UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Reintegração de Servidores em Sistemas Distribuídos

por

MARCIA PASIN

Dissertação submetida à avaliação como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Profa. Taisy Silva Weber
Orientadora

Porto Alegre, junho de 1998.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Pasin, Marcia

Reintegração de Servidores em Sistemas Distribuídos / por Marcia Pasin. - Porto Alegre : CPGCC da UFRGS, 1998.
77f. : il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1998. Orientadora: Weber, Taisy Silva.

1. Reintegracao de servidor. 2. Sistemas operacionais distribuidos. 3. Sistemas de arquivos distribuidos. 4. Tolerancia a falhas. I. Weber, Taisy Silva. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenadora do CPGCC: Profa. Carla Maria Dal Sasso Freitas
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Agradeço à professora Dra. Taisy Silva Weber pela atenciosa orientação; ao professor Dr. Felipe Martins Müller pelo apoio em minha introdução à atividade científica; as amizades que fiz na UFRGS e as amizades da UFSM, onde fiz minha graduação, que foram preservadas durante o curso de mestrado; aos meus pais - Neuton e Lucia - pelo incentivo e apoio constante; aos meus irmãos - Marcelo e Marta; a todos os colegas do Grupo de Tolerância a Falhas; a todos os profissionais do Instituto de Informática que de alguma forma colaboraram para a realização deste trabalho; ao Instituto de Informática da UFRGS e ao CNPq pelo auxílio à realização desta pesquisa.

Sumário

Lista de Abreviaturas

CPU - *Central Processing Unit*
FIFO - *First In First Out*
HA-NFS - *High Availability Network File System*
NFS - *Network File System*
PAS - Protocolo de Atualização do Servidor
RNFS - *Reliable Network File System*
RPC - *Remote Procedure Call*
UDP/IP - *User Datagram Protocol / Internet Protocol*
VFS - *Virtual File System*

Lista de Figuras

Lista de Tabelas

Resumo

Sistemas distribuídos representam uma plataforma ideal para implementação de sistemas computacionais com alta confiabilidade e disponibilidade devido à redundância fornecida por um grande número de estações interligadas. Falhas de um servidor podem ser contornadas pela reconfiguração do sistema. Entretanto falhas em seqüência que afetem múltiplas estações comprometem não apenas o desempenho do sistema, mas também a continuidade do serviço e sua confiabilidade. Assim, servidores falhos, que tenham sido isolados do sistema, devem ser reintegrados tão logo quanto possível para não comprometer a disponibilidade do sistema computacional. Este trabalho trata da atualização do estado de servidores e da troca de informação que o servidor recuperado realiza para integrar-se aos demais membros do sistema através de um procedimento chamado reintegração do servidor. É assumido um ambiente distribuído que garante alta confiabilidade em aplicações convencionais através da técnica de replicação de arquivos. O servidor a ser reintegrado faz parte de um grupo de replicação e volta a participar ativamente do grupo tão logo seja reintegrado. Para tanto, considera-se a estratégia de replicação por cópia primária e um sistema distribuído experimental, compatível com o NFS, desenvolvido na UFRGS para aplicar a reintegração de servidor. Os métodos de atualização de arquivos para a reintegração do servidor foram implementadas no ambiente UNIX.

Palavras-chave: sistemas de arquivos distribuídos, tolerância a falhas, reintegração de servidores.

TITLE: "REINTEGRATION OF FAILED SERVER IN DISTRIBUTED SYSTEMS".

Abstract

Distributed systems are an ideal platform to develop high reliable computer applications due to the redundancy supplied by a great number of interconnected workstations. Failed stations can be masked reconfiguring the system. However, sequential faults, that affect multiple stations, not just decrease the performance of the system, but also affect the continuity of the service and its reliability. Thus, failed stations working as servers, that have been isolated from the system, should be reintegrated as soon as possible to not impair the system availability. This work is exactly about methods to update the state of failed servers. It deals also with the change of information that the recovered server accomplishes to be integrated to the other members of the service group through a process called reintegration of server. It is assumed a distributed environment that guarantees high reliability in conventional applications through replication of files. The server to be reintegrated is part of a replication group and it participates actively of the service group again as soon as it is reintegrated. Our approach is based on a primary copy. The file actualization methods to the reintegration of server were implemented in an UNIX environment. To illustrate our approach we will describe how the integration of repaired server can be made a fault-tolerant system. The experimental distributed system, compatible with NFS, was designed at the UFRGS.

Keywords: distributed file systems, fault tolerance, reintegration of failed server.

1 Introdução

A redundância é o mecanismo que duplica ou aumenta a quantidade de componentes em um sistema distribuído para tolerar falhas. Possibilita que um componente falho tenha seu comportamento mascarado pelos componentes redundantes de *software* ou *hardware*. A redundância em *software*, especialmente em sistemas de arquivos distribuídos, pode ser obtida através da replicação de arquivos nos servidores.

1.1 Replicação de arquivos

A replicação de arquivos consiste em disseminar cópias de arquivos entre os vários servidores de um sistema distribuído. Assim, se uma cópia é perdida acidentalmente, há outra para substituí-la. Em sistemas de arquivos distribuídos, isso implica em aproveitar a distribuição natural dos arquivos em diferentes servidores para tolerar falhas.

A replicação introduz a seus próprios problemas como a manutenção das diferentes cópias quando há uma operação de alteração requisitada por um cliente [PU88]. Problemas de inconsistência podem acontecer devido à concorrência de operações (quando dois ou mais clientes fazem diferentes requisições ao servidor replicado no mesmo instante) ou a falhas de *crash* [CRI86].

Para controlar o problema da consistência para operações concorrentes, o sistema deve garantir ordenação na execução das operações. Se um servidor replicado recebe duas operações x e y de diferentes clientes, todos os servidores que formam o servidor replicado devem executar as operações na mesma ordem. Para evitar inconsistência em caso de falhas de *crash*, o sistema deve realizar as operações atomicamente, isto é, o servidor replicado deve alterar todas as cópias ou não alterar nenhuma cópia (mantendo a versão antiga).

Para detectar falhas de *crash*, o sistema poderá utilizar o mecanismo de *timeout* (limite de tempo). Se o cliente não receber a resposta do serviço requisitado a um servidor até o *timeout* especificado, o cliente assume que o servidor falhou. O mecanismo de *timeout* é usado em um sistema síncrono, onde o limite de atrasos de mensagens é conhecido [JAL94]. Em contraste, um sistema assíncrono não possui qualquer limite para a transmissão das mensagens.

As propriedades de ordenação e de atomicidade, fazer parte da propriedade de linearidade [GUE97]. Para os clientes, o servidor replicado aparece como um único servidor. Isso é desejável sob o ponto de vista de programação: se um cliente precisa estabelecer comunicação com um servidor replicado, faz uma requisição a um servidor replicado da mesma forma que faria se o servidor fosse não replicado. O cliente não precisa de qualquer procedimento adicional para comunicar-se como o servidor replicado. Duas classes fundamentais de métodos de replicação garantem ordenação e atomicidade: cópia primária e réplicas ativas.

1.1.1 Cópia primária

Uma abordagem de distribuição de cópias em um sistema distribuído é a cópia primária [BUD93]. Nesta abordagem, um dos servidores contém a cópia primária e os demais são *backups*. O servidor que contém a cópia primária é chamado de servidor primário. Os servidores *backups*[1] também são chamados de servidores secundários.

Cada cliente sabe qual servidor é o primário e estabelece comunicação somente com este servidor. Os servidores secundários são apenas repositórios de dados. As requisições de leitura são recebidas e respondidas pelo servidor primário, sem consulta aos secundários. Um cliente que quer ler um arquivo faz uma requisição apenas ao servidor primário. O servidor primário realiza a leitura e retorna a informação para o cliente. Na escrita, o cliente envia o arquivo ao servidor primário. O servidor primário atualiza o arquivo em seu sistema de arquivos e envia uma cópia do arquivo para cada um dos servidores secundários.

1.1.2 Réplicas ativas

As réplicas ou cópias ativas [SCH93] também são conhecidas como máquinas de estado. Esta abordagem submete todas as réplicas às mesmas regras. O controle da replicação não é centralizado, como no método de cópia primária.

No procedimento de escrita, a invocação do cliente é recebida por todas as réplicas. Cada réplica processa a alteração e retorna a resposta ao cliente. O cliente espera até receber a primeira resposta. Na leitura, o cliente faz a invocação e, novamente, espera até que receba a primeira resposta.

A abordagem de réplicas ativas requer que as cópias livres de falhas recebam as invocações dos clientes na mesma ordem. Isso pode ser resolvido através de uma primitiva de comunicação que satisfaça a propriedade de ordem e também de atomicidade, descrita em Guerraoui [GUE97].

1.2 Chamada de procedimento remoto

Para que os clientes de sistemas distribuídos possam requisitar serviço aos servidores e para que os servidores possam trocar informações para manter a consistência das suas cópias de arquivos podem ser usadas chamadas de procedimento remoto (RPC ou *remote procedure call*). A RPC foi projetada para ser rápida para possibilitar bom desempenho a estes sistemas e, por isso, não contém uma estrutura com muitas camadas [TAN94].

1.2.1 Modelo cliente-servidor

O modelo cliente-servidor comporta estações de trabalho chamadas clientes que estabelecem comunicação por uma rede com outra estação de trabalho que é chamada servidor (de arquivos) [TAN94]. Neste esquema, os clientes realizam operações sobre arquivos

[1] cópias de reserva

enviando mensagens (pedidos ou requisições) ao servidor, que executa um trabalho e envia a resposta ao cliente.

Um cliente que envia uma mensagem a um servidor e recebe uma resposta funciona como um programa que chama um procedimento e recebe um resultado. O cliente inicia uma ação que é executada em outra estação. O cliente aguarda até que a ação termine e que os resultados estejam disponíveis.

Para esconder a execução de chamadas remotas é possível incorporar a RPC à linguagem de programação. O código é dividido em dois programas distintos. Um programa é local (cliente) e o outro programa é remoto (servidor). A comunicação entre programas clientes e servidor é feita através de chamadas de procedimentos de bibliotecas no cliente que recebem parâmetros. Esses procedimentos enviam mensagens ao servidor de arquivos e aguardam pela resposta. Todos os detalhes de funcionamento podem ser ocultados do programa de aplicação por procedimentos que são denominados *stubs* [TAN94]. O objetivo final é fazer com que uma RPC pareça como uma chamada de procedimento local.

1.2.2 Implementação da RPC

A implementação da RPC pode ser realizada em dez etapas [TAN94] descritas na figura 1.1. Primeiro (etapa 1) o programa do cliente chama o procedimento *stub* localizado no interior de seu próprio espaço de endereços. Os parâmetros podem ser passados de maneira usual, como em uma chamada local. O cliente não percebe nada fora do normal em relação a esta chamada, que aparece como uma chamada local.

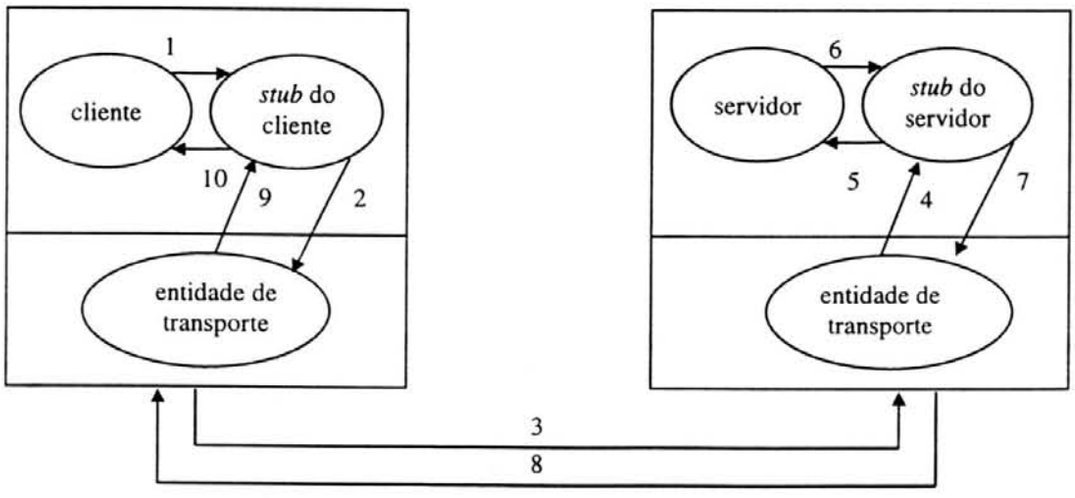

FIGURA 1.1 - Etapas necessárias para realizar uma RPC [TAN94]

O *stub* do cliente recebe os parâmetros e os acondiciona em uma mensagem em uma operação que é chamada coleta de parâmetros. Na etapa seguinte (etapa 2), a mensagem é entregue à camada de transporte para ser transmitida. A camada de transporte anexa um cabeçalho à mensagem e coloca-a na rede (etapa 3).

Quando a mensagem chega ao servidor, a camada de transporte desse lado a entrega ao *stub* do servidor (etapa 4), que separa os parâmetros. O *stub* do servidor chama o procedimento do servidor (etapa 5) que recebe os parâmetros. O procedimento do servidor não sabe que está sendo ativado de forma remota pois seu chamador é um procedimento local.

Somente os *stubs* sabem que a chamada é remota. Depois de finalizar o seu trabalho, o procedimento servidor retorna o resultado (etapa 6). Então o *stub* do servidor coleta o resultado em uma mensagem e entrega à camada de transporte (etapa 7). Quando a mensagem chega à estação cliente (etapa 8), é entregue ao *stub* do cliente (etapa 9). A seguir o *stub* do cliente entrega o resultado ao procedimento do cliente (etapa 10).

Os parâmetros de uma RPC podem ser passados por valores inteiros, ponto flutuante, *strings* de caracteres ou estruturas. O *stub* do cliente simplesmente insere os parâmetros na mensagem.

1.3 Comunicação de grupo

As primitivas de comunicação de grupo podem ser utilizadas para prover tolerância a falhas em sistemas de arquivos distribuídos. Um procedimento eficiente para garantir a consistência das cópias pode ser encontrada em Kaashoer e Tanenbaum [KAA91].

Na RPC, a comunicação ocorre entre dois processos (cliente e servidor). Porém, muitas vezes a comunicação precisa ser estabelecida entre múltiplos processos [TAN95] ou entre um grupo de processos. Isso ocorre, por exemplo, em um sistema com réplicas ativas onde múltiplos servidores precisam atualizar um arquivo. Se for usada RPC, o cliente terá que enviar a mensagem de atualização para um servidor de cada vez. Isso é um procedimento muito custoso.

Para agilizar a comunicação pode ser utilizada uma primitiva que permite comunicar um processo com vários processos [BIR96]. Isso permite, teoricamente[2], que todos os processos recebam a mensagem e atualizem o arquivo no mesmo intervalo de tempo, reduzindo o tempo de realização da operação.

Um grupo pode ser definido como uma coleção de processos que agem juntos [TAN95]. A propriedade que todos os grupos têm é que se uma mensagem é enviada para um grupo, todos os elementos operacionais do grupo devem receber a mensagem. Grupos suportam três diferentes primitivas de comunicação [TAN95]:

a) *multicast*: quando uma mensagem é enviada para um endereço, vai automaticamente para todos os endereços do grupo;

b) *broadcast*: as mensagens que contêm certos endereços são entregues para todas as máquinas;

c) *unicast*: quando um processo envia uma mensagem para outro processo. Se o sistema em questão não comportar *multicast* nem *broadcast*, estas primitivas podem ser implementadas usando *unicast*.

Essas primitivas podem sem aplicadas a dois tipos de grupos (tab. 1.1): estáticos e dinâmicos [BIR96].

---

[2] muito tráfego na rede, por exemplo, pode causar atrasos na entrega das mensagens

TABELA 1.1 - Caracterização de grupos estáticos e grupos dinâmicos

Grupo estático	Grupo dinâmico
Não muda durante a vida do sistema.	Muda durante a vida do sistema.
Quando um servidor sofre falha, o grupo não muda sua configuração; o servidor continua no grupo até ser recuperado ou não.	Quando um servidor falha é removido do grupo (*leave*); se o servidor é recuperado, pode retornar ao grupo (*join*).

Para um sistema de arquivos distribuído, um grupo é um conjunto de cópias ou réplicas. Na cópia primária, o grupo é formado pelo servidor primário e seus secundários. Devido a sua natureza, a cópia primária se adapta perfeitamente aos padrões de um grupo dinâmico [GUE97]. Um servidor em falha, que apresenta suas cópias inacessíveis, é isolado do grupo através da primitiva *leave* no procedimento de recuperação do sistema. No procedimento de reintegração do servidor, o sistema de arquivos do servidor é atualizado e o servidor junta-se ao grupo através da execução de um *join*. Em caso de falha de servidor, uma proposta de consistência de informação pode ser implementada para manter a consistência [GUE97].

1.4 Reintegração de servidor

Quando um servidor é desligado ou quando uma falha é detectada, este servidor deve ser confinado, tornando-se inativo. O servidor deve abandonar o sistema para evitar inconsistências. Quando o servidor é reparado e religado, pode voltar a participar do sistema em um procedimento chamado reintegração do servidor. O procedimento de reintegração envolve a atualização do sistema de arquivos do servidor e a integração deste com os demais servidores do sistema.

Para realizar a atualização do sistema de arquivos, o servidor troca mensagens com o sistema para obter a cópia mais recente dos arquivos. Depois disso, o sistema é notificado que o servidor está recuperado e que deverá voltar a participar ativamente das operações do sistema.

A reintegração de servidor pode ser realizada de forma manual ou de forma automática. A reintegração de forma manual requer a intervenção de operador. O operador tem a tarefa de copiar totalmente ou parcialmente um volume de dados de um servidor para outro.

A forma automática é realizada pelo próprio sistema, através de programas. O objetivo neste trabalho é tratar de mecanismos que utilizem a forma automática de reintegração. A justificativa desta escolha é simples: a forma automática requer menor interferência humana, o que é bem mais confortável para o usuário. Para tanto, o sistema distribuído deve ser capaz de tomar decisões adequadas para tolerar falhas.

A reintegração de servidores não é assunto facilmente encontrado na literatura dos sistemas distribuídos. Na maioria das vezes trata-se de um procedimento manual, executado pelo operador do sistema. O ideal seria que a reintegração fosse um procedimento automático, que fosse iniciado com a partida da estação. A reintegração torna-se necessária para prolongar a vida útil do sistema:

a) para manter a atividade normal do sistema, corrigindo uma eventual degradação de desempenho gerada por falha;

b) para manter o número de componentes (servidores) da configuração original do sistema;

c) para aumentar a confiabilidade e disponibilidade de informação, quando um novo componente é adicionado.

O procedimento de reintegração também pode ser começado devido a aquisição de um novo servidor. A aquisição de um novo servidor está relacionada à necessidade de aumentar a confiabilidade, a disponibilidade ou o desempenho da rede, fato que ocorre naturalmente em sistemas distribuídos.

1.5 Proposta deste trabalho

Este trabalho trata da reintegração de servidores em um ambiente distribuído que suporta replicação de arquivos usando a abordagem de cópia primária e *backups*. Para tanto é considerada a estrutura do sistema UNIX [BAC86], o qual comporta estações clientes e servidores.

Durante a reintegração, o sistema de arquivos do servidor que sofreu a falha deve ser atualizado. Para tanto, devem ser utilizados algoritmos que realizam a atualização de arquivos. Neste trabalho, os algoritmos de atualização foram estruturados em um protótipo com comunicação por RPC [COR91]. Serão utilizadas semânticas de acesso a arquivos disponíveis em servidores UNIX com mínimas alterações nos procedimentos clientes.

Para ilustrar o sistema de cópia primária, será utilizado o sistema experimental RNFS [LEB96, LEB97] que é uma extensão do NFS [SAN85] e propõe alta disponibilidade e alta confiabilidade de armazenamento de dados. O RNFS começou a ser desenvolvido em trabalho anterior no grupo de tolerância a falhas do CPGCC-UFRGS.

É assumido o modelo de falha de *crash* de servidor com tratamento de falhas de particionamento de rede. Um servidor em falha de *crash* simplesmente não processa nenhum serviço. Não recebe, nem emite mensagens. O particionamento de rede pode ocorrer por falhas de conexão entre servidores ou devido a um servidor que sofreu falha e confinou parte do sistema.

1.6 Resultados obtidos

O principal resultado esperado com este trabalho é analisar a viabilidade da implementação de mecanismos que facilitem a reintegração automática de servidor em um ambiente distribuído. A reintegração automática somente poderá ser implementada em um sistema real se for mais eficiente que a reintegração manual.

Para tanto foi implementado um protótipo em linguagem C, utilizando o compilador *rpcgen* [COR91], para executar a atualização do sistema de arquivos do servidor. O protótipo pode ser executado tanto em rede Sun-OS quanto em rede PC-Linux, ambas disponíveis nos laboratórios do Instituto de Informática da UFRGS.

1.7 Estrutura deste trabalho

O capítulo 2 é uma revisão bibliográfica para dar um embasamento teórico ao assunto do capítulo 3, que é a reintegração de servidor em um ambiente com cópia primária. O capítulo 2 começa com uma visão geral de alguns sistemas de arquivos distribuídos. A seguir, o capítulo apresenta uma introdução sobre aspectos de UNIX e NFS relacionados a este trabalho. Depois aparecem sistemas recentes que utilizam replicação de arquivos: o HA-NFS e o Eden File System. O HA-NFS é um produto comercial. O Eden é um sistema experimental, ainda em fase de implementação. Para fechar o capítulo, há uma breve revisão do RNFS e seus algoritmos cooperantes. Os algoritmos cooperantes do RNFS objetivam acrescentar ao NFS alta disponibilidade e alta confiabilidade, através da utilização de distribuição de cópias entre vários servidores.

O capítulo 3 apresenta a reintegração do servidor RNFS. Para realizar a reintegração é necessário atualizar o sistema de arquivo do servidor. Neste ponto, diferentes métodos de atualização de sistema de arquivos são propostos. Alguns desses métodos são implementados em um ambiente distribuído para verificar sua viabilidade.

A implementação dos métodos de atualização de arquivos e os aspectos de implementação estão descritos no capítulo 4. Para fazer a atualização dos arquivos é usado um protocolo recuperador que copia os arquivos de um servidor atualizado para outro desatualizado. Este capítulo também possui informações sobre o ambiente de programação utilizado para a parte prática.

No capítulo 5, são mostrados resultados obtidos com diferentes métodos de atualização de servidor implementados em um protótipo, além dos detalhes de implementação. No capítulo 6 são apresentadas as conclusões finais e sugestões para trabalhos futuros.

2 Sistemas de Arquivos Distribuídos

Muitos sistemas de arquivos distribuídos começaram a ser desenvolvidos nas universidades [ELL83, PU88, SAT93, LEB96], mas poucos tornaram-se produtos comercias. Este capítulo inicia com uma visão geral de alguns sistemas de arquivos distribuídos desenvolvidos tanto nas universidades como nas companhias [PAS97]. Dentre esses sistemas, podem ser destacados o HA-NFS [BHI91], o *Eden* [PU88] e o RNFS [LEB96]. Antes de tratar separadamente de cada um dos últimos três sistemas, serão revisadas os mecanismos do UNIX e do NFS relevantes para a compreensão deste trabalho.

O HA-NFS, o *Eden* e o RNFS foram escolhidos por suportarem replicação de arquivos. O HA-NFS é um produto comercial da *Sun Microsystems* que utiliza um mecanismo de cópia primária restrito com apenas um servidor *backup*. O *Eden*, desenvolvido na Universidade de *Washington* em *Seattle*, mantém a consistência das cópias de arquivos por réplicas ativas. Além de arquivos, o *Eden* suporta a replicação de outros objetos.

