneja los requerimientos en una capa de file system virtual (VFS). El VFS es el responsable de procesar parcialmente el requerimiento y de localizar el/los file system/s específico/s involucrado/s en la resolución del mismo. Usualmente, la información en el path ayuda a localizar los drivers correctos de file system. Luego de un cuidadoso pre-procesamiento, el VFS comienza a llamar a las rutinas exportadas por los drivers de file system. Este es el punto donde comienza el procesamiento del file system específico, en nuestro caso entran en juego las rutinas de NFS que atienden a los system calls mencionados.

En la figura se observa la interacción entre los módulos, dentro del cliente y el servidor. No se muestran los componentes relacionados con el montado de file systems remotos. Los procesos que utilizan NFS se muestran como "Proceso de usuario P cliente" para distinguirlos del módulo NFS cliente que reside dentro del kernel UNIX en cada máquina cliente. En nuestro caso el módulo server es también un proceso de usuario que se ejecuta en cada máquina que actúa como server. Los requerimientos sobre archivos en file systems remotos son traducidos a operaciones del protocolo NFS por el cliente, y enviados al módulo NFS en la máquina que contiene el file system solicitado.

3.3 Secuencia de llamadas

La secuencia de llamadas entre los distintos módulos que intervienen en la resolución de un requerimiento de un cliente NFS a un file system CODA es la siguiente:

Proceso de usuario
→VFS en el SO (sistema de archivos virtual)
→ Cliente NFS (nfs.o)
→ Server NFS (rpc.mountd y rpc.nfsd modificados)
→ Interfase entre CODA y NFS
→ Cliente CODA (Venus con socket UDP)
→ Server CODA (Vice)

3.4 Acceso a CODA desde NFS

En NFS se necesita montar el file system remoto en un directorio local para poder accederlo. Para ello utilizaremos la siguiente sintaxis:

```
# mount –t nfs gateway.codanfs.com:/host.coda.com /coda
```

Donde:

- *gateway.codanfs.com* es la máquina en donde reside el server NFS con la modificación para acceder a file systems CODA.
- *host.coda.com* es la máquina en donde reside el server CODA que se desea acceder.
- */coda* es el nombre del directorio local donde se monta el file system remoto (punto de montado).

Los nombres de las máquinas gateway.codanfs.com y host.coda.com deben poder ser resueltos por los métodos usuales de resolución de nombres, ya sea en forma local (/etc/hosts) o vía DNS.

Debido a los requerimientos de NFS, necesitamos definir en la máquina que ejecuta el server NFS el directorio /host.coda.com para poder autenticarlo cuando lo agregamos en el /etc/exports con los permisos necesarios.

El método que utiliza el server NFS para detectar qué tipo de file system se quiere montar, es tratar de convertir el path /host.coda.com en una dirección IP. Si esto es posible, significa que deberá montar un file system CODA, si no, se interpretará que se desea montar un file system NFS. Esta tarea es realizada por el módulo rpc.mountd, que retorna un descriptor NFS al cliente. Este descriptor inicial contiene información que permite determinar a qué tipo de file system se refiere y se utilizará en cada requerimiento que se envíe al rpc.nfsd que referencie el punto de montado.

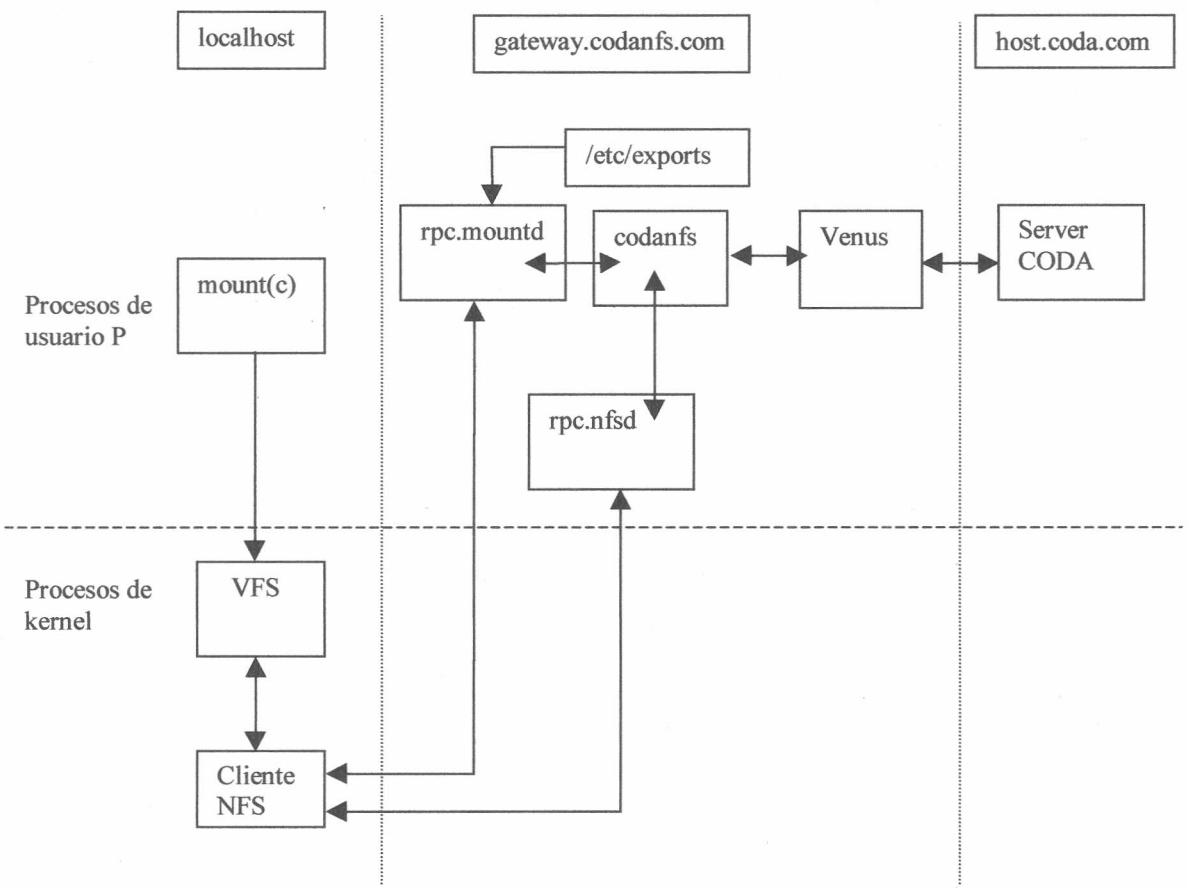

Fig. 4 - Montado de un file system CODA desde un cliente NFS

Las operaciones de archivos que requieran acceso remoto necesitan de un descriptor que determine unívocamente al archivo. En el cliente NFS se conserva un cache local de descriptores. Por cada operación que acceda a un archivo el cliente busca el descriptor en su cache y si no lo encuentra se lo solicita al módulo rpc.nfsd del server NFS. El rpc.nfsd retorna

el descriptor con la información adecuada al tipo de file system correspondiente. Este descriptor será enviado con cada requerimiento que actúe sobre el archivo.

Cada vez que un descriptor llegue al server NFS, se detectará de qué tipo de file system se trata, utilizando para esto la información que se encuentra dentro del mismo. Si se refiere a NFS, el pedido se resuelve en forma local, si no se solicita la información requerida al cliente CODA.

3.5 Decisiones de diseño de la interfase

La interfase bien podría haber sido un proceso que se ejecute independientemente de cada uno de los módulos del server NFS (rpc.mountd y rpc.nfsd). Sin embargo este enfoque agregaría mucho overhead a cada solicitud de datos.

Fig. 5 - Overhead entre procesos.

Cuando se necesita un dato de un server CODA, el cliente NFS se lo pide al server NFS; el server NFS a través de un socket UDP se lo pediría al "proceso interfase"; éste a través de otro socket UDP solicitaría el dato al cliente CODA; finalmente el cliente CODA pedirá el dato por algún medio al server CODA.

Integrando la interfase al server NFS, reducimos el overhead de UDP y el producido por los cambios de contexto de los distintos procesos, ya que los pedidos se hacen a través de llamadas a rutinas de la interfase.

Por lo tanto decidimos integrar la interfase con cada uno de los módulos del server NFS, es decir, cada módulo llama directamente a las rutinas de la interfase sin pasar por ningún socket ni cambio de contexto.

El siguiente paso en la integración de los dos file systems sería incluir las rutinas de la interfase como un thread de Venus, eliminando el overhead que se produce en las llamadas a través de los sockets UDP y cambios de contexto. De esta manera Venus se convertiría en un server NFS.

Dado que esta tesis contribuye al proyecto CODA, que se lleva a cabo en el Carnegie Mellon University (CMU), en este momento liderado por el Dr. Peter Braam, se aceptaron sus recomendaciones y sugerencias para la realización de la interfase.

3.6 Sistema de comunicación entre los distintos procesos

Los módulos que necesitan comunicarse entre sí son:

- rpc.mountd (Server NFS)
- rpc.nfsd (Server NFS)
- interfase CodaNFS
- Venus o Potemkin (Cliente CODA)

La interfase CodaNFS es la que se comunica con todos los módulos mencionados. Por un lado los módulos del server NFS *hablan* con la interfase por medio del **llamado a rutinas** que resuelven las distintas funciones del server (por ejemplo: readdir(), getattr(), etc.). Éstas han sido modificadas para obtener respuestas de un server CODA. Por otro lado, el cliente CODA se comunica con la interfase a través de **sockets UDP** que se abren en cada módulo del server NFS.

Para esto utilizamos ports UDP que no estén en uso. Definimos nuevos servicios en el /etc/services que serán consultados desde el server y el cliente para obtener los ports correspondientes por medio del getserverbyname().

Las entradas UDP del /etc/services que utilizamos son las siguientes[1]:

codanfsmnt: lo usa el rpc.mountd para enviar y recibir mensajes a través de la interfase CodaNFS hacia y desde el cliente CODA (Venus o Potemkin).

codanfsnfs: lo usa el rpc.nfsd para enviar y recibir mensajes a través de la interfase CodaNFS hacia y desde el cliente CODA.

codanfsgtw: se abre en el cliente CODA para recibir mensajes de la interfase CodaNFS, responde al port que originó el mensaje utilizando un port cualquiera.

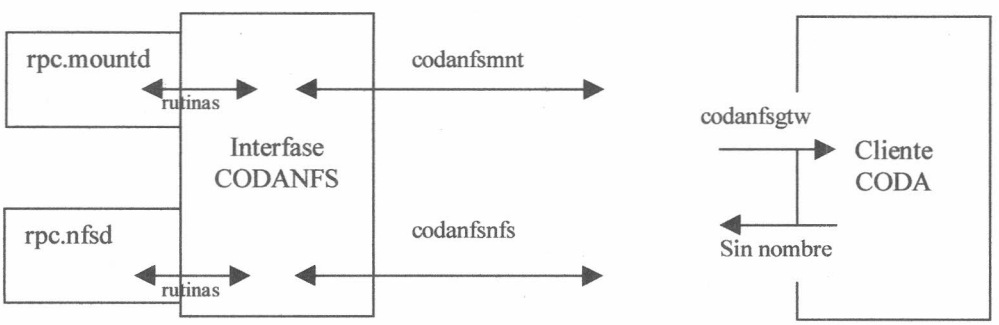

Fig. 6 – Ports utilizados para la comunicación

[1] Ver en el apéndice de archivos de ejemplo el /etc/services.

3.7 Rutinas de comunicación

A continuación se detalla el código de las rutinas utilizadas para la comunicación.

Para abrir el socket en el *mountd*, usamos la rutina `isCodaFs()` que abre el socket en el port **codanfsmnt** cuando detecta que el path que se desea montar se puede resolver como un nombre de host utilizando el gethostbyname(). Se utiliza la rutina `codanfs_SockClose()` para cerrar el socket luego de ser utilizado en el *mountd*.

La apertura del socket en el port **codanfsnfs** en el *nfsd* se realiza usando la rutina `codanfs_SockOpen()` cuando se detecta por primera vez que el dato que se desea acceder pertenece a un server CODA. Este chequeo se realiza a través de la función `isCodaFh()`.

El cliente CODA (Venus o Potemkin) abre el socket en el port **codanfsgtw** en su inicialización.

Para enviar un mensaje desde la interfase al cliente CODA (Venus) se utiliza la rutina **MsgWrite()** que envía la información a través del socket en el port **codanfsgtw**. La interfase espera la respuesta en la rutina **MsgWait()** que se bloquea en un select() hasta que llegue alguna información al port esperado (dependiendo del port que se abrió en el nfsd o mountd). Luego de desbloquearse obtiene la información con la rutina **MsgRead()**. Los pedidos se encolan naturalmente a través de los sockets UDP por orden de llegada. Estas rutinas son usadas solamente en coda_upcall() cada vez que se invoca una operación que requiera acceso remoto al server CODA.

3.8 Estructuras de datos

Un archivo en CODA se determina unívocamente mediante una estructura de 12 bytes llamada **ViceFid** que contiene el *Volume*, *Vnode* y el *Unique*. El *Volume* es un número que identifica un subárbol de archivos dentro del server, el *Vnode* identifica un archivo en particular dentro del volumen y el *Unique* se utiliza para filtrar los ViceFids obsoletos que permanezcan circulando en la red, permite reutilizar los números de *Vnode*. Un ViceFid se convierte en un número de inodo utilizando la siguiente fórmula:

$$(Vol << 20) + (Vnode << 10) + (Unique)$$

Para que un cliente NFS pueda acceder a un archivo debe proveer junto al requerimiento un identificador unívoco del mismo. El protocolo NFS asocia un objeto llamado FileHandle a cada archivo y directorio. El NFS server genera este FileHandle cuando el cliente accede por primera vez al archivo a través de llamadas como lookup() o create(). El server retorna el handle al cliente en respuesta al requerimiento y el cliente puede usarlo subsecuentemente en otras operaciones sobre este archivo.

El cliente ve al FileHandle como un objeto *opaco* de 32 bytes y no intenta interpretar su contenido. El server puede implementar el FileHandle como le plazca mientras garantice la correspondencia unívoca entre archivos y FileHandles.

Para llevar la relación entre FileHandles y ViceFids modificamos la estructura del FileHandle (struct svc_fh) incluyendo el ViceFid dentro de ella. Para ello redujimos el tamaño del hash_path que se utiliza para identificar cada nivel de directorio del path del archivo dentro del FileHandle. Originalmente se podía tener hasta 28 niveles de directorios, con nuestra modificación se redujo a 16 niveles.

```
/* Estas definiciones estan incluidas en el codanfs.h
        #define  VICEFID_LEN  12
        typedef  __u8              vicefid_t[VICEFID_LEN];
*/

#define  HP_LEN                    (NFS_FHSIZE - sizeof(psi_t)-VICEFID_LEN)

typedef struct {
        psi_t           psi;
        __u8            hash_path[HP_LEN];
        vicefid_t       ViceFid;
} svc_fh;
```

En el cache del server NFS (rpc.nfsd), los FileHandles contienen el path -local al server- del archivo que identifican, junto con el svc_fh para poder realizar un lstat en cuanto se soliciten los atributos. En el caso de ser un FileHandle de un archivo CODA, el path estará en NULL pues los atributos se pedirán al server CODA a través de un *codanfs_getattr()*.

El path no viaja en el FileHandle entre el cliente y el server NFS porque el formato del path es dependiente del file system subyacente del sistema operativo. Por ejemplo en DOS se utilizan barras invertidas ("\") mientras que en UNIX se utilizan barras ("/"). Por este motivo el server NFS debe resolver el path al cual se refiere el FileHandle en forma local a partir del punto de montado. Esto lo realiza recreándolo mediante la estructura hash_path en el svc_fh, que contiene identificadores únicos para cada archivo, llamados path_psi, y se calculan mediante una fórmula especial basada en el número de inodo, el minor y major del device donde se encuentra el archivo.

```
typedef struct fhcache {
        struct fhcache *  next;
        struct fhcache *  prev;
        struct fhcache *  hash_next;
        struct fhcache *  fd_next;
        struct fhcache *  fd_prev;
        svc_fh            h;
        int                       fd;
        int                       omode;
        char *            path;
        time_t            last_used;
        nfs_client *      last_clnt;
        nfs_mount *               last_mount;
        uid_t                     last_uid;
        int                       flags;
} fhcache;
```

La interfase es la encargada de convertir las estructuras de datos de los requerimientos de NFS para poder ser procesados por el server CODA, como si viniesen de un módulo de kernel CODA (coda.o). Y a la vez devolver estructuras válidas para NFS, convirtiendo las recibidas desde el cliente CODA (Venus).

Esta conversión de datos se realiza en ambos sentidos con las rutinas: **codavattr_to_statattr()** y **codaiattr_to_vattr()**.

La interfase utiliza lo que se denomina "magic number" para asignar el número de device que identifica a un file system NFS-CODA.

Por cada una de las rutinas que atiende a los systems calls del cliente NFS es necesario incluir el testeo del file system que contiene el dato solicitado (se utiliza la rutina **isCodaFh()**), y en caso de que sea CODA llamar al cliente CODA.

```
int isCodaFh(vicefid_t    * codaFid)
{
        char char0[12];
        memset(&char0, 0,4);
        if (memcmp(codaFid,&char0, 4) != 0)
                return(1);
        else
                return(0);
}
```

3.9 Rutinas utilizadas por el VFS

Existe una rutina que respalda cada una de las operaciones que, como hemos visto, son necesarias en los requerimientos de funcionalidad de la interfase.

- Acceso inicial al file system (`mount()`).

Esta operación se realiza en varios pasos:

- EL cliente NFS le solicita al rpc.mountd el montado de un file system, y espera el file handle inicial del file system.
- El server NFS (rpc.mountd) valida que el directorio que se desea montar está exportado (/etc/exports), en la rutina *mountproc_mnt_1_svc()*.
- Se autentica al cliente NFS.
- El siguiente paso es la creación del file handle inicial, que corresponde al directorio raíz del file system remoto a montar. Esto se realiza en la rutina fh_create. En el caso de un volumen CODA, se llama a la rutina *codanfs_rootfid()*, que retorna el ViceFid del directorio raíz del volumen. Luego, por medio de un *codanfs_getattr()*, se obtienen los atributos de este ViceFid para completar la estructura del file handle con el número de inodo.
- Se guarda en el cache el file handle y se lo devuelve al cliente NFS.
- El cliente NFS toma el file handle recién recibido y solicita los atributos al rpc.nfsd.
- El rpc.nfsd recibe el requerimiento y al verificar que se trata de un file system CODA pide la información utilizando nuevamente la rutina *codanfs_getattr()*.
- Con la respuesta, concluye el montado del file system remoto y asigna los atributos al directorio designado como mount point.

Después de este proceso, estamos en condiciones de hacer un

```
$ ls -ld /coda
```

y obtener los atributos del directorio raíz del file system remoto montado.

