

## PARTE 4 – SISTEMA DE ARQUIVOS DISTRIBUÍDOS

### 16. Introdução

### 17. Projeto de Sistema de Arquivos Distribuído

#### 17.1 Interface do Serviço de Arquivos

#### 17.2 Interface do Serviço de Diretório

#### 17.3 Semântica de Compartilhamento de Arquivos

### 18. Implementação de Sistema de Arquivos Distribuído

#### 18.1 Utilização de arquivos

#### 18.2 Estrutura do sistema

#### 18.3 Caching

#### 18.4 Réplicas

### 19. Estudo de caso

#### 19.1 Sun NFS

## 16. Introdução

O Sistema de Arquivos (SA) é considerado componente chave de qualquer sistema distribuído, ele exerce o mesmo papel que nos sistemas centralizados, ou seja, armazenar programas e dados e disponibilizar os mesmos quando necessário. O componente SA é responsável pela organização, armazenagem, recuperação, nomeação, compartilhamento e proteção dos arquivos. Alguns aspectos dos SA's são diferentes em sistemas distribuídos. É importante em um SD diferenciar os conceitos de Serviço de Arquivo (Serviço) e Servidor de Arquivos (Servidor). O Serviço é a especificação do que é oferecido pelo SA para o usuário (cliente), primitivas disponíveis, parâmetros necessários e ações realizadas. O usuário vê o Serviço como uma definição do que é possível fazer. Entretanto, nada é definido em termos de como o serviço é implementado. O papel do Servidor, então, é implementar o Serviço. Ele é um processo que executa em alguma máquina, ou algumas. A princípio o usuário não sabe quantos destes processos existem nem tampouco qual a localização e função de cada um deles, basta que eles saibam que quando eles requisitam algum procedimento especificado pelo Serviço o mesmo é realizado e os resultados retornados. Considerando que o Servidor é um processo (usuário, ou kernel) executando em alguma máquina, é possível para o sistema contar com vários Servidores oferecendo diferentes Serviços (UNIX, MS-DOS).

Um serviço de arquivos distribuído, é usado para suportar o compartilhamento da armazenagem persistente e de informação. A funcionalidade e desempenho de um SA é crítico em um SD, o projeto do Serviço de arquivos deveria suportar os requisitos de transparência como:

- Acesso
- Localização
- Concorrência
- Falha
- Performance

Outras características e requisitos importantes que afetam a utilidade de um SAD:

- Hardware e SO heterogêneos
- Escalabilidade

Quando o requisito escalabilidade inclui redes com um grande numero de nodos ativos também são importantes os seguintes requisitos de transparência:

- Replicação
- Migração

Existem outras características, não encontradas em serviços de arquivo hoje, que serão importantes em aplicações distribuídas futuramente:

- Suporte a distribuição de dados de grão fino
- Tolerância a partição de redes

## 17. Projeto de Sistema de Arquivos Distribuído

Um SAD tem, tipicamente, dois componentes distintos: Serviço de Arquivos e Serviço de Diretório. O primeiro se preocupa com as operações em arquivos individuais: leitura, escrita, inclusão. O segundo preocupa-se com a criação e gerenciamento de diretórios, inclusão/remoção de arquivos em diretórios, etc.

### 17.1 Interface do Serviço de Arquivos

Em qualquer serviço de arquivos, o mais importante é: O que é um arquivo. Muitos sistemas como UNIX e MS-DOS, utilizam a abordagem de que um arquivo é uma sequência de bytes sem interpretação. O significado e a estrutura da informação é de inteira responsabilidade do programa de aplicação, o SO não tem interesse nisto. Entretanto, existem vários tipos de arquivos. Um arquivo pode ser estruturado como um sequência de registros, onde o SO é chamado para ler/escrever um registro em particular (acesso pela posição ou valor de campo). Arquivos podem tem atributos, que são informações sobre o arquivo: dono, tamanho, data de criação, permissões de acesso. Neste caso, o SAD fornece operações para leitura/escrita dos atributos (mudar direitos de acesso), em alguns sistemas é possível também criar e manipular atributos definidos pelo usuário.