O capítulo termina com um breve resumo do RNFS, que é um sistema experimental que está sendo desenvolvido no II-UFRGS pelo grupo de pesquisa de Tolerância a Falhas. O RNFS suporta vários servidores sujeitos a falhas. Quando um servidor falho é reparado precisa trocar informação com o sistema para receber o estado atual. Uma descrição mais detalhada da especificação do RNFS pode ser encontrada em Leboute [LEB96].

2.1 Comparação entre sistemas de arquivos distribuídos

Há vários sistemas de arquivos disponíveis comercialmente e muitos outros experimentais, desenvolvidos principalmente nas universidades. Após observar cada descrição de alguns destes sistemas em separado [PAS95], foi realizada uma análise comparativa. Os sistemas são comparados quanto à tolerância a falhas, quanto ao modelo de falha[3] tratado pelo sistema, com identificação do tratamento dado às inconsistências ocasionadas por particionamento de rede, e quanto às características principais destes sistemas (tab. 2.1), quanto à unidade de replicação (arquivo, diretório ou volume), quanto ao mecanismo utilizado para prover consistência e integridade dos dados, e quanto à disponibilidade do serviço (tab. 2.2).

[3] segundo a classificação de Cristian [CRI86] um componente pode sofrer falha de *crash*, omissão, temporização, resposta ou bizantina.

TABELA 2.1 - Comparação entre sistemas de arquivos

	Tolerância a Falhas	Modelo de Falhas	Características Principais
Locus	replicação controlada por cópia primária e vetor de versões	*crash*, não trata consistência em caso de particionamento da rede	tolerância a falhas é garantida pelo uso de cópia primária
Roe	replicação controlada por votação balanceada [ELL83]	trata particionamento, *crash* por replicação	disponibilidade, votação balanceada, exatidão e consistência de cópias, habilidade de reconfiguração
Andrew	replicação para arquivos *read-only*	-	escalabilidade e segurança [SAT93]
NFS	operações idempotentes, servidores *stateless*	*crash*	servidores *stateless*
Sprite	não há garantias devido a política de atraso, serviço *stateful* [LEV90]; replicação *read-only*	-	alto desempenho devido ao uso de memórias *cache*, compartilhamento de arquivos [OUS88]
Cedar	replicação com número de versão controlada por *daemon* [GIF88]	-	grupos cooperantes, alta confiabilidade e alta disponibilidade por replicação de arquivos
Eden	réplicas ativas	*crash*	réplicas ativas; replicação *lazy* de objetos; regeneração
Coda	dados armazenados em múltiplos servidores e operação desconectada	falhas de particionamento de tratadas por estratégia otimista, conflitos são detectados e isolados	descendente do Andrew, escalabilidade, segurança, alta disponibilidade [SAT93]
HA-NFS	alta disponibilidade por monitoração de serviços e servidor duplicado	*crash* e conexão através da duplicação da rede de comunicação	alta disponibilidade, alta confiabilidade, servidor *backup*
RNFS	alta disponibilidade, replicação controlada por cópia primária	*crash*; em caso de particionamento apenas a partição com a maioria está apta a eleger novo primário	alta disponibilidade, alta confiabilidade, mecanismo de cópia primária

Fonte: PASIN. Tolerância a falhas em sistemas de arquivos distribuídos. p.30.

TABELA 2.2 - Comparação entre sistemas de arquivos (continuação)

	Unidade de replicação	Consistência e integridade	Disponibilidade
Locus	arquivo e volume [SIN94]	cópia primária + vetor de versão [WAL83]; consistência em caso de particionamento de rede	a cópia primária deve estar disponível para a escrita e cada componente do nome do arquivo deve estar disponível para o arquivo ser aberto [LEV90]; rede totalmente conectada
Roe	arquivo [SIN94]	trata por replicação de arquivos	replicação de arquivos e de informação para realizar acesso aos arquivos [ELL83]
Andrew	volume	*backup* de arquivos em vários servidores	componentes do nome do arquivo devem estar disponíveis [LEV90]; banco de dados replicado com a localização das cópias; replicação evita que servidor torne-se gargalo
NFS	-	operações idempotentes + servidores *stateless*	todos os servidores que são usados para alcançar o arquivo devem estar disponíveis [LEV90]
Sprite	arquivo [SIN94]	número de versão para consistência em arquivos compartilhados	se o servidor do arquivo está disponível, o arquivo também, independente do estado dos outros servidores [LEV90]
Cedar	arquivo [SIN94]	arquivos compartilhados imutáveis [GIF88]	uma lista de servidores é analisada para encontrar o servidor disponível para o arquivo
Eden	objeto [PU88]	réplicas ativas; algoritmo da regeneração [PU88]	replicação *lazy*; réplicas ativas
Coda	volume [SIN94]	replicação de arquivos em servidores	arquivo disponível no servidor preferencial [SAT93]
HA-NFS	disco (volume completo) [BHI91]	espelhamento de disco [BHI91]	arquivo disponível pelo primário na operação normal ou pelo *backup*
RNFS	volume [LEB96]	cópia primária [LEB96]	arquivo disponível no servidor primário

Fonte: PASIN. Tolerância a falhas em sistemas de arquivos distribuídos. p.31.

Muitos sistemas de arquivos distribuídos (tabs. 2.1 e 2.2) apresentam características de tolerância a falhas. Para tanto, esses sistemas fazem uso principalmente da duplicação de

manutenção da consistência das cópias replicadas. No *Andrew*, como apenas arquivos que não permitem atualização[4] são replicados, um procedimento que realize a atualização atômica de escritas não é necessário.

Para controlar a replicação das cópias, os sistemas utilizam dois grupos de soluções: a replicação controlada por votação e a replicação não controlada por votação. O *Roe* utiliza replicação controlada por votação balanceada [ELL83] que garante a consistência das cópias através de *quoruns*. O inconveniente da votação está no próprio uso do algoritmo. Nas escritas, após a obtenção do *quorum* pelos servidores, os clientes ficam retidos até que a escrita seja propagada entre todos os servidores.

O RNFS utiliza cópia primária para controlar a consistência dos arquivos. A principal desvantagem é que a cópia primária é um elemento centralizador. Em caso de falha na cópia primária, uma nova cópia primária deve ser escolhida e os clientes precisam ser avisados.

Outro sistema que utiliza a cópia primária, dessa vez associada a um vetor de versão, é o *Locus*. O *Locus* é um exemplo clássico de sistema com replicação. Objetivo do *Locus* é obter tolerância a falhas totalmente transparente ao usuário. A desvantagem desse sistema é que nem todas as inconsistências são resolvidas em caso de particionamento de rede. Quando as partições se juntam, o sistema pode solicitar ao usuário que escolha a versão dos arquivos que deverá permanecer.

Para compensar a perda de desempenho ocasionada pela tolerância a falhas, sistemas de arquivos utilizam *caches* em clientes e servidores. A *cache* aparece em sistemas voltados ao alto desempenho como o *Sprite* [OUS88] e o *Cedar* [GIF88].

O *Sprite* [OUS88] usa servidores *stateful*, mas praticamente não apresenta características de tolerância a falhas. Para manter a consistência entre a *cache* dos arquivos do servidor e dos clientes, o Sprite usa números de versão.

O *Cedar* [GIF88] possui suporte para grupos cooperantes utilizando arquivos imutáveis. A replicação de arquivos é controlada por um *daemon*. O sistema permite que cada usuário gerencie seus arquivos e auxilia na consistência das versões dos arquivos que são compartilhados. Como o *Sprite*, também usa *cache* nos clientes para obter alto desempenho. Para manter a consistência, o *Cedar* não permite que arquivos compartilhados sejam modificados. Esses arquivos são ditos imutáveis[5]. Para tratar as alterações em arquivos imutáveis, novas versões são criadas localmente nos clientes.

Para prover alta disponibilidade, alta confiabilidade e alto desempenho de acesso, o *Cedar* utiliza replicação de arquivos. Os arquivos estão organizados em diretórios que são

[4] arquivos *read-only*

[5] um arquivo que não pode ser alterado após sua criação. As únicas operações possíveis são criação, leitura e eliminação

componentes de *hardware* (como ocorre no HA-NFS) e replicação de arquivos (como no *Cedar*, *Andrew* e *Roe*). A replicação por *software* parece ser preferida devido a independência do *hardware* existente, já que a replicação por *hardware* requer componentes específicos.

Os sistemas estudados geralmente possuem alguma proposta para a disponibilidade e manutenção do serviço nos servidores. A disponibilidade é adquirida pelo uso de servidores *stateless*, que facilitam a recuperação de falhas em sistemas como o *Andrew* e o NFS. Servidores *stateless* não mantêm informações de estado e *stateful* mantêm estas informações.

A replicação de arquivos é um mecanismo utilizado pela maioria dos sistemas de arquivos distribuídos estudados. Podem ser citados como exemplos o RNFS [LEB96], o *Coda* [SAT93], o *Locus* [WAL83], o *Roe* [ELL83] e o *Andrew* [SAT93] que replica apenas arquivos *read-only*. Estes sistemas, exceto o *Andrew*, foram projetados para controlar a manutenção da consistência das cópias replicadas. No *Andrew*, como apenas arquivos que não permitem atualização[4] são replicados, um procedimento que realize a atualização atômica de escritas não é necessário.

Para controlar a replicação das cópias, os sistemas utilizam dois grupos de soluções: a replicação controlada por votação e a replicação não controlada por votação. O *Roe* utiliza replicação controlada por votação balanceada [ELL83] que garante a consistência das cópias através de *quoruns*. O inconveniente da votação está no próprio uso do algoritmo. Nas escritas, após a obtenção do *quorum* pelos servidores, os clientes ficam retidos até que a escrita seja propagada entre todos os servidores.

O RNFS utiliza cópia primária para controlar a consistência dos arquivos. A principal desvantagem é que a cópia primária é um elemento centralizador. Em caso de falha na cópia primária, uma nova cópia primária deve ser escolhida e os clientes precisam ser avisados.

Outro sistema que utiliza a cópia primária, dessa vez associada a um vetor de versão, é o *Locus*. O *Locus* é um exemplo clássico de sistema com replicação. Objetivo do *Locus* é obter tolerância a falhas totalmente transparente ao usuário. A desvantagem desse sistema é que nem todas as inconsistências são resolvidas em caso de particionamento de rede. Quando as partições se juntam, o sistema pode solicitar ao usuário que escolha a versão dos arquivos que deverá permanecer.

Para compensar a perda de desempenho ocasionada pela tolerância a falhas, sistemas de arquivos utilizam *caches* em clientes e servidores. A *cache* aparece em sistemas voltados ao alto desempenho como o *Sprite* [OUS88] e o *Cedar* [GIF88].

O *Sprite* [OUS88] usa servidores *stateful*, mas praticamente não apresenta características de tolerância a falhas. Para manter a consistência entre a *cache* dos arquivos do servidor e dos clientes, o Sprite usa números de versão.

O *Cedar* [GIF88] possui suporte para grupos cooperantes utilizando arquivos imutáveis. A replicação de arquivos é controlada por um *daemon*. O sistema permite que cada usuário gerencie seus arquivos e auxilia na consistência das versões dos arquivos que são compartilhados. Como o *Sprite*, também usa *cache* nos clientes para obter alto desempenho. Para manter a consistência, o *Cedar* não permite que arquivos compartilhados sejam modificados. Esses arquivos são ditos imutáveis[5]. Para tratar as alterações em arquivos imutáveis, novas versões são criadas localmente nos clientes.

Para prover alta disponibilidade, alta confiabilidade e alto desempenho de acesso, o *Cedar* utiliza replicação de arquivos. Os arquivos estão organizados em diretórios que são

---

[4] arquivos *read-only*

[5] um arquivo que não pode ser alterado após sua criação. As únicas operações possíveis são criação, leitura e eliminação

replicados nos servidores. As alterações nos diretórios dos servidores de arquivos são realizadas serialmente, e parecem invisíveis para o usuário. A transferência de um arquivo para o servidor com atribuição de um novo número de versão e de um novo nome para o arquivo, aparece com uma única ação. Se algum passo falha, então nenhum efeito torna-se visível ao sistema.

O *Coda* [SAT93] apresenta replicação aliada a operação desconectada para os clientes para prover alta disponibilidade. A duplicação de dados permite um repositório de dados compartilhado quando um servidor específico não pode ser contactado ou para contornar algumas falhas de rede.

O sistema *Coda* alterna entre a operação normal e a operação desconectada. A operação normal ocorre quando um cliente contacta um servidor para executar um serviço. A operação desconectada é utilizada quando nenhum servidor está acessível. A intenção da desconexão é que os usuários possam utilizar recursos locais quando os servidores estiverem parcialmente ou totalmente inacessíveis. Quando algum servidor recupera, o sistema volta a utilizar a operação normal. Os conflitos são detectados e isolados tão logo a conexão entre as diversas partições seja restabelecida, mas podem requisitar ao usuário que corrija os conflitos.

Alta disponibilidade é uma característica principal dos sistemas de arquivos recentes e aparece no *Coda*, no RNFS e no HA-NFS. Esses dois últimos sistemas possuem mecanismos para prover tratamento automático de falhas. A alta confiabilidade, outra característica presente, é obtida por replicação da informação. A detecção de falhas e a recuperação do sistema é transparente ao usuário.

Apesar das técnicas de tolerância a falhas empregadas por diferentes sistemas, ainda não há uma solução totalmente satisfatória. Isso deve-se a alguns fatores, como a dificuldade de gerenciar as cópias dispersas e o alto custo de *software* de propósito específico. Ainda deve ser ressaltado que o casamento entre confiabilidade e desempenho ainda não é o desejável. Principalmente porque o baixo custo do *hardware* de propósito geral não compensa gastos com *software* para o gerenciamento da informação replicada. Os sistemas comercias atuais, de um modo geral, ainda não incorporam as características de tolerância a falhas desejáveis.

Características que juntem confiabilidade e alto desempenho aproveitando a organização do sistema de arquivos distribuídos ainda podem ser aprimoradas. A ação de atualizar cópias dispersas ainda é muito cara e dispendiosa.

2.2 Arquivos distribuídos em ambiente UNIX

Para implementar a reintegração de servidores em um sistema distribuído que utiliza um sistema de arquivos baseado no NFS, é necessário ter uma clara noção de funcionamento de algumas primitivas UNIX e características do NFS. Assim, o item 2.2 se limita a abordar algumas dessas primitivas e características. O sistema de arquivos mais usado atualmente com o UNIX é o *Sun* NFS [SAN85].

2.2.1 UNIX

O UNIX [BAC86] é um sistema operacional multiusuário e multitarefa desenvolvido a partir 1969. Desde então o UNIX tem sido amplamente utilizado no meio acadêmico e comercial.

Este item não pretende fazer um tutorial sobre UNIX, apenas apresentar aspectos relevantes para observar a viabilidade da implementação da reintegração automática de servidores neste ambiente. O UNIX não possui todas as primitivas necessárias para suportar replicação de arquivos, mas suporta arquivos dispersos na rede. Leitores familiarizados com UNIX podem ir direto para 2.2.2.

Histórico do UNIX

Em 1964, várias companhias começaram a construir um sistema operacional chamado MULTICS [HAU97]. Em 1969, devido à divergências, os Laboratórios *Bell* da AT&T abandonaram o projeto e resolveram trabalhar no seu próprio sistema operacional chamado UNIX. O UNIX era baseado no MULTICS, que era um sistema para computadores maiores. MULTICS quer dizer *Multiplexed Information and Computing Service* [MUL97]. UNIX era um trocadilho (*uni* é o contrário de *multi* e *x* tem som de *cs* em inglês). O objetivo do UNIX é garantir o mínimo tempo de resposta aceitável para o usuário.

Mais tarde, o UNIX foi escrito em uma linguagem de programação chamada C, que também foi inventada nos Laboratórios *Bell*. No final dos anos 70, quando o UNIX atingiu a versão 7, o código foi distribuído para várias universidades. Então os pesquisadores das universidades começaram a modificar o código, principalmente em *Berkeley*. Logo apareceu o que foi chamado de BSD UNIX (*Berkeley Software Distribution*), distribuído pela universidade.

Nos anos 80, o UNIX da AT&T e o de *Berkeley* já estavam consolidados. O código de *Berkeley,* que era distribuído livremente, foi modificado por um grupo de pesquisadores de da Universidade de *Stanford* para rodar nas suas estações. A primeira estação *Sun* (*Stanford University Network*) foi construída em uma universidade. Com a criação da empresa, o sistema foi chamado de *Sun*-OS (*Sun Operating System*). Mais tarde o sistema foi modificado em *Sun* NFS (*Network File System*) para funcionar em rede, conectando vários computadores ou estações de trabalho.

Estrutura do UNIX

A estrutura do UNIX é formada por um núcleo dividido em módulos. Cada um dos módulos possui uma função bem definida. O módulo mais importante para o nosso trabalho é o sistema de arquivos. O gerente de arquivos é responsável por implementar todos os serviços do sistema de arquivos. Os serviços incluem acesso a arquivos, acesso a dispositivos e chamadas de sistema destinadas a estes acessos. Adicionalmente, o UNIX não tem primitivas para suportar replicação, mas suporta arquivos dispersos na rede.

Chamadas de sistema UNIX

Uma chamada de sistema é uma interrupção de *software* gerada por uma instrução de um programa de usuário. O processamento de cada interrupção executa uma rotina específica do sistema operacional. Internamente, uma chamada de sistema é formada pela execução de uma ou mais funções primitivas.

O UNIX é implementado em linguagem de programação C [KER90], que é amplamente difundida. Essa linguagem oferece ao usuário bibliotecas para utilização de chamadas de sistema UNIX.

Para implementar os métodos de atualização de arquivos, serão utilizadas algumas chamadas dedicadas ao sistema de arquivos. As chamadas do sistema de arquivos estão divididas em diferentes grupos, conforme seu propósito, descritos a seguir.

Chamadas para acesso a arquivos

O tratamento de arquivos no UNIX é realizado através das chamadas *open*, *close*, *read*, *write* e *lseek*. Cada arquivo é associado a um descritor de arquivo (*file handle*) através da chamada *open*. O descritor permite identificar o arquivo nas operações subsequentes (*close*, *read*, *write* e *lseek*). O número retornado no descritor na chamada *open* é válido apenas dentro do processo em questão e em seus processos filhos.

A chamada *open* é o procedimento de abertura do arquivo. Quando o usuário deseja ler ou escrever dados em um arquivo, deve informar sua intenção ao sistema operacional através desse procedimento. O procedimento de abertura realizado pela chamada *open* associa o nome do arquivo a um número descritor. Esse programa utiliza um algoritmo para localizar o *inode* (contração de *index node*) correspondente ao arquivo. Se a busca obteve sucesso, será retornado um valor inteiro na variável do descritor de arquivo.

O *inode* de um arquivo é uma pequena tabela em disco que contém informação sobre o arquivo, exceto seus dados propriamente ditos. No UNIX, cada arquivo de um dispositivo possui um único *inode* correspondente. Desta forma, o *inode* é um descritor perfeito para um arquivo.

A chamada *close* elimina a conexão do descritor de arquivo com o arquivo aberto, permitindo que o descritor seja utilizado com outro arquivo. Como existe um limite de arquivos abertos para cada programa de usuário, a reutilização de descritores pode ser particularmente útil. O limite máximo de arquivos abertos é normalmente 20.

A chamada *read* lê determinado número de *bytes* em um arquivo especificado. Essa chamada possui três argumentos. O primeiro argumento é o descritor para o arquivo, o segundo argumento é um *buffer*[6] para conter os dados lidos e o último argumento é o número de *bytes* a serem lidos. A chamada *read* retorna o número de *bytes* lidos.

A chamada *write* escreve determinado número de *bytes* em um arquivo especificado. Essa chamada possui três argumentos. O primeiro argumento é o descritor para o arquivo, o segundo argumento é um *buffer* para conter os dados escritos e o último argumento é o número de *bytes* a serem escritos. A chamada *write* retorna o número de *bytes* gravados.

Qualquer número de *bytes* pode ser lido ou gravado em uma chamada. Os valores mais comuns são de apenas um caractere ou valores como BUFSIZ[7] que corresponde ao tamanho físico do bloco em um dispositivo periférico do sistema considerado. Tamanhos maiores são mais eficientes pois requerem menos chamadas de sistemas.

A chamada *lseek* posiciona o cursor dentro do arquivo sem leitura ou gravação. As chamadas *read* e *write* são operações seqüenciais que realizam acesso à posição seguinte à corrente. Entretanto, pode ser necessário que um arquivo seja lido ou gravado em posição arbitrária. Uma leitura ou gravação subsequente a um *lseek* começa na posição indicada por *lseek*.

---

[6] espaço de armazenamento temporário; neste caso o *buffer* é um vetor de caracteres

[7] o valor de BUFSIZ está definido na biblioteca C <syscalls.h> e possui o tamanho adequado ao *buffer* do sistema local

Chamadas para criação de arquivos

A chamada *creat* cria novos arquivos ou escreve sobre arquivos antigos:

da = creat (nome, perms);

Essa chamada retorna um descritor para o arquivo especificado em *nome*. Se o arquivo já existe, o seu conteúdo será ignorado. Se o arquivo não existir, *creat* gerará permissões especificadas em *perms*.

No UNIX, as permissões de acesso a leitura, escrita ou execução de um arquivo são identificadas por nove *bits*. Essas permissões controlam o acesso de arquivos ao dono do arquivo, para o grupo de dono do arquivo e para os demais usuários. Essas permissões são informadas a *creat* através do argumento *perms*, que é um número octal formado por três dígitos.

A chamada de sistema *mknod* constrói um arquivo especial usado para a reconfiguração do sistema. É de uso restrito do administrador do sistema..

Chamadas para tratamento de *inodes* e controle de estrutura

As chamadas para tratamento de *inodes* são *stat, fstat, link* e *unlink*. As chamadas *chdir, chroot, chmod* são usadas para controle da estrutura. A chamada *chdir* é dedicada ao tratamento de diretórios. Para o UNIX, os diretórios são arquivos comuns, mas com formatação interna especial. Diretórios são arquivos criados com atributo de diretório em seu *inode* e só podem ser alterados através das chamadas de sistema. A estrutura interna de um diretório é uma tabela contendo o nome de cada arquivo e um apontador, normalmente de 2 *bytes*, indicando o endereço do *inode* que representa o arquivo no bloco de *inodes* no disco. Todos os arquivos são apontados por *inodes*. Cada diretório também possui um apontador para o *inode* que o descreve e um apontador para o *inode* que descreve o diretório imediatamente acima. O diretório raiz possui um apontador para o *inode* raiz do sistema.

A chamada de sistema *stat* recebe como parâmetro um nome de arquivo e retorna a informação do *status* do arquivo. A mesma informação pode ser obtida por *fstat*, mas depois que o arquivo for aberto e associado a um descritor, obtido na chamada *open*. A chamada *fstat* recebe um descritor de arquivo como argumento. A estrutura retornada está descrita no biblioteca *<sys/stat.h>* e tem os componentes da figura 2.1.

```
struct stat {
        // estrutura que contém informação sobre o arquivo

        dev_t st_dev;       // dispositivo onde o arquivo está guardado
        ino_t st_ino;       // número do inode
        mode_t st_mode;     // bits de tipo do arquivo
        nlink_t st_nlink;   // número de links do arquivo
        uid_t st_uid;       // ID do proprietário
        gid_t st_gid;/      / ID do grupo do proprietário
        dev_t st_rdev;      // identificação do dispositivo
        off_t st_size;      // tamanho do arquivo em bytes
        time_t st_atime;    // hora do último acesso
        time_t st_mtime;    // hora da última modificação
        time_t t_ctime;     // hora de alteração do status do arquivo
        long st_blksize;    // tamanho do bloco de I/O
        long st_blocks;     // número de blocos alocados
};
```

FIGURA 2.1 - Estrutura e informação do arquivo [KER90]

A informação retornada pela chamada *stat* pode ser usada para identificar o tipo do arquivo (por exemplo: se é diretório ou arquivo normal). Os campos *st_atime* e *st_mtime* da chamada *stat* não são contíguos. Programas que dependem de tempos contíguos devem usar outras informações.