- Listado de contenidos de un directorio (`readdir()`).

Esta operación también se realiza en varios pasos:

- El cliente NFS solicita al rpc.nfsd el contenido de un directorio indicando el file handle del mismo.

- El server NFS (rpc.nfsd) en la rutina *nfsd_nfsproc_readdir_2()*:
    1- Valida el file handle del directorio, provisto por el cliente (*auth_fh()*)
    2- Si es un file system CODA llama a *codanfs_open_by_path()* para obtener a partir del ViceFid, que se encuentra dentro del file handle, el nombre del archivo (directorio) **local** que se debe leer. La lectura del directorio se realiza en forma local ya que Venus trabaja con una copia en su cache de cada uno de los archivos del volumen montado. La rutina *codanfs_open_by_path()* llama al *venus_open_by_path()* que no estaba implementada originalmente en el cliente CODA, la agregamos en el upcall.c.
    3- Realiza la lectura del archivo local que contiene las entradas del directorio solicitado, de la misma manera para CODA o para NFS.
    4- Retorna al cliente una estructura con el nombre de cada una de las entradas del directorio .
- Por cada entrada de directorio, si es necesario, se realiza un *lookup()* y un *getattr()* para obtener sus atributos.

- Lectura y escritura de archivos regulares (`read()`, `write()`).

Al igual que la lectura de las entradas de un directorio, estas operaciones son "tramposas", ya que se realizan sobre el cache local del cliente CODA. Las rutinas que invoca el cliente NFS son: *nfsd_nfsproc_read_2()* y *nfsd_nfsproc_write_2()*. Estas rutinas fueron modificadas para descubrir el path local del archivo remoto en cuestión, mediante la rutina *codanfs_open_by_path()*. El resto de la operatoria es idéntica para CODA y para NFS. Recordemos que la actualización del server CODA en el caso de un write(), es responsabilidad del cliente CODA (Venus) que transferirá el archivo al server al momento del cierre del mismo.

- Apertura de archivos regulares (`open()`)

La apertura de un archivo regular, está resuelta por estar implementado el lookup() y el getattr().

- Creación de archivos regulares (`create()`).

Si bien el create() es una variante del open(), necesita un tratamiento especial en NFS. Modificamos la rutina *nfsd_nfsproc_create_2()* para que convierta los atributos con que desea crear el archivo y pida a Venus la creación del archivo con *codanfs_create()*.

Esta operación genera un nuevo archivo en el cache de Venus para poder ser accedido luego.

- Eliminación de archivos regulares (`remove()`).

Esta operación solicita a Venus que remueva el archivo de su cache. También se remueve el FileHandle del cache de NFS. La remoción definitiva del archivo en el server CODA, se realizará cuando Venus disponga la sincronización con el servidor.

- Modificación de atributos de archivos y directorios existentes (`setattr()`).

Para modificar los atributos de un archivo cualquiera, es necesario convertir la estructura provista por el cliente NFS con las modificaciones a ser realizadas, a la estructura esperada por el cliente CODA. Esto se realiza con un *coda_iattr_to_vattr()* . Posteriormente se pide a Venus que realice los cambios en los atributos mediante el *codanfs_setattr()*.

- Creación y eliminación de directorios (`mkdir()`, `rmdir()`).

Estas dos operaciones solicitan a Venus la creación o remoción del directorio de su cache. También se remueve el FileHandle del cache de NFS. Sin embargo, la creación o remoción definitiva del directorio en el server CODA, se realizará cuando Venus disponga la sincronización con el servidor.

- Renombramiento de archivos regulares (`rename()`).

Para renombrar un archivo, se solicita a Venus que valide la existencia de la ruta completa del archivo de origen, si no existe, inmediatamente retorna un error. A continuación, se verifica que no exista el archivo de destino, devolviendo un error si se lo encuentra. Otra posibilidad podría haber sido el reemplazo del archivo de destino, tal como lo realiza Linux.

Renombrar parece una operación sencilla, sin embargo, existen ciertas particularidades a tomar en cuenta al momento de implementarla. Una es, como ya nombramos, si se sobreescribirá el archivo de destino, en caso de existir. Otra, es la forma en que el VFS maneja el renombramiento entre dos archivos que se encuentran en file systems diferentes. En el caso del VFS de Linux, cuando descubre que el destino está en distinto file system que el origen, no se utiliza el "rename" de ninguno de los file systems sino que se ejecuta un remove en el origen y un create en el destino, seguido de la copia del archivo de uno a otro. Sin embargo, otros sistemas operativos, no permiten realizar este tipo de operaciones.

3.10 Resumen

Para implementar la interfase se realizó un estudio detallado de la arquitectura de cada uno de los DFSs involucrados, así como la vinculación entre los distintos módulos a través de los system calls.

Se vio cómo se monta un file system CODA, utilizando el montado de NFS, conjuntamente con los parámetros específicos.

Se plantearon varias alternativas de diseño y se justificó la elegida. Se explicaron las estructuras de datos y cómo se incrustó el ViceFid de CODA dentro del FileHandle de NFS.

Finalmente, se explicó el funcionamiento y comportamiento de las rutinas responsables de cumplir con los pedidos del VFS.

CAPITULO 4

Características de los File Systems Distribuidos

En este capítulo se introducen los conceptos básicos de sistemas de procesamiento. Se muestran las diferencias entre sistemas centralizados y distribuidos, se definen los servicios de acceso a datos, diferenciando entre bases de datos y archivos.

La arquitectura de los DFSs es cliente-servidor, se verán las características principales de los DFSs en relación a esta arquitectura. También se definirán las necesidades básicas que debe cumplir un sistema distribuido para independizar a los usuarios de las complejidades inherentes a la distribución de los datos. Esta independencia se logra a través de distintas transparencias que serán explicadas a medida que se revise cada una de las características básicas.

4.1 Sistemas Centralizados vs. Sistemas Distribuidos

En un ambiente de trabajo en donde múltiples usuarios acceden a una misma base de datos, donde base de datos significa conjunto de ítems no necesariamente relacionados entre sí, existen diferentes formas de organizar estos objetos. La administración puede ser centralizada o distribuida.

En una administración centralizada, los objetos residen en un único host, el cual atiende a los múltiples usuarios. Las actualizaciones que se producen por los procesos de usuario se realizan en la base de datos en forma inmediata. Asimismo, si existiese un cache de disco en este host, sería consistente con dicha actualización, es decir, que un usuario podría levantar del cache información que acaba de ser grabada por otro usuario, claro que este proceso es totalmente transparente, por ser este un manejo exclusivo del sistema operativo.

En una administración distribuida, los objetos residen en más de un host, aunque desde el punto de vista de los usuarios parezca que "su" host alberga la base de datos a la que le es posible acceder. En este tipo de administración es de particular interés el tema de actualizaciones de datos en lo que respecta a consistencia de los caches entre los distintos hosts; esta consistencia no es transparente como en la administración centralizada. Otro aspecto importante a tener en cuenta es la autenticación y control de privilegios de usuarios para acceder a los objetos, para lo que existen distintos tipos de administración de usuarios: centralizadas, distribuidas, ACLs, etc.

En el texto que sigue, nos dedicaremos al estudio y especificación de conceptos y problemáticas relacionadas con sistemas distribuidos.

4.2 Servicio de Acceso a Archivos

La mayoría de las aplicaciones en las computadoras utilizan archivos para almacenar en forma permanente la información, o como un medio de compartir información entre diferentes usuarios y programas [2]. Un archivo es una abstracción de un almacenamiento permanente.

Desde la introducción del almacenamiento en disco en los años 60, los sistemas operativos han incluido un componente llamado *sistema de archivos* (file system) el cual es responsable por la organización, almacenamiento, recuperación, nombramiento, compartición y protección de los archivos. Los file systems proveen un conjunto de operaciones de programación que caracterizan la abstracción "archivo" liberando a los programadores de cuestiones concernientes a los detalles de ubicación y estructura de los datos dentro del disco.

El servicio de acceso a archivos es un componente esencial en un sistema distribuido. Cumple idéntica función que un sistema de archivos en un sistema operativo convencional. Brinda el soporte para compartir archivos e información. Habilita a los programas de usuario a acceder a archivos remotos sin la necesidad de copiarlos al disco local y antaño proveía el acceso a archivos desde estaciones de trabajo sin disco. En un sistema distribuido de propósito general, el servicio de archivos es el más usado, por lo tanto son críticos su funcionamiento y performance [2].

4.3 File Systems Distribuidos vs. Bases de Datos Distribuidas

En un sistema distribuido, basándonos en el tamaño y característica de actualización de los objetos a compartir, podemos citar dos tipos de organización. En caso de que los objetos que se comparten sean pequeños, típicamente menores que 1 Mb, en un ambiente donde las escrituras simultáneas son poco frecuentes, podremos hablar de un *sistema de archivos distribuido* (Distributed File System, DFS). Si, en cambio, lo que estuviéramos por compartir fuese una gran base de datos, con ítems totalmente relacionados entre sí, donde las escrituras concurrentes fuese lo normal, entonces estaríamos frente a una *base de datos distribuida* (Distributed Data Base, DDB). De acuerdo a esta clasificación se puede notar que es más fácil distribuir un file system que una base de datos. [15]

4.4 Arquitectura de los DFSs

La arquitectura de los DFSs es siempre cliente-servidor. Un *Sistema Cliente-Servidor* (SCS) divide una aplicación en varios procesos que corren en diferentes computadoras conectados a través de una red, formando un sistema débilmente acoplado [30].

Los servers son físicamente seguros, ejecutan software confiable y son monitoreados por personal de operaciones especializado. Los clientes pueden ser modificados arbitrariamente por los usuarios, están físicamente dispersos y pueden quedar desatendidos o apagados por largos períodos.

Este modelo ha probado ser especialmente conveniente en lo que respecta a escalabilidad, donde existen unos pocos servers y una gran cantidad de clientes. El modelo cliente-servidor descompone un gran sistema distribuido en un pequeño núcleo que cambia relativamente poco y un gran número de clientes periféricos menos estáticos. Desde el punto de vista de seguridad y administración, la escala del sistema parece ser la del núcleo, pero desde el punto de vista de la performance y disponibilidad, un usuario en la periferia recibe servicios casi standalone [17], sin tener en cuenta la carga que soporta el núcleo.

Los siguientes son algunos aspectos clave en el diseño de clientes y servidores [30]:

- El sistema operativo de la estación de trabajo (Windows, OS/2, UNIX) se elige a menudo de acuerdo con el SCS; pero el SCS debería acomodar Clientes basados en sistemas operativos heterogéneos.
- El Servidor debe poder crecer: atender más Clientes heterogéneos o evolucionar a un sistema de múltiples Servidores (heterogéneos) que ofrezca transparencia de ubicación, fragmentación y replicación.
- Los Clientes no deben necesitar conocer las características del Servidor, excepto la interfase estándar (SQL, X-Windows), facilitando la actualización del SCS.
- Los Servidores deben permitir la administración completa de LANS y Servidores desde una misma estación de trabajo.

Cabe destacar que existen varios puntos de vista para poder definir si un sistema de archivos es distribuido. Una opción puede ser la partición de la información a través de la red, es decir que la totalidad de los datos se recupera como resultado de la unión de todas las partes. Esto no significa que una o varias partes no puedan estar replicadas en más de un sitio. Sin embargo desde otro punto de vista podemos pensar en un sistema distribuido como la sola replicación de una base de datos. También la distribución de los datos puede ser realizada con fines de distribución de carga, tolerancia a fallas, disponibilidad, velocidad de acceso o seguridad.

Las implementaciones de los DFSs atacan los siguientes problemas: transparencia de ubicación, preservación de semántica local de actualización (consistencia), escalabilidad, disponibilidad, performance y seguridad. Sin embargo, no es posible optimizar todos los tópicos independiente y simultáneamente. Por ejemplo, no se debe permitir actualizaciones en una red particionada (optimizar disponibilidad), si no se puede tolerar inconsistencias (semántica de actualización) [15].

4.5 Control de concurrencia y consistencia

Los cambios a un archivo realizados por un cliente, no deben interferir con las operaciones de otros clientes que acceden o modifican el mismo archivo en forma simultánea. Esta situación se conoce como *control de concurrencia*. La mayor parte de los servicios de archivos actuales siguen los estándares del UNIX, proveyendo esquemas de lockeo de tipo *advisory* y *mandatory*.

El comportamiento en presencia de requerimientos concurrentes, algunos de ellos conflictivos, agravado por la eventual ocurrencia de fallas, se conoce como semánticas de actualización.

Para mejorar el rendimiento en sistemas distribuidos, se utiliza una área de almacenamiento temporario local en el cliente que contiene datos que se accedieron recientemente, denominada cache. Se necesita proveer de algún método de consistencia para asegurar la integridad de la información en cada componente del DFS. Estudiaremos métodos de consistencia fuerte, en donde se respeta estrictamente la actualización de los datos, y

métodos de consistencia débil, que permite cierta laxitud en las actualizaciones, suponiendo a priori que son esporádicas.

4.5.1 Control de concurrencia – Políticas de lockeo

Junto al control de concurrencia surge una problemática asociada, referida al acceso en forma exclusiva a los archivos, dando lugar a diferentes políticas de lockeo.

Normalmente, se utilizan tres alternativas para el manejo del control de concurrencia, estas son: lockeos, control de concurrencia optimista y ordenamiento por timestamps. Sin embargo, la mayoría de los sistemas usan lockeos. Cuando se usa lockeo, el server establece un lock en cada item etiquetado con el identificador de la transacción antes de ser accedido y remueve este lock cuando la transacción ha sido completada. Mientras un item se encuentra lockeado, sólo la transacción que tiene el lock puede acceder al item. Todas las demás transacciones deben esperar a que el item sea liberado. En algunos casos se puede compartir el lock. El uso de locks puede derivar en deadlocks entre transacciones que esperan mutuamente la liberación de un lock que posee la otra.

En un esquema de control de concurrencia optimista, las transacciones continúan hasta el momento del commit. En ese momento, el server realiza las verificaciones para descubrir si las operaciones se realizaron sobre datos que conflictúan con operaciones de alguna otra transacción concurrente, en cuyo caso el server aborta la transacción y el cliente la puede reiniciar.

En una política de ordenamiento por timestamps, el servidor guarda la hora de última lectura y grabación de cada item. Por cada operación, el timestamp de la transacción se compara con el del item para determinar si se puede realizar en forma inmediata, demorada o rechazarla. Cuando una operación es demorada, la transacción espera pero cuando es rechazada, la transacción aborta.

En un sistema distribuido el otorgamiento de locks se implementa con un módulo separado del servicio de archivos, comúnmente llamado *lock manager*.

4.5.2 Semánticas de actualización

Ante la presencia de actualizaciones concurrentes, es necesario adoptar alguna metodología para resolver en forma uniforme los conflictos que surjan. El objetivo es lograr una *semántica de consistencia fuerte* donde las actualizaciones se reflejan como si se estuviese trabajando en un sistema centralizado. Nombramos las cuatro semánticas de actualización más usadas: *semántica UNIX*, *semántica de sesión*, *archivos inmutables* y *semántica transaccional* [29].

- **Semántica UNIX**: es el resultado de aplicar operaciones básicas sobre un archivo restringidas por una consistencia en tiempo real. Bajo esta semántica, las operaciones de actualización son visibles inmediatamente por todas las operaciones de lectura que siguen, y son siempre aplicadas a la copia del archivo que refleja todas las actualizaciones previas. La rigidez de esta semántica, puede obstaculizar la eficiencia de un file system distribuido

debido a la sobrecarga en garantizar que todas las actualizaciones a un archivo se reflejen en todas las copias del archivo (i.e., copias cacheadas y réplicas en otros servers), y también garantizar que el orden en que se aplican las actualizaciones sea el mismo en todos los servers [29].

- **Semántica de sesión**: cada sesión de archivo obtiene una copia lógica de la versión actual del archivo. Esta copia refleja las actualizaciones de las sesiones de archivos ya cerradas, y ninguna de las actualizaciones de las sesiones que se están llevando a cabo. Todos los accesos de lectura y escritura para la sesión de archivo son realizadas sobre esta copia. Cuando la sesión concluye, la copia lógica pasa a ser la versión actual del archivo (siempre y cuando se haya actualizado el archivo en esta sesión). Una sesión de archivo que actualiza, S1, verá sus actualizaciones completamente sobrescritas cuando otra sesión que actualiza, S2 termine, si S1 comienza después que S2 y termina antes que lo haga S2, incluso aunque las dos sesiones actualicen distintas secciones del archivo. No siempre se define esta semántica de actualización de forma tan estricta, derivando en cuestiones referentes a compartir la copia de un archivo [29].

- **Archivos inmutables**: al igual que la semántica de sesión, a cada sesión se le entrega una copia lógica del archivo accedido, y las actualizaciones serán vistas por otras sesiones sólo después del cierre. La diferencia aquí, es que una sesión que actualiza, crea una nueva versión del archivo tal que tanto la antigua como la nueva versión estarán presentes y serán accesibles. Un *open()* puede requerir una versión antigua. Este método soluciona parcialmente el problema que tiene la semántica de sesión, donde el efecto de una sesión puede ser completamente arrasada por otra. Esta semántica requiere espacio adicional de disco para almacenar las múltiples versiones de cada archivo [29].

- **Semántica Transaccional**: este tipo de semántica requiere que un conjunto de sesiones de archivos sea *serializable* y que la sesión de archivo parezca como una *acción atómica*. Un conjunto de sesiones ejecutadas concurrentemente es serializable, si el resultado de la ejecución es equivalente a alguna ejecución en forma serial del mismo conjunto de sesiones. La atomicidad de una sesión de archivo garantiza que se realicen **todas** o **ninguna** de las acciones de la sesión, y que las acciones aparezcan en forma instantánea, sin estados transitorios detectables. Cada sesión de archivo puede ser equiparable a una transacción en un ambiente de base de datos [29].

4.6 Replicación y Sincronización entre réplicas

Para mejorar la disponibilidad y la *transparencia ante fallas* en un file system distribuido, se utiliza la metodología de replicación. Ésta consiste en mantener copias de los objetos en más de un server, sin que los usuarios o los programas de aplicación conozcan la existencia de las réplicas. Este concepto es conocido como *transparencia de replicación*. Las copias pueden ser de lectura o de lectura/escritura.

Cuando un cliente realiza una actualización a un archivo, esta actualización deberá estar reflejada en las copias de los distintos servers. Cuando se cachea un archivo, las actualizaciones pueden ser hechas en el cache o pueden ser escritas directamente en el server. Si se escriben directamente en el server se pierde el beneficio que tiene el cliente de escribir sobre el cache durante la sesión, pero provee mayor garantía de permanencia. Si se usa *escritura demorada* (write-delayed), las actualizaciones en el cache son retenidas por un lapso de tiempo específico, hasta que se cierre la sesión de actualización o hasta que se vuelque el contenido del cache para hacer lugar. Existen varios métodos para mantener sincronizados los caches de las distintas máquinas, conociéndose esta problemática como políticas de *coherencia de cache*.