Outro aspecto importante em relação ao modelo de arquivos é quando os mesmos podem ser modificados depois de criados. Geralmente, isto é regra, mas em alguns sistemas as únicas operações disponíveis para arquivos são CREATE e READ, ou seja, uma vez criado ele não pode mudar (imutável). Arquivos deste tipo suportam mais facilmente “caching” e replicação ( elimina problema de atualização das cópias).

As técnicas de proteção usadas em SD são essencialmente as mesmas usadas em sistemas com um único processador: capacidades e listas de controle de acesso. As capacidades funcionam como uma autorização (licença) que cada usuário tem para cada recurso (objeto) que ele tem acesso, definindo o tipo de acesso. No caso das listas de controle de acesso, existe uma associação entre arquivo e uma lista de usuários que tem acesso ao mesmo, especificando também a forma de acesso (Ex. UNIX).

Um serviço de arquivos pode ser classificado em 2 tipos dependendo de quando ele suporta o modelo “upload/download” ou o modelo acesso remoto. No primeiro caso, o serviço fornece duas operações: leitura e escrita de arquivo, sendo que a primeira transfere o arquivo inteiro do servidor para o cliente, enquanto que a outra faz a transferência no sentido contrário. Os arquivos podem ser armazenados em memória ou disco. No segundo caso, são fornecidas várias operações (open, close, read, write, lseek, etc.), fazendo com que todo o serviço seja realizado no servidor.

### 17.2 Interface do Serviço de Diretório

Este componente do SA fornece operações para criação e remoção de diretórios, nomeação de arquivos e movimentação dos mesmos de um diretório para outro. Este serviço não depende do tipo de transferência utilizado (arquivo, remoto).

Este serviço define um alfabeto e sintaxe para a composição de nomes de arquivos e diretórios. Tipicamente, nomes de arquivos são compostos por 1 -  $n$  letras, números e caracteres especiais, alguns contêm extensões que definem o tipo do arquivo (txt, c, etc). Em SD, diretórios podem conter sub diretórios que também podem conter outros sub diretórios. Esta estrutura em árvore recebe o nome de sistema de arquivos hierárquico. Em alguns sistemas é possível criar ligações para diretórios arbitrários, o que torna possível a criação de grafos, estrutura mais poderosa que árvore.

Uma característica importante em qualquer projeto de SAD é quando ou não todas as máquinas (processos) deveriam ter exatamente a mesma visão da hierarquia de diretório. Se por um lado visões diferentes (através de montagem remota) são flexíveis e simples de implementar, a desvantagem é que o sistema todo não se comporta como um sistema timesharing, fácil de programar e entender.

Outra questão é a existência ou não de um diretório raiz global, reconhecido por todas as máquinas como raiz (*server/path*). O problema principal, neste caso, é a transparência.

### Transparência de Nomes

Duas formas de transparência são relevantes. A primeira, transparência de localização, significa que o nome (caminho) não fornece nenhum sinal de onde o arquivo está localizado (*/server1/dir1/dir2/x*). A segunda, independência de localização, significa que arquivos podem ser movimentados sem que seus nomes tenham que mudar.

São 3 as abordagens comuns para nomear arquivos e diretórios em um SD:

1. Máquina + nome (caminho), */máquina/caminho* ou *máquina:caminho*
2. Montar arquivos remotos na hierarquia local
3. Um único espaço de nomes igual para todas as máquinas

### Nomeação em 2-níveis

Maioria dos sistemas distribuídos usam alguma forma de nomeação em 2-níveis. Arquivos tem **nomes simbólicos**, *prog.c*, usados pelos usuários, e também tem, **nomes binários**, que são internos para uso do sistema. Os diretórios fornecem um mapeamento entre o nome simbólico e o binário.

A natureza do nome binário pode variar de sistema para sistema. Em um sistema com múltiplos servidores, auto contidos, o nome binário pode ser o número do i-node, como no UNIX. Um esquema mais geral é o nome binário indicar o servidor e o arquivo no servidor, permitindo um diretório de um servidor manter arquivos em outro servidor. Alternativamente, é possível utilizar **ligação simbólica**, uma entrada de diretório contendo um string (servidor, arquivo), que pode ser procurado no servidor para encontrar o nome binário.

Outra idéia é utilizar capacidades como nomes binários, nomes ASCII são traduzidos em capacidades que podem ter formas diferentes: numero da máquina (físico ou lógico), endereço de rede do servidor, juntamente com identificador do arquivo. Em se tratando de SD a busca de um nome simbólico pode resultar em vários nomes binários.