As chamadas de sistema *link* e *unlink* são destinadas a tratamento de arquivos. A chamada *unlink* remove o arquivo do sistema. A chamada *chroot* muda o diretório raiz. A chamada *chmod* muda as permissões para um arquivo. Apenas o proprietário do arquivo ou o administrador do sistema pode alterar as permissões de um arquivo.

Tratamento de *pipes*

Um *pipe* é um canal de comunicação entre processos. Para tratamento de *pipes* são usadas as chamadas de sistema *dup* e *pipe*. A chamada *dup* duplica um descritor de arquivo. A chamada de sistema *pipe* gera um canal de comunicação entre processos. O *pipe* utiliza dois descritores, um para cada arquivo. Um dos descritores é usado para leitura e outro para escrita. A comunicação é realizada lendo informação em um arquivo e escrevendo no outro. Estas duas chamadas não serão utilizadas neste trabalho.

Montagem e desmontagem

Os arquivos do sistema UNIX estão dispersos nos dispositivos da rede. Cada um dos dispositivos contém um sistema de arquivo. Para prover transparência de localização, os diferentes sistemas de arquivos são unidos em uma árvore virtual de diretórios através das chamadas de montagem e desmontagem. Essas chamadas são representadas respectivamente por *mount* e *umount*.

Quando uma estação é ligada, os sistemas de arquivos dispersos nos dispositivos são unidos através do procedimento de montagem a um determinado ponto de montagem até compor toda a árvore virtual.

O procedimento inverso à montagem denomina-se desmontagem. A desmontagem é usada quando a estação é desligada, para desconectar os dispositivos antes da retirada da energia. A relação de dispositivos montados e de seus pontos de montagem é mantida na tabela de montagem. Os comandos *mount* e *umount* correspondem a chamadas de sistema específicas para a atualização desta tabela.

2.2.2 Network File System

Desde 1985, o *Sun Network File System* (NFS) [SAN85] tem sido amplamente utilizado devido à sua independência de *hardware* e sistema operacional. O NFS suporta estações heterogêneas e permite alta portabilidade. Embora o modelo original tenha sido proposto para estações UNIX, atualmente o NFS pode ser portado para outros sistemas operacionais, como PC-DOS.

O NFS convencional implementa um ambiente com clientes e servidor que permite distribuição transparente e compartilhamento de arquivos em redes de estações UNIX. Para que um diretório remoto seja acessível ao cliente de forma transparente, o NFS *monta* o diretório remoto localmente (no cliente) através do procedimento de montagem. Adicionalmente, o NFS convencional não suporta replicação de arquivos.

O item seguinte apresenta uma revisão de alguns mecanismos do NFS. Leitores já familiarizados com o sistema podem ir direto para a seção 2.3.

Arquitetura do NFS

A arquitetura do NFS (fig. 2.2) é formada por diferentes camadas. A interface com o sistema UNIX separa as operações do sistema de arquivos das implementações específicas do sistema operacional.

A VFS ou *Virtual File System* define os procedimentos que operam o sistema de arquivo como um todo. Por suportar diferentes VFS, o NFS suporta diferentes sistemas de arquivo. A estrutura da VFS é chamada de *vnode* (*virtual node*). Cada arquivo ou diretório possui um *vnode* correspondente. Um *vnode* é único no sistema e é a reimplementação dos *inodes* do UNIX. Cada *vnode* é complementado por uma tabela de montagem que possui um ponteiro para o sistema de arquivos imediatamente superior e para o sistema de arquivos sobre o qual é montado. Isso permite que qualquer estação no sistema de arquivos seja um ponto de montagem. Usando as tabelas associadas aos pontos montados, a VFS pode distinguir sistemas de arquivos locais de remotos. As requisições de operação em arquivos remotos são enviados à camada NFS pela camada da VFS.

O NFS usa RPC [COR91] (*remote procedure call*) ou chamada de procedimento remoto para realizar as operações em arquivos remotos. A comunicação entre clientes e servidores NFS é orientada à requisição de serviço, através do envio de mensagens. Um processo cliente (local) empacota a requisição em uma mensagem e transmite-a ao servidor (remoto). O servidor desempacota os dados em variáveis locais, executa o serviço e transmite o resultado, ou a indicação de alguma exceção, através de uma mensagem de resposta. Enquanto o servidor não responde ao serviço, o cliente fica bloqueado, isto é, não executa nenhuma atividade. A RPC será melhor explicada no capítulo 6. Através de RPCs, os pedidos são propagados da VFS para o servidor remoto. A VFS remota realiza o serviço no sistema de arquivo local.

Para que o NFS possa suportar estações heterogêneas independentes de sistema operacional e *hardware*, é usada uma especificação de uma representação externa da dados (XDR ou *eXternal Data Representation*) para descrever os protocolos de RPC. A XDR é implementada como uma biblioteca de funções que deve ser utilizada por um programa de aplicação que utilize RPC.

O mecanismo de RPC é independente de protocolos de transporte. As implementações correntes do NFS utilizam o protocolo de datagramas não confiável UDP/IP e rede com barramento Ethernet [TAN92] e um ambiente com rede local com estações conectadas por fibra ótica ou cabo coaxial.

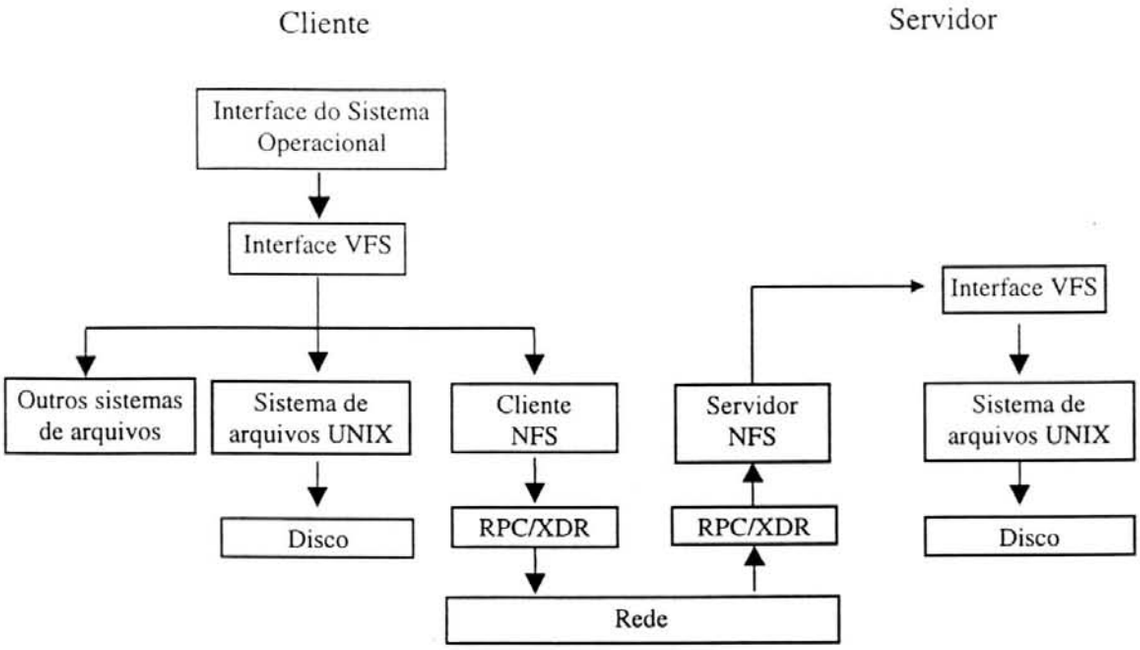

FIGURA 2.2 - Arquitetura do *Sun* NFS [LEV90]

Protocolo do NFS

O protocolo do NFS é formado por um conjunto de 18 procedimentos de chamadas de procedimento remoto. Cada procedimento de RPC passa uma estrutura como argumento e recebe um resultado.

Nas operações de leitura são passados os parâmetros de leitura na estrutura *readargs* (fig. 2.3). Essa estrutura contém um *nfs_fh* (*nfs file handle*) que é um ponteiro indicando o arquivo acessado, dois inteiros indicando a posição e extensão dos dados a serem lidos e um outro campo adicional que não é usado.

A estrutura *writeargs* (fig. 2.3) contém os parâmetros de escrita. São os mesmos parâmetros da leitura além de bloco de dados.

```
struct readargs {
      nfs_fh file;                    // handle do arquivo
      unsigned int offset;            // posição onde iniciar a leitura
      unsigned int count;             // bytes a ler
      unsigned int total_count;       // não usado
};


struct writeargs {
      nfs_fh file;                    // handle do arquivo
      unsigned int begin_offset;// não usado
      unsigned int offset;            // posição do inicio
      unsigned int total_count;       // não usado
      struct {
            unsigned int length;      // tamanho dos dados
            char *buffer;       // dados
      } data;
};
```

FIGURA 2.3 - Parâmetros para as chamadas de leitura e escrita

O NFS convencional possui 18 RPCs descritas na tab. 2.3. Para implementar a reintegração de servidores será necessário utilizar algumas destas RPCs para realizar da troca de informações entre estações.

TABELA 2.3 - RPCs do protocolo NFS

Número	**Nome**	**Natureza**	**Pars. Entrada**	**Pars. Saída**
Descrição				
0	NULL	Nulo	nenhum	nenhum
Procedimento nulo usado para teste do servidor.				
1	GETATTR	Leitura	*nfs_fh*	*attrstat*
Recupera os atributos (permissões) para o arquivo apontado por *nfs_fh*.				
2	SETATTR	Escrita	*sattargs*	*attrstat*
Modifica os atributos de um arquivo				
3	ROOT	Leitura	nenhum	nenhum
Obtém um *handle* para o diretório raiz de um volume. Função obsoleta.				
4	LOOKUP	Leitura	*diropargs*	*diropres*
Obtém um *handler* para um arquivo informado por nome.				
5	READLINK	Leitura	*nfs_fh*	*readlinkres*
Lê o valor de uma entrada de diretório tipo *link* simbólico.				
6	READ	Leitura	*readargs*	*readres*
Lê um bloco de dados de um arquivo informado por um *handle*.				
7	WRITECACHE	Leitura	nenhum	nenhum
Função não usada. Espera potencial para um algoritmo de consistência de *cache*.				
8	WRITE	Leitura	*writeargs*	*attrstat*
Escreve um bloco de dados em um arquivo informado por um *handle*.				
9	CREATE	Escrita	*createargs*	*diropres*
Cria um arquivo a partir do nome e *path* informados.				
10	REMOVE	Escrita	*dirpoargs*	*nfsstat*
Remove um arquivo a partir de seu nome e *path*.				
11	RENAME	Escrita	*renameargs*	*nfsstat*
Atribui um novo nome a um arquivo informado por nome e *path*.				
12	LINK	Escrita	*linkargs*	*nfsstat*
Cria um *hard link* no servidor para o arquivo informado.				
13	SYMLINK	Escrita	*symlinkargs*	*nfsstat*
Cria um *symbolic link* para o arquivo informado.				
14	MKDIR	Escrita	*createargs*	*diropres*
Cria um diretório com o nome fornecido a partir de outro diretório informado.				
15	RMDIR	Escrita	*diropargs*	*nfsstat*
Remove um diretório, informado por nome e *path*.				
16	READDIR	Leitura	*readdirargs*	*readdirres*
Lê uma linha em um arquivo de diretório.				
17	STATFS	Leitura	*nfs_fh*	*statfsres*
Retorna parâmetros gerais do volume contendo o arquivo apontado.				

Fonte: Leboute [LEB96], adaptado de [RFC94] Request For Comments 1094.

Servidores *stateless*

Com a finalidade de facilitar a recuperação após falha de *crash*, o NFS foi projetado com servidores de arquivos *stateless*. Um servidor *stateless* não possui qualquer informação a respeito das requisições de seus clientes. Um pedido de acesso a um arquivo feito por um cliente possui todas as informações necessárias para ser realizado. Quando um cliente não recebe a resposta do pedido, simplesmente retransmite o pedido. Um cliente não diferencia entre um servidor que sofreu uma falha de *crash* e um servidor que é lento. Devido à propriedade do cliente poder refazer o pedido, o pedido precisa ser idempotente[8].

Um servidor *stateless* recuperado após falha, simplesmente reinicia. Este servidor não precisa restabelecer informações como a interação com clientes antes de sofrer a falha e não precisa consultar os clientes para saber o estado dos arquivos antes da falha [SIN94].

A desvantagem dos servidores *stateless* é que o servidor não tem como guardar informações de quais arquivos estão abertos e quais arquivos estão fechados. Isso significa que, quando um arquivo é compartilhado, não há como detectar conflitos de acesso e o projeto da *cache* para o servidor é dificultado.

Quando é desejável detectar conflitos de acesso, os servidores precisam ser *stateful*. Esses servidores mantém informação de estado[9] dos arquivos. Dessa forma é possível implementar um sistema de *cache* eficiente que garanta a consistência de arquivos compartilhados. É o caso do *Sprite File System* [OUS88]. Em compensação, a recuperação de um servidor em falha de *crash* é mais complicada.

Devido ao fato do NFS utilizar servidores *stateless* e pedidos idempotentes, os modos de falhas observados pelos clientes, quando acessam arquivos remotos, são similares aos utilizados para acesso a arquivos locais [COU94]. Quando um servidor falha, o serviço é interrompido até a recuperação do servidor. Um cliente, que fez um pedido, espera até o servidor recuperar e prossegue do ponto no qual o servidor interrompeu o serviço sem saber da ocorrência da falha.

Servidores *stateless* permitem acesso eficiente sem precisar saber dos estados dos arquivos, em caso de falha. O resultado final é que o servidor pode ser interrompido e reiniciado a qualquer momento sem avisar os clientes. As informações necessárias à realização dos acessos podem ser reconstruídas, possibilitando facilidades para implementar a recuperação automática das conexões NFS em caso de falha.

Tornando o NFS mais confiável

Para prover maior confiabilidade ao NFS, alguns sistemas baseados em cópia primária têm sido propostos [BHI91, LEB96, LIS91]. Dentre estes sistemas merecem destaque o HA-NFS [BHI91] (descrito na seção 2.3), um produto comercial da Sun Microsystems, e o RNFS [LEB96] (descrito na seção 2.5), um sistema experimental que está sendo desenvolvido no Instituto de Informática da UFRGS. Estes dois sistemas assemelham-se pela sua capacidade de recuperação automática se ocorrer falha no servidor primário. Em caso de falha no servidor primário do HA-NFS, há um *backup* para substituí-lo, disponibilizando os dados (do disco). O mesmo acontece com o RNFS: se o primário falha, um dos *backups* deve continuar a prover o serviço. Estes dois sistemas diferem, principalmente, porque o HA-NFS possui somente um

---

[8] o efeito de um pedido de operação de um arquivo feito várias vezes deve ter o mesmo efeito de um pedido que foi feito apenas uma vez

[9] quais arquivos estão abertos, por quais clientes e qual modo (leitura ou escrita)

servidor *backup*, enquanto que o RNFS suporta *n backups*. Isso permite ao RNFS suportar maior número de falhas seguidas: se o servidor primário do HA-NFS falhar, o *backup* torna-se o novo primário. Se o novo primário falhar, infelizmente os dados gerenciados por estes servidores estarão indisponíveis. Por outro lado, se o servidor primário RNFS falhar, um dos *backups* se tornará o novo primário. Se o novo primário falhar, haverá outro *backup* para substituí-lo. Assim, o número de pontos de falha do RNFS é dependente da quantidade de servidores.

A principal diferença entre o HA-NFS [BHI91] e o RNFS [LEB96] é que o HA-NFS é um sistema comercial e o RNFS é um sistema experimental e acadêmico, em fase de desenvolvimento. Adicionalmente, o RNFS não necessita de nenhum hardware especial, o que não acontece no HA-NFS (que precisa de um disco espelhado com uma porta dual). O HA-NFS faz parte de um produto que envolve *software* e *hardware*.

Para finalizar, o HA-NFS permite alta confiabilidade dos dados armazenados por espelhamento de disco e o RNFS por cópias replicadas em diferentes servidores, o que parece ser mais atraente (em se tratando de confiabilidade) apesar da necessidade do RNFS ter que tratar do problema de gerenciamento destas cópias.

Uma outra abordagem que pode ser usada para tornar o NFS mais confiável envolve o uso de réplicas ativas. As réplicas ativas aparecem no *Eden System* [PU88] (seção 2.4). Este sistema também possui mecanismos para prover alta disponibilidade através da replicação das cópias em vários servidores.

Contrastando com o *Eden* [PU88], o RNFS [LEB96] replica apenas arquivos (e não objetos) utilizando a cópia primária. O *Eden* replica objetos por réplicas ativas. Outra diferença importante é que o *Eden* utiliza o algoritmo de regeneração para manter a quantidade de cópias do sistema. No RNFS, o administrador decide quantas cópias podem ser feitas de cada arquivo, mas nenhum procedimento foi proposto para a manutenção do número de cópias quando um servidor é perdido. O RNFS prevê a incorporação de um procedimento de reintegração de servidor para que o sistema possa recuperar a quantidade de cópias que havia antes do isolamento do servidor. Comparando com o HA-NFS, o RNFS não requer nenhum *hardware* especial, além do disponível em uma rede NFS comum. Além disso, a replicação do RNFS envolve muitos servidores *backups*, enquanto que o HA-NFS possui apenas um servidor *backup*. Nos próximos itens, os mecanismos básicos do RNFS [LEB96] serão revisados.

2.3 High Availability Network File System

O HA-NFS ou *High Availability Network File System* [BHI91] é uma proposta para tornar o NFS mais confiável. O HA-NFS é parte de um sistema que possui *software* e *hardware* integrados para prover alta disponibilidade de serviços. A detecção de falhas e a recuperação para os serviços de servidor são automáticas. O HA-NFS não permite a replicação de arquivos.

O sistema tolera apenas um ponto de falha, que pode ser de *crash* ou conexão entre servidores. Para tratar falhas de *crash*, o sistema utiliza dois servidores. Um servidor é o primário e o outro é o *backup*. Os servidores são conectados por um disco através de uma porta dual. Apenas um servidor acessa o disco de cada vez. Esse servidor é o primário.

Falhas de disco são toleradas por espelhamento de disco[10]. O espelhamento de disco replica todos os dados de um disco em outro. As duas cópias são controladas pelo servidor primário corrente.

Falhas de conexão são contornadas pela replicação da conexão entre clientes e servidores. A conexão dual é utilizada como um meio de comunicação adicional entre os servidores primário e secundário.

Durante a operação normal, os clientes fazem requisições ao servidor primário. O primário realiza as alterações no disco e as envia aos clientes. Apenas o primário corrente tem acesso lógico ao disco. O *backup* se limita a enviar mensagens *Are you alive?* (fig. 2.4) ao servidor primário e a receber *acks*[11].

Se o secundário não recebeu o *ack* do primário, assume que a conexão com o primário falhou. Então o secundário tenta nova comunicação usando a porta dual do disco. Se a comunicação ainda não for possível, o secundário assume que o servidor primário falhou e passa a controlar o disco.

O HA-NFS usa *timeouts* para o que o servidor secundário descubra se o primário está em falha ou se apenas está muito ocupado. Um servidor muito ocupado pode não estar apto a responder dentro do *timeout* especificado e provavelmente apareça como desabilitado a fornecer o serviço.

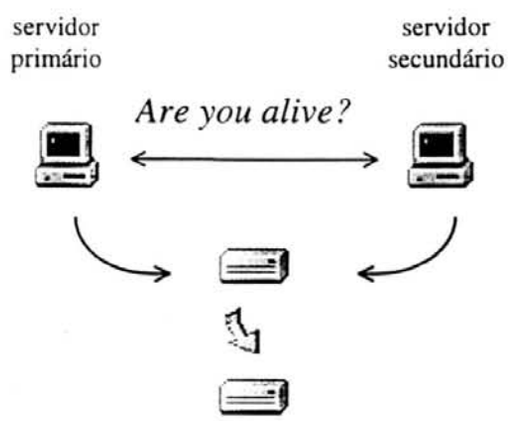

FIGURA 2.4 - Servidores com acesso ao disco espelhado

O servidor secundário também monitora a execução dos seus próprios serviços gerenciando a ocorrência de falhas e qualquer serviço de dados que está sendo executado. Se o servidor secundário detecta um problema com suas tarefas, pode requerer verificação e pedir o seu próprio *backup*, que é neste caso o servidor primário. O encaminhamento dos procedimentos de recuperação é evitado em qualquer situação onde o servidor secundário não pode verificar a disponibilidade dos seus próprios serviços. Desta forma o HA-NFS evita que um sistema não habilitado restaure a integridade do serviço de dados.

No HA-NFS, quando um servidor falha, a recuperação é transparente para os clientes. O serviço que era oferecido por um servidor em falha é substituído pelo serviço de um outro servidor disponível que recebeu o nome e o endereço IP do servidor falho. As aplicações do cliente não precisam ser modificadas e nenhuma intervenção no cliente é necessária para completar o serviço de recuperação.

---

[10] para implementar uma aproximação de armazenamento estável (onde a informação nunca é perdida), é necessário *espelhar* informações em diversos meios não-voláteis (geralmente discos)

[11] mensagem de reconhecimento

Quando o servidor secundário detecta uma falha, um mecanismo de recuperação apropriado é selecionado. Um processo transfere o NFS do servidor falho para o servidor *backup* [BHI91]. O servidor com falha é isolado e não pode interferir no sistema.

O HA-NFS não trata falhas de *crash* nos clientes. Um cliente com falha terá que ser recuperado manualmente. Neste caso a ocorrência de falha é isolada[12]. A falha não altera a disponibilidade ou a integridade dos dados em um outro cliente. Contrastando, falha em servidor pode afetar vários clientes. Falha em cliente afeta apenas uma estação e todas as transações que foram ali originadas.

O tempo de bloqueio[13] é nulo. O tempo de detecção de falha[14] depende do intervalo entre a troca das mensagens *Are you alive?*, do número de vezes que a mensagem está sendo enviada até que a falha seja detectada e do tempo necessário para utilizar o disco como um canal entre os servidores [BHI91].

2.4 Eden File System

O *Eden File System* [PU88] é um sistema distribuído experimental de replicação a nível de objeto que utiliza uma LAN [TAN94]. Os objetos do *Eden* podem ser processos, dados ou procedimentos. A principal característica deste sistema é que as cópias de arquivos perdidas podem ser regeneradas se pelo menos um servidor com cópia do arquivo está em operação normal.

O *Eden* emprega a abordagem das réplicas ativas, que o difere de sistemas como o HA-NFS e o RNFS [LEB96]. O sistema utiliza semânticas simples como as do UNIX para a comunicação de objetos locais e remotos. A diferença é que os objetos do *Eden* podem migrar de uma estação para outra. Portanto a amarração entre clientes e servidores deve ser realizada antes de cada requisição. Para o usuário, a amarração é realizada de forma transparente. Cada objeto do *Eden* é associado a uma *capability* que é formada por um identificador único e um lista de direitos de acesso. A *capability* define as operações que um cliente pode requisitar ao objeto.

O *Eden* permite seleção de objetos arbitrários para serem replicados, escolha do número de cópias para cada objeto e troca de cópias entre servidores operacionais. Cópias podem se tornar inacessíveis devido a falhas em servidores, mas se ao menos um dos servidores com a cópia está operacional, o nível de replicação é restaurado automaticamente utilizando-se o algoritmo da regeneração [PU88].

A idéia básica do algoritmo de regeneração consiste em criar cópias novas e acessíveis para substituírem as cópias inacessíveis na estrutura do diretório. A cópia é feita alterando-se a configuração dos dados do diretório de forma que os dados sempre indiquem onde estão as cópias atuais. Como os diretórios intermediários são também replicados pela mesma técnica, é necessário aplicar este método recursivamente. O algoritmo é composto por duas partes com a finalidade de englobar os dois tipos de operações possíveis: leituras e escritas.