Una vez que las actualizaciones fueron enviadas a un server, pueden ser propagadas a todos los servers al mismo tiempo, asegurando la replicación de la versión actual del archivo; pueden ser propagadas a una cierta cantidad de servers; o a ninguno de ellos. La *actualización perezosa* (lazy update) se refiere a propagar las actualizaciones en un momento posterior, ya sea cuando son requeridas, cuando el server no está ocupado, o después de un período de timeout. Este tipo de actualización origina copias con distinto grado de actualización, derivando en un sistema con consistencia débil, en donde se tolera el acceso a objetos que no están totalmente actualizados.

En cambio, actualizando todos los servers en forma inmediata, se mejora la disponibilidad de la versión actual. Cuanto más servers tengan la copia al día del archivo más fallas de servers podrán tolerarse.

Otro método de reducir la carga de la CPU de un server, es hacer que el cliente realice la distribución de las actualizaciones. En lugar de enviar las actualizaciones a un server y que éste a su vez las distribuya a los demás servers, el cliente envía las actualizaciones a todos los servers en una única operación multicast [29].

4.7 Espacio de nombres

Para el cliente de un DFS existe un único *espacio de nombres* en todo el file system. Esto es lo que da la sensación al usuario de disponer de toda la base de datos en "su" host. Los mecanismos básicos que se utilizan para lograr esta apariencia son: *transparencia de localización* y *transparencia de acceso*. La transparencia de localización se refiere a la posibilidad de acceder a un archivo sin conocer el nombre del server que lo alberga. La transparencia de acceso se refiere a que las aplicaciones que accedían a archivos locales no necesiten ser modificadas para acceder a archivos en localizaciones remotas cuando se usa un DFS.

4.8 Seguridad

Los sistemas centralizados han logrado un grado de seguridad razonable sobre los datos almacenados; los sistemas distribuidos traen aparejado un nuevo conjunto de problemas de seguridad que los convierte en mucho menos seguros, a menos que se tomen precauciones especiales. [2]

El tratamiento de la seguridad enfrenta dos peligros inherentes a los sistemas distribuidos. El primero de ellos es la falta de privacidad e integridad de los datos del usuario mientras viajan por la red. Es muy difícil –en general imposible en la práctica- asegurar una red contra pinchaduras (de cable o lógicas) orientadas a copiar o interferir con los datos que pasan por ella. El segundo, es la posibilidad de ejecutar software que interfiera el sistema en cualquier máquina de la red; no es posible mantener todas las máquinas de la red físicamente protegidas y dicho software podrá ser ejecutado en máquinas vulnerables. [2]

Cada DFS abordará este tema ofreciendo soluciones específicas a cada necesidad. No ahondaremos en esta materia más que lo suficiente para autenticar usuarios y determinar permisos para el uso de archivos y directorios.

4.9 Escalabilidad

La demanda de crecimiento en sistemas distribuidos ha derivado en una filosofía de diseño en donde ningún recurso –de hardware o de software– se asume como de suministro restringido. Mas bien, cuando la demanda por algún crece, debe ser posible extender el sistema para satisfacerla. Por ejemplo, en un sistema distribuido, la frecuencia a la cual se accede a un archivo, tiende a crecer proporcionalmente con el número de usuarios y estaciones de trabajo. Debe ser posible agregar servers para evitar cuello de botella en la performance que surgirían si un único server debiera manejar todos los requerimientos de acceso a dicho archivo. En este caso este archivo debe ser replicado en varios servers y el sistema debe estar diseñado tal que cuando se actualiza el archivo replicado la misma actualización se aplica a todas las copias [2].

Podemos definir entonces *transparencia a la escalabilidad* como la medida en la que un sistema puede crecer sin afectar la disponibilidad de los recursos ni afectar la estructura del sistema o los algoritmos de las aplicaciones.

Dos componentes fundamentales para lograr la escalabilidad de un sistema son:

a) Transparencia a la replicación

   Permite usar múltiples instancias de los objetos de información, con el objeto de incrementar la confiabilidad y la performance sin que los usuarios o los programas de aplicación conozcan la existencia de las réplicas.

b) Transparencia a la migración

   Permite el movimiento de objetos de información dentro del sistema sin afectar la operación de los programas de aplicación o de los usuarios. Es posible seguir usando los archivos mientras los están moviendo.

4.10 Confiabilidad

Uno de los objetivos originales de construir sistemas distribuidos fue hacer que sean más confiables que los centralizados. La idea es que si alguna máquina falla, otra tome su lugar en las tareas a realizar. Por un simple cálculo de probabilidades se prueba que la confiabilidad mejora con la redundancia de recursos.

4.10.1 Disponibilidad

Es importante distinguir varios aspectos de la confiabilidad. La *disponibilidad* se refiere a la cantidad de tiempo en que un sistema es usable. La disponibilidad puede ser mejorada por un diseño que no requiera que un número sustancial de componentes críticos funcionen simultáneamente. Otra herramienta para mejorar la disponibilidad es la redundancia: piezas claves de hardware y software deben ser replicadas tal que, si alguna de ellas falla, las otras podrán tomar su lugar.

4.10.2 Tolerancia a fallas

Otro aspecto relacionado con la disponibilidad es la tolerancia a fallas. Supongamos que un server se cae y rebootea rápidamente. ¿Qué sucederá? ¿El server caerá con todos los usuarios que estén trabajando? Si el server tiene tablas conteniendo información importante de las actividades que se llevan a cabo, la recuperación será muy dificultosa. Veremos que algunos DFS evitan este problema de diferentes maneras.

En general, los DFSs pueden ser diseñados para enmascarar las fallas, esto es para ocultar las fallas a los usuarios y lograr una *transparencia ante fallas*. Si un DFS u otro servicio se construye a partir de un grupo servers cooperativos, entonces será posible construirlo de tal manera que los usuarios no se den cuenta del la perdida de uno o dos servers, a parte de alguna degradación de performance. Por supuesto, que el truco reside en no agregar overhead sustancial cuando el sistema funciona normalmente [27].

4.11 Plataformas soportadas

El concepto de *plataforma soportada* se refiere a la posibilidad de ejecutar cierto software en un determinado sistema operativo. Si el software requiriese vincularse con el kernel, se dice que este sistema operativo tiene (o no tiene), soporte de kernel para dicho software.

Para *portar* un software a un sistema operativo que requiera soporte de kernel, es necesario agregar rutinas de manejo de dispositivos al kernel del sistema operativo que se desea utilizar. La escritura de código kernel es una tarea muy compleja y dependiente de cada sistema operativo específico.

En el caso de los file systems, existe una capa superior que enmascara los pedidos al kernel, es el llamado Virtual File System (VFS). El VFS es el encargado de reconocer el módulo que atenderá el requerimieto de archivo solicitado y direccionar el pedido a través de ese módulo. El VFS y los módulos determinan, dentro del kernel, los sistemas de archivos soportados por dicho kernel.

4.12 Resumen

En este capítulo se definieron conceptos de file systems distribuidos donde los objetos a compartir son pequeños y las escrituras simultáneas poco frecuentes. Para regular el acceso a los datos se utiliza control de concurrencia de acuerdo a una semántica elegida, que podrá derivar en una política de lockeo estricta o en un control de concurrencia optimista, haciendo transparente al usuario el problema de la compartición de los datos.

Se vieron también otras formas de transparencia que ayudan a mejorar la disponibilidad y performance en un sistema distribuido, éstas son transparencia ante fallas y transparencia de replicación. Un único espacio de nombres se logra a través de la transparencia de localización y de acceso.

Para lograr que un sistema crezca sin afectar la disponibilidad de los recursos, se requiere transparencia a la escalabilidad. Será útil algún sistema de redundancia para tolerar fallas de servers y cortes de comunicación.

CAPITULO 5

Estado del arte

El estado del arte en el tema de file systems distribuidos es muy vasto. Por este motivo, hemos decidido estudiar sólo algunos de los file systems distribuidos que se pueden encontrar en el mercado comercial y en el ámbito académico. Cada uno de ellos nos parece representativo de un ámbito específico. NFS es básico como material de estudio cuando se trata el tema de DFSs; AFS es un DFS utilizado en la actualidad a gran escala en Internet y antecesor directo de CODA el cual aporta nuevos conceptos referidos a la computación móvil; CIFS representa la proyección a futuro de los SMB servers presentes en ambientes Windows y UNIX. Nombraremos las características principales de cada uno de ellos y describiremos su funcionamiento.

En todos los casos se trata de simular la semántica UNIX de actualización, donde existe una única copia de cada archivo. Es aquí donde entran en juego las transparencias citadas anteriormente, que permiten que los programas de aplicación no necesiten ser modificados para poder acceder a archivos en localizaciones remotas y distribuidas. Sin embargo, no es posible cumplir esta semántica en forma estricta, por ejemplo la actualización desde el cache local de un cliente que genera modificaciones hacia el server que alberga el archivo sufre demoras que pueden afectar la consistencia de los datos. Cada uno de los DFSs estudiados tiene en cuenta esta situación y plantea su comportamiento frente a la ocurrencia de los casos conflictivos.

5.1 AFS

AFS es un sistema de archivos distribuido basado en tecnología cliente/servidor, que permite a un gran número de computadoras (hosts y workstations) compartir el acceso a un conjunto de archivos de una manera uniforme [28]. Originalmente fue desarrollado en 1983 como proyecto de investigación en el Information Technology Center, en Carnegie Mellon University, bajo el nombre de "Andrew". Actualmente se encuentra disponible en diversos sistemas operativos como producto comercial, habiendo dejado la etapa de investigación. AFS está pensado para funcionar tanto en una red de área local como global [1]. AFS-2 trabaja sobre un UNIX BSD 4.3 con TCP/IP como protocolo de comunicación. Cada workstation en el sistema requiere un disco y trabaja como una workstation UNIX común. AFS provee acceso transparente a archivos remotos para programas UNIX pues el acceso se realiza utilizando las primitivas de archivo estándar de UNIX [22].

El objetivo principal en el diseño de AFS ha sido la escalabilidad. La estrategia clave para lograr la escalabilidad es el cacheado de archivos completos en los nodos de los clientes. AFS tiene dos características de diseño especiales:

Servicio de archivos completos: el servidor AFS transmite la totalidad del archivo o a las máquinas cliente.

Cacheo de archivos completos: una vez que la copia del archivo se ha transferido a la máquina cliente, ésta se almacena en el disco local. El cache es permanente y

sobrevive a rebooteos de la máquina cliente. El cache contiene varios cientos de archivos que han sido utilizados recientemente en esa máquina. Los *open()* en el cliente son satisfechos utilizando la copia local del archivo cuando es posible, en lugar de la copia remota.

Los componentes de software fundamentales de AFS son VICE y VENUS.

VICE es un servidor de procesos multi-thread, con cada thread atendiendo el requerimiento de un cliente. Reside sobre el kernel UNIX en cada server AFS. Provee servicios de compartición de archivos a los clientes.

VENUS es el proceso del cliente que corre en cada workstation AFS y constituye la interfase con VICE. Analiza todas las llamadas y funciones de archivos como un manejador de cache.

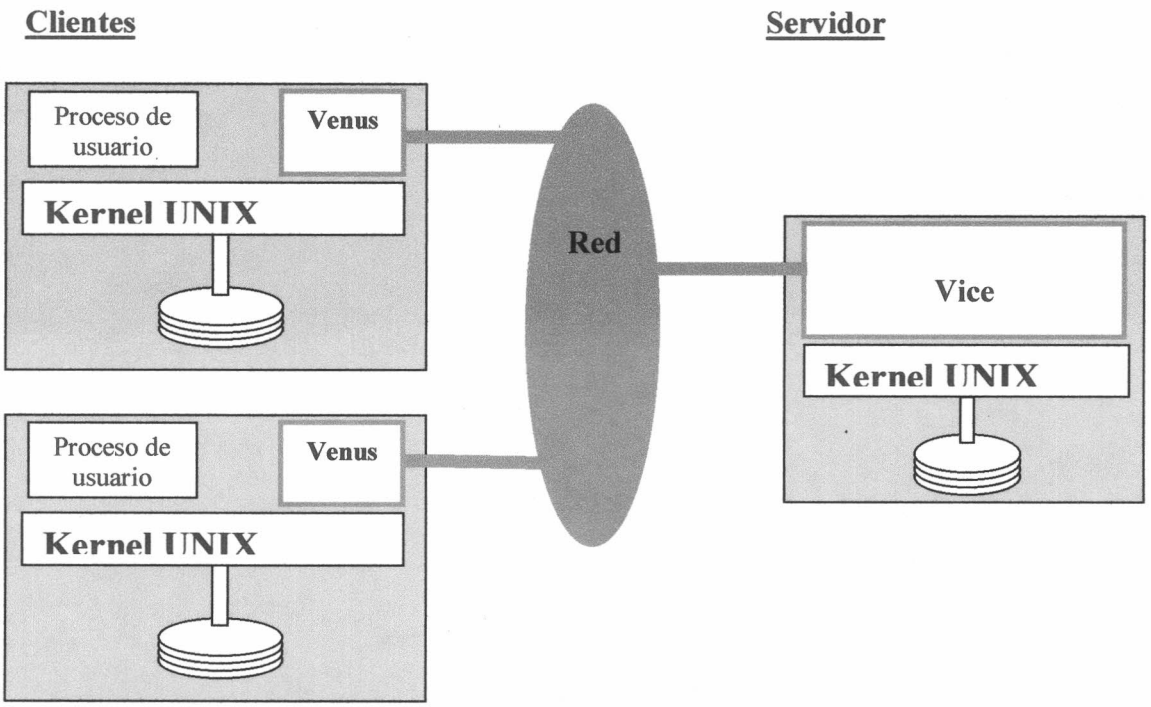

Fig. 7 – Estructura de Cliente y Server AFS

5.1.1 Características de diseño

- Para archivos compartidos que no son frecuentemente actualizados (como ser comandos UNIX o librerías) y para archivos que normalmente son accedidos por un único usuario (como ser el home directory y su subárbol), las copias localmente cacheadas permanecerán válidas por largos períodos. En el primer caso por ser poco actualizados y el

segundo porque la actualización se lleva a cabo en la propia estación de trabajo. Esta clase de archivos representa la gran mayoría de los accesos.

- El cache local debe tener tamaño suficiente para almacenar un conjunto de archivos de trabajo que asegure que los archivos de uso regular en una estación de trabajo se encuentran disponibles en forma local.
- La estrategia de diseño está basada en algunas suposiciones acerca del tamaño promedio y máximo de archivos y el tipo de acceso, basadas en observaciones en ámbitos académicos.
  - Los archivos son pequeños, usualmente menores que 10K.
  - Las operaciones de lectura son alrededor de 6 veces más frecuentes que las de escritura.
  - Son comunes los accesos secuenciales y raros los accesos random.
  - La mayoría de los archivos son escritos por un único usuario. Cuando los archivos se comparten, usualmente un solo usuario es el que lo modifica.
  - Los archivos son referenciados en ráfagas. Si un archivo ha sido referenciado recientemente, hay entonces gran probabilidad de que sea referenciado nuevamente en un futuro cercano.

Estas observaciones se utilizaron para guiar el diseño y la optimización de AFS, sin restringir la funcionalidad vista por los usuarios [29].

Hay un tipo importante de archivos que no está contemplado en la descripción anterior; típicamente las bases de datos son compartidas por muchos usuarios y actualizadas muy frecuentemente. Los diseñadores de AFS han excluido explícitamente de sus objetivos la provisión de facilidades para almacenar bases de datos, argumentando que con las restricciones impuestas por distintas estructuras de nombramiento (accesos en base al contenido) y la necesidad de granularidad fina en el acceso a los datos, control de concurrencia y atomicidad de las actualizaciones, resulta difícil diseñar una base de datos distribuida que sea también un sistema de archivos distribuido. Estas facilidades para bases de datos distribuidas deben abordarse en forma separada.

5.1.2 Espacio de nombres

El espacio de nombres de AFS es muy similar al árbol de directorios de UNIX. Las únicas diferencias aparentes son el agregado de los directorios /afs y /cache.

El file system de AFS tiene dos partes:

- el file system local en el cual residen los archivos de la estación de trabajo local y el área dedicada a cache de archivos remotos.
- el file system compartido soportado por VICE.

El file system local puede verse distinto de workstation a workstation, pero el compartido se ve igual desde todas las workstations.

El file system compartido se monta sobre el directorio /afs en cada estación, y los directorios y archivos que contiene pueden estar situados en distintos fileservers [22].

Es así como los clientes ven al file system como si fuese un enorme disco virtual, compuesto por subdirectorios provenientes de todos los hosts que componen la red.

5.1.3 Celdas

Los subdirectorios que cuelgan del /afs representan *celdas*. Se llama celda a un conjunto de hosts (clientes y servidores) agrupados para su administración. Usualmente pertenecen al mismo dominio en Internet, y el nombre de la celda se deriva del mismo [1].

Un usuario sentado en una workstation, se loguea en una celda, y recibe información que su workstation requiere a los servers de la celda, en su nombre [1].

5.1.4 Estructura de directorios

Los archivos de AFS son almacenados en estructuras llamadas *volúmenes*. Estos volúmenes se encuentran en los servidores de archivos en la celda. Un volumen contiene un subárbol de directorios y archivos con alguna relación entre sí. Normalmente un volumen es considerablemente más pequeño que un file system tradicional. Por ejemplo, cada home directory de un usuario podría almacenarse en un volumen distinto. El acceso a los volúmenes se realiza a través de *mount points*, que aparecen en el file system como directorios descendientes de /afs. Es posible saber si un directorio es o no un mount point, utilizando comandos de AFS [28].

5.1.5 Localización de Archivos

Con sólo saber el nombre completo del archivo, es posible acceder al mismo, sin necesitar conocer el server que lo alberga. Esta propiedad se conoce como *transparencia de localización*. Cada server contiene una copia de una base de datos de localizaciones que mapea los nombres de volúmenes a los servers que los contienen. De todos modos es posible saber qué host contiene a un volumen dado, nuevamente con comandos de AFS [1].

A partir de la trasparencia de localización, se puede cambiar la ubicación de un volumen sin que los clientes se enteren de dicho cambio; esta es otra propiedad conocida como *independencia de localización o transparencia a la migración.*

5.1.6 Manejo del Cache

Los volúmenes pueden ser replicados en varios servers con fines de sólo lectura. Con lo cual sólo existe una única copia accesible para escritura. Este volumen es nombrado, en general, con un punto (".") delante del nombre de su mount point.