## 17.3 Semântica de Compartilhamento de Arquivos

Quando 2 ou mais usuários compartilham o mesmo arquivo, é necessário definir de forma precisa a semântica de leitura e escrita para evitar problemas.

Em sistemas mono processador onde é possível processos compartilhar arquivos, UNIX, a semântica é: quando um READ acontece após um WRITE, a leitura retorna o valor escrito. De forma similar, quando 2 WRITES acontecem sucessivos, seguidos de um READ, o valor lido é o valor armazenado pela ultima escrita. Existe uma ordenação em tempo absoluto em todas operações. Esta é a **semântica UNIX**.

Um solução alternativa pode ser usada, permitindo o relaxamento da semântica de compartilhamento, ao invés de exigir que um READ veja os efeitos de todos os WRITES anteriores, poderíamos aceitar uma regra onde: “as mudanças em um arquivo aberto são inicialmente visíveis apenas para o processo que realiza as modificação. Apenas depois de fechar o arquivo as mudanças serão visíveis para outros processos.” Esta é conhecida como **semântica de sessão** e é muito usada.

Outra abordagem, completamente diferente para a semântica de compartilhamento de arquivos em SDs é fazer todos os **arquivos não modificáveis**. Neste caso, as únicas operações disponíveis são CREATE e READ, não é possível modificar um arquivo apenas substituir o anterior pelo novo, isto quer dizer que os diretórios são atualizados. Nesta abordagem, eliminamos o problema de como tratar com 2 processos onde um esta escrevendo e o outro lendo em um arquivo. Entretanto, temos que tratar com quando 2 processos tentam substituir o mesmo arquivo ao mesmo tempo. A melhor solução parece ser permitir que um dos arquivos novos seja o substituto, o ultimo ou qualquer.

Uma quarta abordagem para semântica do compartilhamento é usar **transações atômicas**. No acesso a um arquivo (grupo de ), um processo executa inicialmente uma operação de INICIO DE TRANSAÇÃO indicando que os comandos seguintes devem ser executados de forma indivisível. O processo então executa operações de leitura e escrita em um ou mais arquivos. No final, o processo executa uma operação do tipo FIM DE TRANSAÇÃO. Neste caso, o sistema garante que as chamadas durante a transação serão executadas em ordem, sem interferência.

## 18. Implementação de Sistema de Arquivos Distribuído

### 18.1 Utilização de arquivos

Antes de implementar qualquer sistema, é interessante que se tenha uma idéia de como ele será usado, como forma de assegurar que as operações mais executadas serão eficientes. Um estudo feito em relação ao padrão de uso de arquivos apresentou os seguintes resultados, que podem auxiliar no projeto de SAD.

1. Maioria dos arquivos tem tamanho menor que 10K, o que sugere a possibilidade de transferir arquivos completos, entre servidor e cliente, ao invés de blocos de disco. Uma vez que transferir é simples e mais eficiente, a idéia é válida. Algo deve ser feito com relação a arquivos grandes.
2. Maioria dos arquivos tem vida curta, o padrão é criar o arquivo, ler o mesmo e depois elimina-lo. Considere uma utilização típica de um compilador que cria arquivos

temporários para transmitir informações entre os seus passos. A conclusão é de que pode ser uma boa idéia criar o arquivo no cliente e mante-lo até a sua eliminação, eliminando tráfego entre cliente e servidor.

3. Compartilhamento de arquivos não é usual. Neste caso, mesmo com todos os problemas da cache, pode ser melhor fazer cache no cliente em função de um melhor desempenho.
4. Na média os processos usam poucos arquivos.
5. Existem classes diferentes de arquivos com propriedades diferentes. Isto pode sugerir a utilização de diferentes mecanismos para tratar as diferentes classes. Por exemplo, arquivos binários talvez pudessem ser replicados (quase não mudam). Arquivos temporários (pequenos, não compartilhados, desaparecem rápido) deveriam ser mantidos localmente. Arquivos de mensagens, arquivos ordinários.
6. Leitura é mais comum que escrita.
7. Leitura e escrita são sequenciais, acesso direto é pouco comum.