Quando a operação é uma leitura, as cópias atuais do diretório são requisitadas. Uma requisição de leitura é difundida (*broadcast*) às cópias até que uma delas responda ao pedido. Se nenhuma das cópias responde ao pedido, a operação é abortada.

---

[12] afeta apenas um usuário
[13] *blocking time*
[14] *failover time*

Quando a operação é uma escrita, uma mensagem é enviada a todas as cópias do diretório requisitando a escrita. Se a requisição falha em todas as cópias, a operação é abortada. Se a operação ocorre em algumas cópias, mas não atingiu um número suficiente, novos servidores operacionais são alocados para conter as novas cópias, até que o número especificado seja atingido. Se não foi possível localizar servidores operacionais ou não foi possível criar o número de cópias necessárias, a operação é abortada. Caso contrário a operação de criação de novas cópias retorna o endereço das novas cópias para o diretório imediatamente superior [PU88].

Este método possibilita que, se um servidor possui uma cópia disponível, isso já é suficiente para torná-la disponível. A estratégia de replicação empregada no Eden é *lazy* [LAD90]. as novas cópias são geradas arbitrariamente. Como este método de recuperação envolve a criação de novas cópias, quando um servidor se recupera após uma falha e se reintegra ao sistema, é necessário realizar a coleta de lixo para eliminar as cópias obsoletas.

2.5 Reliable Network File System

O RNFS ou *Reliable Network File System* [LEB96] é um sistema experimental que utiliza o mecanismo de cópia primária. A distribuição das escritas do servidor primário para os servidores secundários é feita de forma síncrona: cada servidor secundário recebe operação de escrita do servidor primário por vez. Um novo procedimento de escrita só começa depois que o anterior acabar.

Durante a operação normal do RNFS, os clientes fazem suas requisições ao servidor primário. Em caso de falha, os clientes são capazes de eliminar o servidor primário e desencadear um procedimento para substituí-lo, permitindo continuidade no acesso aos dados [LEB96]. O projeto trata falhas de particionamento de rede e a reintegração do servidor, em caso de falha, deve ser realizada de forma automática.

O RNFS prevê mecanismos para tolerar falhas simultâneas, procurando manter compatibilidade com o NFS. Os mecanismos relacionados à replicação e à recuperação de falhas do sistema são transparentes para o usuário. Nos próximos itens, estes mecanismos básicos do RNFS serão revisados.

2.5.1 Mecanismo de replicação do RNFS

O ambiente do RNFS é composto por uma rede de estações com poucos servidores e muitos clientes. Os servidores, que executam o protocolo RNFS e o serviço NFS normal, podem conter vários volumes replicados. O sistema permite a definição de quais volumes devem ser replicados. A escolha do grau e localização das réplicas dos volumes em servidores é feita pelo administrador.

Cada volume replicado forma um conjunto de replicação. Os servidores que contêm os conjuntos de replicação formam um grupo de replicação. Podem existir diversos conjuntos de replicação simultaneamente. Cada servidor pode participar de mais de um grupo de replicação. Um cliente pode estar conectado a vários grupos de replicação. Os conjuntos permitem o mascaramento de falhas e a recuperação automática do sistema de arquivos de servidores.

Os servidores de um grupo de replicação estão divididos em primário e secundários. A atribuição da condição de primário ou secundário é inicialmente estabelecida com base em um

arquivo de configuração, mas pode mudar ao longo do uso do sistema. Diferentes conjuntos de replicação permitem que se tenha diferentes servidores principais para diferentes volumes, evitando que este seja um gargalo.

Cada servidor, para cada grupo de replicação em que participar, estará em um determinado estado associado a um valor (tab. 2.4). Cada servidor mantém uma tabela de estados contendo seu próprio estado e o estado suposto sob seu ponto de vista dos demais membros do grupo. Esta tabela é utilizada por vários algoritmos de controle do RNFS. As tabelas são armazenadas nos servidores em memória volátil.

TABELA 2.4 - Valores possíveis para estados de servidores

Menmônico	**Valor**	**Significado**
PRIMÁRIO	0	Servidor mestre operacional
SECUNDÁRIO	1	Servidor secundário em operação normal
FALHA_HARD	2	Servidor em falha física
FALHA_SOFT	3	Erro no sistema de arquivos remoto
RECUPERAÇÃO	4	Servidor está sendo recuperado ou reintegrado

Fonte: LEBOUTE. Um sistema de arquivos distribuídos tolerante a falhas para o UNIX. p.51.

Os clientes possuem um registro em memória volátil de qual é o servidor primário suposto para cada conjunto de replicação que o cliente estiver conectado. Esta tabela denomina-se tabela de mestres.

2.5.2 Algoritmos cooperantes do RNFS

O RNFS utiliza um conjunto de algoritmos cooperantes para manter o sistema. Esses algoritmos são responsáveis pelo acesso aos serviços de arquivos, repetição de escritas entre servidores, sinalização e monitoramento de atividade dos servidores, escolha de novo servidor primário e reintegração de servidores. A reintegração de servidores é o assunto do próximo capítulo.

O protocolo RNFS foi projetado como uma extensão do protocolo NFS. Novas RPCs foram adicionadas ao protocolo NFS existente para permitir maior confiabilidade e disponibilidade de acesso (tab. 2.5).

TABELA 2.5 - Procedimentos RPC adicionados ao protocolo

Número	**Nome**	**Natureza**	**Pars. Entrada**	**Pars. Saída**
Descrição				
18	IMALIVE	*Broadcasting*	*server_id*	nenhum
Sinal periódico de sinalização de atividade.				
19	REQ_MASTER	*Broadcasting*	*volume_id*	*req_master_result*
Requisição de novo servidor primário a partir dos clientes.				
20	ELECTION	*Unicasting*	*election_in*	*election_out*
Pacote de inquirição usado no algoritmo de eleição.				
21	REQ_RECUP	*Unicasting*	*server_id*	*req_recup_result*
Usado no algoritmo de recuperação de servidores.				

Fonte: LEBOUTE. Um sistema de arquivos distribuídos tolerante a falhas para o UNIX. p.56.

Acesso aos serviços de arquivos

Cada cliente RNFS executa dois algoritmos interligados. Um destes algoritmos é responsável pelo acesso ao serviço e o outro pelo cancelamento de servidor em caso de falha. O algoritmo de acesso ao serviço do RNFS é praticamente o mesmo do NFS. A diferença é que, após determinado *timeout*, o servidor é invalidado e o cliente entra em estado de busca de novo servidor primário.

Para localizar um novo servidor primário, o cliente difunde uma mensagem REQ_MASTER com a identificação do volume. Cada servidor que receber esta mensagem verifica se possui uma cópia do volume e, portanto, se pode ser o novo servidor primário. Em caso afirmativo, retorna sua identificação. Em caso negativo, nenhum retorno é enviado.

Quando o cliente recebe a resposta[15], o servidor que a enviou é anotado como novo servidor primário do volume. Então o cliente enviará os próximos acessos para este servidor. Depois que o novo primário envia a resposta ao cliente, difunde uma mensagem B_VISION com sua visão da rede.

Difusão de escritas no RNFS

O procedimento de difusão de escritas do RNFS (fig. 2.5) é responsável por manter o sistema consistente além de servir para atualizar a tabela de estado dos servidores. O RNFS trata o procedimento de difusão de escritas de forma síncrona. Um novo procedimento de escrita só começa depois que o anterior acabar [LEB96].

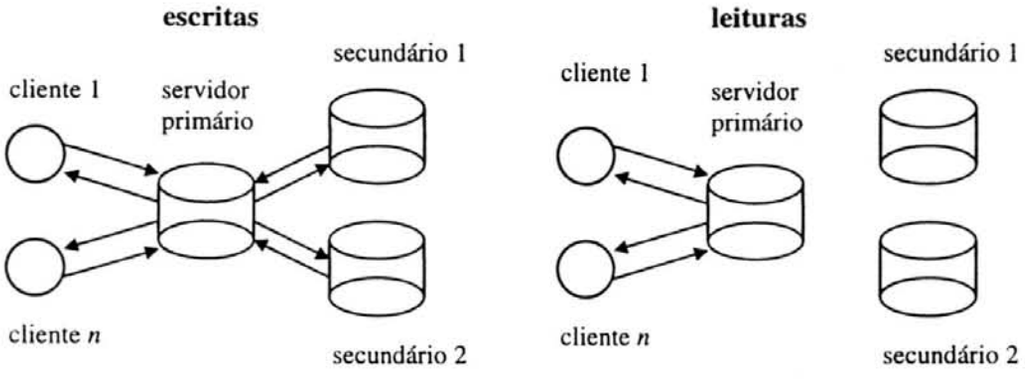

FIGURA 2.5 - Algoritmo de replicação com cópia primária [LEB96]

Quando um cliente solicita um pedido de escrita, o servidor primário altera a sua cópia do arquivo e repassa as alterações aos servidores indicados em sua tabela de estados como SECUNDÁRIOS. As alterações são repassadas aos secundários sempre na mesma ordem. Cada servidor secundário processa a escrita e responde ao primário com um *flag*, indicando que a operação foi realizada. Depois que o primário recebeu o *flag* de um secundário, pode solicitar o mesmo serviço em outro servidor secundário. Quando o primário receber a resposta do último secundário, responde ao cliente dizendo que a alteração foi processada. Durante o procedimento de difusão, o cliente espera bloqueado pela resposta (fig. 2.6). Desta forma, os procedimentos de RPC aparecem aninhados.

[15] para que o cliente não espere indefinidamente, um tempo de espera máximo é definido para este processo

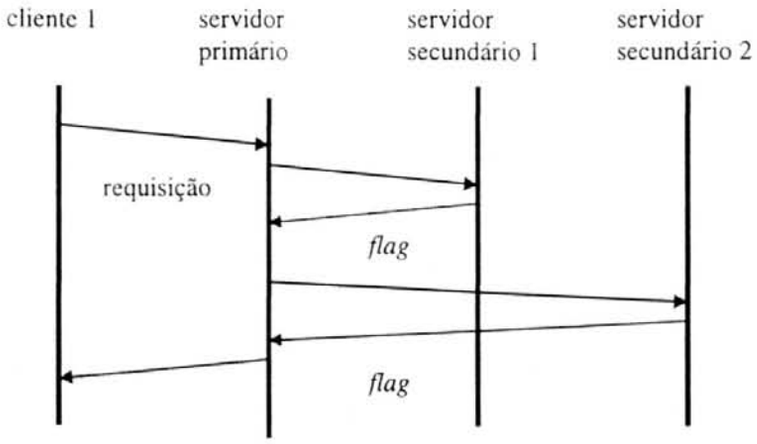

FIGURA 2.6 - Aninhamento de RPCs de escrita

O *flag* é utilizado para atualizar a tabela de estados do primário. Servidores assinalados como SECUNDÁRIO na tabela de estados e que não respondem às solicitações do servidor primário, recebem novas solicitações. Se as novas solicitações não são atendidas, o servidor é marcado como FALHA_HARD na tabela de estados e deixa de receber mensagens do primário. O retorno do estado de FALHA_HARD só pode ser tratado por um administrador do sistema.

Sinalização e monitoramento de atividade dos servidores

O algoritmo de sinalização de atividade do servidor é usado para começar a reintegração de um servidor que esteve desligado por falha física, falta de energia ou desconexão da rede. Este algoritmo consiste na emissão da mensagem IMALIVE[16] pelo servidor recém conectado e na recepção desta mensagem por todos os servidores do sistema. O resultando desta troca de mensagens é a atualização da tabela de estados dos servidores: o servidor em reintegração que possuía o estado de FALHA_HARD ou FALHA_SOFT é alterado para RECUPERAÇÃO.

Escolha de novo servidor primário

A escolha de novo primário é feita através do algoritmo de eleição que também é responsável por resolver possíveis inconsistências como perda de mensagens. Em caso de particionamento, apenas a parte da rede com a maioria dos servidores está apta a realizar a escolha do novo servidor primário. O algoritmo de eleição especificado para o RNFS é uma adaptação do algoritmo de eleição por domínio descrito em Garcia-Molina [GAR82].

2.5.3 O RNFS em presença de falha

O RNFS está sujeito a perda de mensagens que podem ser controladas pelos protocolos já existentes no NFS. O protocolo UDP-IP realiza detecção de erros, assegurando que apenas mensagens íntegras são entregues. Assim, falhas em RPCs envolvem apenas a

[16] *I'm alive*

perda da mensagem de requisição do serviço ou da mensagem de retorno do servidor. Quando uma destas mensagens for perdida, a retransmissão poderá ser solicitada pelo uso de *timeouts* [LEB96]. A única possibilidade de exceção é a perda repetitiva da mensagem de retorno, além do máximo de retransmissões admitidas. Neste caso, o servidor será dado como falho.

Para manter a consistência apesar de falha em servidor, é necessário utilizar um protocolo que mantenha a atomicidade e a ordenação das operações [GUE97] do sistema distribuído. O projeto do RNFS prevê o uso de um protocolo de concordância de duas fases [JAL94] em um modelo restrito de falhas.

Na primeira fase, o servidor primário envia uma mensagem aos secundários com a solicitação de serviço que é anotada em memória estável. Todos os secundários que estiverem em condições de realizar o serviço respondem ao primário. Na segunda fase, se o primário coletar um número suficiente de respostas, envia a mensagem de *commit*[17] aos secundários. Caso contrário, a mensagem enviada é pada abortar a operação.

O protocolo de duas fases é suficiente quando o sistema está livre de falhas e para vários tipos de falhas, que sejam sinalizadas por discordância dos servidores [LEB96]. Alguns casos de falhas, que levam à perda de mensagens, exigem uma terceira fase denominada protocolo de terminação. Nesta fase, os servidores que não receberam a conclusão da operação devem consultar outro servidor. Se isso não for possível, o servidor deve ser colocado como inativo e aguardar a recuperação.

2.5.4 O estado atual do RNFS

O RNFS começou a ser desenvolvido a partir de março de 1996 no grupo de tolerância a falhas do CPGCC-UFRGS. A especificação está praticamente completa. Foram propostos os mecanismos de funcionamento geral do sistema, difusão de escritas com atomicidade apesar de falha, algoritmo de eleição de novo servidor e monitoração de ambiente [LEB96].

O estado atual da implementação, porém, ainda não é o desejável. Muitos módulos ainda precisam ser implementados para tornar o sistema operacional. O único módulo parcialmente implementado é a difusão de escritas [AGN96].

---

[17] mensagem de confirmação

3 Reintegração de Servidores

A reintegração de servidores falhos é um mecanismo presente principalmente em sistemas que empregam réplicas ativas [JAL94]. Porém, antes de começar a reintegração, a falha no servidor precisa ser tratada.

Conforme visto no capítulo anterior, a reintegração automática de um servidor após falha garante a disponibilidade e o desempenho esperados em um sistema. Apesar disso, a maior parte dos sistemas não trata ou trata superficialmente o problema da reintegração de servidores. Isso acontece porque projetistas de sistemas optam por realizar a reintegração ou o acréscimo de novo servidor manualmente.

Schneider [SCH93] relata uma estratégia para reintegrar um objeto reparado em um sistema usando réplicas ativas. Este objeto (que pode ser um servidor) que está sendo reintegrado precisa recuperar o estado consistente com os demais elementos do sistema. Em se tratando de reintegração de servidor em cópia primária, há informações restritas sobre o assunto. A bibliografia trata principalmente pontos de falha e como manter atomicidade em caso de falha do primário.

Um sistema que trata o acréscimo de servidor é o Zebra [HAR 95]. O Zebra provê alta confiabilidade através da divisão de arquivos em pedaços que são armazenados em diferentes servidores. A reintegração de servidor neste sistema envolve a participação dos clientes e a notificação de outros componentes.

O *Locus* [WAL83] também trata a reintegração, relacionada ao tratamento de falha de particionamento de rede. Este sistema emprega um vetor de versão para resolver as inconsistências de diferentes versões de arquivos. Se o sistema não conseguir resolver as inconsistências, será necessária a ajuda do usuário. Outro sistema que permite resolver as inconsistências manualmente em caso de particionamento é o *Coda* [SAT93]. Depois que um servidor é reconectado a rede, os objetos inconsistentes - os quais o sistema não pode corrigir - são disponibilizados como *read-only* para o usuário. Adicionalmente, o RNFS[LEB96] prevê a reintegração automática de servidores sem a interferência do usuário.

Sistemas como o *Locus*, *Coda* e RNFS têm em comum os benefícios do tratamento de falhas de particionamento de rede, apesar de não possuírem um mecanismo transparente para tratar estas falhas. Além disso, a inexistência de um mecanismo automático para tratar a reintegração de servidor compromete a alta disponibilidade durante o funcionamento destes sistemas.

3.1 Fases do Sistema Distribuído Replicado

Pode-se distinguir duas fases no funcionamento normal de um sistema distribuído replicado (fig. 3.1): fase normal plena e fase normal degradada. Neste sistema, dois procedimentos básicos podem ser realizados com servidores durante sua vida útil: reintegração e confinamento.

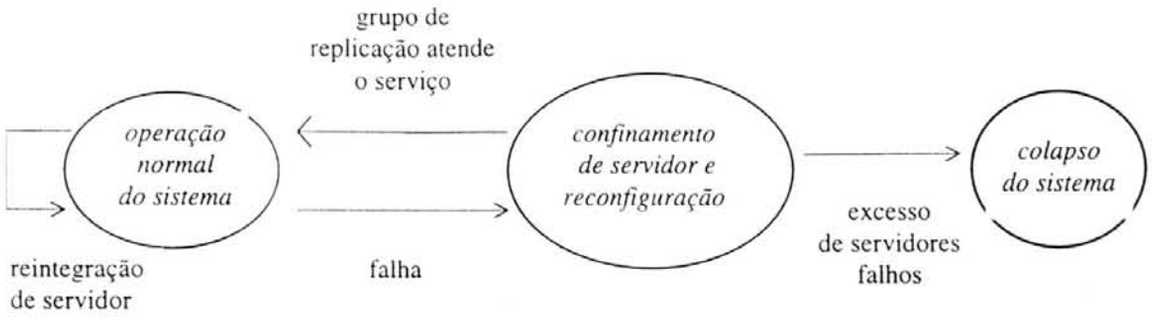

FIGURA 3.1 - Fases em um sistema com grupo de replicação

Durante a fase normal - plena ou degradada - as servidoras de um grupo de replicação estão sujeitas a falhas. Para manter o serviço com alto grau de tolerância a falhas, o grupo de replicação suporta duas operações: *join* e *leave* [BIR 96]. Quando uma falha é detectada em um servidor, o grupo realiza um *leave* iniciando um procedimento que confina o servidor. O sistema perde um servidor e precisa de mecanismos para restaurar o serviço, reconfigurando o grupo de replicação e garantindo que todo o grupo recebeu e realizou atomicamente a última requisição do cliente. O sistema pode requisitar a substituição ou reparo do servidor perdido. Um mecanismo de detecção de falha em um sistema replicado por cópia primária pode ser encontrada em Guerraoui [GUE97], e não será tratado neste artigo.

Depois do confinamento do servidor falho e da reconfiguração dos servidores, o sistema se encontra na fase normal degradada, onde a capacidade de tolerar falhas é reduzida em relação a fase normal plena. Um sistema robusto deve suportar determinado número de falhas subsequentes para não entrar em colapso. Esta capacidade está associada ao número de cópias de arquivos disponíveis nos servidores do sistema.

Quando a falha é recuperada, o servidor pode ser reintegrado ao grupo de replicação, através da execução de um *join*. O *join* mantém ou aumenta a quantidade de servidores do grupo, não permitindo que a tolerância a falhas do grupo decresça.

3.1.1 Fase plena

Durante a fase normal plena, sistemas de arquivo distribuídos estão sujeitos a falhas de *crash* ou a falhas de conexão entre servidores que podem ocasionar particionamento de rede. Por isso, estes sistemas devem prover múltiplas cópias armazenadas em diferentes servidores e guardar informação para recuperar seu estado normal em caso de falha.

O resultado mais indesejável que falhas em servidores de arquivos podem causar é a perda de informações. Por isso é importante que o mecanismo de tolerância a falhas preserve os dados dos arquivos, mesmo que o serviço seja temporariamente suspenso. Com esta finalidade, o sistema usar mecanismos preventivos, executados quando o sistema está *livre de falhas*, e de mecanismos a serem seguidos após a ocorrência de falhas (na fase degradada). Os mecanismos preventivos devem assegurar informações suficientes para permitir a recuperação de falhas. Os mecanismos executados após a ocorrência de falha - na fase degradada - devem assegurar a consistência do sistema de arquivos.

3.1.2 Fase degradada

A fase degrada começa quando ocorre a detecção de falha, que pode ser feita por processos monitores e por *timeouts*. Os processos monitores controlam periodicamente o sistema. O mecanismo de *timeout* pode ser usado para detectar falhas durante a difusão de escritas na cópia primária. Se um cliente faz uma requisição para o primário que não responde dentro do *timeout* determinado, então o cliente assume que o primário falhou. Se o primário faz uma requisição para um secundário que não responde dentro do *timeout*, então o primário assume que o secundário falhou e precisa ser isolado. Um procedimento recuperador torna-se necessário para restaurar o sistema em um estado consistente [SIL93] na fase degradada. Este procedimento deve ser capaz de:

a) isolar o servidor falho do grupo de replicação. Quando um servidor é isolado, o grupo de replicação precisa ajustar sua configuração para continuar a fornecer o serviço. Se o grupo for gerenciado por cópia primária, pode ser necessário escolher um novo coordenador;

b) garantir durabilidade e atomicidade aos demais servidores no grupo de replicação. Atomicidade significa que todas as ações associadas a uma operação precisam ser realizadas por completo mesmo em presença de falhas, ou não devem ser realizadas. Em um sistema de arquivo distribuído, quando uma cópia foi alterada, todas as suas réplicas também devem ser alteradas ou então a operação deve ser abortada. A durabilidade implica na manutenção dos arquivos em memória estável nos servidores;

c) armazenar informações em memória estável e permitir a recuperação do servidor falho;

d) reorganizar o arquivo de recuperação para prover agilidade no procedimento de recuperação.

Adicionalmente, podem ocorrer falhas durante o procedimento recuperador. Um projeto robusto pode prover estratégias para o controle destas falhas. A fase degradada acaba quando o sistema termina o procedimento recuperador. Então o sistema volta a operar na fase normal, porém com a capacidade de tolerar falhas está reduzida pois o sistema perdeu um servidor.

A principal característica da fase degradada é a *perda* de qualidade no serviço, pois o sistema conta com um número menor de servidores. Essa *perda* ou *degradação* pode refletir ou não no desempenho e na tolerância a falhas, dependendo de como o sistema é organizado e da configuração de cada servidor. Quando ocorre falha no servidor primário, se o primário é o servidor mais rápido do grupo então, possivelmente, o sistema irá sofrer perda de desempenho. Como o primário falhou, terá que ser escolhido um novo primário (que pode ser o mais rápido dos secundário, por exemplo).

No RNFS, se a falha ocorrer em servidor secundário, o usuário até poderá notar ganho de desempenho (o que não deveria ocorrer pois o servidor replicado deve ter o mesmo desempenho do servidor não replicado do NFS) pois a difusão de escritas que era realizada em n servidores, agora é realizada em n -1 servidores. Neste caso, o sistema pode até ganhar em

desempenho, mas vai perder na capacidade de tolerância a falhas (que depende do número de componentes do sistema e de quanto estes componentes são confiáveis) pois os dados estão sendo replicados em um local a menos.