Las réplicas son utilizadas por el *manejador de cache* del cliente, Venus, para balancear la carga de la red. Si una copia en un fileserver se torna inaccesible por algún problema de la red, el manejador de cache automáticamente comenzará a acceder los mismos datos desde la copia de otro fileserver [1].

El manejador de cache mantiene en forma local, una copia de los archivos recientemente accedidos, en trozos de un máximo de 64kb, por lo tanto, si un archivo ocupa más que esto habrá varios trozos de 64kb en el cache local que pertenezcan a este archivo. Si en algún momento se interrumpe el acceso a la copia de escritura del fileserver, el manejador de cache podrá acceder a su copia local para lectura pero los intentos de escritura fallarán [1].

Cuando se accede a un nuevo archivo que no se encuentra en el cache, junto al archivo se obtiene del fileserver una promesa de que la versión del archivo/directorio que estará en el cache será la última y estará al día. Esta promesa se llama *callback*. Cuando el archivo/directorio es modificado por otro cliente, el fileserver rompe el callback, entonces, la próxima vez que el usuario acceda al archivo, el manejador de cache buscará una nueva copia.

Cuando un proceso en el cliente realiza un *close()*, si la copia local ha sido modificada, entonces el contenido es enviado de vuelta al server. El server actualiza su contenido y el timestamp del archivo. La copia local en el cliente, se retiene en caso de ser necesitada nuevamente por algún proceso de usuario en la misma workstation [29].

5.1.7 Controles de Acceso

Como en cualquier sistema distribuido, el tema seguridad es de primordial importancia. En AFS todo el tráfico entre workstations y servers está encriptado. El acceso a directorios y archivos es controlado a través de Listas de Control de Acceso (Access Control Lists, ACLs). En AFS solamente los directorios tienen ACLs, por lo tanto el acceso a los archivos depende de la ACL de su directorio. Los nuevos subdirectorios que se crean, heredan una copia de la ACL del padre, entonces si en el futuro se modificasen atributos de la ACL del padre, estos cambios no se reflejarán en el subdirectorio.

Las ACLs reemplazan casi por completo las máscaras de bits de protección en los archivos de UNIX, con excepción de los primeros tres (los del owner) que proveen restricciones adicionales sobre un archivo al cual la ACL permite el acceso.

Por ejemplo, si la ACL permite escribir sobre los archivos pero en particular un archivo tiene los permisos en `'-r-??????'`, entonces no será posible escribir en dicho archivo. Sin embargo, como veremos en los siguientes párrafos, el permiso de escritura en la ACL permite modificar los bits de permisos en UNIX, utilizando el comando `'chmod'`, con lo cual podemos decir que estas restricciones adicionales no son de seguridad sino sólo indicativos del uso del archivo.

Las ACLs tienen siete categorías de permisos que pueden ser controladas individualmente. Estos permisos están divididos en dos grupos: acceso a directorios y acceso a archivos.

Permisos de acceso a directorios:

- `'l'` permiso para examinar la ACL del directorio, ver los archivos que contiene y atravesar subdirectorios supeditado a tener acceso 'l' en las ACLs de los mismos.
- `'i'` permiso de creación (insert), habilita al poseedor a agregar nuevos archivos y subdirectorios al directorio.
- `'d'` permiso de borrado, habilita al poseedor a eliminar archivos y subdirectorios del directorio.
- `'a'` permiso para administrar, habilita al poseedor a cambiar la ACL para ese directorio.

Permisos de acceso a archivos:

- `'r'` permiso de lectura, habilita al poseedor a ver el contenido del directorio (es decir, `'ls -l'`). Para los archivos en el directorio, el poseedor está habilitado a ver los datos almacenados en los archivos.
- `'w'` permiso de escritura, habilita al poseedor a modificar el contenido de los archivos en el directorio, y a cambiar los bits de protección de UNIX con el comando `'chmod'`.
- `'k'` permiso de lockeo, habilita al poseedor a ejecutar programas que necesiten realizar operaciones de *flock()* en archivos del directorio.

En las ACLs se especifican usuarios con sus permisos, y también se puede especificar *grupos protegidos* que actúan en forma similar a los grupos de usuarios en UNIX. Un grupo protegido identifica una lista de usuarios que tendrán determinados permisos de acceso. El administrador de un grupo protegido es el dueño del mismo y es el único que puede modificarlo. El formato del nombre de grupo será usualmente dueño:nombre_grupo.

AFS provee grupos de protección predefinidos que pueden ser usados en cualquier ACL:

`system:anyuser`

Es similar al permiso 'world' de UNIX. Cualquier usuario AFS (en cualquier parte del mundo) puede acceder a archivos y directorios controlados por esta ACL.

`system:authuser`

Esta es una versión más restringida del grupo anterior, que especifica que sólo usuarios autentificados en la celda local podrán acceder a los archivos y directorios controlados por esta ACL.

`system:administrator`

Usuarios que tienen privilegios para ejecutar algunos, pero no todos los comandos de administración del sistema.

Ejemplo de ACL:

```
% fs la /afs/nih/@sys/usr/local/jperez
Access list for /afs/nih/@sys/usr/local/jperez is
Normal rights:
  jperez rlidwk
  cgomez a
  jprerez:contab rlidwk
  system:administrators rlidwka
  system:anyuser rl
```

[8,9,10]

5.1.8 Tokens

Un usuario se autentifica en una celda utilizando el sistema de autenticación Kerberos. Una vez que el usuario ingresa en la celda de AFS, se le asigna automáticamente un *token* que le va a permitir acceder según su identificador de usuario (user ID) a los ACLs de los directorios por un tiempo máximo de 25 hs. Pasado este lapso de tiempo, el token caduca [8,11].

5.2 CODA

CODA es un file system experimental, cuyo objetivo principal es ofrecer a clientes móviles acceso ininterrumpido a los datos, independizándose de las caídas de servers o de red [20]. Hereda muchos de los aspectos de uso y diseño de su antecesor, Andrew File System (AFS). Provee gran disponibilidad a través de dos mecanismos distintos pero complementarios: *operación desconectada* y *replicación* [20]. La finalidad al construir CODA fue la de ofrecer la máxima disponibilidad con la mejor performance y el máximo grado de consistencia que sea posible con estas restricciones [16].

El diseño del mecanismo de consistencia en CODA se basa en que debe usarse siempre la copia más actualizada de un archivo, que sea accesible. A pesar de que esto genere inconsistencias, deben ser escasas y siempre detectables por el sistema [21]. Al igual que AFS, no está recomendado para aplicaciones tales como transacciones en línea o sistemas con gran cantidad de actualizaciones concurrentes [20].

La arquitectura de CODA reconoce tres tipos de máquinas: clientes, servers y SCM (system control machine). Las máquinas cliente son típicamente workstations monousuarias

usadas para acceder a la información compartida. Estas máquinas son de confianza para el server sólo por el lapso de duración de una sesión de login efectuada por un usuario autenticado. Las máquinas server son seguras y su propósito es dar servicio de archivos compartidos a los requerimientos de los clientes. El tercer tipo de máquina es el SCM, cuya función es proveer un único punto de control para facilitar la administración. Lógicamente el SCM es distinto que los servers, pero, físicamente, el SCM puede también actuar como un server [19].

5.2.1 Adaptabilidad

La escasez de recursos locales y de seguridad física en los clientes móviles, justifica apoyarse en los recursos que brindan los servers. Sin embargo, la falta de comunicaciones baratas confiables y los costos crecientes de acceso a los servicios, argumenta a favor de auto-basarse en los recursos locales de los clientes móviles. El desafío para el acceso a datos desde un cliente móvil es conseguir el balance apropiado entre estas dos problemáticas.

Este balance no es estático. A medida que las circunstancias de un cliente móvil cambian, se debe reaccionar y repartir las obligaciones entre los clientes y los servers. En otras palabras se debe *adaptar* [17]. Esta adaptación se realiza en un rango de posibilidades caracterizadas por dos extremos: *adaptación de las aplicaciones* y *transparencia a las aplicaciones*. En el primero, la aplicación debe elegir la mejor manera de adaptarse a los cambios de disponibilidad de los recursos; en el otro extremo, es el sistema operativo el que debe resolver la adaptación, ocultando cualquier evento a las aplicaciones [4,5,6].

5.2.2 Transparencia a las Aplicaciones

CODA provee a los clientes, particularmente a los móviles, una gran disponibilidad de acceso a archivos. Tal como lo hace AFS, CODA presenta a los clientes un único espacio de nombres organizado en volúmenes. Las aplicaciones que ejecutan los clientes CODA, utilizan la interfase standard del file system UNIX. Las aplicaciones en clientes móviles pueden continuar ejecutándose sin modificaciones.

El manejador de cache, *Venus*, es el único responsable de sobrellevar las consecuencias de la movilidad [12].

5.2.3 Operación bajo condiciones de conectividad fuerte

Este es el mejor caso, el de redes LAN de alta calidad. En esta situación, se dice que Venus está *fuertemente conectado*. En estas condiciones, un cliente CODA se comporta como un cliente AFS. Cuando una aplicación abre un archivo en CODA, Venus chequea si el archivo se encuentra en el cache, si no está, trae el archivo completo del server al cache local. Luego realiza las lecturas y escrituras pedidas por las aplicaciones sobre el cache.

Al igual que en AFS, cuando un cliente cachea un archivo, obtiene un callback por parte del server. Cuando se cierra un archivo que fue modificado, una copia de este nuevo

archivo se envía a los servers y éstos notifican a todos los clientes que tenían callbacks para este archivo [12].

5.2.4 Operación desconectada

En el peor caso, Venus no puede contactar ningún server; se dice que el cliente está *desconectado*. Venus continuará dando servicios de lectura sobre los archivos que se encuentran en el cache y, a diferencia de AFS, también permitirá escrituras. Los archivos que no estén en el cache no podrán ser accedidos y Venus retornará un error. Para reducir la posibilidad de faltas en el cache (cache misses) se utiliza un mecanismo llamado *hoarding*, que mejora la eficiencia en el manejo del cache LRU (Least Recently Used). A través del hoarding, un usuario puede indicar de manera explícita qué archivos será importante acceder más adelante. Con esta información se crea una HDB (Hoard Data Base) por cada cliente [12].

Las pasadas de acumulación (hoard walks) periódicas, aseguran que los archivos marcados como importantes, utilizando una combinación de LRU con prioridad de acumulación (hoard priority), permanezcan en el cache ante desconexiones progamadas o inesperadas [18].

Durante la operación desconectada, Venus guarda las actualizaciones en un log, usando optimizaciones para reducir el consumo de recursos. Por ejemplo, si un archivo es creado, renombrado y por último eliminado, ninguno de estos registros es salvado en el log; es como si nunca se hubiera ejecutado ninguna de estas operaciones. Cuando se restablece la conexión, Venus re-ejecuta las actualizaciones en los servers utilizando el mecanismo de *reintegración*, que se verá más adelante [12].

5.2.5 Operación bajo condiciones de conectividad débil

Existe un amplio rango de condiciones de operación entre conectividad fuerte y operación desconectada.

A medida que el ancho de banda de la conexión de la red disminuye, se torna más importante la necesidad de conservar la ilusión de conectividad fuerte. Para lograr esto, Venus intenta satisfacer la mayor cantidad de faltas en el cache, lo más pronto posible, y de ser necesario retrasa otro tráfico. La cuestión de cómo reordenar el otro tráfico se revisa acorde a los cambios en la calidad de la conexión [12]. CODA implementa una serie de modificaciones para aliviar las limitaciones de la conectividad débil. A bajo nivel el protocolo de transporte fue extendido para ser robusto, eficiente y adaptativo para soportar un amplio rango de anchos de banda. En niveles más altos se hicieron cambios para lograr una rápida validación del cache luego de una falla de conexión intermitente; para realizar propagaciones de actualizaciones en background, cuando la conexión de la red es muy lenta; y para dar servicios de faltas de cache asistidos por el usuario, cuando la conectividad es débil [17].

5.2.6 Validación rápida del cache

Cuando la conectividad es fuerte la técnica para mantener la coherencia de los caches esta basada en *callbacks*. Cuando un cliente esta desconectado, ya no puede confiar en los callbacks. Luego de reconectarse debe validar todos los objetos que tiene en el cache para detectar posibles actualizaciones en el server. Desafortunadamente, en un red lenta, el tiempo que lleva realizar esta validación puede ser muy grande [12].

La solución para este caso es el *estampado de tiempo* con múltiples niveles de granularidad. Se utilizan versiones, tanto sobre objetos individuales como sobre volúmenes. Cuando un objeto es modificado el server incrementa la estampa de versión del objeto y del volumen que lo contiene. El cliente cachea en todo momento las estampas de tiempo de los volúmenes. Cuando se restaura la conexión después de una falla, el cliente presenta sus estampas de tiempo de los volúmenes para ser validadas, si la estampa de un volumen coincide, entonces todos los objetos cacheados de ese volumen son válidos. En cambio si la estampa de un volumen no coincide, entonces cada objeto de ese volumen debe ser validado individualmente. Este mecanismo mejora notablemente la velocidad de validación del cache [18].

5.2.7 Reintegración gradual

La *reintegración gradual* (trickle reintegration) es un mecanismo que propaga las actualizaciones a los servers en forma asincrónica, produciendo un mínimo impacto sobre las actividades que se ejecutan en primer plano [18]. Esta reintegración entra en conflicto con la optimización que se logra con el log de actualizaciones que guarda Venus, pues las actualizaciones se envían a los servers antes que en el caso en que se esté desconectado, y hay menos probabilidad de poder optimizar operaciones. Sin embargo se puede atacar este problema utilizando un *sistema de envejecimiento* (aging), donde una operación no es elegible para ser enviada a los servers hasta que no cumple una determinada cantidad de tiempo en el cache [4,6].

5.2.8 Servicios de Faltas de Cache Asistidos por el Usuario

Algunas *faltas en el cache* (cache misses), son más importantes que otras. Por ejemplo, si falta una tipografía mientras se está escribiendo un documento, simplemente se puede sustituir por otra en lugar de esperar a que se realice la trasferencia de la misma, en cambio para un archivo crítico se está dispuesto a esperar el tiempo que sea necesario. Para capturar la esencia de esta idea, Venus incorpora el *modelo de paciencia del usuario* en donde la prioridad del archivo es la indicada por el usuario en el hoarding. Cuando se produce una falta en el cache, Venus estima el tiempo que se necesita para transferir el archivo, si este tiempo sobrepasa el umbral de paciencia preestablecido por el usuario, retorna un error en lugar de traer el archivo [4,6].

Este modelo ayuda a mantener utilizable el sistema en un amplio rango de anchos de banda, balanceando los siguientes dos factores: tiempos de transferencia prolongados

(causando largas esperas) y la necesidad de intervención del usuario (para decidir cuándo vale la pena esperar) [12].

5.2.9 Replicación de Servers

El mecanismo de replicación de servers permite tener volúmenes con réplicas de lectura/escritura en más de un server. El conjunto de servers donde se replica un volumen se denomina el *grupo de almacenamiento de volumen* (VSG, volume storage group). El subconjunto de servers de un VSG que se encuentra accesible es el AVSG (accesible VSG), el cual es idéntico al VSG en ausencia de fallas. Cuando se cierra un archivo que fue modificado, el cliente transfiere el archivo a todos los servers del AVSG utilizando un protocolo de RPC paralelo [2]. Con este procedimiento se maximiza la probabilidad de que cada server del VSG contenga la última copia del archivo/volumen. Además se minimiza la carga de la CPU de los servers, ya que el trabajo de replicación es realizado por los clientes. Esto último es una ventaja para la escalabilidad, ya que los CPUs de los servers son el cuello de botella de muchos DFSs.

5.2.10 Fidelidad: la Base para la adaptabilidad

Definimos *fidelidad* como el grado de similitud entre una copia presentada para su uso y su original o copia "verdadera". En general la fidelidad tiene muchas dimensiones, la consistencia es una dimensión universal bien conocida [18]. Las otras dimensiones dependerán del tipo de datos en cuestión. Por ejemplo, el video tiene por lo menos dos dimensiones: cuadros por segundo y calidad de la imagen de cada cuadro; en audio se puede variar la frecuencia de muestreo. Son entonces las dimensiones los ejes fundamentales de la adaptabilidad en movilidad. Sin embargo, la adaptabilidad no sólo depende del tipo de dato en cuestión sino que también depende de la aplicación que lo utilice, con lo cual dos aplicaciones distintas que actúen sobre el mismo tipo de dato, podrán negociar diferentes valores para los parámetros de una misma dimensión de fidelidad [12].

Consideremos, por ejemplo, una película almacenada en un server y dos aplicaciones que acceden al flujo de imágenes desde un cliente móvil. La primera aplicación es un reproductor de video y la segunda es un editor de escenas. Estas dos aplicaciones deben negociar diferentes fidelidades para acceder al mismo flujo de imágenes; no existirá una política que las satisfaga a ambas. El objetivo principal del reproductor es preservar la correspondencia entre el tiempo de la película y el tiempo real, el objetivo secundario es reproducir la película a la resolución, cantidad de cuadros por segundo y calidad de imagen originales. Cuando los recursos son abundantes, se puede cumplir con ambos objetivos. Sin embargo, cuando el ancho de banda de la red comienza a reducirse, el reproductor tendrá que sacrificar el objetivo secundario para poder cumplir su meta primaria. Entonces, podrá elegir entre descartar cuadros a color o cambiar a blanco y negro con la cantidad de cuadros por segundo original.

En cambio, el objetivo principal del editor de escenas es asegurar que el usuario vea cada cuadro de la película en forma precisa para poder editarla. Para lograrlo el editor podrá disminuir la correspondencia entre el tiempo de la película y el real; así, cuando caiga el

ancho de banda de la red accederá la película a una cantidad de cuadros por segundo menor que la real para evitar perder cuadros [12].

5.2.11 Replicación Optimista (control de accesos)

La decisión de utilizar una estrategia de replicación optimista en lugar de pesimista ha sido fundamental en el diseño de CODA. Cualquier protocolo pesimista, de una u otra manera, debe reservar lugar para almacenar el control de acceso a los objetos cuando los clientes operan desconectados. Esta reserva de lugar tiene que cumplir un difícil compromiso entre disponibilidad y facilidad de uso [20].

Por una parte, la eliminación de la participación del usuario aumenta la responsabilidad por parte del sistema de archivos, disminuyendo la sofisticación de las decisiones de almacenamiento. Una mala decisión de almacenamiento de control de accesos, produciría que a un cliente desconectado le falte un objeto crítico, o que tenga un objeto que no pueda utilizar por falta de permisos, es decir, se reduce notablemente la disponibilidad.

Por otra parte, cuanto más involucrados estén los usuarios en el proceso de alocación de accesos, menos transparente será el sistema.

Una estrategia de replicación optimista evita la necesidad de tomar todas las decisiones de alocación de accesos a priori.

Sin embargo, existe una ventaja de la replicación pesimista sobre la optimista, y es que nunca ocurren fallas de reintegración. Las fallas de reintegración dependen de la naturaleza en la concurrencia a archivos y los hábitos de trabajo de los usuarios.

Herlihy dio la siguiente motivación para control de concurrencia optimista, que es igualmente aplicable al control de replicación optimista:

... [el control de replicación optimista] está basado en la premisa de que es más efectivo pedir disculpas que pedir permiso [20].

5.2.12 Detección y Resolución de Conflictos de Actualización

CODA aborda el problema de actualización concurrente en una red particionada utilizando una estrategia optimista de control de réplicas. Esto ofrece el mayor grado de disponibilidad dado que es posible actualizar los datos en cualquiera de las particiones. Cuando se lleva a cabo la reintegración el sistema asegura la detección de los conflictos de actualización y provee mecanismos para ayudar al usuario a recuperar la consistencia perdida en dicha situación. CODA utiliza diferentes estrategias para archivos que para directorios al manejar actualizaciones concurrentes. Para directorios Venus posee suficiente conocimiento semántico para tratar en forma transparente la *resolución* de conflictos en la reintegración. La resolución falla sólo si el nombre de un archivo recientemente creado colisiona con uno ya existente, si un objeto actualizado por un cliente fue borrado por otro, o si fueron modificados los atributos de un directorio en el server y en el cliente [18].

Como UNIX trata a los archivos como tiras de bytes sin formato, CODA no posee el conocimiento semántico suficiente para resolver conflictos en los archivos. En cambio, ofrece

un mecanismo para instalar e invocar en forma transparente *resolvedores específicos por aplicación* (ASRs Application-specific resolvers). Un ASR es un programa que encapsula el conocimiento específico de la aplicación necesario para distinguir entre una inconsistencia genuina y diferencias reconciliables. Ejemplos de aplicaciones en donde es altamente conveniente usar un ASR son: agendas de citas, registro de avances de proyecto, etc. Si el ASR no puede resolver el conflicto, le presenta la inconsistencia al usuario para su reparación manual [18].

5.2.13 Transacciones sólo de incomunicación (Isolation Only Transactions)

La emulación que hace CODA del modelo de file system UNIX tiene el beneficio de ser compatible con las aplicaciones existentes. Desafortunadamente, el modelo UNIX es débil con respecto al soporte que brinda para acceder a archivos en forma concurrente. En particular, UNIX no tiene la noción de conflictos de lectura-escritura. Esta deficiencia es especialmente importante en computación móvil, porque los largos períodos de desconexión o conexión débil, aumentan la probabilidad de inconsistencias de lectura-escritura.

Consideremos por ejemplo, un CEO usando una notebook desconectada para trabajar en un reporte para una reunión de accionistas. Antes de desconectarse cachea una planilla de cálculo que contiene el último presupuesto disponible, y escribe su reporte en base a esa planilla. Durante su ausencia se genera un nuevo presupuesto, y se actualiza la planilla en el server. Cuando el CEO regresa y su notebook se reintegra a la red, debería darse cuenta que su reporte está basado en datos desactualizados. Notemos que este no es un conflicto escritura-escritura, ya que nadie más ha actualizado su reporte. En cambio, se trata de un conflicto de lectura/escritura entre la planilla y el reporte. Ningún sistema UNIX tiene la habilidad de detectar y manejar dicho problema.

CODA ha sido extendido con un nuevo mecanismo llamado *transacciones sólo de incomunicación* (Isolation Only Transactions, IOTs) para resolver este defecto. El mecanismo IOT ofrece a las aplicaciones, una consistencia mejorada en forma conveniente y fácil de usar [17].

Una IOT es una secuencia de operaciones sobre archivos, que son tratadas como una unidad a los efectos de detección y resolución de conflictos. El nombre "transacciones sólo de incomunicación" se debe al hecho de que este mecanismo sólo enfoca el aspecto de incomunicación por parte del cliente, en relación al concepto clásico de transacción. En otras palabras, IOT no garantiza la falla en atomicidad y sólo garantiza permanencia de datos condicionalmente. Por ejemplo, un programa compilado con IOT en un cliente desconectado, puede ser recompilado en la reintegración si se detectó un cambio en las librerías que usa. El subsistema de IOT en Venus realiza detección de conflictos lectura-escritura automáticamente basado en ciertas restricciones de seriabilidad. Soporta una variedad de mecanismos de resolución de conflictos, tales como la re-ejecución y el uso de ASRs.

5.3 NFS

NFS (Network File System, propietario de SUN Microsystems) fue desarrollado inicialmente como un protocolo para proveer acceso remoto transparente para compartir un file system a través de una red. Fue diseñado para ser independiente de la máquina, sistema operativo, protocolo de transporte o arquitectura de red. Esta independencia se logra con el uso de primitivas de RPC (Remote Procedure Call) construidas sobre XDR (Representación eXternas de Datos) [13].

A pesar que diferentes servicios de archivos distribuidos habían sido desarrollados exitosamente en universidades y laboratorios de investigación, NFS fue el primero que fue presentado como un producto. El diseño y la implementación de NFS tuvieron un considerable éxito técnico y comercial. [2]

Para alentar su adopción como un estándar, la definición de las interfases más importantes fueron hechas de dominio público (rfc1094, 1989), permitiendo a otras compañías implementar clientes y/o servers NFS. Incluso el código fuente fue entregado bajo licencia a otros proveedores de computadoras. Actualmente es soportado por muchos proveedores y es tenido en general como el estándar de facto.

A continuación se describen las funcionalidades de este file system, sin ahondar en los detalles de diseño que se ven más profundamente en el capítulo en donde se documenta la implementación de la interfase NFS-CODA.

5.3.1 Transparencia de Acceso

Los servers NFS son "bobos" y los clientes son "inteligentes". Es el cliente el que realiza el trabajo necesario para convertir el acceso a archivos generalizado que provee el server, en un método local de acceso a archivos, útil para las aplicaciones [14]. Esta propiedad se llama *transparencia de acceso* [2].

5.3.2 Transparencia de Localización

Cada cliente arma un espacio de nombres agregando el file system remoto a su espacio de nombres local. Los file systems deben ser *exportados* por el nodo que los contiene (server), y deben ser *montados* en forma remota por el cliente antes de poder ser accedidos por los procesos locales al cliente. El punto en la jerarquía de nombres del cliente en el cual se monta el file system remoto, es determinado sólo por el cliente. Por lo tanto NFS no contempla un único espacio de nombres en toda la red. Cada cliente ve un conjunto de file systems remotos que es determinado en forma local y los archivos remotos pueden tener diferentes nombres de ruta (pathnames) en clientes distintos. Sin embargo se puede lograr un espacio de nombres uniforme, estableciendo tablas de configuración apropiadas en cada cliente, logrando así el objetivo de *transparencia de localización*, que AFS provee en forma nativa [2].

5.3.3 Independencia de Localización

Las tablas de montado de los clientes contienen la información indicando en qué server se encuentra cada subárbol de archivos. Por lo tanto, un movimiento de subárboles de archivos entre servers implica necesariamente un cambio en las tablas de montado de los clientes. Entonces no es posible lograr completa *independencia de localización* a través de NFS [2].

5.3.4 Montado Automático de File Systems

El servicio de montado en NFS opera en el momento del boot del sistema o cuando un usuario se loguea en su workstation, montando todos los file systems en caso de que vayan a ser utilizados en esa sesión de trabajo. Esto resultó ser muy pesado para algunas aplicaciones y producía gran número de entradas inútiles en las tablas de montado. El servicio de NFS *Automounter* provee una solución, e incidentalmente un método simple de seleccionar dinámicamente entre varios archivos replicados de sólo lectura, permitiendo la distribución de la carga de los servers de archivos. Automounter se ejecuta como un servicio local a nivel de usuario en cada cliente NFS, y permite usar pathnames que se refieren a file systems remotos no montados. Cualquier referencia que realice un proceso de usuario a un pathname ubicado en un subárbol manejado por Automounter, causará que el NFS pase el requerimiento al Automounter, que montará los file systems necesarios para accederlo [2].

5.3.5 Transparencia ante fallas

NFS asume una implementación de servers *carentes de estado*. La carencia de estado consiste en que no es necesario mantener el estado de ningún cliente para funcionar correctamente. La mayoría de las operaciones del protocolo de acceso a archivos son repetibles o idempotentes. Esta capacidad de no mantener el estado es una ventaja distintiva sobre servers que mantienen estado en la eventualidad de una caída del sistema. Con servers carentes de estado, un cliente sólo necesita reintentar el requerimiento hasta que el server responda, sin siquiera notar que se produjo una caída en el server. Cuando un cliente sufre una caída de sistema, la misma no tendrá ningún efecto en los servers que está utilizando, ya que el server no mantiene ningún estado de sus clientes [13]. Estas características determinan lo que se suele llamar *transparencia ante fallas* [2].

5.3.6 Manejo del Cache

Se debe utilizar algún método de cache en los clientes y en los servers para mejorar la performance. En el caso de los servers, se utiliza el manejador de cache nativo del sistema operativo (buffers y reads ahead), pero no se usa delayed writes (mantener páginas en cache hasta que se necesite el lugar o se realice un flush de la memoria), que es reemplazado por write through (grabación inmediata al disco). Esto se debe a que una caída en el server NFS no es detectada por los clientes, y si se implementase delayed writes, se perderían los writes que se encontraren en el cache sin haber sido escritos al disco. En la implementación se

optimiza la apertura de archivos, manteniéndolos abiertos por una determinada cantidad de tiempo, evitando así tener que abrirlos cada vez que el cliente accede a un archivo. En los clientes se usa un método de cache con consistencia débil incorporado al módulo de NFS cliente. Se cachean los resultados de *read()*, *write()*, *getattr()*, *lookup()* y *readdir()*, para reducir el número de pedidos transmitidos a los servers. Este tipo de cacheo en los clientes introduce el problema potencial de la existencia de versiones diferentes de archivos en distintos nodos de clientes.

Las escrituras realizadas por los clientes se conservan en el cache hasta que puedan ser enviadas al server en forma asincrónica (i.e. cuando se ejecute un *close()* del archivo o un *sync()* en el cliente); además no actualizan inmediatamente las copias del archivo que residen en los caches de los demás clientes. Para minimizar estas inconsistencias se desarrolló un método basado en *estampado de tiempos* (timestamp) que valida los bloques cacheados al momento de ser usados. El estampado de tiempo, no necesariamente indica el tiempo real, se puede lograr el efecto deseado con un simple contador incremental de versión. Así, cuando se cachea un bloque de un archivo, también se obtiene la estampa de tiempo de la última modificación del mismo. Al querer abrir el archivo, acceder a un nuevo bloque o pasado el *timeout de validez* se le pide al server la nueva estampa de tiempo para contrastarla con la anterior, si no coinciden se invalidan todos los bloques del cache que pertenezcan a ese archivo. El timeout de validez es el tiempo que se asume como válido un bloque del cache, normalmente 3 seg. para archivos y 30 seg. para directorios [2].

Como se puede observar existen 2 fuentes básicas de inconsistencias al compartir archivos entre clientes. La primera está dada por el tiempo en que una modificación queda en el cache del cliente, y la segunda es producida por la "ventana" de 3 seg. para la validación del cache [2].

5.3.7 Seguridad

Dado que el server NFS no conserva estados, tampoco mantiene registro de los archivos que se encuentran abiertos en sus clientes. Entonces, el server debe verificar la identidad del usuario contra los permisos de acceso de los archivos en cada requerimiento, verificando si se le permite el acceso al archivo de la manera requerida. El protocolo SUN RPC que se usa en NFS, requiere que los clientes envíen la información de autenticación (por ejemplo, los 16 bits en UNIX del User ID y Group ID) con cada requerimiento para ser verificada contra los permisos de acceso de los atributos del archivo. A simple vista esto constituye una falencia de seguridad [16,18].

El server NFS provee una interfase RPC convencional en un port bien conocido. Cualquier proceso puede comportarse como un cliente, enviando requerimientos al server para acceder o actualizar un archivo. Este cliente, puede modificar la llamada RPC para incluir el User ID de cualquier usuario, haciéndose pasar por el usuario sin su permiso o conocimiento. Se han desarrollado métodos alternativos de autenticación para paliar este problema de seguridad. A partir de NFS 3 el tipo de autenticación utilizada puede ser:

-AUTH_NONE acceso irrestricto

-AUTH_UNIX acceso de acuerdo al UID y GID

-AUTH_DES UID y GID encriptados por el método DES y utilización de claves públicas

-AUTH_KERB utilización de encriptación, claves secretas Kerberos y tickets.

Es también el proceso de montado el que permite al server otorgar privilegios de acceso a un grupo restringido de clientes vía control de exportación [2].

5.3.8 Lockeo de Archivos

El manejador de lockeos (Lock Manager) brinda el soporte para lockear archivos en el ambiente de NFS. Surge la disyuntiva entre la inherente conservación de estados necesaria en el lockeo de archivos, y la carencia de conservación de estados en NFS. El protocolo del manejador de lockeos de red (Network Lock Manager, o NLM) resuelve este problema aislando en un protocolo separado la conservación de estados de los archivos lockeados.

5.3.9 Montado Soft vs Hard

Los filesystems remotos pueden ser montados en el cliente con dos modalidades distintas: montado *hard* o *soft*. Si el file system es montado con modalidad hard y el archivo que se desea acceder no está disponible, el proceso de usuario se detiene, esperando indefinidamente hasta que el pedido se satisfaga; por lo tanto en el caso de una caída del server los procesos de usuario quedan suspendidos hasta que el server se reinicie, y luego continuarán como si no hubiese ocurrido falla alguna. En cambio si la modalidad es soft el módulo NFS del cliente retornará un código de error luego de unos pocos intentos. No todos los programas o utilitarios están preparados para resolver fallas de este tipo [2].

5.3.10 Escalabilidad

La escalabilidad del servicio de NFS es limitada. Originalmente fue diseñado para que cada server soporte de cinco a diez clientes cuando los accesos a los archivos en los clientes son 100% remotos, o una cantidad de cincuenta clientes cuando los archivos clave están en el disco local. Las limitaciones de diseño de NFS (overhead de protocolo) y la falta de soporte de replicación de archivos limitan la escala de sistemas distribuidos basados en NFS. El número de clientes que pueden acceder simultáneamente a un determinado archivo en un server, está restringido por la performance del server, y se puede constituir en un cuello de botella que afectará el rendimiento de todo el sistema [2].

5.4 CIFS

El uso de Internet y la World Wide Web se ha caracterizado por el acceso de sólo lectura. Los protocolos existentes como el FTP son una buena solución para transferencia de archivos en un sólo sentido. Sin embargo, cada vez serán más necesarias nuevas interfases de lectura/escritura a medida que Internet se transforme en más interactiva y colaborativa. La

comunidad de Internet obtendría importantes beneficios si se adoptase un protocolo común para compartir archivos, que contenga semánticas modernas tales como archivos compartidos, lockeo de un rango de bytes, coherencia del cache, notificación de cambios, almacenamiento replicado, etc.

CIFS (Common Internet File System) [4,7,9,10] pretende proveer un mecanismo abierto multi-plataforma para que sistemas cliente puedan requerir servicios de archivos e impresión a un server a través de la red. Está basado en el protocolo standard SMB (Server Message Block) ampliamente utilizado por computadoras personales y workstations ejecutando una gran variedad de sistemas operativos.

Sobre el protocolo CIFS se ejecuta el Microsoft Distributed File System (MDFS), que provee una estructura de árbol única para múltiples volúmenes compartidos localizados en diferentes servers de una red. Un árbol de MDFS hace que el acceso a la red sea más fácil para los usuarios, los cuales no necesitarán localizar manualmente qué server contiene un recurso en particular. Luego de conectarse al root de un árbol MDFS, ellos pueden navegar a través del árbol y acceder a todos los recursos contenidos en él, sin importar el server en el cual se encuentran físicamente localizados estos recursos [9].

5.4.1 Acceso a archivos

El protocolo permite el conjunto de operaciones usual para archivos: open(), close(), read(), write() y seek().

Varios mensajes SMB de requerimientos de clientes y respuestas de los servers, como ser el SMB_COM_OPEN, pasan los modos de acceso del archivo. Por cada requerimiento de un proceso de cliente (PID) se obtiene una respuesta del server con los permisos otorgados. En el caso del open() se entrega un file handle (FID) con alcance por cliente que será usado para acceder al archivo en las llamadas subsecuentes. El verdadero acceso permitido a través del FID depende del modo especificado en el requerimiento de open().

La codificación usada es la siguiente:

```
5432 1098 7654 3210
rWrC rLLL rSSS rAAA
```

donde:

W - Modo write through. No se permite read ahead o write behind sobre este archivo o dispositivo.

S - Modo compartido:
- 0 - Compatibility mode
- 1 - Deny read/write/execute (exclusive)
- 2 - Deny write
- 3 - Deny read/execute
- 4 - Deny none

A - Modo de acceso
- 0 - Open for reading
- 1 - Open for writing
- 2 - Open for reading and writing
- 3 - Open for execute

`rSSSrAAA` = 11111111 (hex FF) indica un open de FCB

C - Modo de cacheo
- 0 - Normal file
- 1 - Do not cache this file

L - Localidad de referencia
- 0 - Locality of reference is unknown
- 1 - Mainly sequential access
- 2 - Mainly random access
- 3 - Random access with some locality
- 4 to 7 - Currently undefined

El modo compatibilidad DOS provee exclusión implícita a nivel cliente. Es decir, cuando un cliente abre en modo compatibilidad para escritura ningún otro cliente puede acceder al archivo. Si un cliente abre un archivo en modo compatibilidad para lectura, los demás clientes podrán abrir el archivo en modo compatibilidad sólo para lectura y ninguno para escritura. Como excepción los PIDs del mismo cliente que ejecutó el primer open() en modo compatibilidad podrán abrir el archivo con cualquier combinación de reads y writes.

5.4.2 Lockeo de archivos y registros

El protocolo permite lockear archivos y registros, así como acceso a archivos sin lockeo. Una vez que una aplicación lockea un archivo o registro, cualquier otra aplicación que no solicite lockeo no tendrá acceso al archivo.

El mensaje de lock de un registro es enviado para lockear un determinado rango de bytes. Se puede lockear más de un rango de bytes en un archivo dado, siempre y cuando no se superpongan. Los locks previenen intentos de lockear, leer o escribir las porciones ya lockeadas del archivo por otros clientes o PIDs.

Un lock sólo puede ser desbloqueado por el PID que realizó el lock. El requerimiento de un cliente no espera a que el lock le sea otorgado. Si el lock no puede ser inmediatamente otorgado (dentro de los 200 o 300 ms) el server debe retornar un error al cliente.

Existe también un mensaje que permite realizar el lockeo para lectura de una cantidad de bytes y a la vez especificar un estimado de bytes que se leerán en el futuro. El server utilizará esta estimación para asignar buffers de read ahead que mejorarán la performance en los futuros accesos. Esta estimación no será incluida en el rango de bytes a lockear.

El mensaje que permite grabar y deslockear archivos primero graba en forma segura los bytes especificados y luego los deslockea. Si el requerimiento especifica un rango que supera el final del archivo entonces el archivo se extenderá. Este requerimiento se utiliza en conjunto con el anterior.

5.4.3 Locks Oportunísticos

Un mecanismo para mejorar la performance en acceso a archivos es el empleo de *locks oportunísticos* (OPLOCKS) sobre archivos completos. Se basa en el posibilidad de alterar dinámicamente la estrategia de buffereo en un cliente de forma consistente, de manera tal que un cliente no necesita escribir la información de un archivo al server si sabe que ningún otro proceso está accediendo a estos datos. También, el cliente puede hacer un buffer de read-ahead de los datos si sabe que ningún otro proceso quiere escribirlos.

Encontramos tres tipos de oplocks: LEVEL II, EXCLUSIVE y BATCH.

- Level II oplock: permite a múltiples clientes mantener el mismo archivo abierto, e informa que nadie ha modificado el archivo todavía. Permite al cliente realizar reads y búsquedas de atributos de archivo utilizando información local de su cache o de read-ahead pero todos los demás requerimientos deberán ser enviados inmediatamente al server. Si un cliente A pide acceder a un archivo para lectura y está dispuesto a compartir acceso de lectura/escritura, si ningún cliente tiene abierto el archivo, el protocolo le otorga un exclusive oplock. Cuando el cliente B abre el archivo para sólo lectura, el server debe sincronizar con el cliente A en caso de que existan locks en su buffer y renegociar a oplock level II. Una vez sincronizado, el requerimiento del cliente B puede ser completado. Sin embargo, el cliente B es informado que tiene un oplock level II en lugar de un exclusive oplock. En este caso, ningún cliente que tenga un oplock level II puede bufferear información de lockeo en su máquina. Esto permite al server garantizar que si se realiza cualquier operación de escritura, sólo será necesario notificar a los clientes level II que el oplock debe ser roto, sin necesidad de sincronizar con todos los que acceden al archivo. De este modo el oplock level II se rompe hacia NONE, indicando que algún cliente que tenía abierto el archivo realizó una escritura. Todos los clientes que reciben la ruptura del oplock level II a NONE deben vaciar sus buffers de read ahead. No se espera ninguna respuesta de los clientes cuando se rompe de level II a NONE. De esta forma es posible que las aplicaciones se registren en un server, para ser notificadas cuando el contenido de un archivo o directorio ha sido modificado. Se puede utilizar este mecanismo, por ejemplo, para saber cuándo necesita refrescarse el display sin necesidad de interrogar (polling) el server constantemente.

- Exclusive oplock: este lock es otorgado por el server a un cliente siempre y cuando el archivo no tenga ningún tipo de acceso pedido por otro cliente. Si a un cliente le es concedido un exclusive oplock, éste puede realizar un buffereo local de write behind, de lock y de read ahead, puesto que sabe que es el único que está accediendo al archivo.

Cuando un cliente A abre el archivo con exclusive oplock, si en el futuro otro cliente B requiere una apertura del mismo archivo, entonces el server debe *romper* el oplock para el cliente A. Romper el oplock implica enviar al server cualquier información de lock o datos escritos que se encuentren en el buffer del cliente A, purgar sus buffers de read ahead y después comunicarle al server que el cliente A ha aceptado la ruptura del oplock. Si el cliente B no modificará el estado del archivo, entonces el server puede cambiarle a A su exclusive oplock por un level II. Estos mensajes de sincronización informan al server que ahora es posible que el cliente B complete el open().

- Batch oplock: este tipo de lock permite mantener abierto en un cliente un archivo y realizar cualquier tipo de operación sobre los datos de read ahead o el cache local. Con esto se evitan situaciones de abrir y cerrar el archivo muchas veces en un único proceso. (Se podría pensar en una optimización adicional si se tuviese un "exclusive" batch oplock y un "level II" batch oplock, con lo cual se permitiría compartir el level II batch oplock con otros clientes. Siendo esta una situación frecuente en el procesamiento de archivos por lotes). Como en el caso anterior, si otro cliente desea tener acceso al archivo, se deberá romper el batch oplock lo cual implicará vaciar los buffers de read ahead y transferir la información de lock al server. (No queda claro cuándo se devuelve el batch oplock, si es por timeout, por propagación lazy de las actualizaciones, etc.)

5.4.4 Soporte para DFS

El dialecto del protocolo NT LM 0.12 y los subsiguientes soportan operaciones de file system distribuido. El modelo de file system distribuido empleado es un modelo de *referencias* (referrals). El protocolo especifica la manera por la cual los clientes reciben las referencias. El cliente puede setear un flag en el requerimiento del header SMB indicando que desea que el server resuelva el path SMB con el DFS que el server conoce. El server intenta resolver el nombre requerido a un archivo contenido en el árbol de directorio local indicado por el Tree ID (TID) del requerimiento y procede normalmente. Si el pathname requerido se resuelve a un archivo en un server diferente se devuelve el error STATUS_DFS_PATH_NOT_COVERED, que significa que el server no soporta la parte del espacio de nombres de DFS necesario para resolver el pathname requerido en forma local.

Al recibir este error el cliente debe solicitar al server una referencia (TRANS2_GET_DFS_REFERRAL). La respuesta a este requerimiento contendrá una lista de nombres de servers y shares a intentar, unidos al nombre del archivo del requerimiento. El cliente entonces solicita a cada uno de los servers referidos el archivo solicitado.

5.4.5 Volúmenes virtuales replicados distribuidos

El protocolo soporta subárboles del file system que se muestren al cliente como si fuesen un único volumen y un único server, pero que en realidad abarcan múltiples volúmenes o servidores. Los archivos y directorios de este subárbol pueden ser físicamente movidos a otros servers, y su nombre no necesita ser cambiado, aislando a los clientes de los cambios en la

configuración del server. Estos subárboles pueden también ser transparentemente replicados para compartir la carga. Cuando un cliente requiere un archivo el protocolo utiliza "referrals" para dirigir transparentemente al cliente al server que lo alberga. Sin embargo, no se provee ningún mecanismo para la sincronización de las réplicas, la cual debe ser realizada por algún otro medio manual o automático.

Esta funcionalidad equivale a las transparencias de localización y de migración vistas anteriormente.

5.4.6 Independencia del mecanismo de resolución de nombres

El protocolo permite a los clientes resolver los nombres de los servers utilizando cualquier mecanismo de resolución de nombres. En particular, puede usar DNS (Domain Name Service) permitiendo el acceso a los file systems de otras organizaciones a través de Internet, u la organización jerárquica de servers dentro de una organización.

5.4.7 Modelo de seguridad

Cada server ofrece un conjunto de recursos disponibles a los clientes en la red. Un recurso compartido puede ser un directorio, impresora, etc. Desde el punto de vista de los clientes, el server no tiene dependencias externas de ningún otro server para ofrecer sus servicios; el cliente considera al server el único proveedor del archivo (o cualquier otro recurso) que se accede.

El protocolo CIFS requiere autenticación con el server por parte de los usuarios previo a permitir el acceso a sus archivos, y cada server autentifica sus propios usuarios. Un cliente debe enviar información de autenticación al server antes de que el server le permita el acceso a sus recursos.

Un server requiere que el cliente provea el nombre de usuario y alguna prueba de su identidad para obtener acceso (generalmente algún criptograma derivado de su password). La granularidad de la autorización depende del server. Por ejemplo, puede utilizar el nombre de usuario para verificar la ACL en archivos individuales, o puede tener una única ACL que se aplique a todos los archivos de un árbol de directorio.

Cuando un server valida la cuenta y la password presentada por un cliente, se retorna un identificador representando la instancia autenticada del usuario, a través del user ID (UID) en la respuesta SMB. Este UID debe ser incluido en todos los requerimientos futuros que se realicen en nombre de ese cliente.

5.5 Resumen

Se han estudiado los principales sistemas de archivos distribuidos: AFS, CODA, NFS y CIFS.

Cada uno de ellos introduce nuevos conceptos en el universo de los file systems distribuidos, aportando sus propias soluciones a problemas conocidos. Tal es el caso de:

espacio de nombres uniforme en AFS; adaptabilidad, reintegración gradual y fidelidad en CODA; independencia del protocolo subyacente (RPC) y XDR en NFS; y locks oportunísticos en CIFS.

CAPITULO 6

Análisis comparativo entre DFSs

Luego de haber estudiado con cierto detalle los file systems más representativos actualmente disponibles, hemos visto que cada uno de ellos resalta y optimiza alguna característica específica, en desmedro quizás de otras, que no eran objetivos fundamentales en el diseño. En base a estas diferencias y equivalencias los compararemos de acuerdo a las características descriptas en el capítulo 4.

6.1 Comparación entre los DFSs presentados

NOTA: *En todos los ítems en que no está específicamente citado CODA, es porque se comporta en forma idéntica a AFS.*

Espacio de nombres y Transparencia de localización

AFS: Nombres idénticos para todas las estaciones de trabajo. El espacio de nombres se especifica en forma centralizada, por medio de archivos locales de configuración que apuntan a una base de datos centralizada de celdas y cada celda administra su propio espacio de nombres. El nombre de cada archivo está compuesto por:

/afs/nombre_de_celda/nombre_de_directorio_remoto/nombre_de_archivo

De esta forma se provee en forma nativa transparencia de localización.

CODA: Al igual que AFS, presenta nombres idénticos para todas las estaciones de trabajo. El nombre de cada archivo está compuesto por:

/coda/ nombre_de_directorio_remoto/nombre_de_archivo

Se provee en forma nativa transparencia de localización.

NFS: Nombres dependientes de la estación de trabajo de la cual se accede al sistema de archivos remoto. El espacio de nombres se especifica en forma local a cada estación de trabajo. Por medio de archivos locales de configuración se determina el nombre de directorio que se usará para acceder al file system remoto. El nombre de cada archivo está compuesto por:

/directorio_de_montado_local/nombre_de_directorio_remoto/nombre_de_archivo

Con un esfuerzo adicional de administración de los clientes es posible uniformar el espacio de nombres que de otra forma sería anárquico.

No se provee en forma nativa transparencia de localización, ya que es necesario conocer el nombre del host que contiene un archivo para poder montarlo.

CIFS: Nombres de archivos idénticos para todas las estaciones de trabajo. El espacio de nombres se especifica según una convención de formación de nombres, con partes bien definidas. El nombre de cada archivo esta compuesto por:

//server_dfs/root_dfs/nombre_del_share_remoto/nombre_de_archivo

En el server_dfs se configura el árbol de directorios del root_dfs que contiene los shares de las máquinas remotas.

La transparencia de localización se logra mediante el mecanismo de "referrals" explicado anteriormente. Sin embargo, no existe ningún mecanismo para prohibir el acceso a un servidor remoto conociendo su nombre.

Transparencia de acceso (acceso local vs remoto)

AFS: No existe distinción entre el acceso a archivos remotos y locales. Siempre y cuando se cuente con la autenticación en la celda que contiene los archivos remotos a los que se desea acceder.

NFS: No existe distinción entre el acceso a archivos remotos y locales. Si se cuenta con el servicio de AutoMounter, y se tiene permisos de acceso a los archivos remotos a los que se accede por medio de la exportación del server remoto. No obstante, en el poco frecuente caso de montado soft, la aplicación debería contemplar las situaciones que surgen de los errores de indisponibilidad de los recursos compartidos.

CIFS: No existe distinción entre el acceso a archivos remotos y locales. Siempre y cuando se cuente con los permisos correctos a los archivos remotos a los que se desea acceder.

No está incluido en este punto de evalución el manejo de locks, que se verá en otro ítem.

Transparencia a la replicación

AFS: Soporta nativamente réplicas de sólo lectura, existiendo una sola de lectura/escritura. La propagación de los cambios hacia las copias de sólo lectura se realiza por medio de un procedimiento explícito. La copia que utilizará el requerimiento de cada cliente es seleccionada en base a la carga de los servers y la accesibilidad.

CODA: Soporta nativamente réplicas de lectura/escritura. Al cerrarse un archivo modificado, Venus lo envía a todos los servers del AVSG en paralelo.

NFS: No soporta ningún tipo de replicación en forma nativa. Sin embargo el servicio de AutoMounter puede proveer acceso a réplicas de sólo lectura, permitiendo distribuir la carga. No obstante no existe mecanismo alguno de sincronización de estas copias.

CIFS: El root_dfs puede contener subárboles replicados que accederá transparentemente para balancear la carga. No obstante, no se provee ningún tipo de sincronización entre las réplicas, es por esto que son más apropiadas las réplicas de sólo lectura.

Transparencia a la migración

AFS: Cada server contiene una copia de la base de datos de localización (que está totalmente replicada) que mapea volúmenes a servers. Pueden ocurrir inconsistencias temporales en esta base de datos cuando se mueve un volumen, pero son inofensivas pues se ha previsto dejar información de redireccionamiento en el server del cual se ha removido el volumen. Con este mecanismo y el cacheo de archivos completos se logra total transparencia de migración.

NFS: No es posible mover un filesystem de un server a otro sin afectar la operación de los clientes, es decir los clientes ya no encontrarán los archivos que estaban accediendo. Por otro lado, se deberá modificar manualmente las tablas locales en cada uno de los clientes para que surta efecto la migración. En definitiva, NFS no es transparente a la migración de filesystems.

CIFS: No es posible mover un subárbol de un server a otro sin afectar la operación de los clientes, al igual que en NFS, los clientes ya no encontrarán los archivos que estaban accediendo (es decir, los que se tenían abiertos). CIFS no es transparente a la migración.

Transparencia de escalabilidad

AFS: Fue diseñado con la escalabilidad como característica clave. Esto se logra con la administración centralizada de las bases de datos y de las autorizaciones de los usuarios. También son importantes la transparencia a la replicación, el cacheo de archivos completos y la utilización de callbacks, con lo cual se reduce la carga en los servidores.

NFS: Fue creado para compartir recursos entre unas pocas máquinas en una red de área local. Al no poseer facilidades de replicación y tener una administración distribuida en los clientes, se complica mucho el crecimiento ya que, cuantas más máquinas se

agreguen en la red, más carga deberá soportar el servidor de datos y más compleja se tornará la administración del sistema.

CIFS: Fue pensado para compartir recursos a través de Internet. Permite la replicación y tiene administración centralizada, lo cual contribuye a la escalabilidad. Otras características que facilitan el crecimiento son: manejo de locks oportunísticos y notificación a los clientes de cambios en los archivos y directorios.

Transparencia ante fallas

AFS: Posee dos características que le otorgan cierta transparencia ante fallas: cacheo de archivos completos y replicación transparente.

CODA: Agrega a AFS la capacidad de trabajar en forma desconectada o débilmente conectado y permite la replicación de volúmenes de lectura/escritura. Por lo tanto mejora notablemente la disponibilidad de los datos, inclusive ante la ocurrencia de fallas de las comunicaciones o de los servidores.

NFS: Basa su tolerancia ante fallas en su naturaleza stateless (carente de estado), con lo cual todas las operaciones son repetibles o idempotentes. Cuando un servidor falla, se suspende el servicio que brindaba hasta que se reinicia, los clientes seguirán operando sin notar la caída del server.

CIFS: La única tolerancia ante fallas disponible, está dada por la replicación de subárboles.

Transparencia de concurrencia y consistencia

AFS: Semántica de sesión sobre archivos completos entre procesos de distintos clientes. Conserva la semántica UNIX entre dos procesos ejecutándose en el mismo cliente.

CODA: Semántica de sesión sobre archivos completos con detección de conflictos de actualización. Sin embargo esta semántica se ve debilitada, ya que puede suceder que, en el conjunto de servers que contiene la copia del archivo y que son accesibles (AVSG), no se encuentre la última versión del archivo.

NFS: Respeta la semántica UNIX, basado en una ventana de tiempo. Si la ventana fuese 0 segundos, la semántica sería realmente UNIX, pero no habría cache. El tiempo de la ventana es aproximadamente 5 segundos, y dentro de una ventana se trabaja en una semántica de sesión (de los bloques que están en el cache).

CIFS: Respeta estrictamente la semántica UNIX y se optimiza utilizando locks oportunísticos que permiten la administración dinámica del cache local, ofreciendo read ahead y write behind. Esta optimización no tiene en cuenta la presencia de volúmenes replicados.

Coherencia de caches

AFS: Mecanismo basado en callbacks. Renovación de tokens de acuerdo a una ventana de tiempo, para evitar los efectos de una eventual pérdida de callbacks producida por fallas de comunicación.

CODA: Mecanismo basado en callbacks heredado de AFS. Además, para el manejo de las réplicas de lectura/escritura, Venus detecta cambios en el AVSG con el fin de convalidar los tokens de los callbacks de los archivos que tiene en su cache. Esto lo realiza enviando mensajes de prueba a todos los servers del VSG cada T segundos (normalmente en el orden de los 10 minutos).

NFS: Mecanismo basado en timestamps para los buffers de lectura. Como para las grabaciones se utiliza escritura demorada (hasta cerrar el archivo o *sync()*) se pueden producir inconsistencias entre los caches de los clientes. Empíricamente estas inconsistencias resultaron ser muy esporádicas.

CIFS: Se optimiza el manejo del cache utilizando locks oportunísticos que permiten una administración dinámica. Se ofrece read ahead y write behind en exclusive locks, sólo read ahead en oplock level II y batch. Cuando el oplock es "none" no se ofrece ni read ahead ni write behind.

Locks

AFS: No soporta lockeo de tiras de bytes. Sólo soporta lockeo advisory de archivos completos desde la misma workstation. Procesos en distintas workstations que deseen pedir un *flock()* obtendrán un error EWOULDBLOCK.

NFS: Tiene las mismas características de lockeo que UNIX para procesos en la misma workstation. Para implementar una política de lockeo a nivel del DFS, se utiliza el Network Lock Manager (NLM), que encapsula en un protocolo separado los estados de los archivos lockeados.

CIFS: Honra estrictamente los estándares de lockeo de UNIX tanto para archivos completos como sobre tiras de bytes.

Disponibilidad

AFS: La disponibilidad está dada por las réplicas de sólo-lectura y el cacheo de archivos completos. Ante una caída del server, los tokens de callback de los archivos cacheados por el cliente expirarán y deberá conectarse a otra réplica para continuar. En el caso en que estuviese accediendo a la réplica de escritura, las grabaciones subsiguientes fallarán.

CODA: La disponibilidad fue uno de los objetivos principales en el diseño. A diferencia de AFS, existen varias réplicas de lectura-escritura, lo cual permite, ante la caída de un server, seguir trabajando con alguna de las réplicas. Además, puede operar en forma desconectada, si el archivo se encuentra en el cache, se podrán realizar lecturas y escrituras sobre el mismo.

NFS: Tiene el problema que las aplicaciones van a quedar congeladas durante el tiempo en que el server se encuentre caído, resultando en la indisponibilidad del sistema. Además, la replicación provista es precaria. A favor, podemos citar la transparencia ante fallas que mejora la disponibilidad.

CIFS: La disponibilidad es baja ya que ante caídas del server, se pierde la conexión de los clientes y las aplicaciones cancelan.

Seguridad

AFS: Los usuarios se deben autenticar en la celda utilizando Kerberos. Los tokens tienen tiempo de expiración. Todo el tráfico entre las workstations y los servers viaja encriptado. Se utiliza ACLs para controlar el acceso a directorios y los nueve bits de UNIX para acceder a los archivos.

CODA: La seguridad es idéntica a la de AFS, excepto que en lugar de autenticarse en una celda el usuario se autentica en un server.

NFS: La seguridad es precaria ya que al ser stateless, se debe enviar el UID con cada mensaje. Las versiones más recientes preveen el mejoramiento de los aspectos de seguridad utilizando Kerberos.

CIFS: La seguridad se basa en ACLs para acceder a directorios y archivos. Los usuarios se autentican al DFS utilizando un método de desafíos entre el cliente y el servidor que prueban la identidad del cliente.

Plataformas soportadas

AFS: Se encuentra disponible el módulo server y cliente para BSD UNIX, Mach, Linux y NetBSD; sólo el cliente para OS/2, HP Apollo Domain OS y Windows9x/NT.

CODA: Los módulos server y cliente están disponibles para BSD UNIX y Linux. El módulo cliente para Windows 9x.

NFS: Está disponible para todas las plataformas.

CIFS: Está disponible para Windows NT y UNIX, y sólo el módulo cliente para Windows 9x/2000.

6.2 Resumen

Se ha visto cómo cada DFS tiene sus ventajas y desventajas comparado con los demás. La mayor parte de las virtudes y falencias están dadas por los objetivos perseguidos en el momento del diseño y el momento en que fueron desarrollados. Así es como CODA, cuyo objetivo principal fue el acceso permanente a los datos, provee nativamente replicación de lectura/escritura, cacheo de archivos completos y protocolos de conectividad adaptables. En cambio NFS fue concebido fuertemente tolerante a fallas y para este fin cuenta con su propiedad stateless en detrimento de otras características como ser el mantenimiento de la coherencia de caches entre los clientes.

Sin embargo, ninguno de ellos está congelado sino que se siguen haciendo mejoras a medida que se desarrollan nuevas versiones.

CAPITULO 7

Conclusiones

Esta tesis contribuye a comprender la problemática de los DFSs. Aporta una comparativa entre los distintos DFSs estudiados en el marco teórico basada en la caracterización a través de las transparencias. Muestra una posible forma de resolver algunos de los problemas presentados en el análisis, realizando una interfase entre file systems heterogéneos utilizando como modelo de referencia NFS y CODA. También contribuye con la portabilidad de CODA a sistemas operativos donde es difícil modificar el kernel para dar soporte a Venus.

A partir del uso de la interfase como nexo entre NFS y CODA, surge un "nuevo" DFS, con una semántica propia, que no responde en su totalidad a ninguno de los DFSs vinculados. Esto probablemente sucederá toda vez que se conecten dos file systems de distinto tipo utilizando algún gateway diseñado para tal fin.

Si bien la vinculación de los dos file systems parece una empresa muy complicada, en realidad se resuelve paso a paso con el estudio del funcionamiento de cada uno de los system calls en cada uno de los file systems. La base para la conexión entre los system calls es descubrir el manejo interno de los descriptores (file handles) de cada DFS, así convertir y encapsular el file handle de CODA en el de NFS y al retornar el system call poder devolver estructuras de datos válidas para NFS .

7.1 Proyección de esta tesis

En el futuro las workstations se convertirán en network computers (NC) [6], y necesitarán conectarse con servers remotos con diversos tipos de file systems, existentes hoy en día o resultado de futuros desarrollos. En este escenario, las NC deberán ser lo suficientemente versátiles para conectarse a los distintos file systems.

Como hemos visto en esta tesis, es posible realizar una interfase entre dos file systems heterogéneos. Sin embargo, si las NCs "hablaran" un dialecto universal, llamémoslo OFSC (Open File System Connection), entonces los desarrolladores de file systems sólo tendrían que proveer un driver o interfase entre OFSC y su DFS propietario. Este concepto es equivalente al planteado por el modelo OSI, que separa las redes en capas que ofrecen servicios. Otro ejemplo es el visto en ODBC (Open DataBase Connection), donde se estandariza el acceso a cualquier base de datos sin importar su estructura o implementación.

Habrá que realizar una especificación de OFSC, indicando los servicios disponibles para ser ofrecidos por los servers, como por ejemplo operación desconectada y el comportamiento ante conflictos de reintegración o faltas de archivos en cache producidos por desconexión.

Trazando una analogía entre esta propuesta y nuestra tesis, el OFSC se correspondería con el NFS, el DFS propietario con CODA y el driver con la interfase NFS-CODA.

7.2 Trabajos futuros

7.2.1 Resolución de conflictos

Actualmente, existen varias circunstancias en las cuales el cliente NFS podrá recibir un error mientras está utilizando un archivo o en el momento de cerrarlo, debido a conflictos producidos por inconsistencias de reintegración.

Una posible solución podría ser que la interfase mantuviese los archivos abiertos hasta que no existan clientes que lo tengan abierto. Sin embargo, esta política iría en contra de la naturaleza stateless de NFS. No obstante, se podría customizar el tiempo en el que un archivo se mantendría abierto para minimizar estas situaciones de conflicto.

7.2.2 Archivos especiales

Estudio de la factibilidad, necesidad y utilidad del manejo de archivos especiales (pipes, fifos, devices, etc.).

Aplicaciones que residan completamente en file systems remotos pueden llegar a tener la necesidad de comunicarse a través de un archivo especial con nombre. Actualmente, debería definirse previamente en el file system local. Si el archivo especial fuese parte de la aplicación en el directorio remoto se simplificaría la ejecución de la aplicación que de otra manera necesitaría de algún tipo de instalación especial en el file system local.

7.2.3 Aspectos relacionados con la seguridad

Es necesario resolver aspectos relacionados con la seguridad, principalmente la autenticación inicial del cliente, la cual actualmente (por el uso de NFS 2) se realiza por el archivo */etc/exports*.

Se debería poder utilizar otros mecanismos de seguridad mucho más modernos como Kerberos, que ya utiliza CODA y NFS versión 3.

7.2.4 Concurrencia en los pedidos al gateway

Actualmente todos los pedidos al gateway se serializan resultando imposible atender simultáneamente más de un pedido. Una manera de sobrellevar esta restricción, sería forkear un proceso que quede esperando por la respuesta del cliente CODA y la remita al NFS server que la solicitó.

7.2.5 Acoplamiento de Venus con NFS

Para el elminar el overhead en cada system call, que se produce en las llamadas a través de los sockets UDP y cambios de contexto, se podría integrar las rutinas de la interfase como un thread de Venus. Así, Venus se podría convertir en un server NFS.

APENDICE A

Garantías de Sesión para Datos Replicados con Consistencia Débil

Los sistemas con manejo de datos replicados con consistencia débil se caracterizan por la propagación perezosa de las actualizaciones y la posibilidad de que los clientes vean valores inconsistentes cuando leen datos de diferentes réplicas. Los sistemas de consistencia débil son populares debido a su alta disponibilidad, gran escalabilidad y simpleza de diseño. Estas ventajas surgen de la posibilidad de que las lecturas y escrituras se realicen sin ninguna o muy poca sincronización entre las réplicas. Más recientemente el uso de datos replicados con consistencia débil se necesitó por el acceso móvil a aplicaciones, como así también por la presencia de enlaces de comunicación caros o muy lentos, los cuales dificultan el mantener las copias de datos sincronizados instantáneamente.

Lamentablemente, la falta de garantías respecto al orden de las lecturas y escrituras en un sistema de consistencia débil, puede confundir a los usuarios y a las aplicaciones. Un usuario puede leer un valor para un ítem de datos y momentos después leer un valor más antiguo. Similarmente puede actualizar datos basados en la lectura de otro dato, mientras que otros usuarios pueden ver esta última actualización sin tener acceso a los datos originarios.

Un problema serio con los sistemas de consistencia débil es que pueden aparecer inconsistencias aún cuando sólo un usuario o aplicación esté realizando modificaciones en los datos. Por ejemplo, una aplicación puede grabar un ítem sobre un server y momentos después leer el mismo ítem de otro server con diferentes datos. El cliente verá resultados inconsistentes a menos que los servers se hubieren sincronizado entre las dos operaciones.

Una propuesta para aliviar la problemática de los sistemas de consistencia débil, manteniendo las ventajas fundamentales de leer/escribir cualquier réplica de cualquier server (read-any/write-any), es la llamada *garantías de sesión*. Una *sesión* es una abstracción de una secuencia de lecturas y escrituras realizadas durante la ejecución de una aplicación. La intención de una sesión es la de presentar a una aplicación individual, una base de datos que sea consistente con sus propias acciones, aunque esta aplicación lea y escriba desde varios servers distintos potencialmente inconsistentes. Se desea que el resultado de las operaciones realizadas en una sesión sea consistente con el modelo de un único server centralizado, que puede ser actualizado y leído concurrentemente por múltiples clientes.

Para lograr este objetivo, se proponen cuatro garantías que pueden ser aplicadas independientemente a las operaciones pertenecientes a la sesión:

Leer las escrituras - las lecturas reflejan las escrituras previas.

Lecturas monotónicas - lecturas sucesivas reflejan un conjunto no decreciente de escrituras.

Escrituras después de lecturas - las escrituras se propagan luego de las lecturas de las cuales dependen.

Escrituras monotónicas - las escrituras se propagan en el orden en que fueron ejecutadas.

Estas propiedades se "garantizan" en el sentido que, o bien el sistema de almacenamiento las asegura por cada operación de lectura y escritura perteneciente a la sesión, o informa a la aplicación que la garantía no puede ser satisfecha. Para asegurar que las garantías serán satisfechas, los servers en los cuales se puede realizar una operación deben restringirse a un subconjunto de servers disponibles, que estén suficientemente actualizados.

Como el otorgar garantías restringe el conjunto de servers que pueden ser usados dentro de una sesión, entonces requerir una garantía impacta en forma adversa en la disponibilidad.

Las aplicaciones deben balancear disponibilidad y consistencia. Por esta razón las garantías son pedidas individualmente por sesión. Los pedidos de garantías hechos dentro de una sesión, no tienen ningún efecto en la disponibilidad vista por aplicaciones que están usando otras sesiones u aplicaciones que no piden garantías.

Modelo de almacenamiento subyacente y terminología

Se asume la existencia de un sistema de almacenamiento replicado débilmente consistente, al cual se le agregan garantías. Este sistema consiste en un conjunto de servers, cada uno de los cuales alberga una copia completa de alguna *base de datos* replicada, y *clientes* que ejecutan aplicaciones que acceden a la base de datos. El término "base de datos" no significa ningún tipo de modelo u organización de datos particular. Una base de datos es simplemente un conjunto de *ítems de datos*, donde ítem de datos puede ser cualquier cosa, desde un archivo convencional a una tupla en una base de datos relacional.

Las dos operaciones de lectura y escritura, representarán consultas y actualización a la base de datos respectivamente. Éste es el modelo de almacenamiento subyacente básico para poder definir formalmente estas cuatro garantías sin pérdida de generalidad.

[3]

APENDICE B

Instalación del software

Los pasos que realizamos son:

- Instalación del sistema operativo Linux, versión 5.1 (Manhattan) de RedHat, kernel 2.0.34.
- Instalación de los fuentes de CODA, cliente y servidor, versión 4.6.5 (ftp:ftp.coda.cs.cmu.edu). Los fuentes están empaquetados con el utilitario rpm. El archivo se denomina coda-4.6.5.rpm. Se instala en el directorio /usr/src/redhat/BUILD/coda-4.6.5/.
- Compilado de los fuentes de CODA mediante los siguientes comandos:

  $ ./configure; make coda

  Se generan los ejecutables del cliente y el server CODA.

  $ cd /usr/src/redhat/BUILD/coda-4.6.5/kernel-src/vfs/linux
  $ make coda

  Se genera el módulo coda.o que es la porción del cliente CODA en el kernel.

  $ make oldconfig; make dep

  Reconfigura el kernel para agregar el soporte para CODA.

  $ cd /usr/src/redhat/BUILD/coda-4.6.5/
  $ make client-install
  $ make server-install

  Estos dos comandos instalan en el directorio /etc/rc.d/init.d/ los scripts para levantar en forma automática el cliente y server CODA en el nivel de init 2.

  $ cd /usr/src/redhat/BUILD/coda-4.6.5/utils-src/
  $ make coda

  Genera el ejecutable **potemkin** en el directorio ./potemkin/potemkin/. Este es un proceso que se ejecuta a nivel de usuario y cumple la función de Venus y Vice (server CODA). Simula ser un servidor remoto pero sirve archivos desde la máquina local. Lo utilizamos para realizar el desarrollo de la interfase sin tener la necesidad de ejecutar el verdadero server en otra máquina. Comsume menos recursos y es fácil de estudiar y debuguear.

- Para probar que la instalación fue satisfactoria, ejecutamos potemkin de la siguiente forma:

```
$ insmod coda
$ potemkin –rd /tmp –v
```

Esto monta el directorio /tmp local en /coda, también local, obteniendo una versión reducida de CODA ejecutándose en la máquina.

- Instalación de los fuentes del server NFS, versión 2.2 beta. El archivo que usamos es el nfs-server-2.2beta16-7.src.rpm que se encuentra en las distribuciones de RedHat.
- Compilación del server NFS con los siguientes comandos:

```
$ cd /usr/src/redhat/SOURCES/nfs-server-2.2beta29
$ ./BUILD
```

- Instalación de un sistema de seguimiento de modificaciones de programas fuente. Decidimos instalar el RCS que viene en la distribución de RedHat. Lo utilizamos a lo largo del proyecto para poder identificar claramente las modificaciones introducidas por nosotros a los fuentes originales. El uso de los comandos está documentado en la sección el apéndice D.

Instalación de la interfase

- Instalación del gateway NFS-CODA:

  bajar al directorio /tmp el archivo "gnc.tar" provisto en el diskette

  ```
  $ tar xvf /tmp/gnc.tar
  ```

  Este comando baja las modificaciones del server NFS y del cliente CODA (Venus/potemkin).

- Compilar los fuentes mediante los siguientes comandos:

```
$ cd /usr/src/redhat/BUILD/coda-4.6.5/utils-src/
$ make coda

$ cd /usr/src/redhat/BUILD/coda-4.6.5/
$ make client-install

$ cd /usr/src/redhat/SOURCES/nfs-server-2.2beta29
$ make install
```

- Realizar las modificaciones a los archivos /etc/hosts, /etc/exports y /etc/services de acuerdo a lo indicado en el apéndice D.

APENDICE C

Código de las principales rutinas

```
/*      Routine to find out if the path corresponds to a CODA gateway
        it is only called from rpc.mountd at mount time
        It opens an UDP socket that needs to be explicitly closed after
        used by the calling routine.
*/
int isCodaFs(char * path)
{
        char * s;
        if (strcmp(path, "/") == 0)  return(0); /* only  "/" */
        if (*(strchr(path, '/') + 1) == '\0')
                /*  is it "?ended_in/" */
                return(0);
        s = strchr(path, '/') +1 ; /* whatever comes after  "/" */
        if ((CodaHost = gethostbyname(s)) == NULL) {
                printf("It is not a Coda Gateway.");
                fflush(stdout);
                return (0);
        } else {
                struct servent  *sp;
                int port = 0;
                printf("Coda Host address %s\n",
                        inet_ntoa(*(struct in_addr *) CodaHost->h_addr) );
                fflush(stdout);
                /* Get the default mount port */
                if (!(sp = getservbyname("codanfsmnt", "udp"))) {
                        port = CODAMNT_PORT;
                } else {
                        port = ntohs(sp->s_port);
                }
                SockOpen(port);
                return (1);
        }
}


/* This routine is called only once by rpc.nfsd daemon when it is detected that a fh corresponds to a CODA file
system
*/
void codanfs_SockOpen()
{
        if (KernFD < 0) {
                int port = 0;
                /* Get the default mount port */
                struct servent  *sp;
                if (!(sp = getservbyname("codanfsnfs", "udp"))) {
                        port = CODANFS_PORT;
                } else {
                        port = ntohs(sp->s_port);
                }
                SockOpen(port);
```

```
        }
}

void codanfs_SockClose()
{
        close(KernFD);
}


/*  The same socket is used to receive and to send messages */
/*  Here we get the CODA gateway port */

void SockOpen(int open_port)
{
   int  rc;
   struct sockaddr_in addr;

   KernFD = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   printf("Socket KernFD is %d\n", KernFD);
  /* Aqui va el bind de recepcion */
   addr.sin_family = AF_INET;
   addr.sin_addr.s_addr = htonl(0x7f000001);

   addr.sin_port = htons(open_port);
   rc = bind(KernFD, (struct sockaddr_in *)&addr, sizeof(addr));
   if ( rc != 0 ) {
          printf("Bind de recepcion returns %d\n", rc);
          exit(1);
   }
   else {
     struct servent  *sp;
          /* Get the default gateway port */
          if (!(sp = getservbyname("codanfsgtw", "udp"))) {
                  coda_gtw_port = CODAGTW_PORT;
          } else {
                  coda_gtw_port = ntohs(sp->s_port);
          }
   }
}
```

```
int coda_upcall(mntinfo, inSize, outSize, buffer)

struct coda_sb_info  *mntinfo;
int inSize;
int *outSize;
caddr_t buffer;
{
        union outputArgs                *out;
        int                             error = 0;
        int                             rc;
        char            inbuf[VC_MAXMSGSIZE];
        ViceFid                 *fp=NULL;

        rc = MsgWrite(buffer, inSize);
        MsgWait(0);
        rc = MsgRead(inbuf);
        out = (union outputArgs *) inbuf;

        memcpy(buffer, out, *outSize) ;
        error = out->oh.result;
        return (error);
}

extern int MsgWrite(char *buf, int size)
{
    struct sockaddr_in    addr;
    int                           cc;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(coda_gtw_port);
    addr.sin_addr.s_addr = htonl(0x7f000001);
    cc = sendto(KernFD, buf, size, 0, (struct sockaddr_in *) &addr,
                sizeof(addr));
        if (cc < 0)
printf("El MsgWrite dio un errno: %d, (%s)\n",errno, strerror(errno));
        return cc;
}

extern int MsgWait(rw)
int rw;
{
  int           rc;
  int           nfds;
  struct timeval    to;
  fd_set        readfds;
  fd_set        writefds;
  fd_set        exceptfds;

  while (1) {
    /* Set up arguments for the selects */
    nfds = KernFD+1;
    to.tv_sec = Interval;
    to.tv_usec = 0;
    /* Select on the kernel fid.  Wait at most Interval secs */
    FD_ZERO(&readfds);
    if (rw == 0) FD_SET(KernFD, &readfds);
```

```
        FD_ZERO(&writefds);
        if (rw == 1) FD_SET(KernFD, &writefds);
        FD_ZERO(&exceptfds);
              rc = select(nfds, &readfds, &writefds, &exceptfds, &to);
        if (rc == 0) {
           fprintf(stderr,"Interval - no message\n");
           fflush(stderr);
           continue;
        } else if (rc < 0) {
           if (errno == EINTR) {
           /* I think EINTR won't come back if we are ctrl-c'd.. */
#if 0
              fprintf(stderr,"We were interrupted\n");
              fflush(stderr);
              break;
#else
              continue;
#endif
           } else {
              perror("select'ing");
              exit(-1);
           }
            } /* si hubo error */
            break;
     } /* end while (1) */

     /* Read what we can... */
     return rc;
}

extern int MsgRead(char *m)
{
           struct sockaddr_in          addr;
           int                         len = sizeof(addr);
           int                         rc ;
           union inputArgs             *in;
           union outputArgs            *out;

           in = (union inputArgs *) m;
           rc = 0;

           rc = recvfrom(KernFD, m, (int) (VC_MAXMSGSIZE),
                        0, (struct sockaddr_in *) &addr, &len);
           if (rc<0)
              printf("Error al recibir el msg: %d(%s)\n", errno,
          strerror(errno));
      return(rc);
}
```

```
/*
 * codavattr_to_statattr
 * Given a coda_vattr structure,  fill in it's stat structure
 *
 */
void codavattr_to_statattr(struct stat *sbuf, struct coda_vattr *vbuf)
{
   sbuf->st_dev=773; /*Our own magic number for a codanfs filesytem */
   sbuf->st_ino = vbuf->va_fileid;
   sbuf->st_mode = vbuf->va_mode;
   sbuf->st_nlink = vbuf->va_nlink;
   sbuf->st_uid = vbuf->va_uid;
   sbuf->st_gid = vbuf->va_gid;
   sbuf->st_rdev = 0;                    /* Can't have special devices in /coda */
   sbuf->st_size = vbuf->va_bytes;
   sbuf->st_blksize = vbuf->va_blocksize;
   /* compute the # of blocks */
   sbuf->st_blocks = (sbuf->st_size / 1024 );
   if (sbuf->st_size % 1024) sbuf->st_blocks++;
#ifndef __BSD44__
   sbuf->st_atime = vbuf->va_atime.tv_sec;
   sbuf->st_mtime = vbuf->va_mtime.tv_sec;
   sbuf->st_ctime = vbuf->va_ctime.tv_sec;
#else
   sbuf->st_atimespec = vbuf->va_atime;
   sbuf->st_mtimespec = vbuf->va_mtime;
   sbuf->st_ctimespec = vbuf->va_ctime;
   sbuf->st_flags = vbuf->va_flags;
#endif
}
```

```
/*
 * statattr_to_codavattr
 *
 * Given an iattr return a coda_vattr structure.
 */
void coda_iattr_to_vattr(sattr *iattr, struct coda_vattr *vattr)
{
  int mode;

  /* clean out */
  vattr->va_mode = (int) -1;
  vattr->va_uid = (vuid_t) -1;
  vattr->va_gid = (vgid_t) -1;
  vattr->va_size = (off_t) -1;
  vattr->va_atime.tv_sec = (time_t) -1;
  vattr->va_mtime.tv_sec  = (time_t) -1;
  vattr->va_ctime.tv_sec  = (time_t) -1;
  vattr->va_atime.tv_nsec =  (time_t) -1;
  vattr->va_mtime.tv_nsec = (time_t) -1;
  vattr->va_ctime.tv_nsec = (time_t) -1;
  vattr->va_type = C_VNON;
  vattr->va_fileid = (long)-1;
  vattr->va_gen = (long)-1;
  vattr->va_bytes = (long)-1;
  vattr->va_nlink = (short)-1;
  vattr->va_blocksize = (long)-1;
  vattr->va_rdev = (dev_t)-1;
  vattr->va_flags = 0;

  /* determine the type */

  /* CODA, Sorry, don't determine the type !!
         Potemkin seems not to know about remote types */
  mode = iattr->mode;
  /*
     if ( S_ISDIR(mode) ) {
          vattr->va_type = C_VDIR;
     } else if ( S_ISREG(mode) ) {
          vattr->va_type = C_VREG;
     } else if ( S_ISLNK(mode) ) {
          vattr->va_type = C_VLNK;
     } else {
          vattr->va_type = C_VNON;
     }
  */
  // For CODA the only type is C_VNON
  vattr->va_type = C_VNON;

  /* set those vattrs that need change */
  vattr->va_mode = iattr->mode;
  vattr->va_uid = (vuid_t) iattr->uid;
  vattr->va_gid = (vgid_t) iattr->gid;
  vattr->va_size = iattr->size;
  vattr->va_atime.tv_sec = iattr->atime.seconds;
  vattr->va_mtime.tv_sec = iattr->mtime.seconds;
}
```

```
int codanfs_rootfid(vicefid_t * codaFid)
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_rootfid( sbp, codaFid);
        free(sbp);
        return (codaerr);
}


int codanfs_getattr(vicefid_t * codaFid, struct coda_vattr * codaStat)
{
        struct super_block  * sbp;
        int codaerr;
        unsigned long sizesb ;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_getattr(sbp, codaFid, codaStat);
        free(sbp);
        return (codaerr);
}


/*
        It returns the name given a ViceFid (codaFid)
*/
int codanfs_open_by_path(vicefid_t * codaFid, int flags, const char *name)
{
        struct super_block  * sbp;
        unsigned long sizesb ;
        /* sizesb = sizeof(struct super_block); */
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        venus_open_by_path(sbp, codaFid, flags, name);
        free(sbp);
        return (name != NULL);
}

/*
It returns resFid (a ViceFid) given a ViceFid directory (codaFid)     and a name to find within that directory.
*/
int codanfs_lookup(vicefid_t * codaFid, const char *name, int *type, vicefid_t * resFid)
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_lookup(sbp, codaFid, name, strlen(name), type, resFid);
        free(sbp);
        return (codaerr);
}
```

```
/*
        It returns  a ViceFid of the newlly created name.
*/
int codanfs_create(vicefid_t * dirFid, const char *name, struct coda_vattr * attrs, int mode, vicefid_t *
resFid )
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        /* sizesb = sizeof(struct super_block); */
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        /* The O_EXCL flag is broken by NFS, so we don't care */
        codaerr = venus_create(sbp, dirFid, name, strlen(name), 0, mode,
                                                                        resFid, attrs);
        free(sbp);
        return (codaerr);
}


/*
        It returns  a status of the remove.
*/
int codanfs_remove(vicefid_t * dirFid, const char *name)
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        /* sizesb = sizeof(struct super_block); */
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_remove(sbp, dirFid, name, strlen(name));
        free(sbp);
        return (codaerr);
}

int codanfs_setattr(vicefid_t * codaFid, struct coda_vattr * codaStat)
{
        struct super_block  * sbp;
        unsigned long sizesb ;
        int codaerr;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);

        codaerr = venus_setattr(sbp, codaFid, codaStat);
        free(sbp);
        return (codaerr);
}
```

```
/*
        It returns  a ViceFid of the newlly created name.
        but we don't really need it.
*/

int codanfs_mkdir(vicefid_t * dirFid, const char *name,
                        struct coda_vattr *attrs, vicefid_t * resFid )
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_mkdir(sbp, dirFid, name, strlen(name),
                                                resFid, attrs);
        free(sbp);
        return (codaerr);
}


/*
        It returns  a status of the rmdir.
*/

int codanfs_rmdir(vicefid_t * dirFid, const char *name)
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb;
        sizesb = 256;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_rmdir(sbp, dirFid, name, strlen(name));
        free(sbp);
        return (codaerr);
}

/*
        It returns  a status of the rename operation
        made by Venus.
*/
int codanfs_rename(vicefid_t *olddir_fid, vicefid_t *newdir_fid,
                                const char *old_name, const char *new_name)
{
        int codaerr;
        struct super_block  * sbp;
        unsigned long sizesb ;
        sizesb = 256 ;
        sbp = malloc((unsigned long) sizesb);
        codaerr = venus_rename(sbp, olddir_fid, newdir_fid,strlen(old_name), strlen(new_name),
                                old_name, new_name);
        free(sbp);
        return (codaerr);
}
```

APENDICE D

Archivos de Ejemplo

/etc/exports en la máquina gateway

```
# El directorio /host.coda.com debe estar definido
# para poder montarlo desde NFS.
# También debe existir como directorio en la
# máquina gateway NFS-CODA.

/nfsdir                 (rw,insecure,all_squash)
/host.coda.com          (rw,insecure,all_squash)
/                       (rw,insecure,all_squash)
```

/etc/hosts en la máquina gateway

```
#
# También se puede utilizar DNS para resolver los nombres
#
127.0.0.1               localhost localhost.localdomain
192.168.1.1             gateway.codanfs.com
192.168.1.99            host.coda.com
```

/etc/services en la máquina gateway

```
#
# services  This file describes the various services that are
#           available from the TCP/IP subsystem.  It should be
#           consulted instead of using the numbers in the ARPA
#           include files, or, worse, just guessing them.
#
# Version:  @(#)/etc/services 2.00  04/30/93
#
# Author:   Fred N. van Kempen, <waltje@uwalt.nl.mugnet.org>
#

... aquí va el /etc/services standard ...

# End of services.
linuxconf   98/tcp            # added by linuxconf RPM
#
# Estas son las extensiones del /etc/services para CODA
#
# Iana allocated Coda filesystem port numbers
rpc2portmap     369/tcp
rpc2portmap     369/udp     # Coda portmapper
codaauth2       370/tcp
```

```
codaauth2       370/udp    # Coda authentication server

venus           2430/tcp   # codacon port
venus           2430/udp   # Venus callback/wbc interface
venus-se        2431/tcp   # tcp side effects
venus-se        2431/udp   # udp sftp side effect
codasrv         2432/tcp   # not used
codasrv         2432/udp   # server port
codasrv-se      2433/tcp   # tcp side effects
codasrv-se      2433/udp   # udp sftp side effect
#
# El mountd y el nfsd se deben utilizar en el mismo port
# con la salvedad que si se requiere desmontar y volver
# a montar un file system CODA, es necesario:
# 1-desmontar el file system CODA
# 2-nfs stop y un nfs start
# 3-montar el file system CODA nuevamente
#
codanfsgtw      8000/udp   # NFS to Coda gateway
codanfsmnt      8001/udp   # NFS mountd to Coda gateway
codanfsnfs      8001/udp   # NFS nfsd to Coda gateway
```

RCS

```
CI(1)                                                    CI(1)

NAME
       ci - check in RCS revisions

SYNOPSIS
       ci [options] file ...

DESCRIPTION
       ci  stores  new  revisions  into RCS files.  Each pathname
       matching an RCS suffix is taken to be an  RCS  file.   All
       others  are  assumed  to  be  working files containing new
       revisions.  ci deposits the contents of each working  file
       into  the  corresponding RCS file.  If only a working file
       is given, ci tries to find the corresponding RCS  file  in
       an  RCS subdirectory and then in the working file's direc-
       tory.  For more details, see FILE NAMING below.

CO(1)                                                    CO(1)

NAME
       co - check out RCS revisions

SYNOPSIS
       co [options] file ...

DESCRIPTION
       co  retrieves  a revision from each RCS file and stores it
       into the corresponding working file.

       Pathnames matching an RCS suffix  denote  RCS  files;  all
```

others denote working files. Names are paired as explained in ci(1).

Revisions of an RCS file can be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Checkout with locking fails if the revision to be checked out is currently locked by another user. (A lock can be broken with rcs(1).) Checkout with locking also requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. Checkout without locking is not subject to accesslist restrictions, and is not affected by the presence of locks.

RCSDIFF(1) RCSDIFF(1)

NAME

rcsdiff - compare RCS revisions

SYNOPSIS

rcsdiff [-ksubst] [-q] [-rrev1 [-rrev2]] [-T] [-V[n]] [-xsuffixes] [-zzone] [diff options] file ...

DESCRIPTION

rcsdiff runs diff(1) to compare two revisions of each RCS file given.

Pathnames matching an RCS suffix denote RCS files; all others denote working files. Names are paired as explained in ci(1).

getservbyname

```
GETSERVENT(3)        Linux Programmer's Manual        GETSERVENT(3)

NAME
       getservent, getservbyname, getservbyport, setservent, end-
       servent - get service entry

SYNOPSIS
       #include <netdb.h>

       struct servent *getservent(void);

       struct servent *getservbyname(const char *name, const char *proto);

       struct servent *getservbyport(int port, const char *proto);

       void setservent(int stayopen);

       void endservent(void);

DESCRIPTION
       The getservent() function reads the next line from the
       file /etc/services and returns a structure servent con-
       taining the broken out fields from the line.  The
       /etc/services file is opened if necessary.

       The getservbyname() function returns a servent structure
       for the line from /etc/services that matches the service
       name using protocol proto.
```

---

gesthostbyname

```
GETHOSTBYNAME(3)     Linux Programmer's Manual     GETHOSTBYNAME(3)

NAME
       gethostbyname, gethostbyaddr, sethostent, endhostent, her-
       ror - get network host entry

SYNOPSIS
       #include <netdb.h>
       extern int h_errno;

       struct hostent *gethostbyname(const char *name);

       #include <sys/socket.h>        /* for AF_INET */
       struct hostent *gethostbyaddr(const char *addr, int len, int type);

       void sethostent(int stayopen);

       void endhostent(void);
```

```
       void herror(const char *s);

DESCRIPTION
       The gethostbyname() function returns a structure  of  type
       hostent  for  the  given host name.  Here name is either a
       host name, or an IPv4 address in standard dot notation, or
       an IPv6 address in colon (and possibly dot) notation. (See
       RFC 1884 for the description of IPv6 addresses.)  If  name
       doesn't  end in a dot and the environment variable HOSTAL-
       IASES is set, the alias file  pointed  to  by  HOSTALIASES
       will first be searched for name.  (See hostname(7) for the
       file format.)  The current  domain  and  its  parents  are
       searched unless name ends in a dot.
```

BIBLIOGRAFIA

[1] Blackburn Paul.
AFS Frequently Asked Questions.
Faq.html, versión 1.31, 8 de Marzo, 1996.

[2] Colouris George, Dollimore Jean, Kindberg Tim
Distributed Systems - Concepts and Design, Second Edition 1996.

[3] Douglas Terry B., Demers Alan J., Petersen Karin, Spreitzer Mike, Theimer Marvin M., Welch Brent B.
Session Guarantees for Weakly Consistent Replicated Data.
Computer Science Laboratory, Xerox, Palo Alto Research Center, 1995.

[4] Heizer Isaac, Leach Paul, Perry Dan.
Internet-Draft. CIFS/1.0
Microsoft Corporation. Junio, 1996.

[5] IBM Corp.
DCE Enhanced Distributed File System for AIX.
Septiembre, 1996.

[6] Kumar Manu.
Today's computer a couple of years down the road....
SCS-MU, Mayo 1996.

[7] Leach Paul, Naik.
Internet-Draft. CIFS/1.0
Microsoft Corporation. Diciembre, 1997.

[8] LockHart Harold Jr.
DCE Frequently Asked Questions
Locus Computing Corporation, 23 de Agosto, 1997.

[9] Microsoft Corporation.
Microsoft Distributed File System – Administrator's Guide
Mayo, 1998.

[10] Microsoft Corporation.
Microsoft Distributed File System Version 4.1 for Microsoft Windows NT Server Version 4.0
Release Notes
Julio, 1997.

[11] Mummert Lily B.
Exploiting Weak Connectivity in a Distributed File System

CMU-CS-96-195, Diciembre 1996

[12] Noble Brian D., Satyanarayanan M.
A Research Status Report on Adaptation for Mobile Data Access.
DCS-CMU, 1995.

[13] rfc1094
Nowicki Bill
Sun Microsystems, Inc. March 1989.
Network Working Group.

[14] rfc1813
Network Working Group.
Callaghan, B. Pauloski, P.Staubach, Sun Microsystems, Inc. Junio 1995.

[15] Satyanarayanan M.
Autonomy or Independence in Distributed Systems?
DCS-CMU, 1989.

[16] Satyanarayanan M.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Transactions on Computers 39(4), Abril, 1990.

[17] Satyanarayanan M.
Fundamental Challenges in Mobile Computing.
DCS-CMU, 1997.

[18] Satyanarayanan M.
Mobile Information Access.
CMU-CS-96-107, Enero 1996.

[19] Satyanarayanan M., Ebling María R.,Raiff Joshua, Braam J. Peter.
Coda File System
User and System Administrators Manual
CMU v1.1, Agosto 1997

[20] Satyanarayanan M., Kistler James J., Mummert Lily B., Ebling María R., Kumar Puneet, Lu Qi.
Experience with disconnected operation in a mobile computing environment.
In Proceedings of the 1993 USENIX Simposium on Mobile and Location - Independent Computing.
Cambridge, MA, August 1993.

[21] Satyanarayanan M., Kistler James J., Siegel Ellen H.
Coda: A Resilient Distributed File System.
DCS-CMU, 1990.

[22] Sinniah Rymond Robert
An Introduction to de Andrew File System.
University of East London, Marzo, 1996.

[23] Spasojevic Mirjane, Satyanarayanan M.
A User Profile and Evaluation of a Wide Area Distributed File System.
Transarc Corp., CMU, Enero, 1994.

[24] Spasojevic Mirjane, Satyanarayanan M.
An Empirical Study of a Wide Area Distributed File System.
Transarc Corp., CMU, Noviembre, 1994.

[25] Supynuk Allen.
What is the Andrew File System (AFS)?
Ualberta, CNS, Diciembre, 1994.

[26] Tanenbaum Andrew S.
Computer Networks – Third Edition
Prentice Hall, Inc. 1996.

[27] Tanenbaum Andrew S.
Modern Operating Systems
Prentice Hall, Inc. 1992.

[28] Transarc Corporation.
AFS Beginners Guide.
Edición 1.0 , Febrero, 1994.

[29] Triantafillou Peter, Neilson Carl.
Achieving Strong Consistency in a Distributed File System.
IEEE Transactions on Software Engineering, Vol. 23, No.1:Jan, 1997 pp.35-55.

[30] Yadran Eterovic
Modelos de Computación Cliente-Servidor
UBA – FCEN - ECI 1994.

[31] URL: http://www.coda.cs.cmu.edu/ljpaper/lj.html