## 18.2 Estrutura do sistema

Como servidores, arquivo e diretório, são internamente organizados. A questão básica é : **Cientes e Servidores são diferentes?**

Em alguns sistemas não existe distinção entre cliente e servidor, todas as máquinas executam o mesmo software básico, assim qualquer máquina que quer oferecer serviço de arquivo pode fazer através da exportação de nomes de diretórios selecionados.

Outros sistemas, o servidor de arquivo e servidor de diretório são programas de usuário e assim um sistema pode ser configurado para executar software cliente e servidor na mesma máquina ou não. Por outro lado, estão sistemas nos quais clientes e servidores são máquinas diferentes, seja do ponto de vista de software ou de hardware. Os servidores podem, inclusive, executar em versão diferente de SO, comparado com o cliente.

Uma outra questão de implementação que diferencia os sistemas diz respeito a **como o serviço de arquivo e diretório é estruturado**. Uma organização é combinar os dois em um único servidor que trata todas as chamadas de arquivo e diretório. Outra possibilidade é manter os dois separados, sendo que para abrir um arquivo será necessário chamar o servidor de diretório (mapear nomes) e depois chamar o servidor de arquivo (read, write). O argumento em favor da separação é basicamente que são funções diferentes. O contra é que 2 servidores necessitam mais comunicação. Quando organizado como 2 servidores, o mapeamento do nome simbólico em nome binário pode envolver mais de um servidor se a hierarquia do SA está particionada. Neste caso, buscar nomes sempre pode ser caro. Alguns sistemas mantêm o resultado do mapeamento em cache.

Uma questão final relativa a implementação é quando ou não servidor de arquivo, de diretório e outros, deveriam **manter informação de estado dos clientes**. Existem basicamente duas escolas concorrentes neste caso. A primeira, servidores sem estado, implica em o cliente enviar uma requisição, o servidor atender a mesma, enviar resposta e retirar de suas tabelas internas qualquer informação sobre a requisição. Nenhuma informação específica do cliente é mantida no servidor. A outra, servidores com estado, implica em o servidor manter informação de estado dos clientes. Esta ultima argumenta que se sistemas operacionais centralizados mantêm informação de estado sobre os processos ativos, porque este comportamento agora é inaceitável.

### 18.3 Caching

Em um sistema cliente-servidor (ambos com memória e disco), é possível armazenar arquivos ou parte deles em 4 lugares: disco do servidor, memória do servidor, disco do cliente ou memória do cliente.

O lugar mais comum para armazenar todos os arquivos é no disco do servidor. Normalmente tem bastante espaço e os arquivos podem ser acessados por todos os clientes, com apenas uma cópia de cada arquivo não existe problemas de consistência. O problema neste caso é desempenho, leitura de um arquivo por um cliente implica cópia do disco do servidor para sua memória, sua transferência via rede para o cliente. É possível um ganho considerável em desempenho no uso de cache no servidor, para os arquivos mais recentemente usados, eliminando acesso a disco. Manter arquivos na cache significa determinar que partes do arquivo devem ficar na cache e, portanto, algoritmos para determinar isto. Os algoritmos tem que resolver 2 problemas.

Primeiro, qual é a unidade de gerenciamento? Pode ser arquivo inteiro ou bloco. Se arquivos inteiros são colocados na cache eles podem ser armazenados de forma contígua no disco (ou grandes pedaços), permitindo transferências entre memória e disco em alta velocidade. Cache de blocos, usam espaço de disco e a cache de forma mais eficiente.

Segundo, o algoritmo deve decidir o que fazer quando a cache enche e alguma unidade tem que sair. Geralmente, a política LRU é usada, quando alguém tem que sair o mais velho é escolhido. Apesar de cache no servidor ser fácil de implementar, totalmente transparente para os clientes e eliminar transferência de disco, ainda existe o acesso a rede. A forma de eliminar isto é fazer cache no lado cliente. A maioria dos sistemas que fazem cache no cliente fazem em memória principal e não disco.