3.2 Reintegração de servidor RNFS

A reintegração de servidor em um sistema distribuído corresponde ao *join* do servidor no grupo dinâmico, ou seja, é o mecanismo de juntar ou integrar um servidor a um grupo de replicação. Este mecanismo possui especial importância em um sistema tolerante a falhas, pois é essencial para que o sistema seja usável a longo prazo. A reintegração obedece às seguintes etapas:

a)protocolo de início da reintegração;

b) protocolo de atualização do servidor (PAS): responsável por atualizar o sistema de arquivo do servidor em reintegração, liberar o espaço do arquivo de recuperação e integrar o servidor ao grupo de replicação ou aceitar um novo servidor;

c)protocolo de término da reintegração.

O ideal é que essas etapas sejam realizadas por um mecanismo automático ou semi-automático. A volta da energia pode causar partida automática de servidores, mesmo que isso não seja habitual, sugerindo o começo automático da reintegração. O que ocorre habitualmente é que o administrador interfere diretamente, corrigindo alguma situação causadora de falha. Mesmo neste caso é desejável que a reintegração comece imediatamente após o término da manutenção, sem necessidade de solicitação do administrador [LEB96].

Um ponto importante, relativo à reintegração é a necessidade de estabelecer um controle em relação a seu início e término. Além disso, precisa ser determinado qual servidor pode servir como fonte de dados para recuperar outro servidor.

Em se tratando de servidores RNFS, aqueles que estiverem nos estados PRIMÁRIO ou SECUNDÁRIO conterão sempre cópias atualizadas de seus volumes replicados [LEB96]. A eleição do novo servidor primário deve ocorrer entre servidores que apresentem estados atualizados. Um servidor primário conterá sempre uma cópia atualizada de todos volumes de replicação que gerenciar. Portanto, o servidor primário corrente será sempre considerado incondicionalmente como fonte de dados.

A reintegração começa pela execução de um protocolo de inicialização entre o servidor primário e o servidor a ser reintegrado. No final deste protocolo, os dois servidores estarão no estado RECUPERAÇÃO. Esses servidores sairão deste estado apenas ao término do PAS (Protocolo de Atualização do Servidor). Adicionalmente, como o servidor primário é utilizado como fonte de dados, a realização do PAS pode ocorrer em paralelo com acessos normais de clientes.

3.3 Protocolo de início da reintegração

O protocolo de início da reintegração ocorre depois que um servidor é conectado fisicamente à rede. Quando a estação é religada, difunde (*broadcast*) uma mensagem com o pedido de reintegração ao sistema. O servidor primário responde à requisição e inicia-se o PAS (Protocolo de Atualização do Servidor). Um protocolo adicional será utilizado como término da reintegração. A reintegração de qualquer servidor passa inicialmente pelo protocolo de início da reintegração (fig. 3.2).

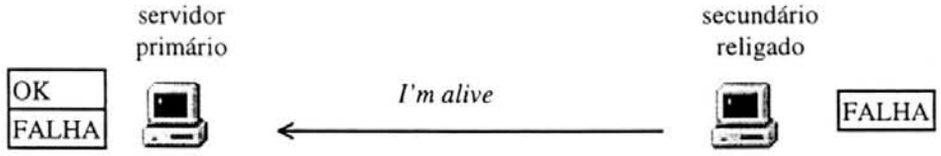

FIGURA 3.2- Servidor avisando que está pronto para reintegrar [LEB96]

O servidor primário monitora os pacotes de IMALIVE emitidos por todos os servidores, inclusive os recentemente religados. Quando um servidor que emite o IMALIVE está em estado de FALHA na tabela de estados do primário, o primário tenta iniciar o PAS. O primário envia uma chamada REQ_RECUP (fig. 3.3) ao secundário, solicitando resposta para o começo do PAS. Caso o secundário responda, a atualização é iniciada imediatamente. Caso o servidor não responda, a atualização voltará a ser tentada na recepção do próximo pacote de IMALIVE e o estado anterior é mantido para o principal.

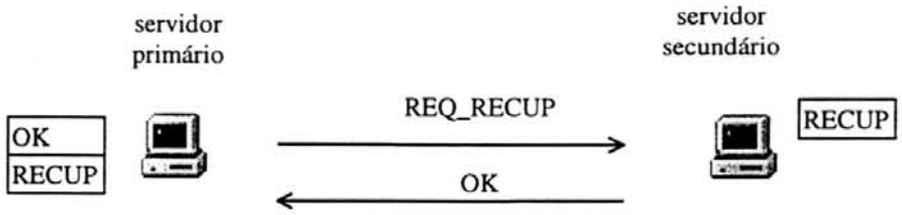

FIGURA 3.3- Servidores concordando para o início da reintegração [LEB96]

Quando o protocolo de inicialização termina, os dois servidores podem executar um PAS escolhido (fig. 3.4). No final do qual, se terminado com sucesso, o servidor recuperado voltará ao seu estado de servidor secundário.

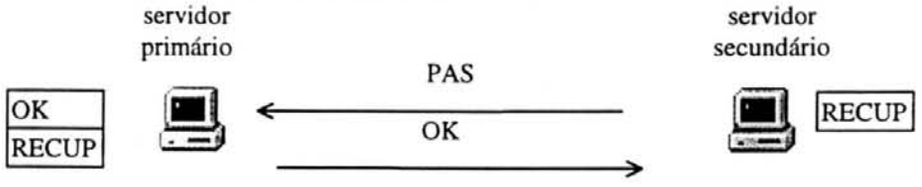

FIGURA 3.4- Protocolo de atualização do servidor [LEB96]

O protocolo de início da reintegração do servidor RNFS assume que rede sempre transporta mensagens íntegras através do protocolo UDP-IP e tolerante a perdas das mensagens através da transmissão periódica de IMALIVEs [LEB96].

3.4 Protocolos de atualização do servidor

O protocolo de atualização do servidor ou PAS restaura os arquivos do servidor com a última versão (versão *up-to-date*) dos dados armazenada em memória permanente. Após a eleição de novo primário, quando necessário, e do início da reintegração, é utilizado um PAS para recuperar (atualizar) o sistema de arquivos do servidor.

O procedimento básico do PAS pode ser resumido na tentativa do servidor primário abrir um arquivo com a versão *up-to-date* em um servidor secundário. Se o servidor secundário possuir o arquivo com a versão *up-to-date*, então o servidor primário tenta ler um outro arquivo a ser conferido. Se o secundário não possui o arquivo, o primário faz uma requisição de um RPC que escreve o arquivo no secundário. O secundário então atualiza a sua versão do arquivo.

Leboute [LEB96] sugere três diferentes PASs: transferência de volumes, cópia de arquivos e *log* de operações. O uso desses procedimentos é especificado para o RNFS mas nenhum foi efetivamente implementado.

3.4.1 Transferência de volumes

A transferência de volumes é o método mais simples para realizar a atualização do sistema de arquivos de um servidor. Depois que dois servidores decidam (através do algoritmo de início da reintegração) que um volume deve ser recuperado, o volume atual é copiado completamente para o volume desatualizado [LEB96]. Para essa operação, foi utilizado um algoritmo de cópia recursiva. Esse algoritmo percorre a árvore de diretórios e transfere todos os arquivos do servidor atualizado para o servidor desatualizado.

Este método possui a vantagem de poder ser implementado usando os procedimentos RPC já existentes no protocolo NFS e serve como abordagem *default* para o PAS.

A transferência de volumes é indicada quando um novo servidor é inserido no grupo e precisa atualizar completamente o seu sistema de arquivos, ou para corrigir desatualizações geradas por falhas de longa duração.

3.4.2 Cópia de arquivos

O algoritmo de transferência de volumes é claramente ineficiente quando ocorrem falhas de curta duração. A quantidade de arquivos desatualizados no servidor a ser recuperado será pequena quando comparada ao número total de arquivos existentes no volume replicado. Este algoritmo pode ser otimizado se forem transmitidos apenas os arquivos que foram alterados enquanto o servidor estava em falha.

Leboute sugere que a atualização por cópia de arquivos seja realizada com a implementação de um algoritmo recursivo semelhante à atualização por transferência de volumes, mas que utilize algum tipo de comparação para cada arquivo ou diretório. Por isso este método é também chamado de atualização diferencial.

A atualização diferencial é indicada para servidores que sofreram falhas de curta duração ou permaneceram temporariamente desligados. A dificuldade deste método é encontrar um critério para a comparação, que distingua quais arquivos estão desatualizados e

devem ser transmitidos de um servidor a outro. São apresentadas duas alternativas para os critérios de comparação: número de versão ou campo de controle de data.

Números de versão de arquivos

Alguns sistemas empregam números de versão para controlar a consistência dos arquivos. É o caso do Locus [WAL83]. O número de versão é um valor nulo quando o arquivo ou diretório é gerado. A cada atualização este valor é incrementado, refletindo o número de vezes que o arquivo foi modificado.

Como o NFS não comporta números de versão, para implementar o RNFS, a camada inferior ao sistema de arquivos deve ser modificada. Isso pode ser feito acrescentado ao *inode* do arquivo mais um campo para conter o valor da versão. Então este valor poderá ser acessado através da chamada de sistema UNIX *stat*. A desvantagem de modificar o *inode*, acrescentando-lhe mais um campo, é que pode ocorrer alguma inconsistência no sistema operacional ou no NFS.

A escrita do arquivo durante o PAS deverá ser uma operação atômica para garantir que o número de versão reflita a última operação realizada com sucesso. O ideal é usar uma estratégia que primeiro atualiza o arquivo e depois incrementa o número de versão. Esta estratégia, aliada às operações idempotentes do NFS, pode resultar em um protocolo muito simples para o RNFS.

Campo de controle de tempo

Outra alternativa para controlar a consistência de informação do RNFS é usar os campos de controle de tempo da última atualização do *inode* do arquivo. A informação deste campo pode ser recuperada pela chamada de sistema UNIX *stat* como tempo de última alteração do arquivo (*file last modify time*) e pode servir como indicação da atualização pelo procedimento de difusão de escritas. O problema é como garantir que os *clocks* do sistema distribuído estão sincronizados [LEB96].

Para manter o sistema consistente apesar de falha, o RNFS realiza a aplicação das escritas nos secundários antes da aplicação no primário [LEB96]. Se o tempo da última alteração de um arquivo no secundário for inferior ao tempo do mesmo arquivo no primário mais a deriva máxima dos *clocks*, pode-se garantir que o arquivo do secundário está atualizado em relação ao primário.

A desvantagem desta alternativa é depender do algoritmo de sincronização dos *clocks*. Além da necessidade de ter este algoritmo sendo executado, a ocorrência de falhas poderá gerar outros problemas. As falhas no algoritmo sincronizador não seriam notadas pelo PAS e levariam a falhas de consistência graves.

Outra dificuldade desta alternativa é que a deriva máxima garantida dos *clocks* deve ser menor que o tempo de retorno de uma operação de um servidor secundário para um primário. Para que o sistema permita maior desempenho esse tempo deve ser bastante curto.

3.4.3 Retenção de *log*

O método de retenção de *logs* aloca um espaço no disco do servidor primário como *cache* de operações para recuperar sistemas de arquivos de servidores secundários falhos ou temporariamente desligados [LEB96]. Sempre que uma operação solicitada por um cliente não for transmitida para um servidor secundário, será anotada na *cache*. Essa *cache* servirá como *log*.

Quando a falha no servidor secundário for corrigida, o servidor poderá ser reintegrado ao sistema utilizando o *log*. Após a execução do protocolo de início da reintegração, o primário passa a transmitir ao secundário todas as operações perdidas, tornando-o atualizado. Devido à necessidade de manutenção do *log*, esta estratégia limita-se ao tratamento de falhas de curta duração [LEB96].

Conceitualmente, o *log* pode ser uma FIFO (*First In First Out*) onde os registros são adicionados no início. Se muitos registros forem adicionados, o *log* pode-se tornar muito longo por conter informação desnecessária. Para controlar o tamanho do *log*, sistemas como o XEL (*eXtended Efhemeral Logging*) [KEE97] usam coletores de lixo.

A desvantagem da retenção de *log* é justamente limitação do espaço em disco para a gravação da FIFO. A FIFO concorre com outros arquivos no espaço em disco do servidor e precisa ter tamanho máximo definido. Quando acabar o espaço disponível, a única alternativa parece ser descartar todas as operações gravadas e utilizar o método de cópia completa [LEB96]. O sistema de arquivos *Harp* [LIS91a] considera a possibilidade de solicitar a gravação em fita dos dados quando este se aproximar de estar lotado.

O RNFS pode comportar uma implementação que controla registros supérfluos através da comparação da última operação armazenada para reduzir o tamanho do *log*. Alternativamente, uma proposta baseada na compressão do *log* [KEE97] pode ser utilizada para eliminar informação desnecessária. A desvantagem de realizar a compressão do *log* é que novos registros não podem ser adicionados no *log* durante a realização da compressão. Métodos para a compressão de *log* para o RNFS podem ser implementados baseados em Kaunitz [KAU84].

O método de retenção de *log* proposto para o RNFS tem a vantagem de guardar poucas informações no *log*. Apenas operações e nomes de arquivos sobre os quais as operações são realizadas são guardados. O espaço ocupado no disco, portanto é bem restrito. O resultado final é que o PAS torna-se um procedimento rápido, e pouco custoso.

Como pode-se observar, o *log* não é a única solução possível para recuperar um sistema tolerante a falha, embora seja a solução mais empregada por razões de desempenho e eficiência [KEE97], principalmente em sistemas de banco de dados.

3.5 Uso de métodos compostos

O método final a ser implementado no sistema poderá ser uma composição de métodos. Leboute sugere iniciar o PAS com o algoritmo de transferência de volumes. Esse método deverá ser mantido como *default*, a ser chamado quando não for possível o uso de métodos mais eficientes. Este método é recomendado quando há grande quantidade de volumes a ser recuperada.

Quando o sistema não contar com números de versão, Leboute sugere a implementação conjunta do algoritmo de retenção de *logs* para cobrir falhas curtas e o algoritmo de transferência de volumes para falhas que ultrapassem o limite da *cache*.

Se o sistema contar com números de versão, pode-se pensar em um método final composto pelos três algoritmos. O método final poderá começar com a retenção de *logs*. Quando o *log* não for suficiente, será executada a atualização diferencial e, em último caso, a transferência de volumes quando houver muitos arquivos a serem atualizados [LEB96].

3.6 Protocolo de término da reintegração

Após a conclusão do PAS ocorre o protocolo de término da reintegração. Este protocolo retorna um *flag* indicando o sucesso ou não da atualização do sistema de arquivo. Quando o algoritmo termina corretamente (fig. 3.5), o servidor primário anota na sua tabela que o servidor recuperado está em estado OK, e o servidor recuperado passa a participar novamente da difusão de escritas.

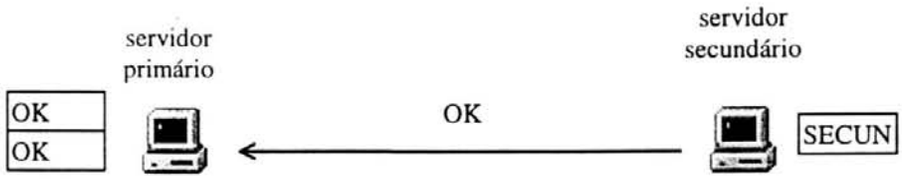

FIGURA 3.5- Término com sucesso da reintegração [LEB96]

Defeitos físicos nos discos, falhas intermitentes nos controladores, ou erro nos dados anotados para a atualização do servidor podem ocasionar falhas [LEB96] no PAS (fig. 3.6).

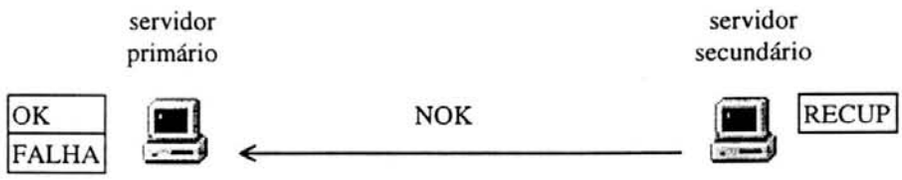

FIGURA 3.6 - Término sem sucesso da reintegração [LEB96]

Quando o algoritmo terminar com erro, o estado do servidor é anotado na tabela do primário como FALHA_SOFT. Neste caso, mensagens devem ser emitidas na console de operação, indicando que o problema deve ser resolvido manualmente por um operador [LEB96].

4 Implementação do Protocolo de Atualização RNFS

Para atender às necessidades de testes e refinamento da reintegração do servidor RNFS, optou-se por observar aspectos dos PASs (protocolos de atualização do servidor) através da implementação de protótipos. Os protótipos foram implementados na camada de aplicação usando *Sun RPC*.

O Sun RPC é projetado para comunicação cliente-servidor no *Sun Operating System* (*Sun* OS). É formado por um compilador chamado *rpcgen* e por uma linguagem de programação chamada RPCL. O Sun OS 4.1 está disponível nas estações de trabalho do Instituto de Informática da UFRGS. O Sun RPC também é disponível para a plataforma Linux (PC-DOS).

4.1 O compilador rpcgen

O *rpcgen* [COR91] é um compilador que auxilia na implementação de aplicações baseadas em serviços RPC. Esse compilador aceita uma especificação de protocolo escrita em RPCL (*remote procedure call language*), que é similar a linguagem C. A especificação do protocolo é formada por procedimentos, tipos de dados e resultados para cada procedimento. Através deste protocolo, o *rpcgen* produz código de linguagem C. O código produzido inclui rotinas de *stub* para clientes, *stub* para servidor e rotinas XDR para argumentos e resultados, além de um arquivo de cabeçalho que contém as definições comuns.

O usuário precisa escrever um programa com os procedimentos do servidor e um programa com os procedimentos do cliente em linguagem C. O programa cliente troca informações com o programa servidor usando funções da biblioteca RPC *Library*.

Os programas cliente e servidor podem ser compilados e ligados aos arquivos produzidos pelo *rpcgen* de maneira usual. Para usar o serviço remoto, o usuário escreve um programa que faz uma chamada de procedimento local chamar o *stub* do cliente produzido pelo *rpcgen*. Ligando esses programas com os *stubs* dos clientes e as rotinas XDR gera-se um programa executável. O compilador *rpcgen* também permite especificar o protocolo de transporte usado pelo servidor (neste caso o UDP, compatível com o NFS).

4.2 Projeto do protótipo com RPC

O projeto de um protótipo ou de uma aplicação com RPC engloba quatro partes principais [TAN94]: interface, cliente, servidor e protocolo.

4.2.1 O projeto da interface

O projeto da interface deve ser cuidadoso quanto à transparência, ao tratamento de exceções e vinculação (*binding*) entre cliente e servidor. As RPCs podem ser implementadas de forma não transparente, identificando-as claramente para não confundir o programador e alertar o compilador. Embora os problemas de RPC pareçam muitos na fase de projeto do

sistema, Tanenbaum [TAN94] considera que na prática falhas de *crash* são raras e grande parte dos outros problemas pode ser evitado por bons projetos de *stubs* e compiladores[18].

Stubs podem ser produzidos pelo programador ou pelo compilador como resultado da compilação. No *rpcgen*, as RPCs são escritas em C pelo programador e são identificadas por procedimentos com números de versão e argumento único [COU94]. O *rpcgen* também permite que o programador escreva as RPCs, sacrificando a transparência para dar maior liberdade de programação. O *rpcgen* produz os *stubs*, o que é mais conveniente ao programador.

Se o sistema de comunicação cliente-servidor não houver facilidades embutidas para tratar erros de RPC que não faça parte do procedimento do cliente, então cada procedimento cliente terá que testar todos os retornos de erros possíveis e terá que tratá-los de forma não transparente.

O projeto precisa definir a exportação de nomes de servidores, o modo pelo qual clientes podem escolher uma instância específica de um servidor quando existem muitas instâncias idênticas e o momento no qual a vinculação funciona. No *rpcgen*, os clientes devem especificar os nomes dos servidores [COU94].

4.2.2 O projeto do cliente

O projeto do cliente inclui temporização e tratamento de processos órfãos [TAN94]. Na especificação do RNFS, um cliente assume que um servidor falhou depois de *n* tentativas de espera pela resposta. Se um cliente descobre que o servidor primário falhou, dispara uma requisição de novo primário. Para tratar órfãos, o cliente refaz o pedido ao novo primário.

O *rpcgen* oferece a facilidade de tratamento de erro por uma biblioteca com várias rotinas. As rotinas dos clientes podem imprimir mensagens de erro ou retornar *strings* contendo mensagens de erro. As rotinas de servidor enviam mensagens aos clientes, se algum erro ocorre.

Na especificação do RNFS, uma mensagem de erro só deve ser emitida em último caso, isto é, se a falha não for sanada pelo mecanismo automático. Ainda, neste caso, o sistema deve garantir um estado consistente.

4.2.3 O projeto do servidor

A principal questão referente ao projeto do servidor é o paralelismo. Cada vez que um servidor recebe um pedido de um cliente, um novo processo é criado sugerindo duas abordagens. Na primeira abordagem, se chegarem outros pedidos de clientes antes que o primeiro seja atendido, mais processos serão criados no servidor. Estes processos são executados em paralelo, com independência uns dos outros. No cliente existe um processo único, associado a um único processo do servidor. Essa abordagem implica na desvantagem de criar um processo no servidor para cada RPC, mas permite a execução dos processos em paralelo. Em uma segunda abordagem, se chegar um pedido no servidor antes que o primeiro seja executado, este deve esperar a sua vez. A segunda abordagem é mais simples e mais

---

[18] certamente o autor (Tanenbaum) está se referindo a computadores utilizados nas grandes companhias internacionais e não aos computadores utilizados nas universidades brasileiras. Infelizmente, sabe-se que falhas de *crash* em disco de servidores são bem mais freqüentes do que se imagina

rápida se houver apenas uma chamada por vez, mas se ocorrerem várias requisições de clientes simultaneamente, o servidor poderá ser um gargalo.

Sistemas dotados de paralelismo devem prever algum mecanismo para controle de concorrência. A implementação de *locks* no servidor primário pode impedir que um arquivo que está sendo atualizado esteja sendo alterado ao mesmo instante por um cliente. Na especificação do RNFS, o servidor deve atender a todos os pedidos de cliente em paralelo, inclusive eventualmente um processo cliente de um PAS. Neste caso, para tratar a concorrência de escrita de processos, podem ser utilizados *locks* no servidor primário.

Um grande obstáculo para o servidor RNFS é alcançar desempenho superior ou compatível com o NFS, mesmo mantendo o sistema replicado. Isso poderá ser possível implementando um mecanismo de RPC eficiente. Uma forma de tornar a RPC mais rápida é executá-la na parte superior da interface com camadas de transporte e rede nulas [TAN94].

4.2.4 O projeto do protocolo

O protótipo com a implementação dos PASs (Protocolos de Atualização do Servidor) é baseado na especificação do RNFS [LEB96] e bibliografia disponível de RPC [COU94, TAN94, COR91]. Foram implementados os PAS descritos nos itens 4.3 e 4.4. Os demais itens são projetos para implementações futuras.

A realização do PAS durante a reintegração de servidor RNFS deve ocorrer em paralelo com a operação normal do sistema; isto é, com a difusão de escritas do primário para os secundários. Adicionalmente, deve ser proposto algum mecanismo para controlar a concorrência com a execução do serviço aos clientes.