Existem pelo menos 3 opções de onde colocar a cache do cliente. A forma mais simples é fazer cache de arquivos diretamente no espaço de endereço de cada processo, sendo a gerência da mesma provida por chamadas de biblioteca. A idéia é apenas manter os mais usados disponíveis. Quando o processo termina todos arquivos são escritos de volta no servidor. Uma segunda opção é colocar a cache no kernel. Neste caso, é necessário uma chamada de kernel em todos os acessos, porém a cache permanece mesmo depois de um processo terminar. A ultima opção é colocar a cache em um processo gerente de cache (nível de usuário), mantendo assim o kernel livre de código de SA e também permitindo maior flexibilidade e facilidade de programação. Páginas do gerente podem ser retiradas.

#### Consistência da Cache

Cache no cliente introduz inconsistência no sistema. Se 2 clientes fazem, simultaneamente, uma leitura no mesmo arquivo e depois modificam o mesmo, vários problemas ocorrem. Um deles, se um terceiro cliente faz uma leitura no arquivo ele vai receber a versão original. Podemos adotar a semântica de sessão para resolver este problema, mas se o usuário espera semântica UNIX, não vai funcionar.

Outro problema é quando os 2 clientes escrevem o arquivo de volta para o servidor, o escrito por ultimo sobrescreve o outro.

Uma forma de resolver o problema da consistência é usar o algoritmo **escrita imediata** : quando uma entrada da cache é modificada (write), o novo valor é imediatamente enviado para o servidor, além de ficar na cache. Desta forma, quando outro cliente ler o arquivo

perceberá o valor mais recente. O problema que ocorre aqui é o fato de que o arquivo permanece na cache e se outro cliente acessar o mesmo arquivo não precisará busca-lo no servidor. Uma forma de resolver é o gerente de cache verificar com o servidor antes de permitir acesso. Outro problema deste algoritmo é de que o tráfego na rede para as escritas continua o mesmo (embora auxilie nas leituras). Assim sendo, ao invés de ir ao servidor no momento da escrita o cliente apenas anota a modificação. De tempos em tempos (30 s) todos os arquivos modificados são enviados (**retardo na escrita**). O retardo afeta a semântica, o que outro processo recebe em uma leitura depende do tempo.

Em busca da facilidade de programação podemos adotar a semântica de sessão e escrever um arquivo de volta ao servidor apenas depois de fechar. Este algoritmo é chamado de **escrita no fechamento**.

Uma abordagem completamente diferente é usar um algoritmo de **controle centralizado**, quando abrir um arquivo a máquina envia uma mensagem avisando para o servidor que mantém a informação (quem abriu, para que, etc). Desta forma é possível controlar a abertura de arquivos em função da operação a ser realizada e também a modificação dos mesmos. Quando um cliente tenta abrir um arquivo já aberto em algum lugar, a requisição pode ser negada ou colocada na fila. Uma outra alternativa, é o servidor enviar uma *mensagem não solicitada* para todos os clientes com o arquivo aberto dizendo para remover o mesmo da cache. Algumas variações do algoritmo de controle centralizado são possíveis, por exemplo, servidores podem manter informação dos arquivos em cache ao invés dos arquivos abertos. Todos os métodos centralizados tem um ponto único de falha e não tem escalabilidade em grandes sistemas.

Finalizando, cache no servidor é fácil de fazer e quase sempre vale a pena, independente de quando tem cache no cliente ou não. Cache no servidor não tem efeito na semântica do sistema vista pelo usuário. Cache no cliente, oferece melhor desempenho pelo preço de uma complexidade maior e possivelmente uma semântica mais confusa. Quando vale a pena fazer ou não depende de como o projetista pensa sobre desempenho, complexidade e facilidade de programação.

## 18.4 Replicação

SADs fornecem replicação de arquivos como um serviço para seus clientes, ou seja, múltiplas cópias de arquivos selecionados são mantidas, sendo cada cópia em um servidor. A razão para isto:

1. Aumentar a confiabilidade tendo cópias independentes de cada arquivo. Se um servidor cair, dados não são perdidos. Muitas aplicações precisam desta propriedade.
2. Permitir que acessos a arquivos sejam feitos mesmo que um servidor esteja fora.
3. Para repartir a carga de trabalho entre os servidores. Com o crescimento do sistema, manter todos os arquivos em um servidor pode representar gargalo.

Uma questão chave na replicação é a transparência. Em um extremo, os usuários podem estar avisados e inclusive controlar o processo. Por outro lado, os usuários não tem conhecimento algum, ou seja, o sistema faz tudo sem que eles saibam (transparência de replicação). A seguir são apresentadas 3 formas como a replicação pode ser realizada. Na primeira, o usuário (programador) controla o processo todo (**replicação explícita**). Um processo cria um arquivo em um servidor específico e depois ele pode fazer cópias



adicionais em outros servidores. O endereço de todas as cópias pode ficar no servidor de diretório. Quando o arquivo é aberto, a tentativa é feita de forma seqüencial até encontrar uma cópia disponível.

Uma abordagem alternativa é a chamada **replicação lazy**, uma cópia do arquivo é feita em um servidor, mais tarde o próprio servidor faz réplicas em outros servidores de forma automática, sem conhecimento do programador. O sistema deve ter inteligência para recuperar qualquer destas cópias se necessário. Importante quando se faz cópias desta forma é estar atento para a possibilidade de o arquivo mudar antes das cópias serem feitas. Por último, a **replicação usando grupo**, funciona da seguinte forma. Todas as chamadas WRITE são transmitidas simultaneamente para todos os servidores, sendo as cópias feitas ao mesmo tempo da original. Se comparado com o método anterior podemos apontar as seguintes diferenças: 1) *lazy* endereça um servidor não um grupo; 2) *lazy* acontece em “background” quando o servidor tem algum tempo livre.

### Protocolos de Atualização

Como os arquivos são atualizados? Apenas enviar uma mensagem de atualização para cada cópia em seqüência não é uma boa idéia pelo fato de que o processo atualizador pode quebrar e neste caso algumas cópias estariam atualizadas e outras não. Alguns algoritmos resolvem este problema.

O primeiro deles, **replicação cópia primária**, tem um servidor que é designado como primário e os outros são secundários. Quando um arquivo replicado é atualizado, a mudança é enviada para o servidor primário que realiza as atualizações e depois envia comandos para os secundários realizarem as mudanças. Leituras são feitas em qualquer cópia. As mudanças deveriam ser escritas em meio de armazenamento estável antes de mudar a cópia primária. Desta forma, se o servidor quebrar antes de sinalizar a modificação para todos os secundários, ele poderá, quando voltar, verificar a existência de uma atualização pendente e continuá-la. A desvantagem deste algoritmo é que se o primário esta fora não pode haver atualizações.

Um método mais robusto proposto por Gifford (1979) inclui **votação**, os clientes requisitam e conseguem permissão de múltiplos servidores antes de ler ou escrever em um arquivo replicado. Um exemplo simples do funcionamento do algoritmo com um arquivo replicado em N servidores. A regra para atualização de um arquivo pode ser que o cliente deve contatar pelo menos metade + 1 dos servidores e ter a concordância dos mesmos para fazer a atualização. Depois disto o arquivo é mudado e um novo número de versão é associado ao mesmo. Para leitura, o mesmo número de servidores deve ser contatado e o cliente pede a eles para mandar a versão do arquivo, se todos tem a mesma ela deve ser a mais recente. De uma forma geral isto quer dizer que para leitura de um arquivo com N réplicas o cliente necessita um *quorum de leitura*, ou seja, um número  $N_r$  de servidores. Da mesma forma, para modificação ele precisa de um *quorum de escrita* com  $N_w$  servidores. Estes valores estão sujeitos a seguinte restrição:  $N_r + N_w > N$ . Apenas depois de conseguir o número apropriado de servidores, uma leitura ou escrita pode ser realizada.

Uma variação deste método é chamado de **votação com fantasmas**. Na maioria das aplicações leituras são mais comuns que escritas sendo que  $N_r$  é um número menor que  $N_w$  que é perto de N. Isto significa que se alguns servidores estão fora, pode ser impossível conseguir o quorum. Assim sendo, o método cria um servidor *dummy*, sem armazenamento, para cada servidor real que esta fora. Um fantasma não é permitido em quorum de leitura, mas ele pode participar de um quorum de escrita. Uma escrita tem sucesso apenas se pelo

menos um servidor é real. Quando um servidor com falha volta, ele deve conseguir um quorum de leitura para localizar a versão mais recente, que é copiada para ele antes de iniciar operação normal. Da mesma forma que o método anterior, este segue a mesma propriedade da votação escolhendo Nr e Nw de forma que não é possível conseguir os 2 quorum ao mesmo tempo.