4.3 Implementação da transferência de volume

O primeiro PAS a ser implementado foi a transferência de volume. Este PAS foi escolhido devido à facilidade de implementação e à existência de outros requisitos para implementar outros PASs, como o fato do NFS não comportar números de versão para arquivos. Para a implementação foi escrito um programa a nível de aplicação, que foi compilado no *rpcgen*. Este programa recebe os parâmetros do servidor e a descrição do volume a ser atualizado. Nenhuma RPC, além das disponíveis no NFS, precisou ser implementada. O PAS executa a função *transferência_de_volume* (fig. 4.1) que é encarregada de realizar a cópia incondicional.

```
transferência_de_volume(servidor.nome, cliente.nome) {

        busca os atributos de arquivo - stat
        se tipo de arquivo é diretório {
                monta nome do diretório para servidor
                cria diretório - mkdir - RPC
                abre diretório - opendir
                lê arquivo em diretório - readdir
                enquanto houver arquivo em diretório {
                        monta nome de arquivo
                        transferência_de_volume(servidor.nome, cliente.nome)
                        lê arquivo em diretório - readdir
                }
                fecha diretório - closedir
        } senão {
                abre arquivo - open
```

```
            lê dados de arquivo - read
            enquanto houver dados
                  lê dados de arquivo - read
            fecha arquivo - close
            monta nome de arquivo para servidor
            escreve o arquivo - RPC - write
      }
      retorna OK
}
```

FIGURA 4.1 - PAS por volume, executado pelo servidor primário

A função *transferência_de_volume* é um algoritmo recursivo que percorre toda a árvore de diretório do servidor primário analisando arquivos e diretórios através da chamada de sistema UNIX *stat*. Diretórios e sub-diretórios são percorridos por completo e imediatamente transferidos ao servidor que está sendo atualizado através da RPC *mkdir*. Cada arquivo é lido no servidor primário (chamada de sistema UNIX *read*) e transferido para escrita no servidor atualizado usando a RPC *write*. O retorno do PAS será OK para sucesso do processo ou NOK para estado de erro.

Do ponto de vista do PAS (fig. 4.2), o servidor primário funciona como cliente da aplicação e o servidor secundário em reintegração como servidor da aplicação O servidor primário realiza RPCs para atualizar o servidor secundário e permanece bloqueado até receber a resposta da atualização. Quando o secundário processa a escrita, responde ao primário. Então o primário pode continuar o PAS.

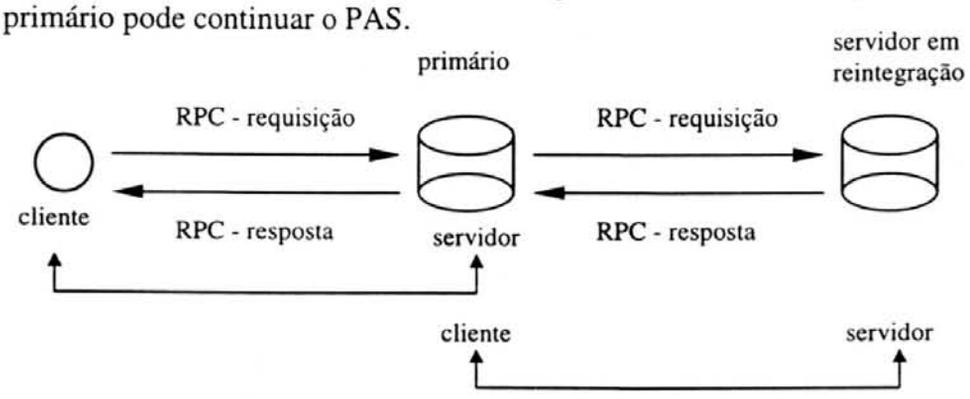

FIGURA 4.2 - PAS com RPCs aninhadas

Do ponto a reintegração do servidor, o servidor em reintegração funciona como um cliente que requisita um serviço - que é a reintegração - ao servidor primário corrente, que é o servidor da aplicação (de reintegração). O servidor primário executa o serviço de reintegração e retorna o resultado para o cliente - servidor que foi reintegrado.

Para finalizar, como o PAS por volume atualiza todos os arquivos do sistema, será usado como método *default*, a ser chamado quando há grande quantidade de informação a ser atualizada.

4.4 Implementação da cópia de arquivos com número de versão

O PAS por cópia de arquivos foi o segundo algoritmo implementado neste trabalho. Este PAS diferencia-se do anterior devido ao uso de um critério de comparação. O mecanismo de número de versões foi usado para prover ao sistema este diferencial. O ideal é implementar os números de versões alterando os *inodes* dos arquivos no UNIX. Para evitar

alterar os *inodes*, e tratar as possíveis inconsistências que isso pode gerar, o protótipo foi implementado usando arquivos de versões.

Os números de versões são gravados em um arquivo denominado *.versão*. Cada servidor do sistema possui o seu próprio arquivo *.versão* no diretório raiz (*/home*). A informação contida em cada linha do arquivo *.versão* é o caminho completo de cada arquivo no servidor e o seu número de versão correspondente. Um servidor chamado *sirius*, por exemplo, pode ter a seguinte configuração (fig. 4.3):

```
./home/sirius/usuario1/mbox          40009
./home/sirius/usuario1/prog.c        30028
./home/sirius/usuario1/teste         22222
```

FIGURA 4.3 - Possível configuração para o arquivo *.versão*

Para manter o sistema de números de versões consistente, toda operação de escrita no servidor deve ser refletida no seu arquivo de versões. Cada vez que um cliente requisita uma alteração em um arquivo, o arquivo e o arquivo de versões são alterados em todos os servidores que contêm cópias do arquivo. É Assumido que há um procedimento capaz de realizar a difusão de escritas mantendo o sistema de número de versões consistente. Portanto, a difusão de escritas é atômica e linear, além de possuir um procedimento de atualização do número de versão também atômico. Mesmo assim, primeiro o arquivo é atualizado e depois a versão.

No protótipo implementado, através de um comando explícito do operador, o PAS por cópia de arquivos pode ser começado. Como no PAS por volumes, este algoritmo recebe os parâmetros do servidor e a descrição do volume a ser atualizado. O PAS utiliza uma função chamada *cópia_de_arquivos* (fig. 4.4).

```
cópia_de_arquivos(servidor.nome, cliente.nome) {

        busca os atributos de arquivo - stat
        se tipo de arquivo é diretório {
                monta nome do diretório para servidor remoto
                cria diretório mkdir - RPC
                abre diretório - opendir
                lê linha de diretório - readdir
                enquanto houver linha em diretório {
                        monta nome de arquivo
                        cópia_de_arquivos(servidor.nome, cliente.nome)
                        lê arquivo em diretório - readdir
                }
                fecha diretório - closedir
        } senão {
                retorna_versão (cliente.nome)
                retorna_versão (servidor.nome) -  RPC
                se servidor.versão < cliente.versão {
                        abre arquivo - open
                        lê dados de arquivo - read
                        enquanto houver dados {
                                lê dados de arquivo - read
                        }
                        fecha arquivo - close
                        monta nome de arquivo para servidor
                        escreve o arquivo - RPC - write
                }
        }
        retorna OK
}
```

FIGURA 4.4 - PAS por cópia de arquivos, executado pelo servidor primário

A função *cópia_de_arquivos* (fig. 4.4) é semelhante à função *transferência_de_volume* (fig. 4.1). A diferença é que em *transferência_de_volume*, os números de versões são tratados, o que impõe alguma complexidade adicional à função.

O algoritmo recursivo percorre toda a árvore de diretório do servidor primário e diferencia arquivos e diretórios através da chamada de sistema UNIX *stat*. Diretórios e sub-diretórios são percorridos por completo e transferidos ao servidor que está sendo atualizado através da RPC *mkdir*. Cada arquivo é lido no servidor primário (chamada de sistema UNIX *read*) e transferido para escrita no servidor que está sendo atualizado usando a RPC *write*, somente se a sua versão estiver desatualizada. O PAS retorna OK para sucesso ou NOK quando ocorre algum erro.

Como esta proposta implementa as primitivas do RNFS em cima do NFS convencional e como o NFS não possui número de versão para seus arquivos, para decidir se o arquivo deve ou não ser atualizado, uma RPC que retorna o número de versão do arquivo precisou ser implementada. De posse do número de versão, o servidor primário compara o valor recebido com o seu valor para o arquivo. Então decide se o arquivo deve ser atualizado ou não. A RPC capaz de realizar esta tarefa é denominada *retorna_versão* (fig. 4.5).

```
retorna_versão (arquivo) {
        abre arquivo_versão - fopen
        se houver erro retorna NOK
        lê linha de arquivo_versão - fscanf
        enquanto não achar versão ou não for fim de arquivo
                lê linha de arquivo_versão - fscanf
        fecha arquivo_versão - fclose
        se achou versão
                retorna versão
        senão retorna NOK
}
```

FIGURA 4.5 - Sub-função retorna versão do arquivo

Como parâmetro, a RPC *retorna_versão* recebe o nome do arquivo. Essa RPC abre o arquivo *.versão* do servidor e analisa todas as linhas até encontrar o número de versão do arquivo ou o final do arquivo. O retorno é o número da versão para o servidor secundário, ou a sinalização de erro.

Além de construir a RPC que retorna o número de versão, a RPC *write*, que realiza a escrita do arquivo foi modificada (fig. 4.6).

```
write_2 (writeargs) {
        ...
        escreve dados - write
        fecha arquivo - close
        ...
        atualiza (arquivo, número)
        ...
        retorna OK
}
```

FIGURA 4.6 - RPC *write* modificada

A RPC *write* precisou ser modificada para comportar o número de versões dos arquivos. Cada vez que um arquivo é escrito, o número de versão do arquivo deve ser ajustado. Esse número é passado como parâmetro para a RPC *write_2* (fig. 4.6) através de *writeargs*. O parâmetro *writeargs* é uma estrutura que contém o nome do arquivo, um ponteiro para uma estrutura *data* e o número de versão para o arquivo. A estrutura *data* contém um repositório de dados e um contador que indica quantos *bytes* há no repositório (fig. 4.7).

```
struct writeargs {
        nfs_fh          file;               // handle do arquivo
        unsigned int begin_offset;          // não usado
        unsigned int offset;                // posição do inicio
        unsigned int total_count;           // não usado
        unsigned int versão;                // modificado para suportar
                                            // número de versão struct
              unsigned int length;          // tamanho dos dados
              char          *buffer;        // os dados
        } data;

};
```

FIGURA 4.7 - Estrutura *writeargs* modificada

Se uma operação de escrita acontece durante a difusão de escritas do primário para os secundários, o número de versão deve ser incrementado por uma unidade, ao menos que seja o procedimento de criação do arquivo. Neste caso, o valor 0 é atribuído ao número de versão. Durante a atualização, o número de versão do servidor que está sendo atualizado deve ser substituído pelo número de versão recebido do servidor primário. Essa tarefa é realizada pela função *atualiza* (fig. 4.8). Essa chamada não faz parte do sistema de arquivos NFS e teve que ser implementada. Recebe como argumento o nome do arquivo e o novo número de versão.

```
atualiza (arquivo, número)  {
        abre arquivo de versão - open
        se houver erro {            // arquivo de versão não existe
              cria o arquivo de versão - creat
              senão {
                    lê linha de versão - read
                    enquanto não for final de arquivo {
                          se achar arquivo
                                posiciona cursor - lseek
                                atualiza versão com número - write
                    }
                    se não achou arquivo
                          posiciona cursor - lseek
                          atualiza versão com 0 - write
              }
        }
        retorna OK
}
```

FIGURA 4.8 - Algoritmo para atualizar número de versão do arquivo

A chamada local *atualiza* (fig. 4.8) mantém o arquivo de versão de cada servidor. Cada vez que uma operação de escrita é realizada, uma única linha é adicionada contendo o *path* completo para o arquivo atualizado no servidor e o seu número de versão correspondente. Se o arquivo que está sendo recuperado já existir, o arquivo de versão é percorrido para encontrar a linha correspondente (ao arquivo que está sendo recuperado). Se a linha for encontrada, o número de versão é atualizado. Se a linha correspondente ao arquivo não for encontrada, uma nova linha com o nome do arquivo e o número de versão é adicionada ao arquivo de versão. Uma estratégia mais eficiente de pesquisa de número de versão pode ser implementada. Para tanto pode ser usada uma função de *hash,* vantajosa quando o sistema possuir muitos arquivos, o que geralmente ocorre em sistemas reais.

Coletor de lixo

Para finalizar o PAS por cópia de arquivos é necessário utilizar um coletor de lixo, como no Sistema Eden [PU88] (seção 2.4). Isso ocorre porque em sistemas de arquivos como

em sistemas de banco de dados [SIL93] pode haver a formação de lixo. Cada vez que um arquivo for atualizado, arquivos contendo versões antigas em servidores que sofreram falha tornam-se inacessíveis. Tais arquivos são considerados lixo, pois não fazem parte do sistema de arquivo corrente e não possuem informação utilizável. O lixo surge como um efeito colateral da queda do servidor. Após o PAS é necessário coletar todos os arquivos de lixo e excluí-los do diretório ao qual fazem parte. Este procedimento é chamado coleta de lixo.

O coletor de lixo pode ser implementado como um procedimento que percorre recursivamente a árvore de arquivos do servidor secundário reintegrado e compara cada arquivo com o seu correspondente no primário. Se não houver arquivo correspondente no primário, é porque este arquivo já foi eliminado quando o secundário estava ausente do grupo de replicação e também precisa ser eliminado no secundário.

4.5 Implementação da cópia de arquivos usando controle de tempo

Se o sistema distribuído comportar um algoritmo de sincronização de *clocks*, o PAS por cópia de arquivos usando como critério de comparação o tempo da última atualização do arquivo pode ser implementado. Além de assumir que este algoritmo está sendo executado em uma camada inferior a camada de replicação, será estabelecido que este algoritmo é livre de falhas ou então, se essas falhas ocorrem não afetam outras camadas. Na prática isso não acontece: falhas no algoritmo de sincronização podem ocorrer e podem se propagar para outras camadas, o que dificulta a implementação.

O algoritmo de cópia de arquivos que usa o tempo da última atualização é basicamente o mesmo algoritmo recursivo dos PASs anteriores. A chamada de sistema UNIX *stat* recebe como argumento o nome do arquivo corrente e retorna os atributos do arquivo. Um destes atributos é o tempo de última alteração do arquivo (*file last modify time*), representado por *st_mtime*.

É interessante observar que a sincronização de *clocks* não necessita ser tão precisa como em sistemas de tempo real [BAC86], nem acuradas em relação a um relógio externo ao sistema. Além da dificuldade de sincronizar os *clocks* do sistema, surgem outros problemas:

a) garantir a consistência de atualização de *st_mtime;*

b) garantir que a deriva máxima[19] dos *clocks* deve ser menor que o tempo de retorno de uma operação de um servidor secundário para o primário;

c) detectar qual arquivo deve ser atualizado;

d) escolher quem será o novo primário, em caso de falha no primário atual.

O problema de consistência de atualização de *st_mtime* (a) ocorre porque os valores escritos em *st_mtime* não são contínuos no sistema. Uma estação que é religada pode voltar com seu *clock* desatualizado. O valor de um *clock* pode ser corrigido pelo uso da chamada UNIX *utimes*, que permite escrever em *st_mtime* um tempo especificado pela aplicação. O problema da deriva máxima dos *clocks* (b) deve ser solucionado pelo algoritmo de sincronização de *clock* utilizado.

---

[19] diferença entre o *clock* mais alto e o mais baixo

Em particular, a efetivação de escritas nos servidores secundários RNFS é concluída antes da sua efetivação no servidor primário, para garantir atomicidade de informação em todos os servidores de arquivo em caso de falha. Assim, para que o protocolo que usa o controle de tempo possa ser implementado para o RNFS, será gravado no campo *st_mtime* do arquivo do primário o valor do tempo quando o cliente acessa o primário (antes da difusão de escritas começar). Mesmo que a atualização efetiva do primário aconteça após os secundários, o valor de *st_mtime* no primário refletirá, se os *clocks* do sistema estiverem sincronizados, sempre um valor inferior a todos os secundários (c). Na prática, a escrita continua sendo processada primeiro em todos os secundários e só então no primário, mas para o sistema de *clocks* aparece como se fosse realizada antes no primário (figura 4.10).

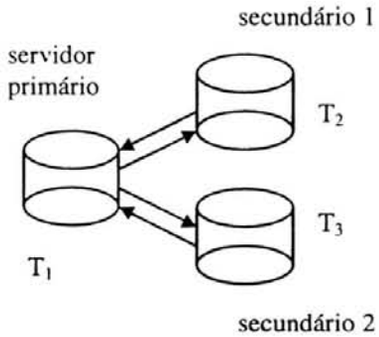

FIGURA 4.10 - Conjunto de replicação livre de falha

No esquema mostrado na figura 4.10 pode-se ter diferentes cenários de falha. Se a falha for no primário, o cliente detecta a falha e começa o algoritmo de escolha de novo primário (d). Como o primário deve sempre ter o tempo de atualização inferior aos secundários, o novo primário será sempre o primeiro secundário que foi atualizado. Na figura 4.11, o primário falhou depois de atualizar todos os secundários e não chegou a alterar o seu sistema de arquivo.

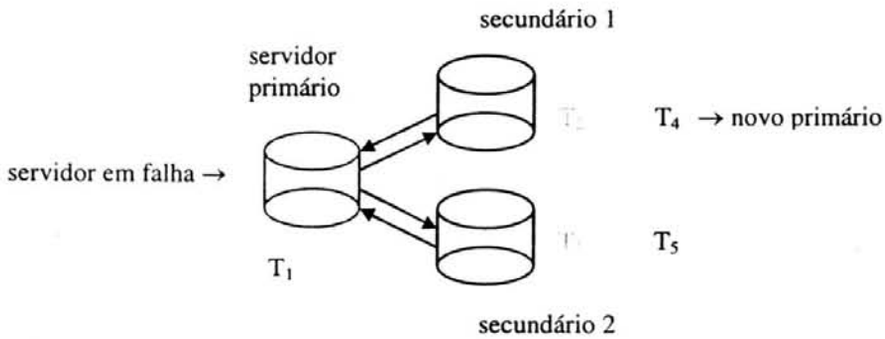

FIGURA 4.11 - Conjunto de replicação com primário em falha

Como o primário deve ter tempos inferiores aos secundários, o novo secundário será o secundário 1. Assim, o secundário 1 (novo primário) compara seus tempos com o secundário 2 para saber se o sistema está consistente ($T_4 < T_5$ (V)). Quando o antigo primário é reconectado ao sistema, o novo primário compara os tempos ($T_4 < T_1$ (F)) e verifica que o arquivo deve ser atualizado no servidor reintegrado.

Quando o primário falha depois de completar a difusão de escritas para os secundários (fig. 4.11), a falha é mascarada para o cliente. Quando o primário falha antes ou durante a difusão de escritas, o cliente não recebe a resposta e suspeita da falha. Antes que o novo

primário seja escolhido, o cliente refaz o pedido. Quando o primário falhou antes de começar a difusão de escritas, o novo primário considera a requisição como nova [GUE97].

Se o primário falhou durante a difusão de escritas, a solução deve garantir atomicidade: ou todos os secundários alteraram o arquivo ou nenhum secundário alterou o arquivo [GUE97]. Se nenhum secundário alterou o arquivo, este caso torna-se similar ao caso em que o primário falhou antes de atualizar os secundários. Se todos os secundários atualizaram o arquivo (fig. 4.11), o arquivo é atualizado mas o cliente não recebeu a resposta. Neste caso o novo primário precisa de um identificador de pedido para evitar refazer a requisição [GUE97].

Neste trabalho, a unidade de atualização (alteração, modificação, recuperação) do RNFS é o arquivo. A cada operação de escrita um arquivo é completamente atualizado. Essa estratégia facilita a implementação, mas não é a ideal. A desvantagem é que blocos de um arquivo que não foram alterados são rescritos, mesmo quando não é necessário. Quando o sistema real for implementado, um arquivo será tratado por um conjunto de blocos, possibilitando que apenas os blocos de arquivos que formam alterados sejam rescritos, sem necessitar alterar todo o arquivo.

Outro caso que precisa ser analisado ocorre quando um dos secundários falhar. Se o secundário 1 falhar na próxima atualização (fig. 4.12), terá o valor de *st_mtime* menor que o primário corrente.

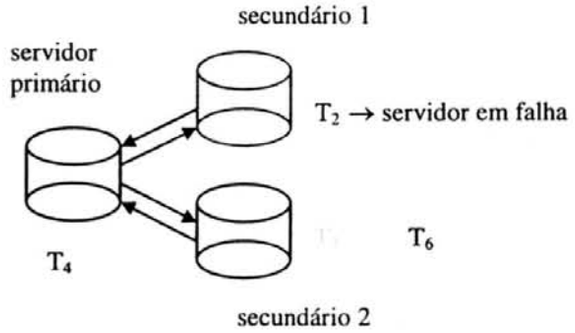

FIGURA 4.12 - Conjunto de replicação com secundário em falha

Como o valor do tempo em secundário 1 é menor que o valor do tempo no primário ($T_4 < T_2$ (F)), o arquivo deve ser atualizado.

4.6 Implementação da retenção de *log*

O *log* é um arquivo guardado em memória não-volátil[20] no servidor primário e suporta até um limite pré-estabelecido de informações. O tamanho do *log* depende da implementação e da capacidade de armazenamento do disco do servidor primário. Como o disco do servidor primário tem tamanho limitado, quando o tempo de indisponibilidade do servidor falho for muito longo não haverá espaço suficiente para reter todas as operações *log*.

Além desta limitação, o PAS por *log* trata apenas um ponto de falha (em servidor secundário) por vez. O *log* não é replicado na nossa implementação, por isso os demais secundários (que estão livres de falha) não estão aptos a continuar o PAS em caso de falha no

[20] isto é, discos. As informações residentes neste tipo de armazenamento geralmente sobrevivem a quedas do sistema.

servidor primário. Pelo mesmo motivo, falha em servidor primário também não pode ser tratada por *log*.

O *log* começa a ser formado quando o servidor primário detecta por *timeout* falha em um servidor secundário durante a difusão de escritas. Se o secundário não responder dentro do *timeout* especificado, é assumido como falho. Quando um servidor é reintegrado ao grupo de replicação, o *log* é utilizado para atualizar o seu sistema de arquivos. Toda a informação do *log* é lida no servidor primário e processada no servidor que está sendo atualizado.

É assumido que a operação de registrar informações no *log* é atômica. Um registro é uma descrição da operação realizada. Enquanto o servidor falho está ausente do sistema, cada vez que uma escrita é difundida, um novo registro é acrescentado ao *log*. Para manter a consistência do sistema, o registro é acrescentado no *log* antes que a difusão de escritas seja feita. Cada registro contém a descrição da operação realizada sobre o arquivo (criação, remoção ou escrita) e nome do arquivo:

log = {registro}
registro = operação + arquivo

Para agilizar o *log*, durante o período de falha do servidor, cada operação realizada na difusão de escritas corresponde a um registro do *log*, exceto se essa operação for repetida. Por exemplo: se chegarem duas operações seguidas para atualizar o arquivo */user/mail* apenas a primeira irá para o *log*.

Existem diferentes estratégias para tratar o *log*, descritas principalmente para sistemas de banco de dados. Nesta pesquisa será utilizada uma estratégia baseada na modificação adiada [SIL93]. A técnica de adiar a modificação garante atomicidade em caso de falha no servidor primário durante a reintegração. Todas as modificações são primeiramente gravadas no *log* do servidor primário, e depois o sistema de arquivos do primário é alterado. Se o servidor primário falhar durante o PAS, a informação do *log* é desconsiderada. Um novo servidor primário é eleito e outro PAS deve ser utilizado (já que o *log* é único e foi descartado). Se o servidor que está sendo atualizado falhar e a operação for abortada, a nova atualização deve ser começada.

O PAS por retenção de *log* segue o algoritmo descrito na figura 4.13. Após a execução do protocolo de início da reintegração, começa a atualização dos arquivos do servidor secundário. Para este protocolo será utilizada a informação mais recente para evitar refazer operações. Se um arquivo é alterado várias vezes durante o período de falha e, consequentemente, aparece em vários registros do *log*, não precisa realizar todas as atualizações de todos os registros do *log*. Basta atualizar todo o arquivo apenas uma vez, utilizando o último registro em que o arquivo aparece no *log*.

```
atualização_por_log {
      lê registro mais recente do log - read
      enquanto houver informação no log {
            se registro.operação = write ou create {
                  se arquivo não está em buffer {
                        escreve arquivo - write - RPC
                        move registro.arquivo para buffer.arquivo
                  }
            } senão se registro.operação = remove {
                  se arquivo está em buffer {
                        remove arquivo - remove - RPC
                  }
                  move registro.arquivo para buffer.arquivo
            }
            remove registro do log
```

```
            lê próximo registro do log
        }

        enquanto houver buffer.arquivo {
            remove buffer.arquivo
        }
        retorna OK
}
```

FIGURA 4.13 - PAS por retenção de *log*, executado pelo servidor primário

A estrutura *buffer* guarda em memória volátil os nomes de arquivos. Durante o PAS, cada vez que um arquivo é atualizado, seu nome é colocado no *buffer*. Quando o arquivo for referenciado, a informação do *buffer* é analisada. Se o arquivo está em *buffer* é porque já foi atualizado (transferido para o servidor que está sendo reintegrado). O *buffer* é utilizado para evitar execuções duplicadas de operações. A reexecução de uma operação sempre implica em procedimento adicional e não necessário, e degrada o desempenho do PAS.

O *log* pode ser implementado como uma pilha, onde o registro mais recente será o primeiro da pilha. A cada atualização do servidor secundário, um registro será eliminado da pilha. Quando o PAS terminar, o *log* estará vazio. Em caso de falha no primário, essas informações serão descartadas.

Uma dificuldade de implementação do PAS por *logs* é como detectar exatamente o ponto de falha no servidor secundário [LEB96], de modo a estabelecer a última operação realizada com sucesso antes da falha. Para este fim, podem ser usados *commit points* [LIS91]: todos os servidores devem manter *commit points* em disco anotando a última operação corretamente transmitida (para os servidores principais) e última operação corretamente aplicada (para os servidores secundários). Se estes *commit points* sobreviverem à falha, isto é, se não houver *crash* de disco, a reintegração poderá ser realizada. Quando ocorrer perda dos *commit points*, será necessário usar outro PAS.

Para melhor compreensão, a atualização baseada em *log* será ilustrada com um exemplo. Seja um sistema distribuído com dois servidores secundários 1 e 2, e um primário (fig. 4.14).

FIGURA 4.14 - Configuração do sistema quando secundário 1 é religado

Na figura 4.14, cada letra representa um nome de arquivo. Os servidores primário e secundário 2 estão funcionando normalmente. O servidor secundário 1 foi desligado em determinado ponto: depois de gravar o arquivo C. O servidor primário não recebeu a resposta de última atualização de secundário 1 e anotou no *log* que a operação <escrever C> não foi realizada. As próximas alterações no sistema de arquivos do grupo de replicação serão guardadas no *log*, até que secundário 1 volte a participar da difusão de escritas. O *log* do servidor primário conterá a informação descrita na figura 4.15 quando o secundário 1 voltar.

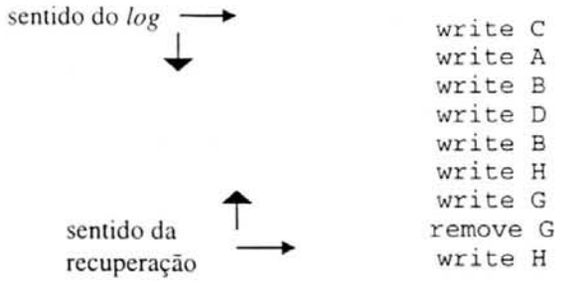

FIGURA 4.15 - Configuração para o *log*

Cada *write* do *log* significa que ocorreu uma operação de escrita, neste caso o arquivo deve ser transferido, através de uma RPC para o servidor que está sendo reintegrado. Cada *remove* significa uma operação de remoção. Neste caso, o arquivo deve ser removido do servidor que está sendo reintegrado. Ainda pode aparecer a operação *create*. Neste caso, o arquivo será criado no servidor que está sendo reintegrado. Quando o PAS inicia, obedece o sentido inverso da figura 4.15, isto é, de baixo para cima. Isso ocorre devido à necessidade de otimizar o PAS, utilizando apenas a informação mais recente do *log*.

O registro <write H> é o primeiro a ser processado. Este registro realiza uma RPC *write* de todo o arquivo do primário para o servidor que está sendo reintegrado e empilha H em *buffer*. A configuração de *buffer* será <H> e o registro <write H> será excluído do *log*. Quando o registro <remove G> for processado, o arquivo G será eliminado através de uma RPC do secundário 1 e o nome do arquivo irá para *buffer*, *buffer* <G, H>. Então, o registro <remove G > será excluído do *log*. O próximo registro <write G> não será processado pois G está em *buffer*. O registro seguinte, <write H> não será processado pois H está em *buffer*. A execução do registro <write B> escreve o arquivo B no secundário 1. Então B será colocado no *buffer* <B, G, H, C> e mais um registro será excluído do *log*. O próximo registro,<write D> será processado. O registro <write B> não será processado pois B está em *buffer* e, finalmente <write A> e <write C> serão processados e excluídos do *log*. Então a atualização do secundário 1 termina e a informação de *buffer* será descartada. Neste ponto, o *log* estará vazio. Após a execução do protocolo de término da reintegração, o servidor secundário 1 poderá participar novamente de difusão de escritas.

4.7 Implementação de protocolo de atualização de servidor composto

Para que um PAS composto possa ser implementado é necessário os PASs sejam compatíveis entre si, de modo que se algum deles falhar, outro possa ser tentado.

Para que os PASs por volume por *log* possam interagir com o PAS por cópia de arquivos, o número de versão em todos os protocolos. Para fazer isso basta atualizar sempre o número de versão cada vez que um arquivo é alterado. Se o critério de comparação utilizado for o tempo da última atualização do arquivo, a chamada UNIX *utimes* deve atualizar um tempo factível e sincronizado para o arquivo.

4.8 Reintegração de servidor em paralelo com o serviço dos clientes

Enquanto o servidor primário reintegra um servidor secundário, os clientes continuam a solicitar pedidos. O servidor primário deveria, então, ser capaz de atender os clientes e o

servidor que está sendo reintegrado. A maneira mais simples de tratar este problema é interromper completamente o serviço para os clientes. Um cliente que faz um pedido para um servidor primário que está executando um PAS, fica bloqueado esperando o término da reintegração. Quando a reintegração termina, o servidor primário volta a responder às requisições dos clientes. Essa estratégia mostra-se mais eficiente na reintegração por volume, onde um grande número de arquivos deve ser copiado. Porém, nem sempre é uma boa alternativa do ponto de vista do cliente pois reduz drasticamente a disponibilidade do sistema.

Uma segunda solução é bloquear apenas a escrita para todos os arquivos do servidor primário[21] durante o PAS. Somente a leitura é permitida para os clientes. No caso do PAS por cópia de arquivos, o bloqueio deve ser desfeito quando o coletor de lixo eliminar os arquivos desnecessários no servidor que está sendo reintegrado, antes do término da reintegração.

A terceira solução consiste em permitir a atualização no primário e demais secundários e utilizar um *log* para guardar as alterações de escrita dos clientes. A desvantagem é que o servidor que está sendo reintegrado deve ser atualizado com o *log* depois que o servidor está reintegrado ao grupo de replicação. Isso pode gerar um problema maior, pois enquanto o *log* está sendo transferido para o servidor reintegrado, novas requisições podem chegar dos clientes.

A quarta solução utiliza *locks* para bloquear escrita em arquivos[22] e desfaz os bloqueios gradativamente (fig. 4.16). Inicialmente, todos os arquivos do sistema (primário, secundário) são bloqueados para escrita no protocolo de início da reintegração. A medida que os arquivos vão sendo atualizados no primário e secundário, os *locks* vão sendo desfeitos. Dessa forma, um arquivo bloqueado significa que o arquivo ainda não foi transferido no PAS.

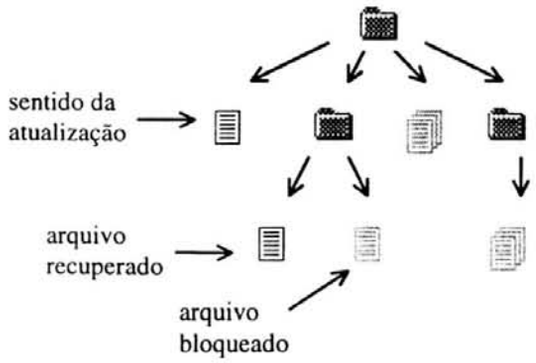

FIGURA 4.16 - Arquivos do servidor primário durante o PAS

Um arquivo representado em cor escura (fig. 4.16) é um arquivo já atualizado; um arquivo em cor clara ainda será atualizado. A seta indica o sentido do PAS (de cima para baixo, da esquerda para a direita). Se chegar uma requisição de cliente para um arquivo que não está bloqueado, a escrita deve ser realizada no sistema de arquivo de todos os servidores do grupo de replicação, incluído o servidor que está sendo reintegrado.

Se o arquivo estiver bloqueado no sistema (primário, secundário), então o arquivo ainda não foi atualizado. Neste caso, a requisição do cliente atualizará o arquivo em todos os servidores grupo de replicação, exceto o servidor que está sendo reintegrado. Quando o PAS atingir o ponto onde se encontra o arquivo em questão, então ocorre a atualização do arquivo servidor que está sendo reintegrado. A atualização será feita com a informação do arquivo já

[21] o primário é o servidor encarregado de reter a informação (*lock*) e não transmitirá as requisições de escrita para o servidor que está sendo reintegrado

[22] novamente o primário é o servidor que retém a informação

atualizado no primário. Dentre as quatro soluções esta parece ser a mais versátil, porém mais complexa para ser implementada por causa do controle dos *locks* e dos processos.

Com relação às soluções apresentadas, falta comentar que o uso de *locks* possui a desvantagem de reter informação de estado, o que viola o princípio de servidores *stateless* do NFS, mas possibilita que a realização do PAS possa ocorrer em paralelo com a operação normal do sistema (fig. 4.17), isto é, com a realização do serviço dos clientes.

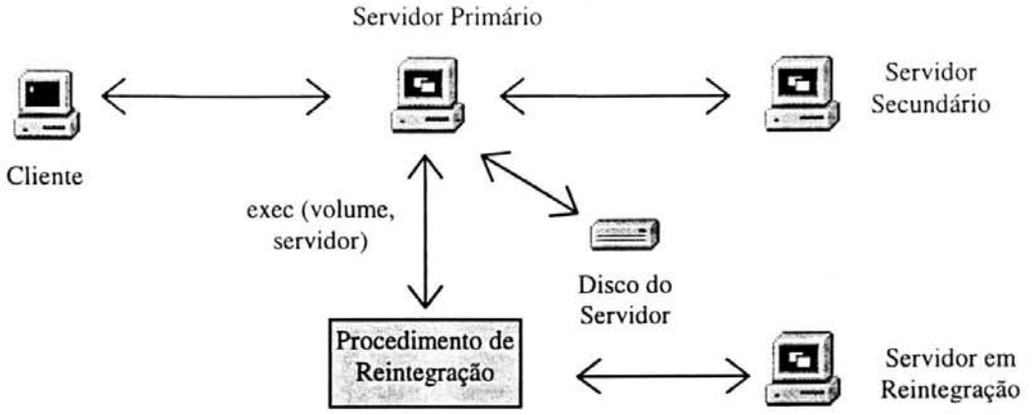

FIGURA 4.17 - Acesso e reintegração de servidor em paralelo [LEB96]

Para servidores UNIX, onde a arquitetura do sistema não permite mais de uma *thread* por processo, não há como comportar de mais de um processo servidor RNFS a cada momento [LEB96]. A solução usa uma abordagem em que o servidor RNFS começa um processo especial responsável pela reintegração e monitora o término do processo [LEB96]. Um processo pode ser começado pela chamada UNIX *exec*.

O PAS deve receber, em sua criação, os parâmetros do servidor e descrição do volume a ser atualizado, e ter os mesmos privilégios de acesso a disco do servidor RNFS. Este protocolo deve implementar os PASs a serem utilizados, e possivelmente parte do algoritmo de acesso RNFS, se este for usado também na reintegração. Se for usado o PAS por retenção de *log*, o acesso a este deve ser compartilhado entre o servidor RNFS e o PAS. Adicionalmente, o servidor RNFS deve ter condições de controlar o término do PAS, e receber ao final um parâmetro indicando o término com sucesso.

4.9 O algoritmo de atualização na presença de falhas

Um aspecto muito importante em aplicações distribuídas é o tratamento de erro. Os erros podem ocorrer em processos cliente e servidor. Dessa forma, alguns aspectos devem ser considerados:

a) no protocolo UDP/IP usado pelo RNFS (que possui detecção de erros e assegura que apenas mensagens íntegras são entregues), a requisição do RPC pode nunca chegar, chegar repetida ou chegar fora de ordem. Para evitar esses problemas com RPC, o NFS possui um mecanismo que distingue mensagens repetidas [TAN94];

b) durante o PAS, um novo arquivo só é enviado do primário para o secundário depois que o primário recebeu a confirmação de que o arquivo anterior foi enviado e alterado com sucesso no secundário.

A respeito dos PASs, se algum erro for detectado durante a sua execução, o PAS é abortado. Então o PAS deverá recomeçar. Um PAS recomeçado é considerado como sendo um novo PAS. Em se tratando de falhas muito graves, onde o sistema não for capaz de realizar a reintegração automática, o operador do sistema será notificado.

Falhas reincidentes em secundários sofrendo PAS podem acontecer, apesar da probabilidade ser pequena. Essas falhas são facilmente detectadas e corrigidas, sem causar inconsistência no sistema.

Uma situação mais complexa ocorre quando o servidor primário falha durante o PAS. Neste caso, ocorrem dois pontos de falha seqüenciais: primeiro a falha no servidor que está sofrendo o PAS e depois a falha no servidor primário. Falhas seqüências podem ser tratadas pelo sistema e serão analisadas a seguir. Um cliente deve detectar a falha seqüencial e requisitar a escolha de novo primário, bem como uma nova visão da rede. A nova visão da rede deve excluir do grupo de replicação os servidores falhos.

4.9.1 Cópia de arquivos e transferência de volume em presença de falhas

Durante a execução do PAS por cópia de arquivos ou por transferência de volumes, se o servidor que está sendo atualizado falhar novamente, um novo PAS será iniciado. O servidor primário é encarregado de detectar falhas nos servidores secundários durante o PAS e os clientes são capazes de detectar falhas no primário usando *timeout*.

Depois do reparo, o servidor precisa executar novamente o protocolo de início da reintegração. O novo PAS recomeça comparando os arquivos, quando for utilizado algum critério de comparação, ou simplesmente transferindo todos os arquivos.

4.9.2 Atualização por *log* em presença de falhas

Também na atualização por *log*, como no caso anterior, se o servidor secundário que está sofrendo o PAS falhar novamente, então o PAS recomeça como um novo procedimento. Isso ocorre porque o *log* conterá outras informações. O servidor primário precisa executar uma outra vez o protocolo de início da reintegração. Durante este procedimento, poderá ocorrer difusão de escritas do primário para os secundários e novos registros poderão ser acrescidos ao *log*. Porém a informação contida no *buffer* do servidor primário, que é uma estrutura auxiliar para a atualização por *log*, deverá permanecer intacta. O ponto de falha no servidor secundário terá que ser considerado novamente.

Se o servidor primário falhar durante o PAS, a informação do *log* será perdida (o disco do primário, onde o *log* está gravado, tornou-se inacessível). Um novo primário será eleito e um PAS alternativo deverá ser usado para recuperar os servidores em falha.

Na prática não é comum ter mais que um ponto de falha. Se isso acontecer, o RNFS deverá estar apto a tratar cada um dos pontos por vez. Assim, ocorrerá duas reintegrações de servidores seguidas.

Um mecanismo alternativo pode ser proposto para tratar dois ou mais pontos de falha simultâneos. Neste caso, o primário terá que realizar uma múltipla difusão de escrita para os

secundários: teria que atender aos pedidos dos clientes difundindo as escritas para os secundários operacionais e reintegrar o antigo primário e o servidor que o antigo primário estava reintegrando.

5 Avaliação dos Protocolos de Atualização de Servidor

Este capítulo apresenta uma avaliação dos PAS (Protocolos de Atualização do Servidor) implementados. Para realizar a avaliação, um protótipo foi implementado. O protótipo é formado por uma aplicação distribuída (programas cliente e servidor, além de arquivos gerados no *rpcgen*). Após a conclusão da implementação, testes foram realizados e estão descritos neste capítulo.

5.1 Código gerado e compilação

Inicialmente o protocolo de atualização por volume foi implementado porque esse algoritmo não precisa de nenhuma RPC, além das existentes do NFS convencional. O PAS por volume utiliza um algoritmo recursivo que transfere todos os arquivos de um servidor atualizado para outro desatualizado. O algoritmo recursivo implementado é uma adaptação do encontrado em Kernighan e Ritchie [KER90].

Depois o PAS por volume foi modificado para comportar a comparação da versão do arquivo. Um arquivo de versão foi adicionado a cada servidor e um algoritmo que incrementa o valor da versão correspondente ao arquivo a cada atualização foi escrito.

O código dos protocolos por volume e por cópia de arquivos foi escrito para ser executado de forma distribuída, onde o servidor atualizado (primário) é o servidor da aplicação e o servidor desatualizado (secundário) é o cliente da aplicação. Para compor a aplicação distribuída, o código foi organizado em diferentes arquivos:

a) *cliente.c* com tamanho de 9259 *bytes*[23], que contém as rotinas locais, realizadas pelo servidor primário para atualizar os arquivos em um servidor secundário que está desatualizado;

b) *server.c* com tamanho de 7006 *bytes,* que contém as rotinas remotas (como *write*, por exemplo), realizadas pelo servidor que está sendo atualizado;

c) *readwrite.x* com tamanho de 693 *bytes,* que é o arquivo que contém as especificações do protocolo; esse arquivo foi compilado no *rpcgen*, que gerou os seguintes arquivos:

   - *readwrite.h* com tamanho de 1124 *bytes*, é o arquivo de cabeçalho (*header*). Contém as macros do pré-processador e os tipos de dados entre outras informações [COR91];

   - *readwrite_clnt.c* com tamanho de 1230 *bytes*, é o arquivo que contém as rotinas do *stub* do cliente;

   - *readwrite_svc.c* com tamanho de 2217 *bytes*, contém a rotina principal do serviço do servidor e a rotina que aloca o tempo da CPU para a execução das rotinas cliente e servidor. A rotina principal gera cabeçalhos de transporte e registra o serviço

[23] os tamanhos de arquivos apresentados em *bytes* são referentes à implementação corrente

[COR91]. A rotina do que aloca o tempo da CPU desvia a execução do programa para a rotina apropriada;

- *readwrite_xdr.c* com tamanho de 1564 *bytes*, é a rotina XDR para a conversão de dados.

Os arquivos *cliente.c* e *server.c* foram compilados e ligados a outros arquivos usando o compilador *cc*, através dos seguintes comandos:

cc -o cliente cliente.c readwrite_clnt.c readwrite_xdr.c
cc -o readwrite_svc readwrite_svc.c server.c readwrite_xdr.c

Neste ponto são gerados os demais arquivos:

a) *cliente*: programa executável do cliente com 32.768 *bytes;*

b) *readwrite_svc*: programa executável do servidor com 24.576 *bytes;*

c) arquivos com código objeto: *cliente.o* (8596 *bytes*), *readwrite_clnt.o* (1692 *bytes*), *readwrite_svc.o* (2816 *bytes*), *readwrite_xdr.o* (1396 *bytes*) e *server.o* (6564 *bytes*).

Para executar o protótipo, foram utilizadas as estações Sun do laboratório do Instituto de Informática da UFRGS. O protótipo também pode ser executado no ambiente Linux dos PCs (também disponível nos laboratórios do Instituto). Para executar a aplicação basta abrir uma sessão (através de *telnet*), disparar o servidor com o comando *readwrite_svc* & e fazer as requisições do serviço de reintegração especificando o nome do servidor primário e o nome do volume a ser recuperado.

O leitor mais interessado pode obter o código dos principais arquivos do protótipo. O código está disponível no diretório *home/sirius/pasin/public_html*. Os demais arquivos podem ser gerados usando o *rpcgen* e o compilador *cc*.

5.2 Resultados de testes

Os testes foram realizados em condições favoráveis de tráfego de rede em estações Sparc. Como o PAS é dependente principalmente do número de mensagens trocadas, o desempenho dos processadores envolvidos não é tão importante quanto a taxa de transmissão da rede.

Para executar os testes, foram geradas diferentes árvores de diretórios. A figura 5.1 mostra uma destas árvores com 44 arquivos. Cada diretório *usern* da figura contém 4 arquivos.

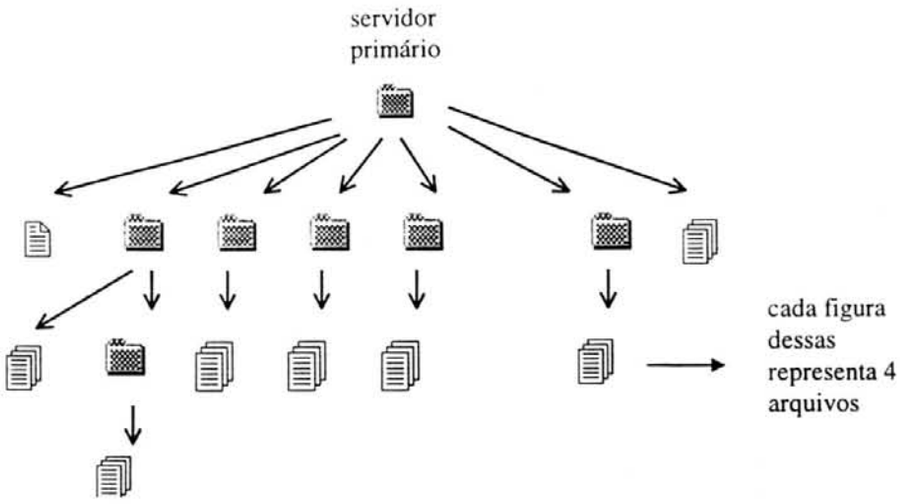

FIGURA 5.1 - Estrutura de diretório com 44 arquivos usada para teste

Os diretórios (figura 5.1) são replicados em dois servidores (primário e secundário). Uma falha foi simulada excluindo-se alguns arquivos no servidor secundário. Para corrigir o resultado da falha deste servidor, um dos protocolos implementados pode ser utilizado.

Como já foi mencionado, o primeiro algoritmo implementado e testado foi o PAS por volume. Nos testes, a complexidade e profundidade das árvores não foram consideradas porque essas duas características são relacionadas ao algoritmo recursivo e não ao PAS implementado. Com o resultado dos testes, foi montado um gráfico (fig. 5.2). Os valores na horizontal representam o número de total de arquivos replicados no sistema e os valores da vertical (à esquerda) representam o tempo de realização do PAS (em segundos). Esse tempo engloba a transferência de arquivos e o processamento que é feito nos servidores primário e secundário.

O tamanho dos arquivos usados nos testes é pequeno (80 *bytes*). Os testes foram realizados na estação rigel (*Sparc Station* IPC). O objetivo final dos testes é estabelecer uma comparação entre os PASs implementados e descobrir quando é conveniente usá-los. Um resultado de teste de desempenho do PAS por volume pode ser descrito com o gráfico da figura 5.2.