## 19. Estudo de caso

### 19.1 Sun NFS (Network File System)

A idéia básica do sistema de arquivos da Sun NFS é permitir uma coleção de clientes e servidores compartilhar um sistema de arquivos comum. Na maioria dos casos clientes e servidores estão na mesma LAN, mas é possível executar NFS em WAN.

Cada servidor NFS exporta um ou mais de seus diretórios para serem acessados por clientes remotos. A lista de diretórios que um servidor exporta é mantida em */etc/exports*, estes podem ser exportados automaticamente sempre que o servidor é carregado. Os clientes acessam os diretórios através da montagem dos mesmos. Quando um cliente monta um diretório, ele torna-se parte da sua hierarquia de diretório. Para os programas em execução no cliente não existe diferença entre um arquivo localizado no servidor remoto ou um no disco local.

O serviço de arquivos NFS é sem estado (stateless), baseado em cache de bloco e utiliza mecanismo de RPC com XDR para conseguir acesso remoto transparente para os usuários do sistema de arquivos.

#### Protocolos NSF

Sendo uma das metas do NFS suportar sistemas heterogeneos, é essencial que a interface entre clientes e servidores seja bem definida. Para conseguir isto, o NFS define 2 protocolos cliente/servidor: um protocolo para montagem do sistema remoto e um protocolo para acesso a diretório e arquivos.

No primeiro, um cliente pode enviar um caminho (nome) para o servidor e requisitar permissão para montar o diretório em algum lugar na sua hierarquia de diretório ( o lugar não é colocado na mensagem porque não interessa ao servidor). Se o nome é legal e o diretório especificado foi exportado, o servidor retorna um **tratador de arquivo** para o cliente. Este tratador contém campos que identificam ( unicamente ) o tipo de sistema de arquivo, o disco, o numero do i-node do diretório e informações de segurança. Outras chamadas ( read, write) usam este tratador. É possível configurar clientes para montar certos diretórios remotos através do arquivo */etc/rc*, que é um script shell contendo comandos de montagem remota e é executado automaticamente quando o cliente é iniciado. Outra forma é a auto montagem, permite que um conjunto de diretórios remotos sejam associados com o diretório local. Os diretórios não são montados quando o cliente é iniciado, e sim quando um arquivo remoto é aberto pela primeira vez, o SO envia uma mensagem para cada servidor o primeiro a responder terá seu diretório montado. Esta abordagem tem duas vantagens em relação a utilização do script: 1) se um dos servidores NFS citados em */etc/rc* está fora, é impossível iniciar o cliente (retardo, erros, dificuldades), 2) permitir o cliente tentar um conjunto de servidores em paralelo significa

conseguir um grau de tolerância a faltas (é preciso apenas 1) e melhorar o desempenho (o primeiro é escolhido). Entretanto, todas alternativas especificadas devem ser idênticas. Sendo que o NFS não tem suporte a replicação, quem deve garantir isto é o usuário (mais usado para arquivos executáveis, binários).

No segundo protocolo, clientes podem enviar mensagens para os servidores para manipular diretórios e ler e escrever arquivos (também é possível acessar atributos do arquivo: modo, tamanho, tempo da última modificação, etc.). Quase todas as chamadas UNIX são suportadas pelo NFS (exceção para OPEN e CLOSE). Não é necessário abrir arquivo antes de ler, o cliente envia para o servidor uma mensagem contendo o nome do arquivo, junto com uma requisição para encontrar o mesmo e retornar um tratador (estrutura que identifica o arquivo). A operação de busca não copia qualquer informação para tabelas internas do sistema. A operação READ contém o tratador, o deslocamento para o início da leitura e o número de bytes a ler. Cada mensagem destas é auto contida. Este método usado pelo NFS torna difícil conseguir a semântica UNIX.

O NFS utiliza o mecanismo de proteção do UNIX, mensagens de requisição contém ID do usuário e do grupo do chamador, usados para o servidor NFS validar o acesso. Atualmente, é possível usar criptografia com chave pública para estabelecer uma chave segura para validação do cliente e servidor em cada requisição e resposta. Assim, não é possível um cliente malicioso personalizar outro porque ele não conhece a chave. Criptografia é usada apenas na autenticação não para transmissão dos dados.

As chaves e outras informações para autenticação são mantidas pelo NIS (Network Information Service), conhecido como páginas amarelas, cuja função é armazenar pares (chave, valor). Ele mantém chaves, armazena mapeamento de nomes de usuários para senhas (criptografadas), mapeamento de nomes de máquinas para endereços de rede e outras informações. Estes servidores são replicados usando um arranjo mestre/escravo, para ler dados um processo pode usar o mestre ou qualquer uma das cópias. As mudanças devem ser feitas no mestre (propaga para os outros dentro de um certo tempo).

### Implementação NSF

O NSF é composto por 3 níveis. No topo está o nível de chamada de sistema, tratando de chamadas como OPEN, READ e CLOSE. Depois de identificar a chamada e verificar os parâmetros, ele chama o segundo nível, sistema de arquivo virtual (VFS). A tarefa deste é manter uma tabela com uma entrada para cada arquivo aberto (UNIX). O nível VFS tem uma entrada, chamada **v-node**, para cada arquivo aberto. Estes são usados para dizer quando o arquivo é local ou remoto. Vamos verificar a utilização dos v-nodes acompanhando uma sequência de chamadas de sistema MOUNT, OPEN e READ. A montagem de um sistema de arquivos remoto é feita através do programa *mount* especificando o diretório remoto e o diretório local onde deve ser montado, ele identifica o diretório remoto e descobre o nome da máquina onde ele está, contata a mesma pedindo um tratador para o diretório. Se o diretório existe e está disponível para montagem remota, o servidor devolve um tratador de arquivo para o diretório. Por último, executa a chamada de sistema MOUNT, passando o tratador para o kernel que constrói um **v-node** para o diretório remoto e pede ao cliente NFS para criar um **r-node** nas suas tabelas internas para manter o tratador do arquivo ( **v-node** aponta **r-node** ). Cada **v-node** no nível VFS contém um apontador para um **r-node** ( no NFS cliente) ou para um **i-node** ( no SO local). A partir do **v-node** pode-se saber se o arquivo ou diretório é local ou remoto (encontrar tratador).

Quando um arquivo remoto é aberto, o kernel encontra o diretório no qual o sistema remoto esta montado verifica que o mesmo é remoto e no seu **v-node** encontra o apontador para o **r-node**. Então ele pede para o NFS cliente abrir o arquivo, o mesmo procura a parte do nome no servidor remoto e retorna um tratador para o arquivo. O NFS faz um **r-node** para o arquivo remoto nas suas tabelas e volta para o VFS que coloca um **v-node**, nas suas tabelas, apontando para o r-node. O chamador recebe um descritor de arquivo, para o arquivo remoto, que é mapeado em um v-node pelas tabelas no nível VFS. Quando o descritor é usado, para um READ, o VFS localiza o v-node correspondente e a partir dele determina quando ele é local ou remoto (qual i-node ou r-node).

As transferências entre cliente e servidor são feitas em grandes volumes, 8192 bytes, por questões de eficiência. Na leitura, depois do cliente VFS receber 8K, ele automaticamente emite um requisição para outros 8K, assim se precisar já esta disponível ( read ahead ). Na escrita, se um WRITE tem menos que 8K os dados são acumulados localmente e enviados, no fechamento do arquivo os dados são enviados imediatamente.

Também para melhorar o desempenho é usado cache nos servidores, para evitar acessos a disco. Clientes mantém 2 caches, uma para atributos de arquivo (i-nodes) e um para dados do arquivo. Sempre que estes são necessários, é verificado se a cache pode atender. Como visto a utilização de cache traz problemas. O NFS faz várias coisas para amenizar o problema: 1) associado com cada bloco de cache existe um timer, quando o mesmo expira a entrada é descartada. O timer é 3s para blocos de dados e 30s para blocos de diretório; 2) quando um arquivo que esta em cache é aberto, uma mensagem é enviada para o servidor para saber quando o arquivo foi modificado pela ultima vez ( se depois, nova cópia); 3) a cada 30s todos os blocos modificados são enviados para o servidor. Mesmo assim, a semântica UNIX não é implementada. Ainda mais, quando um arquivo é criado ela pode não ser visível para fora por até 30s. O funcionamento do NFS depende do tempo.