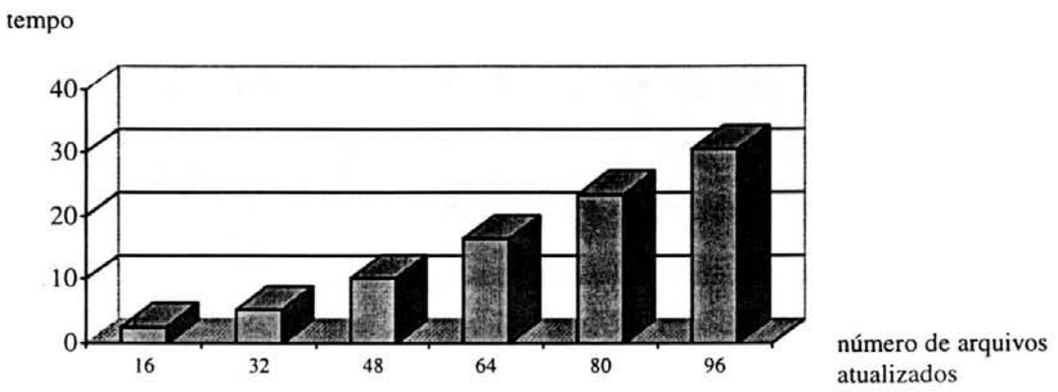

FIGURA 5.2 - Execução do protocolo de atualização por volume

No gráfico (fig. 5.2) é notado que o tempo de execução do PAS cresce exponencialmente, como uma curva. Para poder realizar a comparação, o PAS por volume foi alterado para

suportar números de versão de arquivos. Um arquivo com números de versão foi gerado em cada servidor e rotinas para manter os números de versão foram criadas. Essa nova bateria de testes foi realizada na estação vega (Sparc Station 1+) com arquivos de 80 *bytes* em média. Com os resultados, o gráfico descrito na figura 5.3 foi obtido.

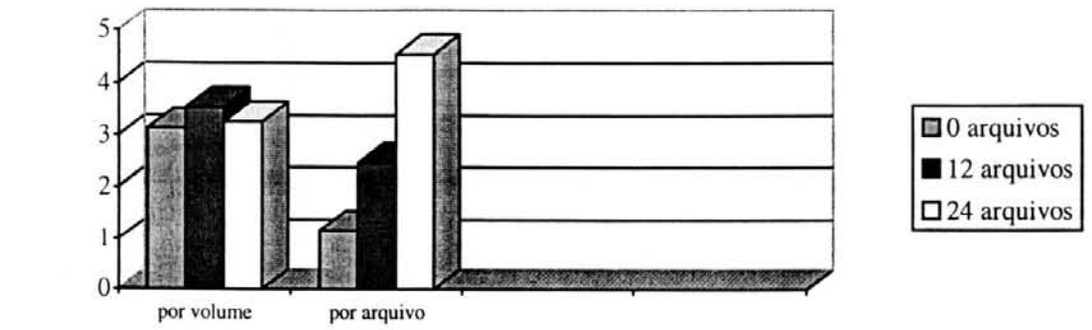

FIGURA 5.3 - Comparação entre PASs com 24 arquivos

O gráfico da figura 5.3 apresenta na vertical (à esquerda) os valores de tempo. A legenda mostra a correspondência de arquivos que faltam no sistema de arquivos. Para testar, foram eliminados todos (24), a metade (12) ou nenhum (0) arquivo. Então foram executados os dois PASs (por transferência de volume e por cópia de arquivos), para as três diferentes cenários de falha. Para obter o melhor parâmetro de comparação, foram executados dois protocolos no mesmo cenário: uma árvore com sub-diretórios (cada um destes sub-diretórios com pelo menos um arquivo), totalizando 24 arquivos.

Observando os resultados (fig. 5.3) pode ser observado que o PAS por volume obteve desempenho praticamente constante. Isso acontece porque para este método não importa a quantidade de arquivos que faltam (todos, metade ou nenhum), sempre serão atualizados todos os arquivos.

No PAS por cópia de arquivos pode ser observado que se há poucos ou nenhum arquivo a ser transferido, este método é mais eficiente que o método por volume. Se há uma quantidade média, esses métodos são praticamente equivalentes (pelo menos para os sistemas de arquivos testados), mas o PAS por cópia de arquivos ainda é um pouco mais eficiente que o PAS por transferência de volumes.

No último caso, quando todos os arquivos do sistema precisam ser atualizados, se for usado o PAS por cópia de arquivos a comparação precisa ser feita em todos os arquivos, com o seu correspondente no servidor primário e transferir todos os arquivos. Neste caso, certamente, a transferência de volumes será o método menos oneroso. Isso pode ser observado no gráfico (fig. 5.3), onde para um sistema com 24 arquivos, todos eles precisam ser recuperados.

Para reforçar as observações, os mesmos testes foram realizados para a árvore de diretórios da figura 5.1, com 44 arquivos e foi obtido o seguinte gráfico (fig. 5.4).

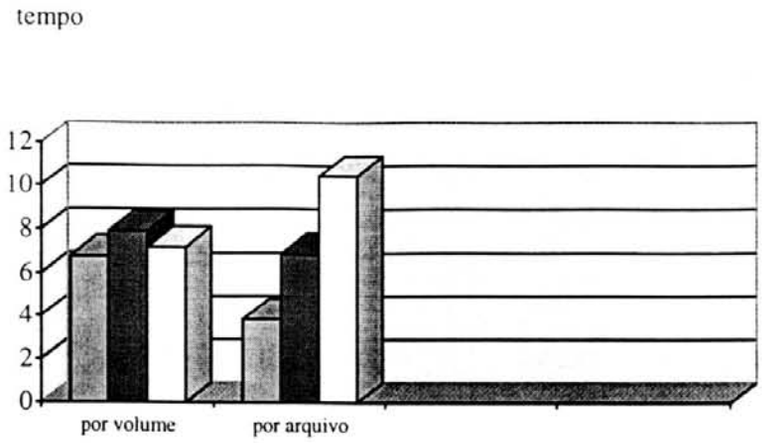

FIGURA 5.4 - Comparação protocolos por volume e cópia de arquivos.

A primeira coluna é referente a um sistema de arquivos completamente atualizado, a segunda é referente a metade dos arquivos desatualizados e a última coluna é referente a um sistema de arquivos totalmente desatualizado. Novamente, o PAS por cópia de arquivos foi mais eficiente quando há poucos (ou nenhum) arquivo a ser recuperado. Isso acontece porque o custo de realização de uma RPC (neste caso de escrita de arquivo) é muito alto. Assim, o PAS por cópia de arquivos busca somente a informação da versão do arquivo e transfere apenas o arquivo que é necessário para atualizar o servidor que está sendo reintegrado.

5.3 Comparando os protocolos de atualização de servidor

Uma análise geral dos PAS por transferência de volume, por cópia de arquivos e por retenção de *log* permite montar um quadro comparativo resumido (tab. 5.1).

TABELA 5.1 - Comparação entre os PASs

	Transferência de volume	Cópia de arquivos	*Log* de operações
Desvantagens	recupera inclusive arquivos atualizados	necessita manter os *clocks* sincronizados; método de comparação	limitação do *log*; necessário manter *commit points*
Indicado para sistemas	que não possuem número de versão	com sincronização de *clocks* ou método de comparação	que não possuem número de versão
Indicado para tratar falhas de	longa duração	curta duração	curta duração
Quantidade de informação a ser recuperada	muitos arquivos	poucos arquivos	poucos arquivos
Tempo de indisponibilidade do servidor primário	alta	depende da quantidade de arquivos a ser recuperada	depende da quantidade de arquivos a ser recuperada

Comparando os dois primeiros métodos, foi observado que o protocolo por cópia de arquivos usando número de versão é realmente eficiente quando há poucos arquivos para serem recuperados, se comparado com o protocolo por volume. No protocolo por cópia de arquivos, o tempo de transferência de arquivos é reduzido devido à comparação das versões de arquivo. Esse tempo aumenta consideravelmente se o número de arquivos que deve ser transferido é muito grande.

A principal desvantagem dos métodos de retenção de *log* e de cópia de arquivos é a manutenção do arquivo de recuperação (arquivo de *log* ou arquivo com o número de versão). As operações realizadas neste arquivo precisam ser atômicas e deve ter alguma garantia de que a informação armazenada pelo arquivo é consistente. Se a informação não for consistente, o sistema não estará apto a tratar falhas por esses métodos. Um mecanismo de tratamento de falhas neste arquivo deve ser proposto (como a replicação do arquivo).

6 Conclusões

O capítulo final apresenta as conclusões obtidas com a implementação do protótipo, o seu estado atual e sugestões para trabalhos futuros.

6.1 Resultados obtidos

A reintegração automática de servidores em ambiente distribuído foi validada parcialmente. Foram implementados dois PASs (por volume e por cópia de arquivos). Faltou implementar a troca de mensagens dos protocolos de início e término da reintegração. O método de atualização do servidor por cópia de arquivos com número de versão parece ser a estratégia mais atraente, se o sistema comporta número de versão (o que não acontece no NFS normal). Uma opção é modificar a estrutura de *inode* do arquivo UNIX inserindo um campo com número de versão quando ocorrer a implementação do RNFS. A desvantagem é que a inserção do número de versão necessita de alteração na estrutura do NFS e da recompilação do núcleo do sistema operacional, o que pode ferir a sua compatibilidade.

O protótipo foi implementado usando o compilador *rpcgen*. Este protótipo atualiza um servidor a partir de outro atualizado. O protótipo pode ser executado tanto em rede Sun quanto em rede PC-Linux, pois procurou-se manter o padrão ANSI.

Os testes do protótipo foram feitos nas estações do Sun do Instituto de Informática. Durante várias execuções de um mesmo cenário de teste, é possível notar alguma diferença de resultados de tempos de execução devido ao tráfego da rede. Isso pode ser ilustrado na Fig. 6.1, onde a atualização por volume foi realizada usando a estação *antares* como cliente e *regulus* como servidor.

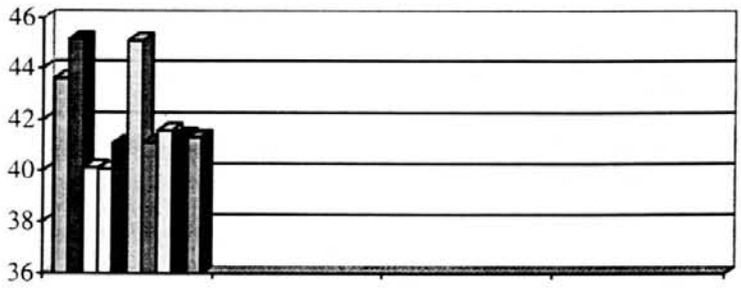

FIGURA 6.1 - Diferentes resultados para um mesmo sistema com 100 arquivos.

Para testar o protótipo com mais segurança seria necessário isolar pontos da rede, o que é inconveniente pois uma rede é compartilhada por muitos usuários.

O *rpcgen* mostrou-se didático, fácil de usar e de entender. Apesar do desempenho não ser compatível com o sistema UNIX real, serviu para validar os PASs e para a obtenção de resultados qualitativos. Os resultados de tempo obtidos certamente não serão os mesmos na implementação real do RNFS, porém é possível obter uma comparação proporcional.

6.2 Estado atual da implementação

O protótipo para o PAS por volumes está concluído e foi o primeiro a ser implementado por não precisar de nenhuma RPC, além das encontrados no NFS convencional. O protótipo do protocolo por cópia de arquivos também está pronto. Para implementar este protótipo cada servidor recebeu um arquivo com números de versão. O algoritmo utilizado compara a informação armazenada nos arquivos de versão para saber se o arquivo precisa ser atualizado.

O PAS por retenção de *log* bem como os protocolos de início e fim da reintegração ainda precisam ser implementados.

6.3 Sugestões para trabalhos futuros

O estudo da reintegração da servidores para o RNFS foi primeiramente sugerido por Leboute [LEB96]. Outras sugestões encontradas no mesmo são a avaliação de novos mecanismos de difusão de escrita (como o uso de *multicast*), a implementação de um algoritmo de escolha de novo servidor primário e um mecanismo de monitoramento do ambiente do RNFS. Além das sugestões de Leboute, pode-se acrescentar mais idéias para trabalhos futuros que surgiram no estudo da reintegração de servidores e estão interligadas, já que elas fazem parte do projeto do RNFS.

6.3.1 Implementar a difusão de escritas com atomicidade em caso de falha

A difusão de escritas do primário para os secundários poderá ser implementada usando um procedimento mais eficiente. Isso pode ser feito usando *multicast*, para garantir a atomicidade da informação dos servidores do grupo de replicação em caso de falha.

6.3.2 Proposta de configuração para o sistema de cópia primária

Para que o sistema distribuído apresente desempenho satisfatório sob o ponto de vista dos clientes, é necessário propor uma configuração ideal dos servidores do sistema. Se o servidor primário for mais rápido que os secundários, o procedimento de difusão de escritas poderá se tornar um gargalo. Apesar da operação de escrita no primário ser muito rápida, a mesma operação será muito demorada nos secundários. Neste caso, a possibilidade de utilizar a difusão por escritas por *multicast* para agilizar o procedimento pode ser analisada.

6.3.3 Prioridade de clientes e servidores que estão reintegrando

Para tratar as prioridades de serviço entre clientes e servidores, podem ser atribuídas diferentes prioridades a clientes e servidores, conforme as necessidades e o perfil de usuário que se quer atender. A atribuição de prioridades desencadeia três diferentes abordagens. Na primeira abordagem, quando o sistema está realizado a reintegração de algum servidor, pode

ser estabelecido que o servidor que está sendo reintegrado terá prioridade maior sobre todos os clientes. Isso significa que o sistema deve integrar o servidor o mais rápido possível para recuperar o número tolerável de pontos de falha antes da degradação do sistema. Em outras palavras o sistema prioriza a confiabilidade em relação à disponibilidade (que é atender os clientes).

Uma outra abordagem é atender às necessidades dos clientes e reintegrar o servidor quando houver tempo sobrando (entre uma solicitação de cliente e outra). Essa abordagem recairá sob um sistema com maior disponibilidade que o anterior. A terceira abordagem é a composição das duas primeiras, para chegar a um meio termo.

6.3.4 Implementação do sistema real

A última sugestão é começar a implementar o sistema real propriamente dito, ou o RNFS. O sistema Linux (que é distribuído livremente) pode ser utilizado como base, já que os códigos fontes estão disponíveis. Inicialmente, o núcleo do sistema deve ser modificado para suportar o número de versão de arquivo. A seguir, a difusão de escritas precisa ser implementada para dividir os servidores em primário e secundários e formar os grupos de replicação. Neste ponto, a reintegração de servidores pode ser implementada.

6.3.5 Reintegração de um servidor em vários grupos de replicação

Como um sistema distribuído permite que vários grupos de replicação existam simultaneamente, deve ser considerada a hipótese de reintegrar um servidor em vários grupos de replicação. Neste caso, um servidor reintegrado *n*, quando atualizado, conterá *m* diferentes árvores de diretórios para cada *m* primários que replicar (fig. 6.2):

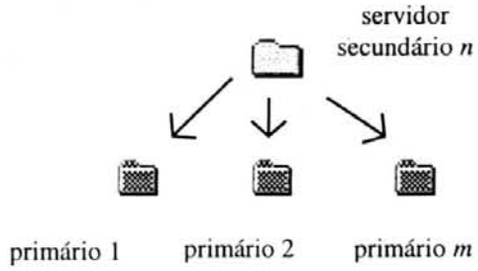

FIGURA 6.2 - Diretórios do servidor secundário *n* para *m* grupos de replicação

Essa organização para os servidores secundários permite que a atualização dos diferentes diretórios seja realizada concorrentemente. Na medida em que é concluída a atualização relativa a um grupo, o servidor torna-se membro ativo daquele grupo. A reintegração terminará totalmente quando o último procedimento concorrente terminar, se não houver falha. Um mecanismo de *timeout* evita que o servidor espere indefinidamente por atualizações de um grupo do qual não mais faz parte. Ao final da atualização dos arquivos, será necessário executar um coletor de lixo para os m primários replicados pelo secundário *n*.

Bibliografia

[AGN96] AGNOLETTO, Alessandro Dario. **Difusão de escritas em um sistema de replicação de arquivos com cópia principal no RNFS**: projeto de diplomação. Porto Alegre: Instituto de Informática da UFRGS, 1996. 47p.

[BAC86] BACH, Maurice J. **The Design of the UNIX Operating System.** Englewood Cliffs: Prentice-Hall, 1986. 471p.

[BHI91] BHIDE, A.; ELNOZAHY, E. N.; MORGAN, S. P. A highly available network file server. In: USENIX, 1991. **Proceedings...** [S.l.: s.n.], 1991. p.199-205.

[BIR96] BIRMAN, Kenneth. **Building Secure and Reliable Network Application**. Greenwish: Manning, 1996. 591p.

[BUD93] BUDHIRAJA, N. *et al.* The primary-backup approach. In: MULLENDER, Sape (Ed.). **Distributed Systems**. 2 ed. New York: ACM Press, 1993. p.199-216.

[CRI86] CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock synchronization in the presence of omissions and performance faults, and processor joins. In INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, 16., 1986. **Proceedings...** [S.l.: s.n.], 1986.

[COR91] CORBIN, J. R. **The art of distributed applications:** programming techniques for remote procedure calls. New York: Springer-Verlag, 1991. 321p.

[COU94] COULORIS, George; DOLLIMORE, Jenn; KINDBERG, Tim. **Distributed systems:** concepts and design**.** Wokinghan, England: Addison-Wesley, 1994. 644p.

[ELL83] ELLIS, C. S.; FLOYD, R. A. The Roe File System. In SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS, 3., 1983, Clearwater Beach. **Proceedings...** New York: IEEE, 1983. p.175-181.

[GAR82] GARCIA-MOLINA, Hector. Elections in a distributed computing system. **IEEE Transactions on Computers**, New York, v.c-31, n.1, p.48-59, Jan. 1982.

[GIF88] GIFFORD, D. K.; NEEDHAM, R. M.; SCHROEDER, M. D. The Cedar File System. **Communications of the ACM**, New York, v.31, n.3, p.288-298, Mar. 1988.

[GUE97] GUERRAOUI, Rachid; SCHIPER, André. Software-based replication for fault tolerance. **Computer**, New York, v.30, n.4, p.68-74, Apr. 1997.

[HAR 95] HARTMAN, John H., OUSYERHOUT, John K. The Zebra Striped Network File System. **ACM Transactions on Computer Systems**, New York. Aug. 1995. v.13, n.3. p.274-310.

[HAU97] HAUBEN, Ronda. **History of UNIX.** Disponível por WWW em http://www.dei.isep.ipp.pt/docs/unix-Part_I.html e http://www.dei.isep.ipp.pt/docs/unix-Part_II.html (17 Nov. 1997).

[JAL94] JALOTE, Pankaj. **Fault Tolerance in distributed systems**. Englewood Cliffs: Prentice Hall, 1994. p.257-306.

[KAA91] KAASHOEK, M. Frans; TANENBAUM, Andrew S. Fault tolerance using group communication. **Operating Systems Review**, New York, v.25, n.2, p. 71-74, Apr. 1991.

[KAU84] KAUNITZ, J.; VAN EKERT, L. Audit trail compaction for database recovery. **Communications of ACM**, New York, v.27, n.7, p.678-683, July 1984.

[KEE97] KEEN, John S.; DALLY, Willian J. Extended ephemeral logging: log storage management of applications whith long-lived transactions. **ACM Transactions on Database Systems**, New York, v.22, n.1, p.1-42, Mar.1997.

[KER90] KERNIGHAN, Brian W.; RITCHIE, Dennis M. **C: a linguagem de programação padrão ANSI**. Rio de Janeiro: Campus, 1990. p.173-194.

[LAD90] LADIN, Rivka; LISKOV, Barbara; SHRIRA, Liuba. Lazy Replication: Exploiting the Semantics of Distributed Services. **Operating Systems Review**, New York, v.25, n.1, p.49-55, Jan. 1991.

[LEB96] LEBOUTE, Mario Magalhães. **RNFS - Um sistema de arquivos distribuídos tolerante a falhas para o UNIX**. Porto Alegre: CPGCC da UFRGS, 1996. 85p.

[LEB97] LEBOUTE, Mario Magalhães; WEBER, Taisy Silva. Um sistema de arquivos distribuídos tolerante a falhas para o UNIX compatível com o NFS. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES, 9., 1997. **Anais...** Campos do Jordão: [s. n.], 1997. p.167-183.

[LEV90] LEVY, Eliezer; SILBERSCHATZ, Abraham. Distributed file systems: concepts and examples, **ACM Computing Surveys**, New York, v.22, n.4, p.321-374, Dec. 1990.

[LIS91] LISKOV, Barbara *et al.* A Replicated UNIX File System. **Operating Systems Review,** New York, v.25, n.1, p.60-64, Jan. 1991.

[LIS91a] LISKOV, Barbara *et al.* Replication in the Harp File System. **Operating Systems Review,** New York, v.25, n.5, p.226-238, 1991.

[MUL93] MULLENDER, Sape J., Kernel support for distributed systems. In S. MULLENDER (Ed.) **Distributed Systems.** 2. ed. New York: ACM Press, 1993. p.385-409.

[MUL97] MULTICS. Disponível por WWW em http://www.best.com/~thvv/multics. com (17 Nov. 1997).

[OUS88] OUSTERHOUT, J. K. *et al.* The Sprite Network Operating System. **Computer,** New York, v.21, n.2, p.23-35, 1988.

[PAS97] PASIN, Marcia. **Tolerância a falhas em sistemas de arquivos distribuídos**. Trabalho Individual I. Porto Alegre: CPGCC da UFRGS, 1997. 36p.

[PUP88] PU, Calton; NOE, J. D.; PROUDFOOT, A. Regeneration of replicated objects: a technique and its Eden implementation. **IEEE Transactions on Software Engineering**, New York, v.14, n.7, July 1988. p. 936-945.

[RFC94] Request For Comments 1094. **The NFS Protocol Especification.** Disponível por www.internic.net (Jan. 1996).

[SAN85] SANDBERG, R. *et al.* Design and implementation of the Sun Network File System. In: THE SUMMER USENIX CONFERENCE, 1985. **Proceedings...** [S.l.: s.n.], 1985. p.119-130.

[SAT93] SATYANARAYANAN, M. Distributed File Systems. In: MULLENDER, Sape (Ed.). **Distributed Systems.** 2. ed. New York: ACM Press, 1993. p.199-216.

[SCH93] SCHNEIDER, F. B. Replication management using the state machine approach. In: MULLENDER, Sape (Ed.). **Distributed Systems.** 2. ed. New York: ACM Press, 1993. p. 169-198.

[SIL93] SILBERSCHATZ, Abraham, KORTH, Henry F. **Sistema de Banco de Dados.** 2. ed. São Paulo: Makron Books, 1993. p.339-374.

[SIN94] SINGHAL, M.; SHIVARATRI, N. G. **Advanced concepts in operating systems**: distributed, database, and operating systems. [S. l.]: McGraw-Hill, 1994. 522p.

[TAN92] TANENBAUM, Andrew S. **Modern Operating Systems**. Englewood Cliffs: Prentice-Hall, 1992. 728p.

[TAN94] TANENBAUM, Andrew S. **Redes de Computadores**. Rio de Janeiro: Campus, 1994. 786p.

[TAN95] TANENBAUM, Andrew S. **Distributed Operating Systems**. Englewood Cliffs: Prentice-Hall, 1995. 614p.

[WAL83] WALKER, B. *et al.* The Locus distributed operating system. **Operating Systems Review**, New York, v.17, n.5, p.49-70, Dec. 1983.

Informática  **UFRGS**

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Reintegração de Servidores em Sistemas Distribuídos"

por

Marcia Pasin

Dissertação apresentada aos Senhores:

Prof. Dr. Ricardo de Oliveira Anido (UNICAMP)

Profa. Dra. Ingrid Eleonora Schreiber Jansch Porto

Prof. Dr. Jose Palazzo Moreira de Oliveira

Vista e permitida a impressão.
Porto Alegre, 07/08/98.

Profa. Dra. Taisy Silva Weber,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS