UNIVERSIDADE FEDERAL DA PARAÍBA

Centro de Ciências e Tecnologia
Coordenação de Pós-Graduação em Informática

DISSERTAÇÃO DE MESTRADO

SALIUS - SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS

por

Tatiana Simas Stanchi

Campina Grande / Paraíba – 2000

UNIVERSIDADE FEDERAL DA PARAÍBA

Centro de Ciências e Tecnologia
Coordenação de Pós-Graduação em Informática

Tatiana Simas Stanchi

SALIUS - SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS

Dissertação apresentada ao curso de Mestrado em Informática da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

Área de Concentração: Sistemas Distribuídos

Orientador: Francisco Vilar Brasileiro

Campina Grande / Paraíba – 2000

S784S

STANCHI, Tatiana Simas

SALIUS - SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS.

Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande - Pb, fevereiro de 2000.

127 p. Il.

Orientador: Francisco Vilar Brasileiro.

Palavras Chave:

1. Sistemas Distribuídos
2. Armazenamento Estável
3. Sistema de Arquivos Robusto
4. Tolerância a Falhas

CDU - 681.3.066D

Resumo

Armazenamento estável é um requisito importante para muitas aplicações. Os sistemas de arquivos tradicionais realizam a gravação síncrona em disco, como uma forma de garantir a estabilidade dos dados armazenados. Contudo, a gravação síncrona provoca uma degradação no desempenho das aplicações, em decorrência da lentidão dos acessos a disco.

À medida que a tecnologia evolui, aumentam os prejuízos no desempenho, causados pelo uso da gravação síncrona. Os avanços tecnológicos não acontecem no mesmo ritmo para todos os componentes de hardware: a velocidade dos processadores aumenta em proporções bem maiores que a velocidade de acesso a disco. Por isso, o tempo de acesso a disco é um entrave crescente ao desempenho de aplicações que realizam muitas operações de escrita, especialmente se a gravação for síncrona.

Esta dissertação apresenta a especificação de uma nova técnica de armazenamento estável, denominada **SALIUS — SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS**. A técnica proposta substitui a gravação síncrona pela replicação remota de *buffers* alterados na *cache*. Assim, a aplicação deixa de esperar pela operação de saída para disco, passando a aguardar pelo envio dos dados alterados através da rede e o armazenamento desses dados na memória principal de uma máquina remota. Como a velocidade de acesso à memória principal é progressivamente maior que a velocidade de acesso a disco, enquanto a transmissão de dados, através de uma rede de alta velocidade, é bem mais veloz do que a gravação da mesma quantidade de dados em disco, o principal benefício oferecido é a garantia de estabilidade dos dados, com um desempenho potencialmente superior ao de soluções baseadas na gravação síncrona em disco. A técnica proposta tende a ser cada vez mais vantajosa, porque a tecnologia de memória e a tecnologia de rede progridem num ritmo mais acelerado que a tecnologia de disco.

Para assegurar a estabilidade das informações, a técnica proposta inclui um mecanismo de recuperação para frente, capaz de restaurar o sistema de arquivos para um estado consistente, após um *crash.*

Abstract

Stable storage of data is an important requirement for many applications. Traditional file systems use synchronous writing to disk as a way of guaranteeing the stability of stored data. However, synchronous writing impairs application performance because of the slow speed of disk access.

As hardware technology advances, impairment to performance caused by the use of synchronous writing increases. The hardware component technologies develop at varying rates. The speed of processors is increasing at a greater rate than the speed of disk access. Thus disk access time is limiting the performance of applications which perform many disk access operations, especially in the case of synchronous writing.

This dissertation presents the specification of a new stable storage technique, called **SALIUS — STABLE STORAGE SERVICE WITH FORWARD RECOVERY BASED ON REMOTE REPLICATION OF BUFFERS**. The proposed service substitutes synchronous writing with remote replication of buffers that are modified in the cache. In this way the application does not wait for output to disk but instead waits for the sending of modified data through a network and the storage of this data in the main memory of a remote machine.

The main advantage of the proposed service is the guarantee of stability, with a potentially better performance than the solutions based on synchronous writing, due to the fact that the speed of memory access is progressively greater than that of disk access, and the transfer of data through a high speed network is considerably faster than writing the same quantity of data to disk. Since memory and network technologies are developing at a considerably greater speed than that of disk technology, this approach could be even more advantageous.

To assure the stability of the information, the technique includes a forward recovery mechanism which allows it to collect enough remote information to restore the file system to a consistent state after a crash.

Dedico esse trabalho à minha querida filha Malu, fonte de toda a minha inspiração.

Agradecimentos

O percurso trilhado durante um curso de mestrado nem sempre é fácil e tranqüilo. Agradeço a Deus a oportunidade de realizar esse trabalho e de aprender um pouco mais sobre a Arte de Viver.

Minha eterna gratidão aos meus pais, Arlindo (*in memorian)* e Ana Lúcia, sempre presentes com palavras de carinho e incentivo, de prontidão para atender às minhas solicitações. Sem eles, seria impossível cumprir essa jornada.

Agradeço à minha filhinha Malu, pela cumplicidade e compreensão, diante da falta de tempo e de tantas prioridades, que subtraíram-lhe horas de convívio e atenção. Sou igualmente grata à minha irmã Kátia, pela valiosa ajuda nos momentos difíceis e por torcer pelo meu sucesso, e a Edil, pelo companheirismo e por desejar a realização de meus sonhos e projetos.

Um agradecimento muito especial ao meu grande amigo e orientador Fubica, que tornou possível a realização de um sonho antigo, sabendo desempenhar, com competência, cada um desses papéis. Minha gratidão aos meus professores e amigos Raimundo Macêdo e Francisco Dórea (*in memorian*), que despertaram o meu interesse para a área acadêmica e foram grandes exemplos, e ao Prof. Marcus Sampaio, por instigar a minha vinda para esse curso. Agradeço, também, aos colegas da Procuradoria, que tanto me incentivaram.

Não poderia esquecer de agradecer à inestimável contribuição daqueles que fazem tudo acontecer: os funcionários do DSC, especialmente as secretárias da COPIN, Aninha e Vera, sempre muito prestativas; D. Inês, pela maravilhosa tapioca com queijo, e as minhas ajudantes, Conça e Zoraide, que cuidaram bem de minha casa, da minha filha e de mim.

Finalmente, um agradecimento enorme aos meus amigos e amigas, companheiros de vida, de Campina Grande (vindos de tantos lugares), de Salvador e de Recife, que me deram a força necessária para realizar esse sonho, compartilhando risos e lágrimas. A tentativa de citá-los poderia incorrer na injustiça de algum esquecimento.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

. Introdução

A evolução tecnológica exerce uma forte influência nas pesquisas de sistemas computacionais. À medida que a tecnologia avança, as características dos componentes de hardware mudam, exigindo que os algoritmos utilizados para gerenciar os sistemas sejam reexaminados e que novas técnicas sejam desenvolvidas. As influências tecnológicas são particularmente evidentes nos projetos de sistema de arquivos: algumas técnicas desenvolvidas nas duas últimas décadas já se tornaram obsoletas.

Um grande problema enfrentado pelos sistemas de arquivos é o ritmo evolutivo heterogêneo dos componentes de hardware. O projeto de um sistema de arquivos depende da tecnologia de processador, de memória principal, de disco e, mais recentemente, da tecnologia de rede. Nas duas últimas décadas, processadores, memórias principais e redes melhoraram por várias ordens de grandeza, tanto em desempenho, quanto em capacidade. A tecnologia de disco não acompanhou esse desenvolvimento, no que tange ao desempenho, em decorrência das limitações impostas pela natureza mecânica dos discos magnéticos.

A evolução desigual dos componentes de hardware desafia os sistemas de arquivos a compensarem a lentidão dos discos, permitindo que o desempenho cresça com a tecnologia de processador, de memória e de rede. O aumento do desempenho dos sistemas de arquivos deve superar o dos discos. Caso contrário, as aplicações serão incapazes de utilizar o rápido aumento da velocidade de processadores e dos demais componentes de hardware, para oferecer um melhor desempenho aos seus usuários.

Tradicionalmente, os sistemas de arquivos adotam técnicas de otimização de desempenho baseadas na utilização de memória principal. Porém, essas técnicas reduzem a confiabilidade do sistema, em decorrência da volatilidade da memória.

Algumas aplicações necessitam de armazenamento estável de dados, ou seja, não admitem a perda de informações, mesmo que ocorram falhas no sistema. Nesse caso, as técnicas de otimização baseadas apenas em memória principal não podem ser utilizadas, porque um *crash* no sistema apaga todo o conteúdo da memória volátil. Uma solução bastante simples, para *crashes* provocados pela interrupção do fornecimento de energia, é a utilização de *nobreaks*. Porém, essa solução não atende aos casos de *crash* do sistema operacional.

Os sistemas de arquivos tradicionais oferecem a gravação síncrona, como uma forma de prover armazenamento estável: a aplicação solicita ao sistema que inicie imediatamente a gravação, em disco, dos dados de uma requisição de escrita e espera pela finalização da operação de saída. Portanto, os sistemas de arquivos tradicionais obrigam as aplicações a pagarem um alto custo pelo armazenamento estável de dados, porque a gravação síncrona vincula o desempenho do sistema ao desempenho dos discos.

1.1 Objetivo

Este trabalho tem como objetivo apresentar a especificação de uma nova técnica de armazenamento estável, denominada **SALIUS[1] — SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS —**, como uma alternativa à gravação síncrona em disco. A idéia fundamental é garantir a sobrevivência de todos os dados armazenados através do uso das primitivas do serviço, no caso da ocorrência de um *crash* no sistema, e, durante o funcionamento normal, possibilitar um desempenho superior ao obtido por sistemas que adotam a gravação síncrona.

1.2 Armazenamento Estável

Um armazenamento é dito estável quando o seu conteúdo é preservado de falhas, ou seja, ocorrendo uma falha, os dados armazenados não são destruídos [Jal94]. Vários tipos de falhas podem ameaçar o conteúdo de um sistema de arquivos, tais como: *crash* no sistema, falhas no dispositivo de armazenamento, falhas na memória principal, dentre outras.

Este trabalho concentra-se apenas em falhas do tipo *crash*: após uma primeira omissão, o sistema deixa de produzir respostas para as entradas subseqüentes, até que seja reiniciado [Cri91], fazendo com que todo o conteúdo da memória volátil seja apagado. Portanto, para este trabalho, um armazenamento é considerado estável quando os dados estiverem gravados em um meio de armazenamento não-volátil, como os discos.

[1] SALIUS (do latim) eram os doze sacerdotes de Marte, encarregados da guarda dos escudos sagrados que protegiam a estabilidade da Roma Antiga.

1.3 Replicação

A replicação possibilita a construção de serviços tolerantes a faltas, através da reprodução de recursos vitais para o sistema, em componentes com modos independentes de falha. A replicação pode envolver tanto recursos de software, quanto de hardware [Lit92]. A replicação de recursos numa mesma máquina é denominada replicação local, enquanto a replicação de recursos em diferentes máquinas de um sistema é uma replicação remota.

A replicação de dados e informações de controle possibilita a recuperação de *crashes*: ao reiniciar, um sistema pode restaurar seu estado anterior ao *crash* e voltar a operar normalmente. O serviço de armazenamento estável apresentado nesta dissertação realiza a replicação remota de *buffers* da *cache* de arquivos. Assim, após um *crash*, os dados e os metadados de um sistema de arquivos podem ser recuperados, a partir de uma cópia existente em outra máquina, restaurando o sistema de arquivos para um estado consistente.

1.4 Enfoque da Pesquisa

A pesquisa apresentada nesta dissertação focaliza o sistema de arquivos do UNIX [RT78] [MJL+84], como ponto de partida para a discussão sobre armazenamento estável. A pesquisa parte do estudo de um modelo tradicional de sistema de arquivos, aqui representado pelo sistema de arquivos do UNIX, para analisar como os avanços tecnológicos influenciam na evolução das técnicas de otimização de desempenho, ao tempo em que tornam iminente o surgimento de novas técnicas de armazenamento estável.

Existem muitas vantagens em eleger o sistema UNIX como o foco inicial de uma pesquisa na área de sistema de arquivos: a maioria de aplicações UNIX faz uso intensivo do sistema de arquivos; as requisições de armazenamento no ambiente UNIX são diversificadas, variando tanto em tamanho, quanto na natureza, seqüencial ou randômica. Trata-se de um sistema de arquivos amplamente utilizado, o qual tem sido objeto de muitas pesquisas, havendo uma rica bibliografia disponível.

O trabalho apresenta uma especificação do Serviço de Armazenamento Estável com Recuperação para Frente Baseado na Replicação Remota de Buffers, voltada para o ambiente UNIX [Bac86]. Como idéia fundamental do serviço proposto é a replicação remota de dados alterados na *cache* de arquivos, trata-se de um projeto destinado a um ambiente distribuído. Por isso, o serviço de armazenamento estável proposto utiliza a capacidade de comunicação via rede do UNIX, para realizar a replicação.

Embora a especificação apresentada neste texto enfoque o sistema UNIX, a replicação remota de informações da *cache* de arquivos, como uma alternativa à gravação síncrona de dados, também pode ser aplicada a outros sistemas de arquivos. Assim, a técnica de armazenamento estável proposta tem um sentido genérico, embora o sistema de arquivos do UNIX seja utilizado como contexto para a sua descrição.

1.5 Estudo de Viabilidade

O primeiro passo dessa pesquisa consistiu na realização de testes de viabilidade, com o objetivo de comparar o tempo da gravação síncrona de um *buffer* com o tempo de replicação do mesmo *buffer*, na memória principal de outra máquina do sistema.

Os testes foram realizados no Laboratório de Sistemas Distribuídos da Universidade Federal da Paraíba, utilizando-se dois microcomputadores, ambos com processadores Intel 486 e discos com tempo de acesso de oito milissegundos, executando sistema operacional Linux [BBD+97], cuja versão do núcleo era 2.0.29. Os equipamentos estavam interligados através de uma rede padrão Ethernet, com taxa de transmissão de até dez megabits por segundo.

Os programas de teste foram escritos na linguagem C e a comunicação entre processos das duas máquinas foi estabelecida através de *sockets* [Ste92], utilizando protocolos TCP/IP [CS91] [CS93].

Foram realizados dois procedimentos de medição, sendo um para a gravação síncrona de um *buffer* e o outro para a replicação de um *buffer* idêntico. Os procedimentos foram executados para tamanhos de *buffers* diferentes, com uma repetição de cem vezes, para cada tamanho de *buffer*. O procedimento de medição da gravação síncrona utilizou um programa que preenchia o *buffer* e contava o tempo em milissegundos, necessário para realizar a gravação em disco.

O procedimento de medição da replicação utilizou dois programas: o primeiro, denominado de cliente, executado na máquina de origem, cuja função era preencher o *buffer* e computar o tempo em milissegundos, necessário para enviá-lo a uma outra máquina e receber um reconhecimento. O segundo programa, denominado servidor, executado na máquina destinatária, que recebia o *buffer* e retornava um reconhecimento.

A pretensão desses testes nunca foi chegar a resultados deterministas. Visaram apenas evitar a insistência em desenvolver um projeto que não tivesse possibilidades de êxito. Os resultados obtidos foram animadores e serviram de incentivo para que o projeto fosse posto a termo.

A Tabela 1.1 exibe o tempo médio de replicação e o tempo médio de gravação síncrona, em milissegundos, calculados para cada tamanho de *buffer*. Os resultados demonstraram que a replicação é bastante vantajosa para *buffers* pequenos: replicar um *buffer* de um *kilobyte* é vinte e cinco vezes mais rápido do que gravá-lo no disco.

Tamanho do *buffer* (KB)	Tempo médio de replicação (ms)	Tempo médio de gravação síncrona (ms)
0,5	0,9	20
1	1	25,5
4	6,2	28,1
6	15	29
8	16,4	30,1
12	24,3	31,5
14	25,2	32,8
16	34,1	33,8
24	43,2	35,4

Tabela 1.1: Resultados dos testes de viabilidade.

O gráfico da Figura 1.1 demonstra o comportamento do tempo médio de replicação e do tempo médio de gravação síncrona: à medida que o tamanho do *buffer* aumenta, a vantagem da replicação diminui; quando o *buffer* atinge dezesseis *kilobytes*, o tempo médio é praticamente o mesmo, para ambos os procedimentos; para *buffers* maiores, a replicação deixa de ser vantajosa. Portanto, existe um tamanho máximo de *buffer* (B-max), após o qual a replicação passa a ser mais lenta do que a gravação síncrona em disco.

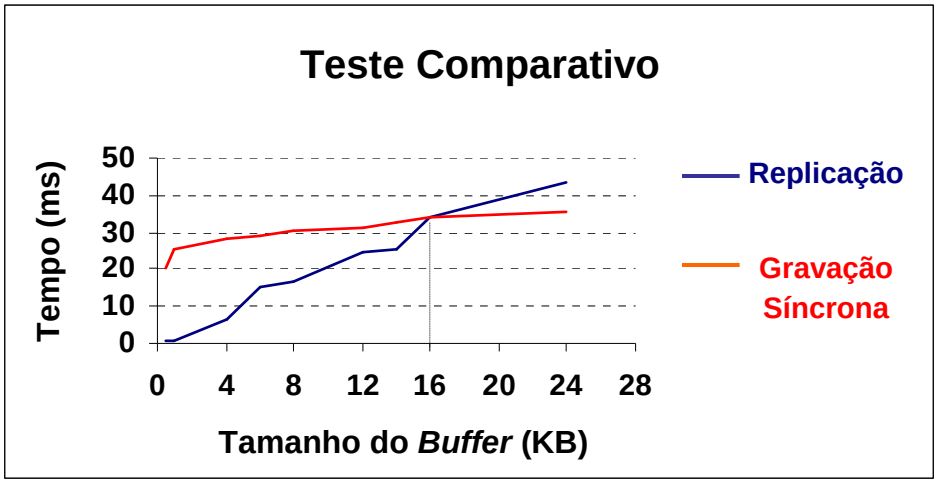

Figura 1.1: Gráfico comparativo entre a replicação e a gravação síncrona de um *buffer*.

À primeira vista, portanto, a replicação seria vantajosa, apenas, para sistemas de arquivos com blocos de tamanho pequeno. Porém, uma análise mais detalhada poderia demonstrar que o tamanho máximo de *buffer* (B-max) varia de acordo com as características dos componentes de hardware do sistema (o item 5.7 apresenta as influências dos componentes de hardware no desempenho da replicação). O B-max é derivado da relação entre o tempo médio de gravação síncrona (TmG) e o tempo médio de replicação (TmR): se o TmG diminui, então, o B-max diminui; se o TmR diminui, então, o B-max aumenta.

O tempo médio de gravação síncrona depende do tempo de acesso a disco: quanto mais rápido o acesso a disco, menor o TmG e também menor o B-max. No entanto, as tendências tecnológicas apontam para uma melhoria muito tímida do tempo de acesso a disco, em relação aos demais componentes de hardware. Assim, o TmG tende a uma variação muito pequena.

O tempo médio de replicação depende da taxa de transmissão de dados na rede e do tempo de acesso à memória principal: quanto mais velozes forem os acessos à memória e a transmissão através da rede, menor o TmR e maior o B-max. Assim, com o uso de rede de alta velocidade e de memória com tempo de acesso reduzido, pode-se aumentar significativamente o B-max. As tendências tecnológicas apontam para um grande aumento na velocidade das redes e para um aumento na velocidade de acesso à memória principal bem superior ao crescimento da velocidade de acesso a disco. Assim, as tendências da tecnologia indicam um provável crescimento progressivo de B-max.

Os testes realizados foram considerados satisfatórios, tendo-se em vista as tendências da tecnologia e os seguintes fatores:

- os testes utilizaram uma rede local, com taxa de transmissão máxima de dez megabits por segundo, sendo que atualmente estão disponíveis redes com taxas de transmissão bem maiores, de cem megabits, ou na ordem de gigabits por segundo, que devem levar a um aumento no tamanho máximo de *buffer*;
- os testes utilizaram os protocolos de comunicação TCP/IP. Por ser orientado à conexão e garantir a entrega confiável de mensagens, o TCP incorpora uma sobrecarga de processamento que retarda a transmissão de dados em uma rede convencional;
- os testes utilizaram processadores Intel 486, capazes de executar, aproximadamente, vinte MIPS (Milhões de Instruções Por Segundo). Atualmente, estão disponíveis equipamentos com processadores mais velozes (mais de 500 MIPS) [Int97];
- a quantidade de informações normalmente trafegadas pelo serviço é pequena. Quando um bloco é alterado, o serviço envia uma mensagem de replicação contendo o conteúdo do bloco e um cabeçalho com informações de controle, cujo tamanho é de apenas algumas poucas dezenas de bytes. Assim, a quantidade de dados trafegados é apenas ligeiramente maior que o tamanho do bloco do arquivo (a maioria dos sistemas de arquivos do UNIX utiliza blocos pequenos, sendo 4Kb o tamanho mais adotado).

1.6 Cenário de Aplicação

À medida que proliferam os sistemas computacionais ligados através de rede, crescem os clientes em potencial do SALIUS. Com o grande desenvolvimento da Internet, muitos serviços novos passaram a ser oferecidos. Alguns destes serviços requerem alta confiabilidade no armazenamento secundário de informações.

Um cenário típico de aplicação do SALIUS consiste de um servidor de transações, que utiliza um sistema de arquivos local para a gravação de seus *logs*. Como as informações contidas em um *log* são imprescindíveis para a recuperação de informações gravadas por transações já confirmadas, os dados de um *log* precisam de armazenamento estável. Num sistema de arquivos tradicional, esses dados seriam armazenados através de gravação síncrona, o que acarretaria uma perda no desempenho. Estando o servidor de transações ligado a uma rede local, o SALIUS pode ser utilizado para prover armazenamento estável, evitando os atrasos introduzidos pelas gravações síncronas.

Sistemas de banco a domicílio e de comércio eletrônico são exemplos típicos de aplicações que poderiam usar esse serviço de transações munido do SALIUS. Por exemplo, quando um cliente de um sistema de banco a domicílio deseja solicitar uma movimentação em sua conta, ele envia um formulário com os dados necessários. Ao receber tais informações, o servidor deve realizar a transação solicitada e enviar uma resposta ao cliente. Os dados da transação precisam ser assegurados. Por isso, o servidor de transações grava toda a movimentação num arquivo de *log*, que será utilizado para ações de recuperação após um *crash*. Num sistema de arquivos convencional, a aplicação deveria esperar pela gravação síncrona dos dados do *log*. Utilizando o SALIUS, os dados do *log* são replicados na memória principal de máquinas da rede local do servidor de transações. Como a replicação é potencialmente mais rápida do que a gravação síncrona em disco, com o serviço oferecido pelo SALIUS, a aplicação pode garantir um melhor tempo de resposta. Adicionalmente, o mecanismo de recuperação para frente do SALIUS garante a restauração dos dados do *log*, caso o servidor de transações sofra um *crash*.

1.7 Contribuições da Dissertação

As principais contribuições deste trabalho são:

- a especificação de uma nova técnica de armazenamento estável, potencialmente capaz de substituir, de forma vantajosa, as soluções baseadas apenas na gravação síncrona em disco. Tomando-se como base as velocidades de acesso a disco, as taxas de transmissão das redes de alta velocidade e as velocidades de acesso à memória principal, disponíveis atualmente, uma implementação eficiente do SALIUS deverá levar a um desempenho superior ao armazenamento através de gravação síncrona em disco;

- o serviço proposto é independente de hardware especial. A maioria dos trabalhos recentes, que visam aumentar a confiabilidade do armazenamento de dados, depende da utilização de um hardware especial, o que quase sempre acarreta altos custos. A técnica de armazenamento estável proposta nesta dissertação, pelo contrário, utiliza a tecnologia atualmente disponível na maioria dos sistemas computacionais;
- o SALIUS possibilita que os avanços tecnológicos em curso beneficiem mais o sistema, porque consegue diminuir a influência da tecnologia de disco no desempenho global do sistema: eliminando a urgência de gravar dados em disco, o serviço permite que o sistema utilize mais agressivamente as técnicas de otimização de desempenho, baseadas no armazenamento em memória principal. O desempenho do sistema passa a depender mais dos processadores, da capacidade de memória disponível, da velocidade de acesso à memória, das taxas de transmissão da rede e da banda passante da rede disponível para esse serviço, enfraquecendo o vínculo existente com a tecnologia de disco;
- o SALIUS pode ser utilizado por sistemas de arquivo de propósito geral;
- o SALIUS assegura a estabilidade de todos os dados armazenados, através do uso de suas primitivas, incluindo as últimas alterações realizadas sobre o sistema de arquivos antes de um *crash*. Para isso, o serviço adota um protocolo de replicação capaz de garantir a existência de, pelo menos, uma cópia dos dados alterados na *cache* de arquivos, e um mecanismo de recuperação que utiliza essa cópia para restaurar o sistema de arquivos;
- o SALIUS é capaz de tolerar *crash* de sistema operacional;
- o mecanismo de recuperação do serviço é simples e potencialmente rápido: não é preciso verificar todos os metadados do sistema de arquivos, detendo-se apenas aos dados e metadados possivelmente afetados pelo *crash*, ou seja, aqueles que foram alterados na memória principal e replicados.

1.8 Organização da Dissertação

Este documento está organizado em seis capítulos. O próximo capítulo descreve o sistema de arquivos do UNIX, que é utilizado como contexto do trabalho, e a técnica de armazenamento estável baseada na gravação síncrona em disco. Adicionalmente, discutem-se as principais técnicas de otimização de desempenho adotadas pelos sistemas de arquivos tradicionais. Esse capítulo foi incluído na dissertação, com o objetivo de agregar ao documento todos os conceitos prévios importantes à sua compreensão. Entretanto, para um leitor familiarizado com a estrutura interna do sistema de arquivos do UNIX e com as técnicas de otimização de sistemas de arquivos, pode ser conveniente avançar diretamente para o Capítulo 3.

O Capítulo 3 revisa as questões relevantes para as pesquisas na área de sistema de arquivos, provendo a motivação e a direção deste trabalho. Apresenta os efeitos dos avanços tecnológicos em curso e as necessidades de armazenamento das novas aplicações, evidenciando os novos requisitos que precisam ser contemplados pelos projetos de sistema de arquivos.

O Capítulo 4 apresenta o estado da arte em sistemas de arquivos robustos, descrevendo e analisando, de forma crítica, os trabalhos relacionados, que visam a melhoria do desempenho e da confiabilidade dos sistemas de arquivos. Esse capítulo também evidencia o diferencial da técnica de armazenamento estável proposta.

O Capítulo 5 apresenta a especificação do Serviço de Armazenamento Estável com Recuperação para Frente Baseado na Replicação Remota de Buffers. Inicialmente, esse capítulo descreve os principais objetivos e requisitos do serviço. Em seguida, apresenta o funcionamento geral do serviço e descreve mais detalhadamente cada parte componente: a Interface, o Servidor de Arquivos Complementar e sua interação com o Serviço de Replicação de Buffers, além do Procedimento de Recuperação. Finalmente, o capítulo apresenta considerações sobre o desempenho e a confiabilidade do SALIUS.

O Capítulo 6 apresenta o Projeto de um Serviço de Replicação de Buffers, descrevendo os requisitos do serviço, as características do grupo de réplicas, o modelo de replicação utilizado, o protocolo de comunicação e o protocolo de replicação do serviço.

O Capítulo 7 apresenta as conclusões obtidas e as sugestões de trabalhos futuros, de continuação dos estudos apresentados nesta dissertação.

Capítulo 2

Sistema de Arquivos Tradicional

Um sistema de arquivos é um subsistema do sistema operacional, cujo propósito é prover armazenamento de informações a longo prazo [LS90]. Para isso, o subsistema armazena dados em unidades denominadas *arquivos*, em dispositivos de armazenamento secundário, como os discos [Tan92]. Um sistema de arquivos atende às requisições de usuários, realizando operações sobre os arquivos, tais como: criação, remoção, leitura e escrita.

Projeto de sistema de arquivos é uma área de pesquisa da computação relativamente antiga. Muitos projetos já foram propostos e implementados. Esses projetos adaptam-se às mudanças dos sistemas computacionais, provocadas pela evolução da tecnologia. Este capítulo apresenta uma análise breve da evolução dos sistemas de arquivos.

A especificação do Serviço de Armazenamento Estável com Recuperação para Frente Baseado na Replicação Remota de Buffers, apresentada nesta dissertação, é voltada para o ambiente UNIX [RT78], em decorrência de vários fatores, dentre eles: o sistema de arquivos do UNIX é amplamente utilizado, tanto na academia, quanto comercialmente; existe uma vasta bibliografia disponível, documentando o funcionamento desse sistema e as pesquisas realizadas ele. Por isso, este capítulo descreve as abstrações básicas e as técnicas de implementação do sistema de arquivos do UNIX, como ponto de referência para a discussão sobre técnicas de otimização de sistemas de arquivos e técnicas de armazenamento estável.

Mais propriamente, este capítulo descreve a técnica de armazenamento estável empregada pelo sistema de arquivos do UNIX, cujo modelo é adotado pela maioria dos sistemas de arquivos tradicionais. Adicionalmente, este capítulo apresenta algumas técnicas de otimização de sistemas de arquivos, e descreve um sistema de arquivos em rede, o NFS [SGK+85], muito utilizado para a integração de dados de sistemas de arquivos do UNIX.

2.1 Evolução dos Sistemas de Arquivos

Inicialmente, todos os sistemas de computadores eram centralizados, ou seja, compostos por um único processador, uma memória, alguns periféricos e terminais [Tan92]. Num sistema centralizado, todos os dispositivos de armazenamento secundário e demais periféricos estão ligados a uma única CPU. O sistema operacional, normalmente formado por um grande núcleo monolítico, permite que os usuários compartilhem os recursos disponíveis. O sistema de arquivos, como um subsistema do sistema operacional, também é centralizado, atendendo às requisições de todos os usuários. O sistema de arquivos e seus usuários são implementados como processos, executados numa mesma máquina.

Durante a década de oitenta, o desenvolvimento de microprocessadores com grande poder computacional e a implementação de redes locais, com taxas de transferência superiores a dez megabits por segundo, alteraram o perfil dos sistemas computacionais. Surgiram sistemas compostos por várias CPUs, conectadas através de redes de alta velocidade [Tan95]. Os sistemas operacionais foram acrescidos de novas funcionalidades, como, por exemplo, os subsistemas de comunicação, que permitem a troca de informações entre processos executando em máquinas diferentes. Essa facilidade de comunicação possibilitou uma interação cada vez maior entre os sistemas operacionais das diversas máquinas componentes do sistema, até o surgimento de sistemas distribuídos:

> "*Um sistema distribuído é uma coleção de computadores independentes, que aparecem como um único computador para os usuários do sistema*" [Tan95].

A discussão sobre sistemas distribuídos requer a definição prévia dos conceitos de serviço, servidor e cliente [Mitchell. Apud LS90:322]. Um *serviço* é um software que pode ser executado em uma ou mais máquinas, provendo uma função específica para os clientes. Um *serviço* especifica um conjunto de operações, cuja execução pode ser disparada por entradas de clientes, ou pela passagem do tempo [Cri91]. A execução de uma operação pode resultar em saídas para os clientes, ou em mudanças de estado. Um *servidor* é uma implementação do serviço, na forma de um processo, executado em uma máquina do sistema. Um *cliente* é um processo que solicita serviços a um servidor, através de uma interface.

Usando a terminologia previamente definida, um sistema de arquivos distribuído provê serviço de arquivos para seus clientes, sendo implementado através de um ou vários servidores de arquivos. A interface do serviço de arquivos é formada por um conjunto de primitivas e seus respectivos parâmetros, usadas pelos clientes, para solicitar que um servidor realize operações com arquivos, tais como: criação, remoção, leitura e escrita, dentre outras. Os clientes, os servidores e os dispositivos de armazenamento estão espalhados em diversas máquinas [LS90].

2.2 Sistema de Arquivos do UNIX

O sistema de arquivos do UNIX [RT78] é um modelo clássico de sistema de arquivos centralizado. No UNIX, um arquivo é tratado como uma seqüência não-estruturada de bytes. O sistema de arquivos atende às requisições de acesso das aplicações, armazenando ou recuperando uma quantidade de bytes solicitada. A interpretação do conteúdo do arquivo é uma tarefa das aplicações.

Cada arquivo está associado a uma estrutura de dados, denominada de *nó-i* (nó índice), onde o sistema armazena os atributos do arquivo, tais como: tipo, tamanho, tempos da última modificação e do último acesso, proprietário e permissões de acesso. O sistema de arquivos mantém um conjunto de *nós-i*, numerados seqüencialmente. O número de um *nó-i* é um inteiro, que identifica unicamente o arquivo associado. O *nó-i* é o ponto de partida para o acesso aos dados de um arquivo, porque armazena uma tabela com os endereços dos blocos do arquivo.

As aplicações referenciam os arquivos por nomes. O sistema de arquivos traduz o nome de um arquivo no número do *nó-i* correspondente. Para isso, o sistema utiliza um tipo especial de arquivo, denominado *diretório*, que contém pares de nomes de arquivos e seus respectivos números de *nó-i*.

Um diretório pode conter entradas, tanto para nomes de arquivos, quanto para nomes de outros diretórios, permitindo uma organização hierárquica do sistema de arquivos. Por isso, a estrutura de diretórios é representada na forma de uma árvore ou de um grafo. O topo da hierarquia é um *diretório raiz*, os nós intermediários são os demais diretórios e as folhas são os arquivos [Tan92]. Quando existem entradas para um mesmo arquivo em mais de um diretório, a hierarquia assume a forma de um grafo. O *nó-i* do arquivo guarda a informação do número de diretórios que referenciam o arquivo.

A localização de um arquivo é realizada através de um *nome de caminho*, formado de uma seqüência de nomes componentes, separados por barras simples. Cada componente é um nome de arquivo contido no diretório representado pelo componente anterior [Bac86]. Por exemplo, o nome de caminho "*/usr/docs/curso*" referencia o arquivo *curso*, contido no diretório *docs*, o qual está contido no diretório *usr*. O sistema converte um nome de caminho no *nó-i* do arquivo correspondente. Para isso, realiza uma pesquisa linear em cada diretório do nome de caminho, buscando o número do *nó-i* do componente seguinte. Se este for outro diretório, o sistema utiliza o *nó-i* para localizar os seus blocos de dados. A seguir, lê o conteúdo do diretório, localizando o *nó-i* do próximo componente do nome de caminho e assim, sucessivamente, até chegar ao *nó-i* do arquivo.

2.2.1 Manipulação de Arquivos

O UNIX adota o modelo tradicional de acesso a arquivos: primeiro, o arquivo deve ser aberto; seguem as operações de leitura e/ou escrita e, finalmente, o arquivo deve ser fechado. Para cada arquivo aberto, o sistema mantém um indicador da *posição corrente* dentro do arquivo. As operações de acesso começam na posição corrente. A operação de escrita permite à aplicação transferir uma quantidade informada de dados para um arquivo, a partir da posição corrente. A operação de leitura permite que a aplicação leia dados do arquivo para a memória principal, iniciando na posição corrente. Após cada operação de acesso, o sistema atualiza a posição corrente. Para realizar acessos aleatórios, a aplicação deve, previamente, solicitar que o sistema altere a posição corrente do arquivo.

As aplicações não manipulam diretamente os dados armazenados: requisitam que o sistema de arquivos realize as operações desejadas [AT89]. O diagrama da Figura 2.1 mostra que o sistema de arquivos do UNIX é um componente do núcleo do sistema operacional. Os processos de usuário comunicam-se com o núcleo, invocando as *chamadas de sistema*, ou através de primitivas das *bibliotecas de interface*, que são vinculadas aos programas do usuário em tempo de compilação [Bac86]. As chamadas de sistema ordenam operações ao núcleo e realizam a troca de informações entre o núcleo e os processos de usuário.

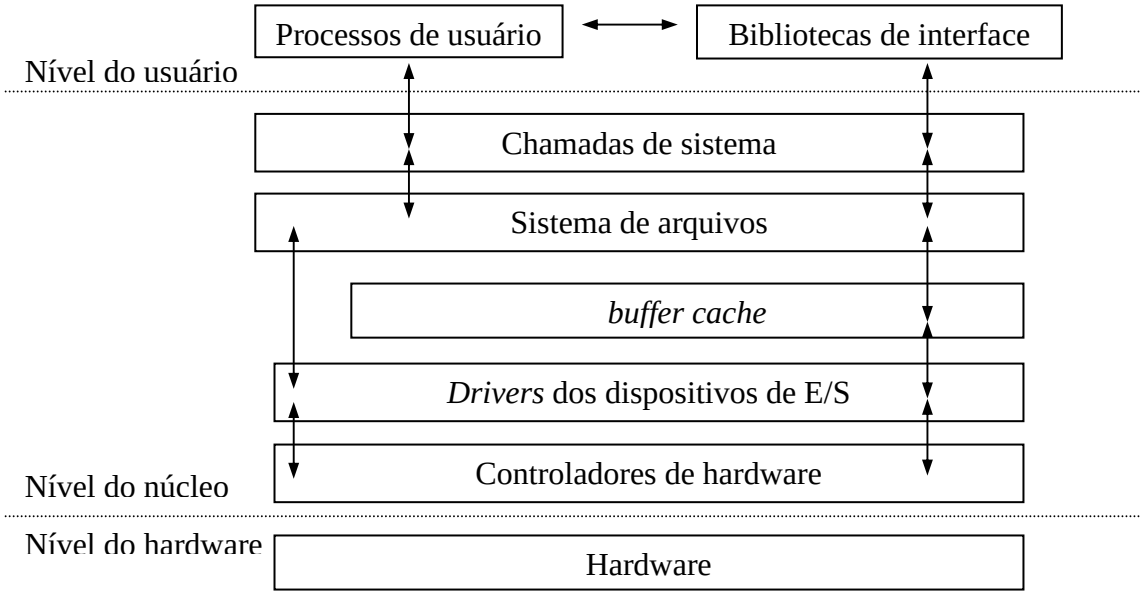

Figura 2.1: Diagrama da manipulação de arquivos no UNIX.

O UNIX oferece aos programas de usuário uma interface com muitas opções de chamadas de sistema [AT89]. As chamadas mais utilizadas são: *open*, utilizada para abrir ou criar um arquivo; *read*, que permite a leitura de uma quantidade informada de bytes de um arquivo; *write*, usada para armazenar dados em um arquivo; *close*, utilizada para fechar arquivos; *mkdir*, que serve para criar um novo diretório; *link*, que permite a criação de um novo nome de caminho para um arquivo; *unlink*, utilizada para remover um nome de caminho, ou um arquivo inteiro, e *seek*, que permite alterar a posição corrente no arquivo. Existem, ainda, chamadas de sistema para recuperar e alterar os atributos do arquivo.

O sistema de arquivos do UNIX utiliza um mecanismo de *buffering*, exibido na Figura 2.1 como *buffer cache*, que regula o fluxo de dados entre o núcleo do sistema e os dispositivos de armazenamento secundário. O mecanismo de *buffering* interage com os *drivers* dos dispositivos de entrada/saída, para iniciar a transferência dos dados. Alternativamente, o sistema de arquivos pode interagir diretamente com os *drivers* dos dispositivos de entrada/saída, sem a intervenção do mecanismo de *buffering*.

2.2.2 Implementação

A implementação inicial do sistema de arquivos do UNIX [RT78] é muito simples e não utiliza muitas técnicas de otimização. No entanto, a sua descrição é importante, pois forma a base para as implementações subseqüentes de sistemas de arquivos para UNIX.

2.2.2.1 Organização do Disco

Uma instalação do sistema operacional UNIX pode possuir vários discos, cada um contendo um ou mais sistemas de arquivos [Tan92]. O núcleo trata um sistema de arquivos como um dispositivo lógico, identificado por um número de dispositivo e composto de uma seqüência de blocos lógicos [Bac86]. Na implementação inicial, o tamanho dos blocos de um sistema de arquivos é fixo e múltiplo de 512 bytes, homogêneo num sistema de arquivos, mas podendo variar de um sistema para outro. O *driver* de disco realiza a conversão de um número lógico de bloco em um endereço físico, no disco. A Figura 2.2 ilustra a estrutura de um sistema de arquivos, composta das seguintes regiões:

- **Bloco de boot**. Não é utilizado pelo sistema de arquivos. Quando coincide com o bloco zero do disco, contém o código dos procedimentos iniciais, executados assim que o computador é ligado.
- **Superbloco**. Contém um sumário de informações acerca do sistema de arquivo, incluindo a localização das outras regiões; o número de arquivos existentes; o próximo *nó-i* livre; o número total de blocos livres na região de dados; informações para a localização de blocos livres; além de informações de estado, indicando se o sistema está disponível apenas para leitura e se o conteúdo do superbloco foi modificado.

- **Lista de *nós-i*.** Uma seqüência de *nós-i*, com os atributos de arquivos e diretórios.
- **Blocos de dados.** Blocos contendo os dados de arquivos e diretórios.

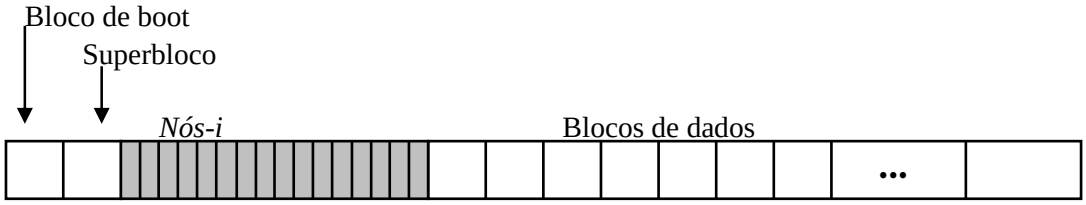

Figura 2.2: Organização de um sistema de arquivos do UNIX, no disco.

2.2.2.2 Endereçamento de Blocos

No *nó-i* estão armazenados os atributos do arquivo e uma tabela de endereços composta de treze entradas, com endereços de blocos do arquivo. A Figura 2.3 ilustra a tabela de endereços de um *nó-i*. As dez primeiras entradas apontam para blocos diretos, que armazenam dados. As três entradas restantes endereçam blocos indiretos. Um bloco indireto simples armazena endereços de blocos diretos, um bloco indireto duplo armazena endereços de blocos indiretos simples e um bloco indireto triplo armazena endereços de blocos indiretos duplos.

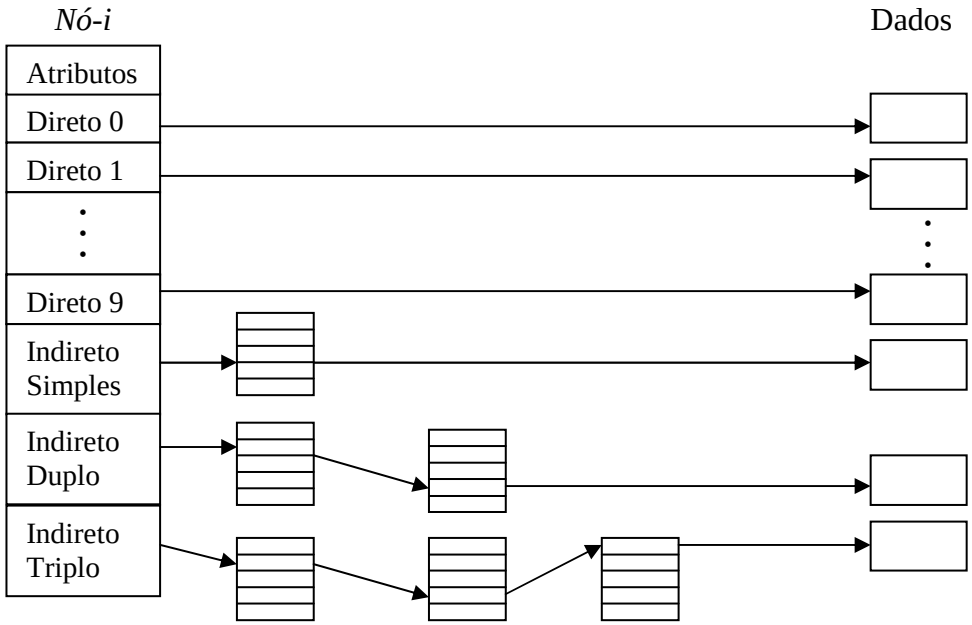

Figura 2.3: Blocos endereçados por um *nó-i* de um arquivo Unix.

2.2.2.3 Gerência do Espaço Livre

O sistema de arquivos do UNIX [RT78] gerencia o espaço livre em disco, através de uma lista ligada, chamada *lista de blocos livres*, formada pelos blocos da região de dados que não estão correntemente alocados para um arquivo ou diretório. O início da lista de blocos livres é um ponteiro no superbloco, para o primeiro bloco de dados livre. Quando uma operação de escrita, ou de criação de arquivo, requer espaço na região de dados do disco, o sistema aloca blocos da lista. Quando uma operação de remoção, ou de redução do tamanho de um arquivo, libera espaço em disco, os blocos retornam para a lista.

2.2.2.4 Tabelas Internas

Para executar as chamadas de sistema, o sistema de arquivos manipula estruturas de dados auxiliares. São tabelas internas, armazenadas na área de memória principal reservada ao sistema operacional e, portanto, não são acessíveis aos processos de usuário [Tan92].

- **Tabela de *nós-i*.** O sistema mantém uma entrada na tabela de *nós-i* para cada arquivo aberto. A cópia de um *nó-i* na tabela é única, mesmo que vários processos estejam trabalhando com o arquivo, concorrentemente. Além do *nó-i*, cada entrada da tabela armazena um contador de ligações, para controlar quantas referências existem para o arquivo, num determinado momento. O *nó-i* só é retirado da tabela quando todos os processos fecham o arquivo e o contador de ligações assume o valor zero.

- **Tabela de arquivos abertos.** Todos os arquivos abertos no sistema possuem entradas na tabela de arquivos abertos. Cada vez que um processo abre um arquivo, o sistema cria uma nova entrada nessa tabela. Assim, num dado momento, podem existir várias entradas na tabela referenciando um mesmo arquivo. Em cada entrada, estão registrados: o modo como o arquivo foi aberto (apenas para leitura, para leitura e escrita, ou apenas para escrita), a posição corrente no arquivo, onde será realizada a próxima operação e um apontador para o *nó-i* do arquivo, na tabela de *nós-i*. Existe, ainda, um contador para controlar o número de processos apontando para essa entrada.

- **Tabela de descritores de arquivos.** Enquanto a tabela de *nós-i* e a tabela de arquivos abertos são estruturas globais, o núcleo aloca uma tabela de descritores de arquivos, para cada processo. Quando um processo abre, ou cria um arquivo, o sistema aloca uma nova entrada nessa tabela e retorna para o processo um descritor de arquivo. Nas chamadas de sistema subseqüentes, o processo passa a referenciar o arquivo pelo seu descritor, que é usado como índice na tabela de descritores. Cada entrada na tabela de descritores aponta para uma única entrada na tabela de arquivos abertos.

A existência de dois tipos de tabelas, para controlar os arquivos abertos no sistema, tem o objetivo de possibilitar o compartilhamento da posição corrente de um arquivo entre dois processos. Se a entrada na tabela de descritores apontasse diretamente para o *nó-i* do arquivo, a posição corrente desse arquivo seria armazenada nessa tabela, sendo, portanto, de uso exclusivo do processo. Armazenando a posição corrente do arquivo numa tabela separada, o sistema permite que processos filhos utilizem a mesma posição corrente de um arquivo aberto pelo processo pai [Bac86]. Assim, num dado momento, podem existir vários apontadores para uma mesma entrada na tabela de arquivos abertos, provenientes de processos diferentes, mas que são relacionados.

A Figura 2.4 ilustra as três tabelas mantidas pelo núcleo. Observando dois arquivos abertos no sistema, o arquivo A e o arquivo B, nota-se que cada um possui uma entrada única na tabela de *nós-i*. Seguindo as setas, pode-se observar que o processo 1 abriu o arquivo A duas vezes, primeiro para leitura e, a seguir, para escrita, enquanto o processo 2 abriu o arquivo B para leitura.

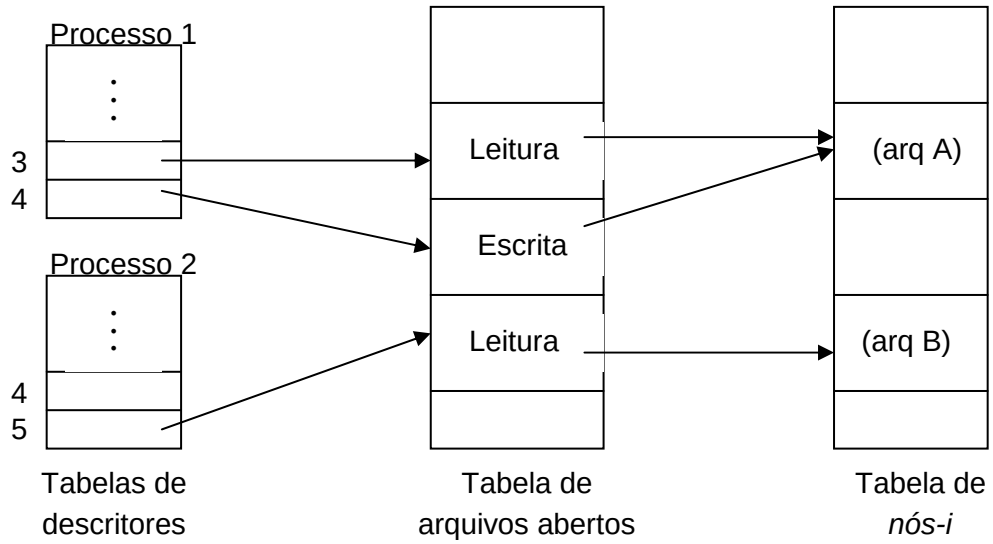

Figura 2.4: Tabelas internas do sistema de arquivos do UNIX.

2.2.2.5 Buffer Cache

Quando um processo de usuário precisa ter acesso a informações gravadas no disco, ele realiza uma chamada de sistema, requisitando ao núcleo a leitura de blocos do disco para uma área de trabalho, na memória principal. Se a operação for de escrita, o núcleo realiza a gravação em disco dos dados armazenados na área de trabalho do processo. Como o acesso a disco é muito lento, o sistema reserva uma área da memória principal, denominada de *buffer cache*, onde mantém os blocos de disco mais recentemente usados [Bac86].

A *cache* é implementada numa área de memória de acesso exclusivo do núcleo. Portanto, não pode ser manipulada diretamente por processos de usuário. O sistema armazena os dados lidos do disco na *cache*, antes de enviá-los para a área de trabalho do processo de usuário. Uma única chamada de sistema, para realização de uma leitura, pode preencher vários *buffers* da *cache*. Os dados permanecem por algum tempo na *cache*, de modo que requisições seguintes, dos mesmos dados, podem dispensar o acesso ao disco. Portanto, o sistema só precisa ler um dado do disco, se este não estiver disponível na *cache*.

De modo similar, os dados das operações de escrita são transferidos da área de trabalho do processo de usuário para a *cache*. Normalmente, o sistema de arquivos do UNIX realiza a *propagação retardada* das alterações: os dados permanecem na *cache*, por trinta segundos, antes de ser gravados em disco. Nesse intervalo de tempo, várias operações de escrita podem ser realizadas sobre um mesmo bloco, sem que sejam propagadas para o disco.

Durante o processamento de uma requisição de escrita, o sistema armazena os dados em *buffers* da *cache*, marca tais *buffers* como "sujos" e devolve imediatamente o controle para o processo de usuário. Portanto, quando uma operação de escrita retorna como bem-sucedida, o processo de usuário não tem a garantia de que os dados serão efetivamente gravados em disco. A ocorrência de um *crash*, nos trinta segundos próximos, ocasiona a perda das modificações realizadas. Para minimizar o risco de perder dados, a cada trinta segundos, o sistema executa a chamada *sync*, que realiza a gravação em disco de todos os *buffers* "sujos" da *cache*.

2.2.2.6 Unix FFS

A maior revisão do sistema de arquivos do UNIX foi realizada em Berkeley, resultando no UNIX FFS (*Fast File System*) [MJL+84]. As alterações realizadas se relacionam com a utilização do espaço em disco, com o objetivo de melhorar o desempenho do sistema de arquivos. As principais mudanças são:

- no UNIX FFS, os *nós-i* não estão concentrados no início do disco. O sistema divide o disco em regiões de cilindros adjacentes, denominadas *grupos de cilindros*. Cada grupo de cilindro possui sua região de *nó-i* e sua região de dados. Assim, mesmo em discos grandes, os *nós-i* estão mais próximos da região de dados;
- o UNIX FFS permite a existência de até dois tamanhos de bloco, iniciando com o tamanho mínimo de 4096 bytes, até o limite imposto pela controladora de disco, ou pelos *drivers*. A utilização de blocos maiores reduz, em potencial, a quantidade de acessos a disco, mas aumenta a probabilidade de ocorrer um fenômeno denominado de *fragmentação interna*: freqüentemente, o último bloco de um arquivo não é totalmente preenchido, provocando um desperdício de espaço. Para evitar a fragmentação interna, o UNIX FFS introduziu o conceito de *fragmento de bloco*, que ocupa apenas uma porção de um bloco do sistema de arquivos;

- o gerenciamento do espaço livre foi alterado, para tornar a alocação de blocos mais rápida. A lista de blocos livres foi substituída por um *mapa de bits*, que registra o estado de alocação de cada bloco e fragmento do disco. Como o *mapa de bits* é pequeno, ele pode ser mantido na memória, dispensando acessos a disco durante a alocação de blocos;
- o UNIX FFS possui um novo algoritmo de alocação de blocos, que procura: alocar o *nó-i* e os blocos de um arquivo no mesmo grupo de cilindros; alocar os blocos de um arquivo da forma mais contígua possível e concentrar, sempre que houver espaço, os arquivos de um mesmo diretório em um único grupo de cilindros. Dessa forma, o UNIX FFS consegue diminuir a quantidade de posicionamentos do braço, durante o processamento das requisições de acesso.

2.2.3 Armazenamento Estável no UNIX

O sistema de arquivos do UNIX consegue prover armazenamento estável através da *gravação síncrona* de dados. Na gravação síncrona, o sistema ativa o *driver* de disco, para iniciar imediatamente a transferência do conteúdo alterado nos *buffers* da *cache*, para os blocos de disco apropriados. O processo de usuário requisitante da chamada de sistema é bloqueado, até que a gravação termine.

O sistema oferece duas maneiras de uma aplicação solicitar a gravação síncrona de dados. Primeiro, a aplicação pode informar ao sistema, no momento em que abrir o arquivo, que todas as operações de escrita devem ser realizadas de forma síncrona. A aplicação faz essa opção, passando o parâmetro O_SYNC na chamada de sistema *open* [AT89]. O sistema guarda essa informação na tabela de arquivos abertos, realizando a gravação síncrona dos dados de todas as requisições de escrita para esse arquivo. Quando uma operação de escrita retorna como bem-sucedida, o processo de usuário tem a certeza de que os dados foram realmente gravados no disco.

Uma segunda forma de realizar a gravação síncrona é invocando uma operação complementar, através da primitiva *fsync*, logo após a operação de escrita no arquivo [Bac86]. A primitiva *fsync* bloqueia o processo do usuário, enquanto transfere para o disco o conteúdo de todos os *buffers* "sujos" da *cache*, com dados do arquivo informado. O processo de usuário só continua a sua execução, quando a gravação dos dados termina.

Existe, ainda, a primitiva *sync*, que transfere o conteúdo de todos os *buffers* "sujos" da *cache* para o disco. Porém, ela é inadequada para implementar o armazenamento estável de um arquivo: além de gravar conteúdos do sistema de arquivos, que não se relacionam com o arquivo em questão, o processo de usuário não espera pelo final da operação. Assim, embora o sistema de arquivos dispare imediatamente a gravação dos *buffers* para disco, a aplicação não tem a garantia de que a operação será concluída com sucesso.

Quando uma aplicação realiza muitas operações de escrita, a espera pela gravação síncrona pode comprometer o seu desempenho. Portanto, nem sempre as aplicações podem arcar com o custo do armazenamento estável de dados, no sistema de arquivos do UNIX.

2.2.4 Procedimento de Recuperação do UNIX

Algumas circunstâncias, como a falta de energia ou falhas no hardware, podem levar a um *crash*, deixando o sistema de arquivos num estado inconsistente. O comando *fsck* é utilizado para investigar inconsistências e reparar o sistema de arquivos [AT89]. Trata-se de um programa interativo, que utiliza a interface de caracter, ou de bloco, para ter acesso direto ao sistema de arquivos, sem valer-se das chamadas de sistema regulares [Bac86].

O *fsck* prevê uma série de possíveis situações de inconsistência, resultantes da perda de memória volátil, como, por exemplo: blocos de dados referenciados por mais de um *nó-i*; blocos de dados que não são referenciados; superbloco com informações que não condizem com o estado real do sistema; lista de blocos livres corrompida; *nó-i* com informações inválidas; diretórios desconectados, ou com entradas incorretas. Para realizar tais verificações, o *fsck* examina todos os metadados existentes no disco, incluindo todos os *nós-i*, blocos indiretos e diretórios, além da lista de blocos livres. Pesquisando essas estruturas de dados e corrigindo as inconsistências, o *fsck* consegue levar o sistema de arquivos a um estado consistente.

Uma limitação desse procedimento de recuperação é a necessidade de conferir, incondicionalmente, todos os metadados do sistema de arquivos, a despeito da quantidade de metadados inconsistentes [RO92]. Certas operações, como a verificação de cada entrada de diretório, podem requerer muitas leituras não-seqüenciais, que exigem múltiplas operações de posicionamento. No caso de discos grandes, com muitos arquivos, a leitura e o exame de todos os metadados, em geral, duram um longo tempo. Quanto maior a capacidade do disco, maior o tempo de execução do procedimento de recuperação. Normalmente, apenas um pequeno número de metadados está sendo atualizado e pode tornar-se inconsistente, no momento de um *crash*. Se o procedimento de recuperação pudesse identificar as estruturas afetadas pelo *crash*, haveria a possibilidade de restaurar a consistência do sistema de arquivos mais rapidamente.

Outra limitação do procedimento de recuperação do UNIX é o confinamento das ações corretivas aos metadados do sistema de arquivos. O *fsck* restringe-se a corrigir os valores dos metadados, para que voltem a corresponder ao estado real do sistema de arquivos, armazenado em disco. Esse procedimento é incapaz de recuperar os blocos de dados que estavam armazenados na *cache*. Por isso, as aplicações podem perder as últimas atualizações realizadas.

2.3 Técnicas de Otimização do Desempenho

Ao longo das últimas décadas, muitas técnicas de otimização do desempenho de sistema de arquivos foram desenvolvidas. Algumas transferem dados para a memória principal, visando reduzir a quantidade de acessos a disco, realizados para atender às requisições de aplicações. Outras procuram utilizar os discos de forma mais eficiente, para diminuir o tempo de acesso. A maioria dos sistemas de arquivos atuais adota algumas dessas técnicas, que serão descritas a seguir.

2.3.1 Técnicas que Utilizam Memória Principal

As técnicas de otimização de sistemas de arquivos, que se fundamentam na utilização da memória principal, procuram satisfazer às requisições de armazenamento, sem ler dados dos discos. A justificativa para o uso da memória principal no atendimento das requisições é a grande diferença entre o tempo de acesso à memória, na ordem de nanossegundos, e o tempo de acesso a disco, na ordem de milissegundos.

2.3.1.1 Caching de Leitura

Explorando a observação de que *"se um dado é referenciado, existe uma alta probabilidade de que seja referenciado novamente, num futuro próximo"* [Sat93], a maioria dos sistemas de arquivos implementa algum tipo de *caching*, com o objetivo de aprimorar o desempenho.

A *cache* é uma área de memória onde o sistema mantém os dados utilizados com maior freqüência. Além dos dados de arquivos, os metadados e os diretórios também podem ser armazenados em *cache*. A *caching* de leitura contribui para a melhoria do desempenho, porque possibilita que muitas requisições de leitura das aplicações sejam atendidas com uma cópia do dado na *cache*, sem a realização de acesso a disco. O sistema de arquivos do UNIX faz uso ostensivo de *caching* de leitura.

Quanto maior a quantidade de memória utilizada para *caching*, menor o número de acessos a disco realizados pelo sistema de arquivos. No entanto, o aumento demasiado da área de memória dedicada à *cache* é prejudicial ao sistema operacional, porque a área de memória da *cache* não pode ser utilizada para satisfazer outras demandas do sistema.

O sistema operacional Sprite [OCD+88] implementa *cache* de tamanho variável. Nesse sistema, o mecanismo de memória virtual negocia continuamente o espaço de memória com o mecanismo de *caching* [NWO88].

2.3.1.2 Leitura Antecipada

A leitura antecipada de dados consiste em transferir para a memória principal os dados que a aplicação pode referenciar no futuro. O sistema precisa prever os dados que serão solicitados e ler antecipadamente esses dados do disco. Se a previsão se confirmar, as aplicações não precisarão esperar por acesso a disco.

Apostando na probabilidade de que, *"quando um arquivo é aberto, ele normalmente é lido por inteiro"* [Sat93], o sistema de arquivos do UNIX monitora os acessos a um arquivo. Ao detectar acessos a blocos adjacentes de um arquivo, o sistema assume a ocorrência de acesso seqüencial e realiza a leitura antecipada. Essa técnica é bastante eficiente, pois grande parte dos acessos realizados é seqüencial [BHK+91].

2.3.1.3 Caching de Escrita

Na *caching* de escrita, a memória principal é utilizada para melhorar o desempenho das operações de escrita. No processamento de uma requisição de escrita, o sistema altera os dados na *cache* e retorna o controle para a aplicação, antes que os dados alterados sejam transferidos para o disco. Assim, a aplicação pode dar continuidade ao seu processamento mais rapidamente, melhorando o seu desempenho individual.

A *caching* de escrita também otimiza o desempenho global de um sistema de arquivos, pois reduz a quantidade de dados transferidos para o disco. Enquanto um bloco permanece na *cache*, várias operações de escrita sobre esse bloco podem ser combinadas e transferidas para o disco, numa única operação de saída. Além disso, arquivos inteiros podem ser apagados enquanto estão na *cache*, diminuindo a quantidade de operações de saída necessárias para atualizar o sistema de arquivos em disco.

Não obstante o benefício do desempenho, a *caching* de escrita reduz a confiabilidade do sistema, porque a memória principal, utilizada na maioria dos computadores, não é um meio confiável para o armazenamento de dados, por ser volátil [Tan95:146]. Por isso, a *caching* de escrita abre uma janela de vulnerabilidade no sistema: entre a modificação de um dado e a sua propagação para o disco. Ocorrendo um *crash* nesse intervalo, uma aplicação pode perder modificações realizadas. Em resposta, os sistemas de arquivos restringem a utilização da *caching* de escrita: o UNIX original limita em trinta segundos o tempo de permanência de um bloco na *cache*; o UNIX FFS não permite a *caching* de escrita de estruturas fundamentais para a recuperação do sistema, como os diretórios e os *nós-i*.

2.3.2 Técnicas de Otimização de Disco

Essas técnicas exploram as características dos discos magnéticos, para ler e gravar dados armazenados em disco, com maior eficiência. O principal objetivo é abreviar o tempo de espera das aplicações que realizam acessos a disco.

Uma requisição de acesso a disco especifica o cilindro, a cabeça de leitura/gravação, o setor inicial e o número de setores que devem ser transferidos. Existem três componentes que influenciam no tempo de acesso a disco: *tempo de posicionamento*, que é o tempo necessário para que o *braço* seja posicionado no cilindro correto; a *latência rotacional*, que é o tempo gasto no movimento de rotação, que posiciona o setor inicial sob a cabeça de leitura/gravação, e o *tempo de transferência*, que é o tempo gasto lendo, ou gravando dados nos setores.

Nos discos atuais, o *tempo de posicionamento* ainda é o maior componente do tempo total do processamento de requisições de acesso. Por isso, as técnicas de otimização visam diminuir a quantidade de posicionamentos, seja através de uma nova organização do disco, utilizando técnicas mais otimizadas de alocação do espaço em disco, seja através do escalonamento das requisições de acesso, ou mesmo implementando mais um nível de *cache* dos dados armazenados no disco.

2.3.2.1 Caching de Trilha

A *caching* de trilha é implementada por alguns *drivers* de disco, que mantêm as trilhas mais recentemente utilizadas em *buffers* [Tan92]. Assim, as requisições envolvendo setores presentes na *cache* não precisam realizar transferência de/para o disco. Durante o processamento de uma requisição, após o *posicionamento* do *braço*, o *driver* fica livre durante a transferência de dados. O *driver* pode aproveitar para ler uma trilha inteira no tempo correspondente a uma rotação. Para isso, a controladora deve ser dotada de um sensor, que permita ao *driver* identificar o setor sob a cabeça de leitura/gravação e enviar uma requisição para o próximo setor.

A *caching* de trilha implementada pelo *driver* de disco apresenta a seguinte desvantagem: o processador é o responsável pela transferência de dados entre a *cache* e a área de memória do programa de usuário. Algumas controladoras implementam a *caching* de trilha em sua própria memória interna, de modo que a transferência de dados pode ser realizada pelo hardware de acesso direto à memória, sem ocupar o processador.

2.3.2.2 Técnicas de Alocação

Os algoritmos de alocação visam arrumar os dados no disco, de forma a minimizar os posicionamentos necessários para atender às requisições de acesso. Na técnica de *alocação contígua*, o sistema armazena os dados de um arquivo em setores consecutivos, para que os acessos seqüenciais sejam realizados sem a ocorrência de posicionamentos entre os blocos.

A alocação contígua favorece o desempenho de aplicações que realizam acessos seqüenciais a grandes arquivos. No entanto, não é vantajosa para aplicações que manipulam pequenos arquivos, ou que realizam muitos acessos randômicos aos grandes arquivos. Essa técnica apresenta, ainda, a desvantagem de provocar o fenômeno da *fragmentação externa*: em ambientes com a criação e remoção freqüente de arquivos, de tamanhos diferentes, a alocação contígua gera o aparecimento de espaços desperdiçados no disco, difíceis de ser reutilizados.

O método de gerenciamento do espaço livre, do sistema de arquivos original do UNIX, não favorece a implementação da alocação contígua. Os blocos são alocados a partir de uma lista de blocos livres. Logo após a criação de um sistema de arquivos, a lista de blocos livres está ordenada pelo número de bloco. Os arquivos são armazenados em posições contíguas, porque os blocos são alocados na ordem em que aparecem na lista. À medida que os blocos do disco vão sendo alocados e liberados, o espaço em disco torna-se fragmentado e os blocos passam a aparecer na lista numa ordem aleatória, tornando a alocação contígua impossível. Por isso, os acessos seqüenciais apresentam um desempenho tipicamente ruim, nesse sistema.

No UNIX FFS, a utilização de um *mapa de bits* facilita a alocação de regiões contíguas de blocos livres. O algoritmo de alocação tenta armazenar os dados de um arquivo em blocos contíguos, tanto quanto possível, melhorando o desempenho dos acessos seqüenciais.

O *agrupamento* é uma outra técnica de alocação, que também é utilizada para aumentar o desempenho das operações de acesso a disco. O sistema de arquivos armazena os arquivos que são normalmente requisitados juntos, em posições próximas do disco. Assim, as requisições de acesso não provocam muitos posicionamentos. Por exemplo, o Unix FFS utiliza a alocação por agrupamento quando procura armazenar os arquivos de um mesmo diretório num único grupo de cilindros. Essa técnica é favorável apenas em ambientes onde os acessos a arquivos são previsíveis. No entanto, num sistema de arquivos de propósito geral, existem aplicações de naturezas diversas, com comportamentos bastante variados, existindo, inclusive, aplicações que mudam suas tendências de acesso ao longo do tempo.

2.3.2.3 Técnicas de Escalonamento

Quando uma requisição de acesso a disco está sendo atendida, as outras requisições que chegam são enfileiradas. O *driver* de disco faz o escalonamento da fila de pedidos pendentes, ou seja, determina a ordem em que as requisições serão processadas. Esse escalonamento pode ser realizado de forma a diminuir o tempo de posicionamento e aumentar o desempenho do sistema. Por exemplo, se um conjunto de requisições alterna entre uma trilha inicial e uma trilha do final do disco, o *driver* pode reordenar tais requisições, processando primeiro todas que fazem acesso à trilha inicial e, a seguir, aquelas relacionadas com a trilha do final do disco.

Vários algoritmos de escalonamento foram propostos, conseguindo aumentar a eficiência dos discos em vinte e cinco por cento, no máximo [SCO90]. O algoritmo do *menor posicionamento primeiro* atende sempre à requisição que precisar do menor deslocamento do braço, reduzindo o tempo de posicionamento. No entanto, quando o volume de requisições é muito grande, esse algoritmo tende a manter o *braço* no meio do disco, prejudicando as requisições em qualquer dos extremos. O *algoritmo do elevador* não prioriza um conjunto de requisições, em detrimento de outras. O *driver* move o *braço* do disco sempre em uma determinada direção, até que não existam requisições pendentes nessa direção. Então, o *braço* muda de direção. Esse algoritmo garante que, dado um conjunto qualquer de pedidos, o limite máximo de movimentação do braço do disco é fixo e igual a duas vezes o número de cilindros a serem percorridos [Tan92].

As técnicas de escalonamento contribuem para melhorar o desempenho de um sistema de arquivos apenas quando o fluxo de requisições de entrada/saída é grande o suficiente para que existam filas de pedidos pendentes. Caso contrário, o *driver* de disco atende de imediato toda requisição que chega.

2.4 Sistema de Arquivos em Rede

O NFS (*Network File System*) [SGK+85] é um Sistema de Arquivos em Rede, que tem a flexibilidade de suportar sistemas heterogêneos: permite a existência de clientes baseados em arquiteturas variadas, como processadores Intel, RISC, Motorola, além de permitir a interação de clientes utilizando sistemas operacionais diferentes, como clientes Macintosh, Windows, OS/2, Windows NT e vários tipos de UNIX.

O NFS possibilita que vários sistemas de arquivos centralizados possam compartilhar dados, através de um *mecanismo de montagem*: cada servidor exporta um ou mais de seus diretórios; um cliente monta um diretório exportado por um servidor no seu sistema de arquivos local, passando a ter acesso a toda a sub-árvore enraizada por este diretório.

O NFS é vastamente utilizado em ambientes distribuídos, para possibilitar o compartilhamento de arquivos entre sistemas UNIX.

Utilizando as operações de montagem, cada usuário pode criar uma hierarquia de nomes customizada. Assim, cada cliente NFS pode ter uma visão diferente da árvore de diretórios. A Figura 2.5 mostra árvores de clientes NFS: cada cliente pode importar e montar, em qualquer ponto de sua árvore local, diretórios que foram exportados pelos servidores. Dois clientes podem formar árvores bem diferentes.

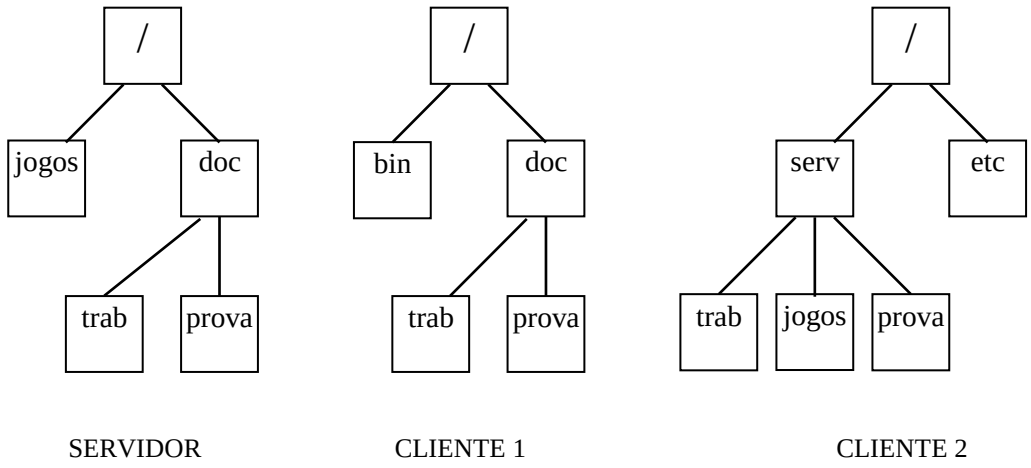

Figura 2.5: Visões do sistema de arquivos nos clientes NFS.

O NFS é um sistema sem informações de estado. Para esse tipo de sistema é mais fácil permanecer em operação após uma falta. Cada requisição do cliente contém todas as informações necessárias ao seu atendimento pelo servidor. Por outro lado, grandes mensagens trafegam pela rede, podendo comprometer o desempenho.

2.4.1 Arquitetura do NFS

O NFS foi projetado como um serviço de rede, que permite o compartilhamento de arquivos entre máquinas com sistemas de arquivos independentes, de forma transparente. O compartilhamento é baseado no relacionamento cliente-servidor, utilizando c*hamada remota a procedimento* (RPC) para a troca de solicitações e respostas. Qualquer máquina pode ser um cliente ou um servidor. Para tornar um diretório remoto acessível, um cliente deve montá-lo em sua hierarquia local de arquivos. A partir desse momento, o cliente usa as mesmas chamadas para referenciar arquivos locais e remotos.

O sistema possui dois protocolos: o *Protocolo de Montagem*, com a função de permitir que os servidores exportem diretórios e que os clientes os montem em seu sistema de arquivos local, e o *Protocolo NFS*, para realização de operações sobre arquivos remotos.

O NFS está organizado em três níveis, descritos a seguir, e exibidos na Figura 2.6.

- **Nível de Chamadas de Sistema.** Verifica a sintaxe da chamada e valida parâmetros, como o descritor de arquivo.
- **Nível VFS (Virtual File System).** Distingue um arquivo local de um arquivo remoto. Para isso, verifica uma tabela com entrada uma para cada arquivo aberto, mantida pelo núcleo, contendo um identificador único do arquivo em toda a rede, chamado de *nó-v* (nó virtual). Quando o arquivo é local, o *nó-v* aponta para um *nó-i* do sistema de arquivos local. O VFS, então, ativa procedimentos específicos desse sistema de arquivos para tratar a requisição. Quando o arquivo é remoto, o *nó-v* correspondente aponta para uma entrada na tabela de *nós-r* (nós remotos), no nível de Protocolo NFS. Neste caso, o VFS ativa o citado nível para realizar as requisições remotas.
- **Nível de Protocolo NFS.** É responsável por realizar as chamadas remotas a procedimento para o nível de Protocolo NFS do servidor apropriado a atender a uma requisição de arquivo remoto. Ele possui uma tabela com um *nó-r* para cada arquivo remoto aberto. Em um *nó-r* estão as informações necessárias para o acesso ao arquivo, como o endereço do servidor e a autorização de manipulação do arquivo.

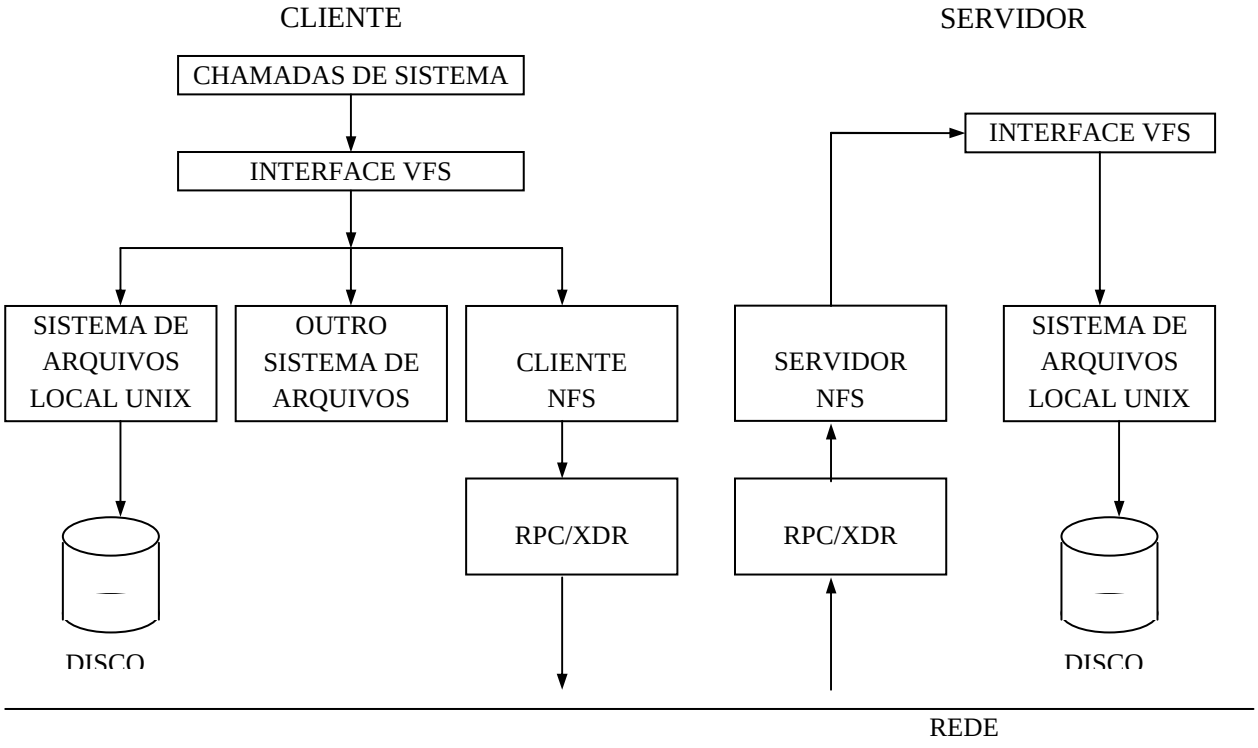

Figura 2.6: Arquitetura do NFS.

2.4.2 Localização de Arquivos no NFS

Quando um cliente deseja ter acesso a um arquivo (ou diretório) remoto, ele faz uma chamada de sistema comum para o núcleo, que analisa o nome de caminho e verifica que o arquivo é remoto, com base nas informações da tabela de montagem. A seguir, o núcleo procura uma entrada para o arquivo na tabela de *nós-v*, do VFS. Caso ainda não exista, passa o controle para o cliente NFS, que requisita acesso ao arquivo, através de uma RPC enviada ao servidor. No servidor, o nível de Protocolo NFS repassa a requisição ao nível VFS, que encontra o arquivo no disco do servidor. Uma resposta é enviada ao cliente. O Protocolo NFS do cliente cria um *nó-r* para o arquivo em suas tabelas e passa as informações recebidas para o VFS. Este acrescenta um *nó-v* em suas tabelas, apontando para o *nó-r* do cliente NFS.

A tradução de um nome de caminho é realizado quebrando o mesmo em seus diretórios componentes. A seqüência de passos descrita no parágrafo anterior é repetida para cada diretório remoto do nome de caminho, até chegar ao arquivo desejado, quando o processo cliente recebe um descritor de arquivo, que serve de índice para a tabela de *nós-v*.

O núcleo do cliente sabe quando um ponto de montagem é atravessado, através das informações da tabela de montagem. O NFS permite a montagem em cascata, ou seja, montar um sistema de arquivos remoto no topo de outro sistema de arquivos remoto já montado. Um servidor, entretanto, não pode agir como um intermediário entre um cliente e outro servidor. Durante a localização de um arquivo, o cliente deve estabelecer uma conexão cliente-servidor em separado, com cada servidor que armazena diretórios componentes do nome de caminho.

Para agilizar a localização de arquivos, os clientes podem manter *caches* de diretórios, com os *nós-v* dos diretórios remotos. Assim, arquivos com o mesmo nome de caminho inicial podem ser localizados mais rapidamente.

2.4.3 Caching no NFS

O NFS realiza *caching* tanto no cliente, quanto no servidor. Em geral, o sistema transfere grandes blocos de dados para a *cache* (tipicamente de 8KB), explorando a observação de que “quando um arquivo é aberto, ele normalmente é lido por inteiro” [Sat93]. Por isso, é comum o uso de *leitura antecipada* de blocos do arquivo para a *cache* de cliente, com o objetivo de otimizar as operações de leitura seqüencial de arquivos.

O NFS utiliza a política de *propagação retardada*, associando um temporizador a cada bloco de *cache*. Quando ele expira, o bloco correspondente é enviado de volta ao servidor. Esse tempo é, em geral, de três segundos para diretórios e de trinta segundos para arquivos.

Essa política de atualização de *cache* de cliente torna a semântica de compartilhamento do NFS confusa e dependente de temporizações: um novo arquivo pode demorar trinta segundos para tornar-se visível aos demais clientes do sistema.

No NFS, um cliente faz a validação dos dados armazenados em sua *cache* no momento em que o arquivo é aberto. Essa estratégia apresenta um problema em potencial: se um determinado cliente A abre um arquivo para leitura, copiando o mesmo para a sua *cache* e, a seguir, o cliente B abre o mesmo arquivo para escrita, as alterações do arquivo realizadas por B não invalidam a cópia de A, pois A não tentará abrir o arquivo novamente. Para solucionar esse problema, os fornecedores do NFS oferecem um mecanismo opcional de bloqueio de arquivos.

O sistema NFS implementa apenas *caching* de leitura nos servidores, gravando diretamente em disco os dados das operações de escrita [Ste91] (note que o NFS implementa *caching* no cliente, tanto para operações de leitura, quanto para operações de escrita).

Uma aplicação que requer armazenamento estável dos dados, ela não pode se beneficiar com a *cache* de cliente, nem com a *cache* de servidor. Ao término de uma operação de escrita, o sistema precisa garantir de que os dados foram efetivamente gravados no disco do servidor. Os principais Sistemas de Arquivos Distribuídos do mercado não oferecem esse tipo de serviço. Tanto no UNIX quanto no NFS, o cliente precisa invocar explicitamente uma operação *flush*, para obrigar que os dados em *buffer* sejam gravados em disco. A aplicação espera pelo final da transferência dos dados para disco. Por isso, o ambiente NFS também requer um Serviço de Armazenamento Estável mais eficiente.

2.5 Conclusões

O conteúdo descrito neste capítulo evidencia cinco pontos importantes, que devem ser considerados na elaboração do projeto de um serviço de armazenamento estável. Primeiro, tanto um sistema de arquivos tradicional, como o UNIX, quanto um sistema de arquivos em rede, como o NFS, oferecem a gravação síncrona como método de prover estabilidade no armazenamento de dados. Segundo, a gravação síncrona apresenta um alto custo de desempenho. Terceiro, as técnicas de otimização baseadas em memória conseguem aprimorar o desempenho, mas diminuem a confiabilidade do sistema. Quarto, as técnicas de otimização de disco não conseguem melhorar significativamente o desempenho dos sistemas de arquivos de propósito geral. Quinto, com o advento dos sistemas distribuídos, surgiram novas facilidades de comunicação, possibilitando a interação de processos executados em máquinas diferentes.

Capítulo 3

Evolução Tecnológica e o Novo Desafio

Este capítulo apresenta a motivação para novas pesquisas em técnicas de armazenamento estável de dados. O argumento básico é que o método de armazenamento estável adotado nos sistemas de arquivos tradicionais é inadequado para a tecnologia atual. Mais especificamente, o armazenamento estável de dados, baseado na gravação síncrona em disco, é um entrave, cada vez maior, ao desempenho de muitas aplicações, devido às tendências da tecnologia corrente. Enquanto estiverem aguardando o processamento das requisições de armazenamento, as aplicações não serão capazes de explorar plenamente as melhorias tecnológicas, tais como: processadores, memórias e redes mais velozes.

As necessidades de armazenamento das novas aplicações exigem que os sistemas de arquivos sejam mais eficientes no atendimento aos requisitos de desempenho e confiabilidade. No entanto, as tendências da tecnologia atual colocam os sistemas de arquivos diante de um impasse entre esses dois requisitos. Basicamente, as técnicas de otimização de desempenho adotadas fundamentam-se no uso ostensivo da memória principal. Porém, a volatilidade da memória diminui a confiabilidade do sistema, pois leva ao risco de perda de dados. Assim, as aplicações que necessitam gravar dados de forma estável continuam pagando um alto custo de desempenho, para realizar a gravação síncrona dos dados em disco.

Paralelamente, ocorrem avanços significativos na tecnologia de redes: as altas taxas de transferência, disponíveis atualmente, representam um recurso em potencial, a ser utilizado nas técnicas de otimização de sistemas de arquivos. A técnica de armazenamento estável descrita nesta dissertação considera essa e as demais tendências da tecnologia, assim como as necessidades de armazenamento das aplicações. O serviço proposto não requer tecnologia de hardware especial, nem alterações no comportamento das aplicações: possibilita que o sistema de arquivos utilize as mudanças tecnológicas de modo favorável.

3.1 Tecnologia para Sistemas de Arquivos

Os principais componentes de hardware integrantes de um sistema de arquivos são: processador, disco e memória principal. Adicionalmente, um sistema de arquivos distribuído necessita de uma estrutura de rede, para viabilizar a comunicação entre clientes e servidores. Esta seção examina como a tecnologia desses componentes está mudando.

Todos os componentes de hardware citados estão em constante evolução, beneficiando os sistemas de arquivos, especialmente no que se refere à capacidade de armazenamento e ao desempenho. Porém, para que os sistemas de arquivos possam usufruir plenamente dos avanços da tecnologia, é preciso que o projeto desses sistemas seja capaz de lidar com um aspecto importante do progresso tecnológico: ele não acontece no mesmo ritmo para todos os componentes.

3.1.1 Tecnologia de Processador

Os avanços na tecnologia de semicondutores permitiram uma grande evolução na tecnologia de processador. Na última década, a velocidade dos processadores cresceu cerca de cinqüenta por cento ao ano [Mud+96] [Int97]. Considerando sistemas SPP (*Scalable Parallel Processors)*, que podem ser configurados com centenas, ou mesmo milhares de processadores em paralelo, o poder de processamento tornou-se quinhentas vezes maior nesse período [MCP+98]. O grande aumento na capacidade de processamento pressiona os outros elementos de hardware dos sistemas computacionais a tornarem-se mais velozes, para que o sistema permaneça o mais balanceado possível.

A tecnologia de processador é importante para um sistema de arquivos, porque, quanto maior o poder de processamento, melhor será o desempenho do sistema. Infelizmente, porém, o desempenho de um sistema não cresce na mesma proporção que a velocidade dos processadores, pois depende, também, de outros componentes de hardware. Particularmente, o desempenho de um sistema de arquivos está intrinsecamente subordinado à tecnologia de disco.

Por exemplo, assumindo, hipoteticamente, que a tecnologia de disco permanece estática e considerando uma aplicação que realiza um acesso a disco a cada cem milissegundos de tempo de processador e leva quinhentos segundos para executar no seguinte sistema: constituído por uma CPU (Unidade Central de Processamento) de um MIPS (Milhões de Instruções Por Segundo) e um disco, cujo tempo médio de acesso é de dez milissegundos. Substituindo a CPU por outra de cem MIPS, a aplicação irá finalizar em 94,2 segundos. Assim, a aplicação melhora seu desempenho em apenas 5,3 vezes, mesmo com uma CPU cem vezes mais rápida.

O exemplo ilustra o conceito conhecido como *Lei de Amdahl* [Amd67]: se parte de um sistema torna-se mais veloz e a outra parte não, então o desempenho total será prejudicado pela parte não melhorada. Na tecnologia atual, a parte que mais evolui são os processadores, enquanto os discos constituem a parte que não progride no mesmo ritmo. Na época, Amdahl previu que grandes aumentos na velocidade de processadores resultariam apenas em pequenas melhorias no desempenho global do sistema, a não ser que fossem acompanhadas por avanços correspondentes na tecnologia de armazenamento secundário.

3.1.2 Tecnologia de Disco

Os discos magnéticos ainda constituem o principal meio de armazenamento não-volátil de dados, nos sistemas de arquivos atuais. A evolução da tecnologia de disco também contribui para a melhoria dos sistemas de arquivos. Contudo, os avanços dessa tecnologia concentram-se mais nas áreas de custo e capacidade, do que no desempenho. Na última década, a capacidade de armazenamento dos discos magnéticos cresceu cerca de vinte e sete por cento ao ano [CLG+94], enquanto o desempenho não evoluiu no mesmo ritmo.

Existem dois componentes básicos, que devem ser considerados na avaliação do desempenho de um disco: o tempo de transferência e o tempo de acesso. O tempo de transferência é a medida da rapidez na qual os dados são transferidos de/para o disco. Essa medida é importante para as requisições que armazenam grandes quantidades de dados, nas quais a maior parte do tempo de processamento é gasta com a transferência de dados. Os valores apresentados em [CLG+94] demonstram que a taxa de transferência dos discos magnéticos cresce, aproximadamente, vinte e dois por cento ao ano.

O tempo de acesso é o tempo que uma requisição individual de armazenamento leva para finalizar: o somatório do tempo de posicionamento, do tempo de latência rotacional e do tempo de transferência de dados. O tempo de acesso a disco vem evoluindo em proporções bem menores que a velocidade dos processadores: enquanto o poder de processamento aumenta cerca de cinqüenta por cento ao ano, a tecnologia de disco precisou de uma década para reduzir o tempo de acesso, nessa mesma proporção [CLG+94].

O tempo de acesso a disco influencia no desempenho de toda aplicação que grava dados de forma síncrona, ou seja, que precisa esperar pelo término de cada requisição de saída antes de prosseguir. Assim, o tempo de acesso a disco é crucial para o armazenamento estável de dados nos sistemas de arquivos tradicionais.

O tempo de posicionamento e o tempo de latência dos discos são difíceis de aprimorar, porque dependem de movimentos mecânicos. Por isso, os projetos que objetivam o aumento do desempenho dos discos se concentram na melhoria do tempo de transferência, como a tecnologia RAID, descrita a seguir.

3.1.2.1 Tecnologia RAID

O tempo de transferência das operações de acesso a disco pode ser melhorado com a utilização da tecnologia RAID - *Redundant Array of Inexpensive Disks* [CLG+94]. RAID é um método de armazenamento proposto para simular um disco de alta capacidade, explorando o paralelismo de um conjunto de discos independentes. Os dados de uma requisição de escrita são segmentados e espalhados em múltiplos discos.

A intercalação de partes de dados provê um alto desempenho, pois permite que várias operações de entrada/saída sejam realizadas paralelamente. Existem dois aspectos nesse paralelismo. Primeiro, várias requisições podem ser atendidas ao mesmo tempo, por discos diferentes. Por exemplo, se um arquivo é espalhado em cem discos, então o sistema pode processar, simultaneamente, até cem requisições de entrada/saída, envolvendo partes diferentes do arquivo e atingindo, desse modo, um desempenho bem superior a um único disco. Segundo, uma mesma requisição, envolvendo múltiplos blocos de um arquivo, pode ser atendida por vários discos, operando de forma coordenada.

Para explorar o benefício de alto desempenho provido pela tecnologia RAID, os sistemas de arquivos devem arrumar os dados nos discos, de modo que as requisições possam utilizar vários discos, de forma concorrente. Caso contrário, o RAID não será mais veloz do que um único disco.

Quanto maior a quantidade de discos utilizados no RAID, maior o potencial para proporcionar melhorias no desempenho. Por outro lado, um grande número de discos diminui a confiabilidade do sistema. Para solucionar esse problema, a tecnologia RAID emprega redundância, na forma de códigos de correção de erro. Infelizmente, a redundância influencia negativamente no desempenho, pois toda operação de escrita precisa atualizar a informação redundante. Para amenizar o prejuízo no desempenho, os sistemas utilizam memória adicional para realizar *caching* dos dados necessários para o cálculo da informação redundante.

Existem várias organizações de RAID que oferecem diferentes níveis de desempenho, de confiabilidade e de relação custo/benefício. Elas diferem em dois aspectos principais: o tamanho dos dados intercalados e a técnica utilizada para implementar a redundância. Alguns esquemas de RAID intercalam pequenas unidades de dados, como o RAID Nível 3, que espalha os bits de cada palavra em diferentes discos. Nesse tipo de esquema, toda requisição de entrada/saída, independentemente do seu tamanho, faz acesso a todos os discos do RAID, resultando em altas taxas de transferência para todas as requisições. Porém, somente uma requisição pode ser atendida de cada vez. Outros esquemas de RAID, como RAID Níveis 4, 5 e 6, intercalam blocos inteiros de dados. Assim, pequenas requisições fazem acesso somente a um número reduzido de discos, possibilitando que várias requisições sejam atendidas ao mesmo tempo.

Existem diversas técnicas para a implementação de redundância, que variam quanto ao método utilizado para calcular e distribuir a informação redundante através do RAID. Embora alguns esquemas de RAID façam uso da codificação de Reed-Solomon ou Hamming, a maioria utiliza paridade. Alguns esquemas de RAID concentram a informação redundante num pequeno número de discos, enquanto outros distribuem a informação redundante uniformemente, através de todos os discos.

A Figura 3.1 exibe um exemplo típico de RAID Nível 4. Os dados são segmentados em blocos e espalhados em quatro discos. Um quinto disco armazena os blocos de paridade (P1, P2, ...) contendo os valores calculados através da operação XOR (OU-exclusivo) dos blocos correspondentes nos discos de dados. Por exemplo, caba byte em P1 é o XOR dos bytes correspondentes nos blocos B1, B2, B3 e B4. Se um disco falhar, sua informação pode ser restaurada, mediante uma simples operação XOR dos blocos de paridade com os blocos de dados correlatos dos outros discos. No exemplo apresentado, a redundância é de apenas vinte e cinco por cento.

O *espelhamento de disco* é o RAID Nível 1, uma forma mais simples de RAID, com redundância de cem por cento, ou seja, o número de discos é dobrado [McK96]. O sistema de arquivos do NetWare [MMP94], do Windows NT [Cus94] e de outros realiza espelhamento de discos.

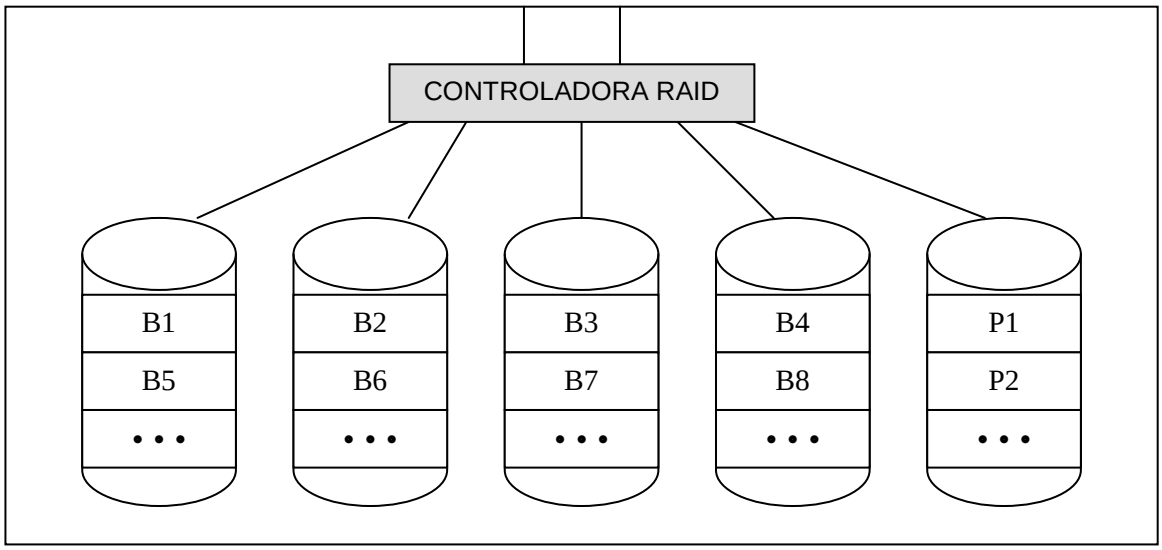

Figura 3.1: Exemplo de RAID Nível 4.

Apesar das melhorias do tempo de transferência propiciadas pelo uso de RAID, o desempenho de um disco magnético é afetado por outros componentes do tempo de acesso, que não possuem muitas opções de otimização.

3.1.2.2 Tecnologia de Armazenamento Ótico

Além da tecnologia RAID, que conseguem prover um melhor desempenho do que a tecnologia de disco magnético convencional, existe uma tecnologia de armazenamento ótico, capaz de prover alta densidade de armazenamento e maior integridade de dados.

Mais propriamente, existem duas tecnologias de armazenamento ótico, cada uma com diferentes características e algumas aplicações distintas: a tecnologia de CD-ROM e a tecnologia magneto-ótica. Nenhuma delas representa uma alternativa viável para a substituição do armazenamento magnético convencional, no o uso generalizado e cotidiano, mas cada uma possui a sua aplicabilidade, para o armazenamento secundário de informações [Nor95].

A operação de todos os dispositivos de armazenamento ótico depende da tecnologia laser. O raio laser é um feixe de luz altamente controlado e concentrado, utilizado, neste caso, para gravar dados em meios óticos especiais, para permitir a gravação magnética tradicional em material magneto-ótico especial, ou para ler dados previamente gravados.

Na tecnologia de CD-ROM, os discos são constituídos por uma lâmina de plástico rígido e recobertos por uma película protetora. A gravação é realizada através de laser de baixa intensidade, que “queima” pontinhos correspondentes a “1” na superfície. A vantagem desse dispositivo é a capacidade de armazenar uma grande quantidade de informações de forma confiável.

A confiabilidade do CD-ROM deve-se aos seguintes fatores: é constituído de material altamente resistente e durável; a superfície do disco é selada em seu revestimento plástico, sendo tocada apenas pela luz; o cabeçote de leitura/gravação opera duas mil vezes mais afastado da superfície do disco do que os cabeçotes dos discos magnéticos; grande espaço do disco é utilizado para gravar informações de detecção e correção de erros; e finalmente, o laser não lê a face exterior da superfície do disco, sendo capaz de ignorar pequenos danos físicos, como arranhões e sujeiras.

Inicialmente, a tecnologia de CD-ROM estava limitada a um único ciclo de gravação; atualmente, existem discos de CD-ROM regraváveis. No entanto, esta nova capacidade é restrita a alguns ciclos de gravação, não podendo ser utilizada indefinidamente, porque, de fato, as informações não podem ser apagadas da superfície do disco (as marcas feitas pelo laser são permanentes), sendo as novas gravações realizadas mediante o aproveitamento do espaço ainda disponível no disco.

A tecnologia de CD-ROM é bastante utilizada para a realização de cópias de segurança e para a distribuição de dados: atualmente, existe uma grande quantidade de softwares comercializados em CD-ROM.

As unidades magneto-óticas utilizam meios magnéticos para armazenar informação por controle a laser. Essas unidades empregam cartuchos preenchidos com meios magnéticos de leitura/gravação. Para gravar informações, os discos utilizam as mudanças de polaridade no material magnético quando ele é aquecido. Quando o laser aquece o material magnético do disco, torna-se fácil mudar a polaridade do material. Depois que o material esfria, a polaridade é fixada dentro da nova orientação.

A tecnologia MO oferece armazenamento de dados mais durável do que a de discos magnéticos e disquetes convencionais: os discos são feitos de plástico policarbonato de alta resistência, semelhante ao material de vidros à prova de balas; a camada de dados é mantida em segurança entre um sanduíche de policarbonato. Adicionalmente, os dados são imunes a campos magnéticos, pois sua gravação também depende da tecnologia laser.

As unidades magneto-óticas são utilizadas em aplicações de arquivos *off-line* e, principalmente, para a realização de cópias de segurança.

Embora a tecnologia de armazenamento ótico possa oferecer grandes densidades de armazenamento, com alta confiabilidade, seu desempenho ainda é bastante inferior ao dos discos magnéticos: o tempo médio de acesso a um CD-ROM é cerca de vinte vezes maior que maior que o tempo médio de acesso a um disco magnético, enquanto uma unidade magneto-ótica é mais lentas tanto em transferência, quanto em acesso de dados, do que uma unidade de disco magnético. Por isso, a tecnologia de disco magnético continua sendo o meio padrão de armazenamento secundário nos sistemas computacionais.

3.1.3 Tecnologia de Memória

Assim como a tecnologia de processador, a tecnologia de memória valeu-se das melhorias nos semicondutores, aprimorando tanto a capacidade de armazenamento, quanto a velocidade de acesso. Esses avanços englobam toda a hierarquia de memória principal dos computadores, incluindo os registros de processador e os vários níveis de *cache*.

A memória DRAM (*Dynamic Random Access Memory*) é o padrão de memória principal mais utilizado nos computadores atuais, devido ao seu custo cada vez menor, associado a velocidades de acesso e capacidades cada vez maiores. Na última década, a velocidade de acesso da memória DRAM cresceu aproximadamente dez por cento ao ano [Mud+96]. Embora essa taxa de crescimento seja inferior ao aumento da velocidade dos processadores, ela é bem maior que o aumento da velocidade de acesso a disco. Por isso, praticamente, todos os sistemas de arquivos atuais utilizam a memória principal para implementar técnicas de otimização de desempenho, que minimizam o impacto negativo causado pela tecnologia de disco, tais como: leitura antecipada, *caching* de leitura e *caching* de escrita [Tan92].

Com a disponibilidade de grandes memórias principais, é possível implementar técnicas agressivas de leitura antecipada, que reduzem o tempo médio de espera das aplicações. De forma similar, à medida que as *cache*s de arquivo aumentam, cresce também a fração de requisições de leitura que podem ser atendidas sem a necessidade de fazer acesso a disco, contribuindo positivamente para o desempenho das aplicações.

No caso de *caching* de escrita, a maior disponibilidade de memória principal não leva a ganhos substanciais no desempenho. O maior benefício consiste no aumento da eficiência dos acessos a disco, porque permite que o sistema faça a reordenação de um número maior de requisições de escrita, quando existe uma fila de pedidos pendentes. No entanto, essa reordenação leva apenas a um aumento modesto do desempenho. O benefício que a *caching* de escrita consegue prover ao desempenho de um sistema é mais afetado pelo tempo que os dados permanecem na *cache*, do que pelo tamanho da *cache*. Quanto mais tempo os dados de escrita permanecem na *cache*, maior a chance de que sejam absorvidos (reescritos ou apagados). Entretanto, o aumento desse tempo também leva a um maior risco de perder dados. Portanto, existe uma limitação das *cache*s de arquivo para reduzir o tráfego de escrita em disco, que independe do tamanho da *cache* [BAD+92].

As aplicações que precisam gravar dados de forma estável não admitem qualquer risco de perder dados. Por isso, realizam a *gravação síncrona* dos dados, sem tirar proveito da *caching* de escrita. Assim, um sistema de arquivos tradicional não consegue utilizar a memória principal para melhorar o desempenho do armazenamento estável de dados.

3.1.4 Tecnologia de Rede

A evolução da tecnologia de rede beneficia tanto as redes locais, quanto as redes de longa distância, com o aprimoramento da confiabilidade e das taxas de transmissão de dados. No início desta década, a idéia de taxas de transmissão de cem Mbps (megabits por segundo) em redes locais, até o usuário final e a preços acessíveis, ainda era vislumbrada como um futuro distante. Hoje, essa taxa de transmissão é uma realidade comum.

As normas para cabeamento estruturado contribuíram para a melhoria da qualidade das redes implementadas. Por exemplo, a norma TIA/EIA-568-A [TE95], que abrange, inclusive, a integração de voz e dados, especifica: os requisitos mínimos para cabeamento de telecomunicações dentro de edifícios comerciais; topologia e distâncias recomendadas; meios de transmissão; designações de conectores e pinos; critérios para teste de desempenho e a vida útil dos sistemas de cabeamento (como sendo superior a dez anos).

Atualmente, já estão padronizadas redes de alta velocidade, como Gigabit Ethernet [IEE98], ATM ou Myrinet [Boden. Apud ADN+95:41]. Essas redes oferecem taxas de transferência cada vez mais altas: Gigabit Ethernet, a 1 Gbps e ATM OC24, a 1.2 Gbps.

A infra-estrutura de redes de alta velocidade permite a construção de sistemas distribuídos capazes de realizar comunicação de alto desempenho. Essa facilidade pode ser utilizada nas técnicas de otimização de sistemas de arquivos. Por exemplo, a replicação de arquivos em várias máquinas de um sistema distribuído é uma técnica utilizada para aumentar a disponibilidade e a confiabilidade do sistema [Tan95] [Sat93] [LS90]. Quando um sistema mantém cópias de arquivos em vários servidores, a falta de alguns servidores não impede o acesso a um arquivo, a partir de uma cópia existente em um servidor em funcionamento. O sistema de arquivos precisa garantir a consistência das réplicas, refletindo as modificações de um arquivo em todas as cópias existentes. A atividade extra de propagar as atualizações de arquivos, para todas as suas respectivas réplicas, provoca uma perda no desempenho, que é minimizada quando os dados são transmitidos através de redes de alta velocidade.

As redes de alta velocidade também representam um recurso estratégico para novas técnicas de armazenamento estável de dados. Estudos demonstraram que a latência decorrente de um acesso através de uma rede de alta velocidade é menor do que a latência introduzida pela leitura de um bloco num disco local [Dah+94]. Baseado nesse fato, o SALIUS utiliza as redes de alta velocidade para replicar dados da *cache* de arquivos em várias máquinas de um sistema distribuído, como forma de prover confiabilidade ao sistema de arquivos, eliminando a necessidade de realizar gravação síncrona de dados.

3.2 Demanda de Armazenamento das Aplicações

Outro fator que influencia no projeto de um sistema de arquivos é a demanda de armazenamento das aplicações. Um sistema de arquivos deve ser preparado para atender às necessidades das aplicações, considerando os tipos de requisições de armazenamento, a freqüência na qual elas acontecem e os requisitos que devem ser atendidos.

As requisições podem ser classificadas em pequenas ou grandes, conforme a quantidade de dados armazenados e a tecnologia de disco utilizada. Uma requisição é considerada pequena quando o tempo de acesso é mais dominado pelo tempo de posicionamento e tempo de latência, do que pelo tempo de transferência de dados. As grandes requisições são aquelas onde o tempo de transferência é o maior componente do tempo total necessário para o processamento da requisição.

Como os avanços da tecnologia de discos magnéticos, no campo do desempenho, concentram-se na melhoria do tempo de transferência, as aplicações com predominância de grandes requisições de armazenamento são mais favorecidas do que aquelas que fazem pequenas requisições [RO92].

As requisições de armazenamento também podem ser classificadas em seqüenciais e randômicas. Uma requisição seqüencial provoca o acesso a um objeto numa ordem logicamente contígua, como, por exemplo, a leitura de um arquivo do início ao fim. Qualquer requisição é dita randômica, quando não é seqüencial.

A tecnologia RAID beneficia mais as requisições seqüenciais do que as randômicas, pois pequenos acessos seqüenciais podem ser combinados num grande acesso seqüencial, cujos dados podem ser gravados em múltiplos discos, paralelamente. Esse é o caso, por exemplo, das aplicações de supercomputadores, caracterizadas principalmente por acessos seqüenciais a grandes arquivos. As requisições randômicas, pelo contrário, normalmente sofrem maior influência do tempo de latência e do tempo de posicionamento dos discos.

As aplicações que realizam operações de escrita com muita freqüência sofrem maior influência da tecnologia de disco. Em ambientes de escritório e engenharia, predominam aplicações que atualizam dados com freqüência: o desenvolvimento de programas, a preparação de documentos, as simulações e os projetos interativos. Na área de suporte a decisão, de modo inverso, as aplicações raramente modificam dados e prevalecem as operações de leitura. Nesse caso, o desempenho pode ser otimizado com técnicas que utilizam memória principal, como *caching* e leitura antecipada.

As aplicações diferem, ainda, quanto ao nível de confiabilidade e de desempenho exigidos. Para algumas aplicações, o desempenho é um fator crítico, ao tempo em que admitem a perda eventual de uma pequena quantidade de dados. Por exemplo, no processamento de imagens, caracterizado pela exibição de seqüências de quadros, a perda eventual de alguns quadros não compromete o funcionamento da aplicação. Contudo, um atraso na exibição dos quadros pode chegar a invalidar a aplicação.

Outras aplicações, ao contrário, exigem alta confiabilidade, não permitindo qualquer perda de dados. Um exemplo típico é um sistema de automação bancária, onde grande parte das operações envolve valores monetários, que precisam ser preservados. O desempenho também é um requisito importante nessa aplicação, embora pequenos atrasos no processamento sejam toleráveis. Em geral, esse sistema caracteriza-se pelo processamento de transações com muitas requisições de armazenamento, pequenas e randômicas, ou seja, aquelas cujo processamento individual não é otimizado pela tecnologia RAID. No entanto, o paralelismo provido pelo uso de RAID possibilita o processamento concorrente de várias transações, melhorando o desempenho global do sistema. Como a confiabilidade é um requisito fundamental, o sistema deve garantir a estabilidade dos dados armazenados. Portanto, a aplicação precisa realizar a gravação síncrona dos dados. Assim, mesmo com o auxílio de RAID, o desempenho é prejudicado pelo tempo de acesso a disco.

3.3 Tendências da Tecnologia

O crescimento acelerado do poder de processamento dos computadores tende a prosseguir, pelo menos, nos próximos dez anos [MCP+98]. A arquitetura de computadores propende para a exploração, cada vez maior, do paralelismo externo e do interno aos processadores. Atualmente, existem vários sistemas SPP em funcionamento, com mais de cem processadores em paralelo, tais como: *Sun NOW*, na Universidade da Califórnia, em Berkeley [ACP95]; *Tera MTA-1*, no SDSC (*San Diego Supercomputer Center*) [ABC+97]; o *HP Exemplar X-Class*, conhecido como *SPP 2000* [AD98] e o *IBM SP Family* [SSA+95].

A tecnologia de rede também tende a continuar evoluindo, aumentando progressivamente as taxas de transferências, enquanto os custos se tornarão cada vez mais acessíveis. As tecnologias emergentes, como Gigabit Ethernet e 1.2 Gbps ATM, ainda consideradas de alto custo, serão amplamente utilizadas, em aproximadamente cinco anos.

As tecnologias de rede de alta velocidade possibilitam comunicação de alto desempenho entre os computadores. Existe, inclusive, uma tendência em utilizá-las na construção de sistemas distribuídos, fortemente acoplados, capazes de suportar tanto as aplicações distribuídas, quanto as aplicações paralelas. Muitas pesquisas têm sido desenvolvidas, com o objetivo de prover comunicação de alto desempenho, tanto sobre redes ATM [ECG+92] [PLC95], quanto sobre redes Gigabit Ethernet [STH+99].

Como as melhorias na velocidade de acesso à memória não acompanham o ritmo de evolução dos processadores, a latência de memória, em relação à execução de instruções, tende a crescer substancialmente [Mud+96]. Essa propensão, aliada à utilização cada vez maior de memórias fisicamente distribuídas, está levando a memórias NUMA (*Nonuniform Memory Access*) [LEK91], com latências que variam de poucos ciclos de processador, para dados em *cache*, até centenas ou milhares de ciclos. Para diminuir essa latência, os projetos tendem a adotar vários níveis de *cache*, procurando aprimorar os esquemas de coerência de *cache*. Também existe a tendência de incorporar instruções de reordenação e leitura antecipada, nos processadores de alto desempenho, e de adicionar inteligência às placas de memória, para otimização dos acessos.

Adicionalmente, os sistemas tendem ao uso cada vez mais intenso de *multithreading*. Uma *thread* é uma abstração da computação, para denotar um grupo de instruções postas para executar como uma unidade. Decompondo o trabalho a ser executado em *threads*, os programas podem mascarar a latência de memória, mantendo *threads* suspensas, enquanto o acesso a dados da memória estiver sendo realizado, e colocando *threads* para executar, quando os dados requisitados estiverem disponíveis.

Diferente das tecnologias citadas anteriormente, que possuem muitas possibilidades de evolução, a tecnologia de discos magnéticos está fadada a tornar-se um entrave progressivamente maior ao desempenho dos sistemas. O tempo de acesso a disco depende de movimentos mecânicos, que são muito difíceis de aprimorar [WZ94]. Portanto, existe uma tendência de ampliação da discrepância entre as velocidades de processador e de disco. Assim, se uma aplicação gera um grande número de pequenas transferências para disco, separadas por posicionamentos, mesmo que conte com processadores muito rápidos, não alcançará um bom desempenho, devido ao atraso introduzido pelos acessos a disco.

A adição de memória não-volátil na *cache* dos sistemas computacionais [BAD+92] é uma das tendências das novas arquiteturas de computadores, permitindo a utilização de esquemas mais agressivos de *caching* de escrita. No entanto, sistemas com alta demanda de operações escrita ainda precisarão de uma forma eficiente de enviar os dados alterados para o disco, quando as *cache*s estiverem cheias. Além disso, quanto mais tempo os dados armazenados permanecerem na *cache*, maior a chance de que sejam corrompidos, em função de um erro de software.

O armazenamento não-volátil de dados deve passar, então, por mudanças radicais, para permitir o surgimento de sistemas mais balanceados, com velocidades crescentes. Outras tecnologias de armazenamento devem substituir os discos magnéticos. A tecnologia de discos de estado sólido é uma forte candidata.

3.3.1 Discos de Estado Sólido

Um disco de estado sólido é um dispositivo de armazenamento de alto desempenho, constituído basicamente de *chips* de memória DRAM e de uma bateria própria, que assegura a manutenção dos dados, caso o fornecimento de energia seja interrompido [Cas98] [BGI97]. Ele possui a mesma interface de um disco magnético convencional, de modo que as diferenças entre esses dois dispositivos são transparentes para o processador. Os discos de estado sólido possuem tempos de acesso extremamente pequenos (menores do que cem microssegundos), inclusive para acessos randômicos, e altas taxas de transferência.

Quando comparados com os discos magnéticos, os discos de estado sólido apresentam as seguintes vantagens: taxas de transferência melhores; o acesso a dados não requer operações de posicionamento, nem apresenta a latência de rotação dos discos magnéticos, levando a tempos de acesso também melhores; contribuem para aumentar a utilização da CPU, porque aceleram as operações de entrada e saída; são mais resistentes a choques e conseguem prover mais confiabilidade ao sistema, apresentando um excelente MTBF (*Medium Time Before Fault*), tipicamente de um milhão de horas.

O maior empecilho para a utilização maciça dos discos de estado sólido são os custos elevados. Portanto, os discos magnéticos ainda devem permanecer como o meio de armazenamento secundário padrão, pelo menos na próxima década. Por essa razão, uma técnica eficiente de armazenamento estável de dados, utilizando a tecnologia de discos magnéticos, continua sob demanda.

3.4 Tendências das Aplicações

Estudos realizados por Baker, no início da década de noventa, já demonstravam o aumento da quantidade de operações de leitura e escrita nas aplicações tradicionais [BHK+91]. Mais recentemente, surgiram novas aplicações, com novos requisitos de armazenamento, tais como[Mud+96]: o processamento de multimídia, que inclui conversas, reconhecimento de voz, codificação e decodificação de imagens de vídeo e videoconferência; grandes bancos de dados interativos; servidores Web altamente carregados; sistemas críticos de tempo real; transações de comércio eletrônico; ambientes de desenvolvimento de aplicações distribuídas e linguagens orientadas a objetos. Essas novas aplicações pressionam os sistemas de arquivos a melhorarem o desempenho e a confiabilidade.

Algumas dessas novas aplicações exigem alta confiabilidade no armazenamento de dados. Por exemplo, as transações de comércio eletrônico e os serviços de banco a domicílio requerem que os servidores de arquivos sejam capazes de armazenar dados, de forma estável. Adicionalmente, precisam oferecer um tempo de resposta aceitável para o usuário.

Portanto, tais aplicações necessitam de um método de armazenamento estável, com baixo custo de desempenho. Considerando que a maioria das novas aplicações utilizam uma infra-estrutura de rede, a técnica de armazenamento estável, proposta neste trabalho, constitui uma alternativa viável.

3.5 Enfoque das Pesquisas Atuais

As pesquisas atuais em sistemas de arquivos visam atender à demanda crescente das aplicações, por armazenamento confiável e de alto desempenho. As técnicas de otimização de desempenho disponíveis, aplicadas à tecnologia corrente, só conseguem atender às necessidades de alguns tipos de aplicações. Com o uso de grandes memórias principais, uma aplicação caracterizada por muitas operações de leitura pode aprimorar o seu desempenho. As aplicações pautadas em acessos seqüenciais a grandes objetos são capazes de explorar o paralelismo da tecnologia RAID, para melhorar, igualmente, o desempenho. Infelizmente, aplicações que realizam muitas operações randômicas de escrita continuam prejudicadas pela tecnologia de disco, principalmente quando necessitam garantir a estabilidade dos dados armazenados.

O problema reside no fato da tecnologia de memória ser a principal alternativa utilizada para a implementação de técnicas de otimização do desempenho. A volatilidade da memória DRAM conduz os sistemas de arquivos a um impasse entre confiabilidade e desempenho [CNR+96].

- As aplicações que necessitam de alto desempenho e admitem alguma perda de dados adotam uma política de consistência de *cache* do tipo *write back*, gravando os dados alterados em disco, de forma assíncrona, somente quando requisitado pelo mecanismo de substituição de páginas da *cache*. Essa estratégia diminui a quantidade de acessos a disco, aumentando, dessa forma, o desempenho do sistema, mas não garante a estabilidade dos dados armazenados: enquanto permanecem na memória principal, os dados são vulneráveis a *crashes*, ou seja, todo o conteúdo da memória pode perder- se.

- As aplicações que necessitam de armazenamento altamente confiável adotam uma política de consistência de *cache* do tipo *write through*, ou seja, assim que um dado é alterado na *cache*, o sistema providencia a transferência desse dado para o disco. O sistema poderia iniciar a gravação do dado alterado e retornar imediatamente o controle para a aplicação. Entretanto, para garantir a estabilidade dos dados, essa gravação é realizada de forma síncrona: o controle só retorna para a aplicação quando a informação já está realmente gravada no disco. Essa estratégia leva à degradação do desempenho, porque aumenta o número de acessos a disco e obriga a aplicação a esperar pelo término da gravação física dos dados.

- O sistema de arquivos do UNIX adota como padrão uma política de consistência de *cache*, denominada *dalayed write*, que espera trinta segundos antes de enviar os dados alterados na *cache* para o disco [Bac86]. Essa estratégia minimiza a degradação do desempenho, causada pelas operações de escrita, se comparada com a política de *write through*. No entanto, torna inevitável a perda total dos dados escritos nos últimos trinta segundos, anteriores a um *crash* [OCH+85]. Além disso, o ganho no desempenho é limitado, pois estudos demostraram que 1/3 a 2/3 do total de dados escritos continuam válidos após os trinta segundos [BHK+91] e são, portanto, inevitavelmente gravados em disco. As aplicações que necessitam armazenar dados de forma estável, no UNIX, também forçam uma gravação síncrona, através dos comandos *sync* ou *fsync* [AT89].

A menos que a tecnologia corrente ou o comportamento das aplicações sofram mudanças radicais imediatas, atender simultaneamente aos requisitos de confiabilidade e desempenho é um importante desafio para os sistemas de arquivos. Muitos trabalhos foram desenvolvidos, com o objetivo de solucionar esse impasse, e alguns deles estão descritos no próximo capítulo.

3.6 Conclusões

A discussão sobre as influências da evolução tecnológica evidenciou pontos importantes:

- os avanços tecnológicos trouxeram benefícios ao desempenho de sistemas de arquivos;
- os componentes de hardware dos sistemas de arquivos evoluem em ritmos diferentes: a velocidade dos processadores apresenta uma taxa de crescimento bem maior do que a velocidade de acesso aos discos magnéticos, condenando a tecnologia de disco a tornar-se um entrave progressivamente maior ao desempenho dos sistemas de arquivos;
- a tecnologia de rede apresenta taxas de transferência cada vez maiores, mas esse recurso ainda é pouco utilizado nas técnicas de otimização de sistema de arquivos;
- existe uma tendência unânime dos sistemas de arquivos existentes, de utilizar a memória principal para amenizar o impacto negativo da tecnologia de disco. Entretanto, as técnicas de otimização baseadas em memória pouco beneficiam as operações de escrita, especialmente quando a aplicação requer armazenamento estável;
- os sistemas de arquivos carecem de uma nova técnica de armazenamento estável de dados, capaz de atender aos requisitos de desempenho e confiabilidade das aplicações.

O SALIUS tem o objetivo de oferecer alta confiabilidade no armazenamento. Adicionalmente, considerando os dados existentes sobre as velocidades de acesso: processador-disco, processador-memória e memória-memória (via rede), uma implementação eficiente do serviço provavelmente levará a um sistema capaz de oferecer armazenamento estável, com desempenho superior à gravação síncrona de dados.

Capítulo 4

Sistemas de Arquivos Robustos: Estado da Arte

O contexto atual dos sistemas de arquivos exige soluções para que os requisitos de desempenho e confiabilidade sejam atendidos satisfatoriamente. Muitos trabalhos foram desenvolvidos nesse sentido. Alguns enfocam somente a questão do desempenho e buscam soluções para diminuir a quantidade de acessos a disco, ou para tornar as operações de escrita em disco mais eficientes. Outros preocupam-se também com a confiabilidade e sugerem métodos para tornar a memória principal capaz de armazenar dados, de forma persistente. Este capítulo descreve trabalhos importantes e apresenta uma análise crítica dos resultados obtidos.

4.1 Sistema de Arquivos Baseado em Log

Um sistema de arquivos baseado em *log* - *Log File System* (LFS), como o Sprite LFS [RO92] e o BDS LFS [SBM+93], é aquele que organiza fisicamente as informações do sistema de arquivos na forma de um único e contínuo *log* seqüencial. O *log* é a única estrutura de dados do disco. Os blocos de dados e todas as estruturas do sistema de arquivos, porventura alterados durante uma operação de escrita, são agrupados em *buffers*, na memória principal, sendo posteriormente gravados no disco, de forma seqüencial, através de uma única operação de saída. Assim, um sistema baseado em *log* consegue minimizar a quantidade de acessos a disco e de posicionamentos decorrentes de operações de escrita, promovendo um ganho substancial no desempenho [OD88].

A disposição física das informações no disco influencia no tempo consumido nas operações de leitura e escrita de dados. O acesso a um bloco de dados, pertencente a um determinado arquivo, requer a leitura prévia de metadados, contendo informações de localização. A escrita de um bloco de dados pode implicar na escrita de metadados que o referenciam.

Num LFS, que mantém os blocos de um arquivo e seus respectivos metadados situados em posições físicas consecutivas, o acesso a um bloco de dados pode ser realizado através de um único acesso a disco. Em contraposição, na maioria dos sistemas de arquivos, que seguem a organização de disco do UNIX FFS (*Unix Fast File System*) [MJL+84], mantendo dados e metadados fisicamente espalhados no disco, uma operação de leitura, ou de escrita de dados, pode provocar vários acessos a disco, intercalados com posicionamentos da cabeça de leitura e gravação.

Por exemplo, o UNIX FFS precisa realizar, pelo menos, cinco operações de saída durante a criação de um arquivo: dois acessos para os atributos de arquivo, pelo menos um acesso para os dados do arquivo, um acesso para os dados do diretório e outro para os atributos do diretório. Nos sistemas de arquivos baseados em *log*, a criação de um arquivo pequeno pode ser realizada com uma única operação de escrita em disco.

O LFS possui estruturas de índices similares àquelas utilizadas pelo UNIX FFS (diretórios, *nós-i*, blocos diretos e blocos indiretos). A principal diferença é que no LFS os *nós-i* não estão localizados em posições fixas no disco. Quando o LFS altera um bloco de dados, ele sabe que deverá gravá-lo no final do *log*. Então, atualiza o *nó-i* do arquivo, para apontar para a nova posição do bloco alterado e, finalmente, grava o bloco de dados, seguido do *nó-i*, de uma só vez, no final do *log*. Essa diferença é a chave para reduzir o número de posicionamentos e de operações de saída, em requisições contendo pequenas modificações de arquivos. Quando os *nós-i* estão em posições fixas, como no UNIX FFS, as alterações de *nós-i*, blocos de diretório e blocos de dados de um arquivo causam pequenas transferências não seqüenciais para o disco.

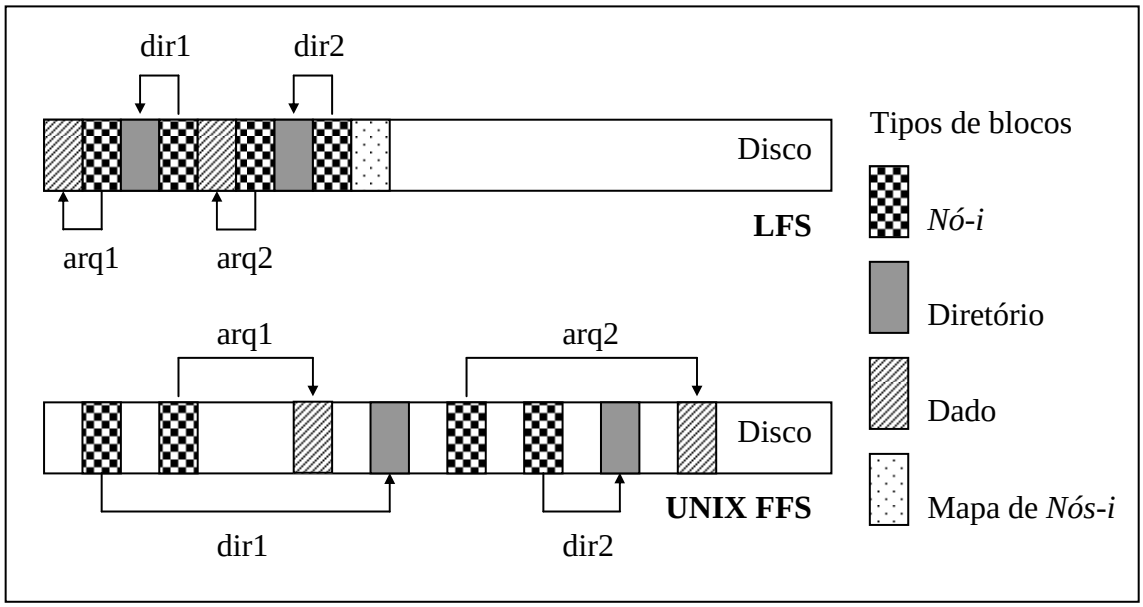

Figura 4.1: Uma comparação da organização de disco do LFS com a do UNIX.

A Figura 4.1 mostra como os atributos, os dados e os índices são alocados no *log* do LFS e compara a organização do disco com a do UNIX FFS. A figura apresenta o conteúdo do disco, nos dois sistemas de arquivos, após a criação de dois novos arquivos, denominados respectivamente de *dir1/arq1* e *dir2/arq2*, cada um contendo apenas um único bloco de dados. Cada sistema de arquivos grava o novo bloco de dados e o *nó-i* para *arq1* e *arq2*, mais o novo bloco de dados e o *nó-i* para os diretórios *dir1* e *dir2*. O UNIX FFS requer várias operações de saída não-seqüenciais, enquanto o LFS armazena todas as informações necessárias em uma única e grande operação de escrita para disco.

4.1.1 Localização e Leitura de Dados

Embora os sistemas de arquivos baseados em *log* simplifiquem as operações de escrita, eles complicam potencialmente as leituras randômicas, pois um bloco pode estar localizado em qualquer lugar do *log*, a depender de onde ele foi gravado. Além disso, os *nós-i* se movimentam no *log*. Para localizar os *nós-i*, o LFS adiciona um nível de indireção, chamado de mapa de *nós-i* (*imap*), que contém a posição corrente de cada *nó-i*.

O mapa de *nós-i* é consultado e atualizado com muita freqüência. Assim, para assegurar um bom desempenho, o LFS realiza *caching* do mapa de *nós-i* e o divide em blocos que, quando alterados, são gravados no final do *log*, sem a necessidade de posicionamentos extras. Os endereços dos blocos do mapa de *nós-i* permanecem sempre na memória. Periodicamente, o LFS grava esses endereços numa região fixa de *checkpoint* (salvaguarda) do disco.

4.1.2 Recuperação de *Crash*

A região de *checkpoint* contém as informações básicas para o procedimento de recuperação do LFS, como a localização dos blocos do mapa de *nós-i*. Após um *crash*, o LFS lê as informações do último *checkpoint* e realiza uma recuperação para frente, varrendo o *log*, a partir da primeira posição gravada após o último *checkpoint*, e realizando ações corretivas.

Quando uma operação de recuperação termina, o mapa de *nós-i* contém as localizações de todos os *nós-i* do sistema e os *nós-i* apontam para todos os blocos de dados. Quanto maior a freqüência na qual os *checkpoints* são gravados, menor o tempo gasto na recuperação do sistema de arquivos, após um *crash* [Mck96]. Porém, o intervalo de tempo entre os *checkpoints* deve ser grande o suficiente, de forma a compensar o tempo gasto com a gravação deles.

Adicionalmente, o LFS mantém um *Log de Operações de Diretórios*, que é utilizado pelo procedimento de recuperação, para restaurar a consistência entre arquivos e diretórios do sistema.

4.1.3 Gerência de Espaço Livre

Outro aspecto importante do LFS é a necessidade de gerenciar o espaço do disco, de modo que grandes extensões de espaço livre estejam sempre disponíveis para a gravação de novos dados. Quando o sistema de arquivos é criado, todo o espaço livre está numa única extensão do disco. No momento em que o *log* atingir o final do disco, o espaço livre já terá sido fragmentado em pequenas extensões, correspondentes aos arquivos ou blocos, apagados ou reescritos.

O LFS divide o espaço do disco em segmentos e utiliza um mecanismo de limpeza que gera segmentos livres, através de uma estratégia que combina encadeamento e cópia. Os blocos válidos de segmentos parcialmente ocupados são copiados e agrupados num segmento livre. Os segmentos restantes são marcados como limpos e encadeados numa lista de segmentos livres, através da qual o *log* cresce.

4.1.4 Sistemas Distribuídos Baseados em Log

Os sistemas distribuídos Zebra [HO95] e xFS [ADN+95] combinam LFS com RAID (*Redundant Array of Inexpensive Disks*) [CLG+94]. Os conceitos de RAID são implementados por software, utilizando-se os discos espalhados ao longo da rede. Assim, conseguem resolver o problema de custo apresentado pelos sistemas que implementam RAID, através de um hardware especial.

Tanto no xFS, quanto no Zebra, as alterações realizadas sobre um arquivo são mantidas num *log* privativo do cliente. Quando as alterações terminam, o *log* é dividido em fragmentos, que são enviados para diferentes discos de servidores, espalhados pela rede. Um disco é dedicado a manter informações de paridade referentes aos diversos fragmentos, adotando o mesmo mecanismo de espalhamento de dados utilizado nos sistemas RAID implementados por hardware.

A Figura 4.2 ilustra como os sistemas Zebra e xFS espalham os dados nos discos distribuídos ao longo da rede. Cada cliente agrupa os dados das operações de escrita num *log* seqüencial, na memória principal. O cliente 1 divide seu *log* nos segmentos 1, 2 e 3, enquanto o cliente 2 divide seu *log* nos segmentos A, B e C. Cada cliente monta um segmento de paridade, calculado a partir dos segmentos de dados de seu respectivo *log*. A seguir, os clientes enviam os seus segmentos para diferentes discos de servidores. Assim, o disco 1 armazena os segmento 1 do cliente 1 e o segmento A do cliente 2, e assim sucessivamente, até o último disco, neste caso, o disco 4, que armazena os segmentos com as informações de paridade.

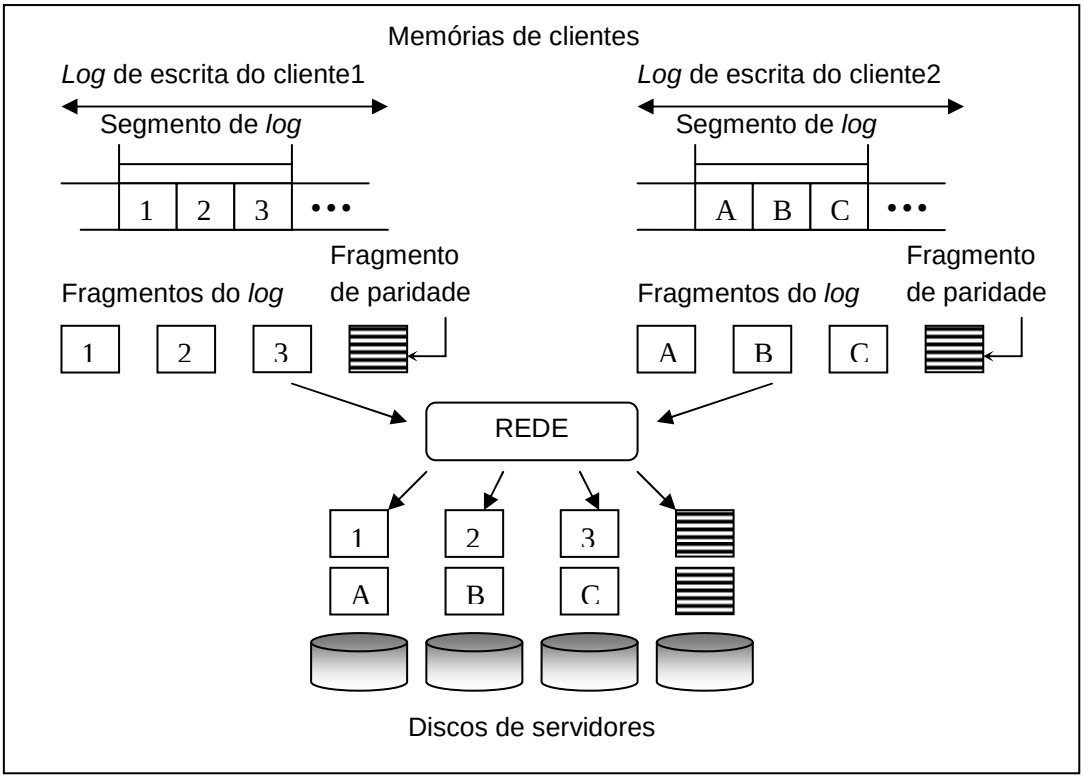

Figura 4.2: Distribuição de dados nos sistemas Zebra e xFS.

Nos dois sistemas, é preciso existir um mecanismo de limpeza de segmentos que cria segmentos limpos e um gerente de arquivos que informa onde os clientes armazenam os dados ao longo do *log*. As principais diferenças entre os dois sistemas são:

- no sistema Zebra, os clientes espalham os dados dos seus *logs* por todos os discos da rede, enquanto no xFS, existem grupos de espalhamento de dados, ou seja, cada cliente espalha os dados de seu *log* apenas nos discos que pertencem ao seu grupo;
- no sistema Zebra, o gerenciamento é centralizado, existindo apenas um gerente de arquivos e um único limpador de segmentos, enquanto no xFS, o gerenciamento é descentralizado, ou seja, existem vários gerentes de arquivos e uma tabela, globalmente replicada, associa cada arquivo ao seu gerente, possibilitando que qualquer cliente possa identificar rapidamente o gerente de um arquivo. Além disso, o xFS implementa um limpador de segmentos distribuído.

4.1.5 Resultados e Análise Crítica

LFS é uma técnica de armazenamento em disco, cujo objetivo principal é obter excelente desempenho nas operações de escrita, que gravam pequenas quantidades de dados. Os sistemas de arquivos baseados em *log*, citados neste texto, conseguem atingir tal objetivo.

Agrupando os dados alterados na memória, para depois gravá-los, através de uma única e contínua operação de saída, o LFS consegue utilizar plenamente a banda passante do disco. Porém, essa estratégia diminui a confiabilidade do sistema. A ocorrência de um *crash* provoca, necessariamente, a perda de dados. O mecanismo de recuperação para frente leva o sistema a um estado consistente, considerando apenas o conteúdo já gravado no *log*, antes da falta. O sistema não consegue recuperar os dados que estavam na *cache* no momento imediatamente anterior ao *crash*. Portanto, o LFS não consegue prover armazenamento estável de dados.

Os resultados das pesquisas realizadas demonstraram que o LFS só apresenta vantagens de desempenho sobre o UNIX FFS nas pequenas operações de escrita. Em escritas longas e nas operações de leitura, ambos os sistemas apresentam desempenhos equivalentes. Assim, os resultados do LFS são vantajosos apenas no ambiente para o qual foi projetado, ou seja, aquele cujas aplicações realizam um grande número de pequenas operações de escrita.

A contribuição dos sistemas baseados em *log*, aumentando eficiência da utilização dos discos magnéticos, durante o processamento de pequenas requisições de escrita, apenas ameniza, mas não resolve o impacto negativo dessa tecnologia no desempenho. A natureza mecânica dos discos magnéticos dificulta a redução do tempo de acesso, predestinando o aumento da discrepância existente entre a velocidade dos processadores e a velocidade de acesso a disco.

O LFS também apresenta problemas de segurança, pois não consegue garantir privacidade de dados [Sta97]. Ele sempre aloca blocos novos, no final do disco, para gravar os novos dados. Portanto, as informações apagadas do sistema de arquivos não são sobrescritas de imediato e permanecem no disco por algum tempo.

4.2 Cache de Memória Não-Volátil

A utilização de pequenas quantidades de memória não-volátil, denominada NVRAM (*Non-Volatile Random Access Memory*), contribui para o aumento do desempenho e da confiabilidade de sistemas de arquivos. Uma memória não-volátil, tal qual uma memória RAM (*Random Access Memory*) com bateria sobressalente, permite a diminuição do tráfego de escrita. Armazenando dados em uma NVRAM, pode-se garantir sua permanência, sem o custo de transferir tais dados da *cache* de cliente para a *cache* de servidor, ou da *cache* de servidor para o disco.

Muitos trabalhos já comprovaram os benefícios da adição de memória não-volátil na *cache* de servidor: a introdução de NVRAM em *cache*s de servidores de sistemas de arquivos UNIX FFS [MJL+84], num ambiente NFS [SGK+85], reduziu o tráfego para disco em torno de cinqüenta por cento [MSC+90]; de forma similar, a controladora de disco 3990-3, da IBM, utilizava quatro megabytes de NVRAM [MH88].

O trabalho de Mary Baker [BAD+92] estendeu os benefícios da memória não-volátil para o escopo de sistema de arquivos distribuído, demonstrando que a adição de pequenas quantidades de NVRAM, na memória *cache* de clientes, pode diminuir o tráfego de rede cliente-servidor, decorrente de operações de escrita. Esse trabalho foi implementado no sistema de arquivos Sprite LFS [RO92] e pode ser dividido em dois tópicos principais: a utilização de NVRAM na *cache* de cliente e a utilização de NVRAM na *cache* de servidor.

4.2.1 Memória Não-Volátil na *Cache* de Cliente

A utilização de NVRAM na *cache* de cliente permite que os dados alterados permaneçam mais tempo na *cache*, antes de serem enviados de volta ao servidor. O sistema Sprite adota uma política de consistência de *cache*, denominada *Propagação Retardada* [NWO88]: o sistema espera um tempo de trinta segundos, antes de enviar um bloco modificado na *cache* de um cliente para o servidor; todo cliente executa um *procedimento de limpeza de cache*, a cada cinco segundos, que envia de volta ao servidor os dados gravados na *cache* há mais de trinta segundos.

Com a adição de NVRAM na *cache* de cliente, o sistema Sprite passou a assegurar a estabilidade dos dados gravados na *cache* não-volátil. Por isso, o envio periódico de dados modificados foi desabilitado. Os dados só retornam ao servidor por solicitação do mecanismo de substituição de páginas da *cache*, ou do mecanismo de controle de concorrência do sistema. As operações *sync* e *fsync* também foram modificadas e passaram a retornar, imediatamente, sem forçar o envio de dados para o servidor.

Foram implementados dois modelos de *cache* com NVRAM:

- no modelo de *cache "write-aside"*, a NVRAM é utilizada apenas como cópia de segurança dos dados alterados na *cache*. Todas as operações de escrita são direcionadas tanto para a *cache* volátil, quanto para a NVRAM. Isso aumenta o tráfego de dados no barramento de memória. As operações de leitura não fazem acesso à NVRAM;

- no modelo de *cache "unified"*, a NVRAM é intensamente integrada com a *cache* volátil. Os blocos modificados por operações de escrita residem somente na NVRAM. Assim, as operações de escrita geram tráfego apenas para a NVRAM. Os blocos lidos do servidor podem residir em ambas as *cache*s: quando a *cache* volátil está cheia, o bloco lido pode ser armazenado na NVRAM, caso exista espaço disponível.

O modelo de *cache "unified"* apresenta vantagens sobre o modelo *"write-aside"*, pois gera menos tráfego no barramento de memória, contribuindo, dessa forma, para um melhor desempenho do sistema. Além disso, o espaço da NVRAM também é utilizado para armazenar dados solicitados por operações de leitura.

4.2.2 Memória Não-Volátil na *Cache* de Servidor

A memória não-volátil é utilizada na *cache* de servidor para reduzir a quantidade de operações de escrita para disco. Em BAD+92, a propagação periódica de dados para disco foi desabilitada. Um bloco alterado na NVRAM somente é gravado em disco quando ele é escolhido pelo mecanismo de substituição de blocos da *cache*. Muitos dados são apagados antes da gravação, o que reduz ainda mais a quantidade de acessos a disco. Adicionalmente, os dados podem ser agrupados e ordenados, de forma que a operação de gravação gere o menor número possível de posicionamentos mecânicos no disco.

4.2.3 Dificuldades na utilização de NVRAM

Existem dificuldades para incorporar a NVRAM ao projeto de um sistema de arquivos. A primeira grande dificuldade está relacionada com a disponibilidade dos dados alterados na *cache* de cliente. Quando um cliente falha e existem blocos alterados em sua NVRAM, se o cliente não conseguir se recuperar rapidamente, esses dados permanecem indisponíveis. Assim, o servidor não pode atender às solicitações de leitura desses dados. A solução apresentada em [BAD+92] é transferir a NVRAM para outra máquina operante, com o objetivo de tornar disponíveis os dados armazenados. Para que seja possível realizar a leitura dos dados em outra máquina, os blocos da NVRAM devem ser capazes de se auto-identificarem.

A segunda dificuldade diz respeito ao protocolo de *cache* de cliente. O sistema deve garantir a persistência dos dados armazenados na *cache* não-volátil de um cliente. Por isso, o protocolo de *cache* de cliente precisaria ser alterado, para não liberar imediatamente da *cache* os dados enviados de volta ao servidor, mas esperar que esses dados sejam gravados em disco. Uma segunda opção seria forçar uma gravação síncrona desses dados, no disco do servidor. O artigo BAD+92 sugere a utilização de NVRAM na *cache* de servidor, permitindo que os dados da *cache* de cliente sejam liberados, tão logo sejam recebidos na *cache* não-volátil do servidor.

4.2.4 Resultados Obtidos

Os testes utilizaram como plataforma o sistema distribuído Sprite [OCD+88], com sistema de arquivos Sprite LFS, e chegaram aos seguintes resultados [BAD+92]:

- a introdução de apenas um megabyte de NVRAM na memória *cache* de clientes conseguiu reduzir o tráfego de rede cliente-servidor em quarenta a cinqüenta por cento;
- o aumento da quantidade de NVRAM utilizada não melhorou significativamente os resultados obtidos;

- o modelo de *cache "unified"* atingiu resultados melhores que o modelo *"write-aside"*, contribuindo para reduzir o tráfego cliente-servidor, tanto nas operações de leitura, quanto nas operações de escrita;
- a introdução de memória NVRAM na *cache* de servidor resultou numa redução de, pelo menos, vinte por cento dos acessos a disco, chegando até noventa por cento, numa simulação com o mesmo sistema de arquivos, sob utilização intensa.

4.2.5 Análise Crítica

A utilização de *cache* de memória não-volátil melhora o desempenho de um sistema de arquivos, pois consegue reduzir tanto quantidade de acessos a disco, quanto o tráfego cliente-servidor, possibilitando que os dados passem mais tempo na *cache* e aumentando, assim, a probabilidade de que esses dados sejam absorvidos na própria *cache*. Porém, as melhorias no desempenho são limitadas, porque a NVRAM está atrelada a um barramento de memória: a sobrecarga do barramento restringe o desempenho dos acessos à NVRAM.

A NVRAM consegue prover armazenamento estável, até certo ponto, pois os dados gravados em memória não-volátil sobrevivem a *crashes* do sistema. Entretanto, se o fluxo de dados das operações de escrita torna-se maior que a capacidade da NVRAM, o sistema passa a transferir dados para o disco. Para garantir a estabilidade desses dados, a gravação em disco precisa ser realizada de forma síncrona. Nesse caso, a NVRAM não apresenta vantagens sobre o armazenamento estável em um sistema de arquivos tradicional.

4.3 Rio File Cache

O principal objetivo do Rio File *Cache* [CNR+96] é tornar a memória RAM capaz de sobreviver a *crashes* de sistema operacional. Para isso, o trabalho adota duas linhas de ação: impede que a memória seja sobregravada acidentalmente pelo sistema, durante um *crash*, e provê o sistema de um mecanismo de recuperação do tipo *"warm reboot"*.

4.3.1 Proteção de Memória

O mecanismo de proteção de memória do Rio File *Cache* fundamenta-se no controle de acesso às páginas da *cache*: utiliza uma técnica denominada *"code patching"*, que impede que rotinas não-autorizadas gravem dados na *cache*. Essa técnica é implementada através da alteração do código do núcleo do sistema operacional. Todas as rotinas do núcleo, que gravam dados na memória, são acrescidas de instruções que verificam se o endereço de armazenamento é de uma página de *cache*. Somente as rotinas de manipulação de *cache* têm permissão explícita para gravar dados na área de *cache*. A utilização do mecanismo de proteção de memória é opcional e aumenta a confiabilidade do sistema de arquivos.

4.3.2 Mecanismo de Recuperação

O mecanismo de recuperação do Rio File *Cache* pressupõe que o sistema é capaz de preservar, por algum tempo, o conteúdo da memória principal e da *cache* de processador, após um reinício normal do sistema. Assim, depois de um *crash*, o sistema atualiza o sistema de arquivos com o conteúdo presente na *cache*. O sistema mantém e protege uma área de memória chamada *registro*, contendo as informações necessárias para localizar, identificar e restaurar dados e metadados de arquivos, a partir dos *buffers* da *cache*. A recuperação é realizada em duas fases:

- logo que o sistema reinicia, o mecanismo de recuperação grava todo o conteúdo presente na memória principal, antes do *crash*, na partição de permuta do disco. A seguir, grava os metadados atualizados no sistema de arquivos do disco, usando para isso os endereços de disco armazenados no *registro*. Assim, o sistema de arquivos fica intacto antes do programa de recuperação *fsck* ser executado;
- quando o sistema finaliza todos os procedimentos de reinício, um processo, no nível de usuário, analisa os dados gravados na partição de permuta e restaura o conteúdo da memória principal.

4.3.3 Efeitos no Projeto de Sistema de Arquivos

O Rio File *Cache* operou algumas alterações no comportamento do sistema de arquivos, para reduzir a freqüência de operações de saída para disco: a gravação periódica dos dados em *cache* foi desabilitada; a rotina *panic* foi modificada, evitando gravar dados para disco antes de um provável *crash*. As rotinas *sync* e *fsync* foram alteradas para retornar imediatamente, sem forçar a gravação síncrona de dados. Assim, as operações de escrita para disco passaram a ocorrer somente durante a substituição de páginas de memória.

Adicionalmente, o Rio File *Cache* possibilitou a ordenação das alterações de metadados na *cache*, do mesmo modo como é feito nas operações de escrita em disco, minimizando a possibilidade de produzir inconsistências durante um *crash*.

Esse sistema conseguiu, também, aplicar atomicidade à escrita de metadados críticos, utilizando um esquema de cópias de páginas: antes de alterar uma página da *cache* contendo metadados críticos, o sistema copia o seu conteúdo, numa nova página de memória, e faz com que o *registro* aponte para essa cópia; quando a operação termina, o sistema faz com que o *registro* volte a apontar para a página original da *cache*, com os metadados já alterados.

4.3.4 Suporte de Arquitetura Necessário

A memória do sistema e a *cache* do processador devem ser capazes de preservar seus respectivos conteúdos, após um reinício normal do sistema. Equipamentos do tipo DEC Alphas possuem essa característica [DEC94]. No entanto, a maioria dos computadores não possui essa facilidade.

O mecanismo de proteção causa um impacto negativo no desempenho do sistema: as rotinas do núcleo precisam executar instruções adicionais de verificação, toda vez que vão escrever dados na memória, para assegurar que apenas as rotinas de manipulação de *cache* gravem dados na área de *cache*. Otimizações de código foram implementadas, com o objetivo de minimizar esse impacto.

Para tornar o mecanismo de proteção simples e eficiente, seria necessário um suporte de hardware, para implementar um controlador de memória capaz de impedir escritas não-autorizadas em certas páginas físicas [BMR+91], nesse caso, as páginas de *cache*. Uma implementação simples desse controlador pode ser realizada, acrescentando-se a cada página de memória um bit de permissão de escrita e mapeando as permissões no espaço de endereçamento do processador [CNR+96]. Assim, o sistema poderia utilizar os bits de permissão de escrita para substituir o esquema de *"code patching"* do mecanismo de proteção.

4.3.5 Resultados e Análise Crítica

O Rio File *Cache* consegue, de fato, tornar a memória principal apta a sobreviver a *crashes* de sistema operacional [CNR+96]. A confiabilidade atingida, mesmo sem o mecanismo de proteção de memória, é equivalente a de um sistema de arquivos que adota a política de *cache write through*, enquanto seu desempenho é vinte vezes maior. O Rio torna todas as operações de escrita imediatamente permanentes e ainda executa mais rápido do que sistemas como o UNIX FFS, que adotam a política de *Propagação Retardada* dos dados da *cache* para o disco. O mecanismo de proteção de memória pode ser utilizado para tornar o sistema ainda mais confiável, causando, entretanto, uma queda no desempenho.

Apesar dos resultados obtidos com esse trabalho serem bastante positivos, o sistema depende da disponibilidade de um hardware especial, que não é comum nos computadores utilizados em larga escala. A adaptação de um sistema distribuído, de modo que todos os computadores contenham esse recurso, implica em altos custos. Portanto, o Rio File *Cache* não consegue prover armazenamento estável de dados a um custo acessível.

4.4 Sistema de Armazenamento em Memória Não-Volátil

Flash RAM é um tipo de memória não-volátil, isto é, consegue armazenar dados de forma persistente. Porém, apresenta problemas básicos, quando é utilizada para prover um sistema de memória não-volátil de propósito geral. Não obstante, alguns trabalhos, dos quais pode-se destacar o eNVy [WZ94], fazem uso da tecnologia Flash, com esse propósito.

Um *chip* Flash é estruturalmente e funcionalmente muito similar a um *chip* EPROM (*Erasable Programable Read-Only Memory*), exceto pelo fato de seu conteúdo ser eletricamente removível [Int94]. Cada *chip* consiste de um extenso vetor de *bytes*, formado por células de memória não-volátil. Embora a estrutura simples desses *chips* permita que sejam fabricados em larga escala e a um baixo custo, a memória Flash RAM não é utilizada vastamente porque apresenta algumas deficiências básicas [WZ94]:

- nos *chips* Flash, os dados não podem ser gravados sobrepondo o conteúdo anterior, ou seja, o conteúdo antigo precisa ser apagado, antes que novos dados sejam gravados. Adicionalmente, a remoção de dados num *chip* Flash só acontece em grandes volumes, o que significa que todo o conteúdo do dispositivo é removido numa única e demorada operação, que dura cerca de cinqüenta milissegundos. A tecnologia Flash mais recente permite alguma flexibilidade, através da divisão do vetor de memória em blocos, que podem ser apagados de forma independente. Desse modo, cada operação de remoção apaga somente um bloco, em vez de apagar todo o banco de memória. Ainda assim, a operação de remoção é bastante lenta, se comparada com outros tipos de memória;
- uma operação de programação/remoção de dados gasta muito mais tempo que uma operação de leitura. Uma leitura arbitrária pode ser realizada com tempo de acesso próximo ao de uma memória DRAM (setenta nanossegundos) [AMD99]. A tecnologia Flash mais sofisticada divide a memória em páginas, para otimizar a leitura, sendo que o acesso inicial a uma página pode ser realizado em sessenta e cinco nanossegundos e, desse ponto em diante, a leitura de um dado, localizado em uma posição qualquer da página, pode ser realizada em apenas vinte e cinco nanossegundos [AMD99a]. Em contraposição, a programação de *bytes* individuais é bem mais lenta, durando de quatro a dez microssegundos, e ainda deve esperar que o conteúdo anterior seja apagado;
- a memória Flash RAM possui tempo de vida útil limitado. O método de programação utilizado pela tecnologia Flash degrada, gradualmente, o tempo de programação e de remoção do conteúdo da memória, cada vez que essas operações são executadas. Um *chip* falha quando uma operação de programação ou de remoção leva mais tempo do que o permitido na especificação. Cada fabricante garante um número mínimo de ciclos de programação/remoção dentro de uma faixa de tempo preestabelecida. Esse número varia de dez mil a um milhão de vezes [AMD99].

4.4.1 Sistema eNVy

O eNVy [WZ94] utiliza a memória Flash RAM como a base para um sistema de armazenamento não-volátil em memória principal, adotando uma variedade de técnicas para superar as deficiências apresentadas pela tecnologia Flash.

O eNVy adota um esquema de cópias de páginas na escrita, simulando para o processador que os dados sobrepõem o conteúdo antigo. O vetor de memória é tratado como um espaço linear de endereçamento, dividido em páginas. Uma tabela de páginas mapeia o endereço lógico linear, apresentado ao processador, para um endereço físico no *chip* Flash. Quando o processador requisita que dados sejam armazenados num endereço específico, o eNVy faz uma nova cópia da página correspondente, incluindo os dados atualizados, e altera a tabela de páginas para que aponte para a nova versão.

Como a programação da memória Flash RAM é uma operação lenta, durante a atualização de uma página de memória, a nova versão desta é criada num outro tipo de memória: o eNVy utiliza uma pequena quantidade de memória SRAM (*Synchronous Random Access Memory*), com bateria sobressalente. A tabela de páginas passa a apontar para a página escrita na SRAM. A tabela de páginas também é mantida na SRAM, pois sofre atualizações constantes.

O gerenciamento da SRAM é realizado através de uma política simples de fila. À proporção que a SRAM vai atingindo um nível de enchimento preestabelecido, suas páginas vão sendo gravadas na memória Flash. Se não existir espaço livre na memória Flash, uma operação de limpeza é realizada. O algoritmo de limpeza utilizado pelo eNVy é funcionalmente similar ao algoritmo de limpeza do LFS [RO92].

Algumas simulações do sistema eNVy foram realizadas, utilizando-se as memórias Flash RAM e SRAM, com as características apresentadas na Tabela 2.1.

TIPO DE MEMÓRIA	QUANTIDADE	TEMPO DE LEITURA	TEMPO DE ESCRITA
Flash RAM	2 Gb	85 ns	4 –10 ms
SRAM	16 Mb	85 ns	85ns

Tabela 2.1: Características das memórias utilizadas na simulação do sistema eNVy.

Os resultados da simulação foram os seguintes:

- tempo de latência médio de cento e oitenta nanossegundos, para as leituras, e de duzentos nanossegundos, para as escritas;

- conforme a taxa de transações por segundo for aumentando, mais dados vão sendo armazenados na memória. Quando o nível de enchimento da Flash RAM ultrapassa oitenta por cento, o tempo das operações de escrita aumenta para aproximadamente sete microssegundos, devido ao tempo consumido pelo mecanismo de limpeza.

4.4.2 Análise Crítica

O sistema eNVy foi projetado para aplicações com base de dados de pequeno e médio portes. O desempenho do sistema eNVy está atrelado à taxa de transações por minuto e à quantidade de dados armazenada pela aplicação. Assim, o eNVy não pode ser utilizado de forma genérica. Somente algumas aplicações, com o comportamento previsível, podem beneficiar-se desse sistema.

Portanto, a tecnologia de memória Flash ainda precisa superar suas deficiências, para que possa ser utilizada como principal dispositivo de armazenamento estável de um sistema de arquivos de propósito geral.

4.5 Conclusões

A Tabela 4.1 resume as principais características dos diversos trabalhos apresentados neste capítulo. Todos eles visam melhorar o desempenho dos sistemas de arquivos e apresentam resultados positivos, em maior ou menor escala. Em alguns casos, o desempenho do sistema só é vantajoso para um ambiente específico, como alguns sistemas baseados em *log* [RO92] [SBM+93], projetados para aplicações com predominância de pequenas operações de escrita, como os sistemas de automação de escritório, e o sistema eNVy [WZ94], projetado para aplicações com base de dados de pequeno e médio portes.

A melhoria no desempenho provida por sistemas baseados em *log* [ADN+95] [HO95] [RO92] [SBM+93] resulta da utilização mais eficiente dos discos, durante as operações de escrita. Assim, conseguem amenizar, temporariamente, o impacto negativo da tecnologia de disco magnético no desempenho. Porém, as tendências da tecnologia, apresentadas no capítulo anterior, apontam a necessidade de desvincular o desempenho do sistema do desempenho dos discos, o que não acontece em tais sistemas.

A utilização de cache de NVRAM [BAD+92] só consegue prover ganhos no desempenho enquanto a memória NVRAM tiver capacidade para armazenar os dados das operações de escrita, evitando acessos a disco. Um aumento do tráfego de escrita pode esgotar o espaço da NVRAM, gerando a necessidade de transferir dados para o disco.

Os sistemas baseados em *log* não garantem a estabilidade dos dados armazenados. Os demais trabalhos apresentados conseguem garantir confiabilidade no armazenamento de dados: a cache de NVRAM, o Rio File Cache [CNR+96] e o sistema eNVy [WZ94]. Eles buscam capacitar a memória principal para armazenar dados de forma estável. Todos utilizam algum hardware especial, ou seja, dispositivos que não fazem parte da arquitetura dos computadores utilizados em larga escala. A adoção desses dispositivos especiais geralmente implica em custos altos.

Somente os sistema baseados em *log* e o Rio File Cache incluem um mecanismo de recuperação de *crash*. No caso dos sistemas baseados em *log*, o procedimento de recuperação utiliza apenas as informações gravadas no *log*, para restaurar a consistência do sistema de arquivos. Desse modo, as informações presentes na cache, no momento do *crash* são irremediavelmente perdidas.

Com exceção, do sistema eNVy, desenvolvido para um ambiente específico, os demais trabalhos podem ser utilizados como sistemas de propósito geral. Embora o LFS [RO92] tenha sido desenvolvido para atender as necessidades de um ambiente bem definido, os sistemas baseados em *log* podem ser utilizados por aplicações de diversas naturezas, sendo que o desempenho apresentado pode variar de um ambiente para outro.

SISTEMAS	ARMAZENAMENTO ESTÁVEL	MELHORIA NO DESEMPENHO	HARDWARE ESPECIAL	MECANISMO DE RECUPERAÇÃO	PROPÓSITO GERAL
BASEADOS EM LOG	NÃO	SIM	NÃO	SIM	SIM
CACHE DE NVRAM	SIM	SIM	SIM	NÃO	SIM
RIO FILE CAHE	SIM	SIM	SIM	SIM	SIM
SISTEMA ENVY	SIM	SIM	SIM	NÃO	NÃO

Tabela 4.1: Quadro comparativo de trabalhos relacionados.

Nenhum dos trabalhos apresentados constitui uma técnica de armazenamento estável de propósito geral a um custo acessível, com desempenho superior à gravação síncrona de dados, capaz de desvincular o desempenho do sistema do desempenho dos discos e de oferecer um mecanismo de recuperação para frente.

Capítulo 5

Especificação do SALIUS

Este capítulo apresenta a especificação de uma nova técnica de armazenamento estável, denominada SALIUS — SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS —, que visa atender, simultaneamente, aos requisitos de confiabilidade e desempenho de aplicações que necessitam gravar dados de forma estável, utilizando a tecnologia ora disponível. Ele descreve os conceitos básicos e as técnicas empregadas, incluindo como os sistemas de arquivos são alterados, como a replicação é utilizada e como a recuperação de *crash* é realizada.

O SALIUS é um software capaz de realizar operações sobre arquivos, de forma estável, sempre que requisitadas pelos seus clientes. O SALIUS é um software complementar ao sistema de arquivos, disponibilizando novas operações de manipulação de arquivos para os clientes. Um cliente do serviço é um programa de usuário, ou seja, o mesmo cliente usual de um sistema de arquivos. Um cliente faz requisições ao SALIUS, quando deseja manipular um arquivo de forma estável. Assim, o SALIUS é um complemento do serviço de arquivos, capaz de garantir a estabilidade dos efeitos de operações realizadas sobre arquivos, a despeito da ocorrência de *crashes*.

Este Capítulo apresenta os principais objetivos do SALIUS e os requisitos que devem ser cumpridos pelo serviço, descreve o funcionamento geral do SALIUS e a interação entre seus componentes. A seguir, cada parte componente do SALIUS é descrita em detalhe. Finalmente, algumas considerações sobre o desempenho e a confiabilidade do serviço são apresentadas.

5.1 Principais Objetivos

- **Estabilidade de Dados.** O principal objetivo do SALIUS é garantir a estabilidade dos dados, ou seja, preservar os dados armazenados por suas primitivas, independentemente da ocorrência de *crashes* no sistema.
- **Desempenho.** O serviço pretende oferecer um desempenho melhor do que o armazenamento estável baseado na gravação síncrona de dados.
- **Custo.** O serviço tem como meta não introduzir custos adicionais ao sistema.
- **Capacidade de Recuperação de *Crash*.** O serviço visa oferecer um mecanismo de recuperação capaz de restabelecer o sistema de arquivos a um estado consistente, após um *crash* na máquina onde o sistema de arquivos reside, garantindo a estabilidade de todos os dados armazenados através de primitivas do SALIUS.
- **Serviço de propósito geral.** O serviço deve atender às necessidades de armazenamento de aplicações de diversas natureza, não se restringindo a um contexto específico.

5.2 Requisitos

O SALIUS deve atender a certos requisitos, para que consiga atingir os seus objetivos.

5.2.1 Semântica de Falha

O projeto de um Serviço de Armazenamento Estável deve especificar a semântica de falha assumida para o sistema. O tipo de semântica de falha influencia na especificação do comportamento do serviço, para que possa garantir a estabilidade dos dados armazenados. Por exemplo, se um projeto assume uma semântica de falha por *crash* e/ou valor, para a operação de recuperação de dados armazenados em disco, então ele deve prever a replicação de dados em discos diferentes. Se o projeto também assume que o sistema apresenta uma semântica de falha por valor, em resposta às requisições de leitura de dados armazenados na memória principal, significa que os dados em memória podem ter seus valores corrompidos e, portanto, o serviço deve possuir um mecanismo de proteção de memória, como em [CNR+96].

O SALIUS assume uma *semântica de falha por crash***.** Quando ocorre um *crash*, o sistema pára, precisando ser reiniciado, para que volte a funcionar. Assim, todo conteúdo da memória principal volátil é perdido. O serviço precisa garantir a sobrevivência dos dados provenientes de todas as operações de escrita, realizadas através de suas primitivas e concluídas com sucesso, mesmo que tais dados não tenham sido propagados da memória para o disco, antes do *crash*. Falhas no meio de armazenamento não-volátil e na memória principal fogem ao escopo deste trabalho.

5.2.2 Semântica de Compartilhamento

Semântica de compartilhamento é uma especificação dos efeitos de operações feitas por múltiplos usuários que estão realizando acessos a um mesmo arquivo, simultaneamente. Na prática, essa semântica deve definir o momento exato em que as modificações de dados, realizadas por um processo, tornam-se visíveis aos demais processos [Tan95].

Como o enfoque da pesquisa foi direcionado para o sistema de arquivos do UNIX, o SALIUS deve manter a mesma semântica de compartilhamento do UNIX. Assim, deve obrigar a uma ordenação das operações de leitura e escrita, retornando sempre o valor mais recente. *"Quando uma leitura segue uma escrita, o sistema retorna o valor que acabou de ser escrito. Quando uma leitura segue duas operações de escrita, o sistema retorna o valor resultante da última escrita"* [Tan95]. Desse modo, os efeitos de uma operação de escrita tornam-se imediatamente visíveis, para todos os processos, e o sistema mantém uma imagem única de cada arquivo.

5.2.3 Transparência

A transparência pode ser analisada sob vários aspectos. Na abordagem do SALIUS, a transparência se traduz na ilusão do usuário de que o comportamento do sistema de arquivos original permanece inalterado. O serviço deve esconder da aplicação as ações internas, executadas com a finalidade de prover estabilidade. Para isso, deve ser transparente em alguns aspectos:

- **transparente quanto à localização**, ou seja, o programa de usuário não precisa saber onde os dados estão sendo replicados, nem de qual máquina serão recuperados;
- **transparente quanto à replicação**, ou seja, a aplicação não deve tomar conhecimento de quantas réplicas existem, nem das ações necessárias para manter a consistência;
- **transparente quanto à falha:** deve oferecer meios de mascarar falhas, quando possível, e, quando não, deve efetuar medidas de recuperação.

O SALIUS não precisa prover *transparência de acesso*. O modo como um usuário invoca uma operação no serviço pode ser diferente da forma como a mesma operação é invocada no sistema de arquivos original. Isso porque, para a maioria das operações, a semântica assumida no SALIUS difere da semântica do sistema de arquivos original. Se comparado com a gravação assíncrona de dados, o serviço, normalmente, consome mais recursos e apresenta um desempenho inferior, em decorrência da replicação de dados. Em muitas ocasiões, a aplicação não necessita pagar pela estabilidade de dados. Para prover *flexibilidade* ao usuário, de optar pela utilização do serviço, de acordo com a sua conveniência, a transparência de acesso pode ser sacrificada.

5.2.4 Facilidade de Administração

O SALIUS não deve introduzir dificuldades na administração do sistema. O esforço exigido para gerenciar o novo serviço precisa ser mínimo. Basicamente, o administrador deve realizar tarefas de configuração, no momento da instalação do serviço. Também é tarefa do administrador executar o procedimento de recuperação, após um *crash*.

5.2.5 Independência de Hardware Especial

A implementação do SALIUS não deve necessitar de um hardware especial. A introdução de um hardware especial num sistema, geralmente, acarreta custos adicionais. Assim, para atender ao objetivo de não onerar custos ao sistema, é necessário que o serviço possa funcionar com os componentes de hardware comuns na maioria dos sistemas computacionais.

5.3 Funcionamento Geral do SALIUS

A idéia fundamental do SALIUS é garantir a estabilidade de dados, minimizando ao máximo a necessidade de realizar gravações síncronas em disco. Baseando-se no fato de que um acesso à memória principal (na ordem de nanossegundos) é muito mais rápido do que um acesso a disco (na ordem de milissegundos) e de que a latência decorrente de um acesso através de uma rede de alta velocidade é menor do que a latência introduzida pela leitura de um bloco num disco local [Dah+94], o SALIUS substitui a gravação síncrona de dados pela replicação remota de *buffers* alterados na *cache* de arquivos.

A estabilidade dos dados é garantida, porque todas as estruturas de dados, mantidas na memória principal pelo sistema de arquivos, são copiadas para as memórias principais de diferentes máquinas do sistema distribuído: o sistema de arquivos do UNIX mantém em memória a *buffer cache*, os *nós-i* e o superbloco; qualquer alteração nessas estruturas é replicada pelo SALIUS. Como a replicação garante a sobrevivência de, pelo menos, uma cópia dos dados alterados, após a ocorrência de um *crash*, o conteúdo replicado pode ser recuperado, a partir de uma outra máquina, e gravado no sistema de arquivos em disco.

O SALIUS é constituído de quatro componentes básicos:

- **Interface**: um conjunto de primitivas, incorporadas à interface do sistema de arquivos e utilizadas pelos programas de usuário para requisitar operações ao SALIUS;

- **Servidor de Arquivos Complementar (SAC)**: é a implementação das operações invocadas através de primitivas do SALIUS. O SAC realiza operações sobre arquivos, requisitadas pelos clientes, e interage com o servidor de replicação de *buffers*, solicitando a replicação dos dados alterados;

- **Serviço de Replicação de Buffers**: um software que provê a replicação confiável de *buffers*, ou seja, garante o armazenamento de cópias das alterações realizadas num sistema de arquivos. Adicionalmente, esse serviço fornece as informações necessárias para restabelecer o sistema de arquivos a um estado consistente, preservando os dados estáveis. O Serviço de Replicação de Buffers é implementado por um ou mais processos Servidores de Replicação de Buffers (SRB) e possui dois tipos de clientes diferentes: o SAC e o Procedimento de Recuperação;

- **Procedimento de Recuperação**: um software que restabelece a consistência do sistema de arquivos, após um *crash*. O procedimento de recuperação interage com o SRB para recuperar as informações replicadas, necessárias para restaurar a consistência do sistema de arquivos, mantendo as informações estáveis.

Se uma aplicação quiser atualizar um arquivo, de forma estável, basta invocar uma primitiva do SALIUS. Então, o controle passa para o Servidor de Arquivos Complementar (SAC), que executa a operação requisitada: para isso, o SAC atualiza as estruturas do sistema de arquivos, em memória, e interage com o Servidor de Replicação de Buffers (SRB), solicitando a replicação de todas as alterações efetuadas. O SRB realiza a replicação e envia uma resposta ao SAC, confirmando a estabilidade dos dados.

Uma mensagem de confirmação só pode ser enviada quando o SRB puder garantir, com uma determinada probabilidade, que a informação está estável. Somente após receber a confirmação do SRB, o SAC retorna o controle para o programa de usuário, informando que a operação requisitada foi realizada com sucesso. Se a mensagem de confirmação não chega dentro de um intervalo de tempo predefinido, o SAC assume que o SRB falhou. Nesse caso, o SAC força a gravação de todos os dados do sistema de arquivos alterados na memória e ainda não propagados para o disco, passando a estabilizar as informações através de gravações síncronas. Quando o SRB se recupera de uma falha, o SAC volta a utilizar o serviço de replicação.

O funcionamento básico do SALIUS pode ser observado na Figura 5.1. A máquina 1 possui um sistema de arquivos, acrescido do Servidor de Arquivos Complementar do SALIUS, e um processo usuário ‘U’, que invoca uma primitiva do SALIUS para alterar um arquivo, passando os dados alterados como parâmetro, num *buffer* de usuário. Quando a primitiva é executada, o SAC copia esses dados para a *buffer cache*. Se necessário, o superbloco e o *nó-i* do arquivo também são atualizados. A seguir, num tempo **t**, o SAC envia as estruturas alteradas para o Servidor de Replicação de Buffers, executando na máquina 2. O SRB recebe as informações enviadas e, depois de armazená-las na memória principal, envia uma mensagem de confirmação para o SAC. Se o SAC receber uma confirmação, dentro de um intervalo de tempo predefinido (Δt), então retorna o controle para o processo ‘U’, informando o sucesso da operação. Caso contrário, o SAC detecta que o SRB falhou e realiza a gravação síncrona dos dados em disco, antes de retornar.

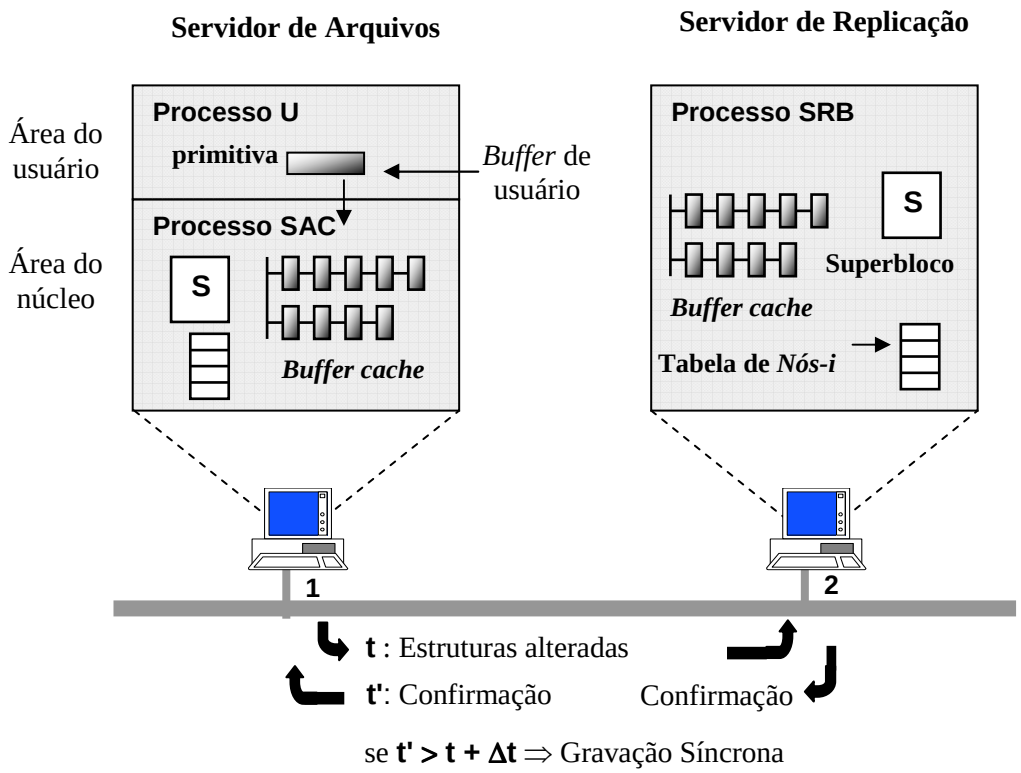

Figura 5.1: Funcionamento básico do SALIUS.

Se a replicação é confirmada no tempo esperado ($t' \leq t + \Delta t$), os dados alterados na memória principal da máquina 1 permanecem na *cache* por um certo tempo e depois são gravados em disco, de forma assíncrona, como no funcionamento padrão do sistema de arquivos do UNIX. Assim, o SALIUS possibilita que o sistema de arquivos do UNIX realize *caching* de escrita e ofereça armazenamento estável, simultaneamente. A depender da relação entre o volume de dados alterados e a capacidade de memória principal disponível nas máquinas do sistema, o tempo de permanência dos dados alterados na *cache* pode ser aumentado para um valor maior do que o valor padrão (que é de trinta segundos, no UNIX), contribuindo para um melhor desempenho do sistema.

O Servidor de Replicação de Buffers recebe os dados replicados e os armazena num *log*, mantido inicialmente na memória principal. Quando o espaço na memória torna-se escasso, uma parte das informações replicadas é transferida para um disco. Periodicamente, o SRB realiza uma operação de limpeza do *log*, descartando as informações que já tenham sido gravadas em disco, na máquina 1.

Finalmente, após um *crash*, o procedimento de recuperação do SALIUS deve ser executado, para restaurar a consistência do sistema de arquivos. O procedimento de recuperação envia uma mensagem ao SRB, solicitando todos os dados replicados antes do *crash*. O SRB responde, fornecendo as informações requisitadas. A seguir, o procedimento de recuperação grava tais informações no sistema de arquivos, em disco.

O Serviço de Replicação de Buffers, ilustrado na Figura 5.1, é implementado por um único processo servidor. Porém, esse serviço pode ser implementado por várias réplicas desse processo, executando em diferentes máquinas do sistema, com o objetivo de aumentar a sua confiabilidade, como no projeto apresentado no Capítulo 6.

5.3.1 Tratamento de Falha na Replicação

Quando uma replicação não é confirmada no tempo esperado, o SAC assume que o SRB falhou. Por isso, realiza a gravação síncrona, garantindo a estabilidade dos dados provenientes da operação de atualização em curso. Porém, essa ação não é suficiente para preservar a semântica de compartilhamento do serviço. Na prática, o SRB pode continuar operacional, e a falha detectada ser decorrente de algum problema na comunicação entre o SAC e o SRB. Se o sistema sofrer um *crash* logo após da gravação síncrona dos dados, o procedimento de recuperação gravará no disco os dados replicados no SRB, antes do *crash*. O SRB pode conter dados desatualizados, que irão sobrepor informações mais recentes, armazenadas com a gravação síncrona.

A situação descrita pode ser demonstrada com o seguinte exemplo: um bloco x é atualizado e replicado no SRB; em seguida, outra operação modifica o mesmo bloco e tenta realizar a replicação, mas o SRB falha e, portanto, x é gravado de forma síncrona; a seguir, o sistema de arquivos sofre um *crash* e o procedimento de recuperação grava o antigo conteúdo do bloco x, fornecido pelo SRB, violando a semântica de compartilhamento adotada pelo serviço, porque uma atualização antiga sobrepõe uma atualização mais recente.

Para preservar a semântica de compartilhamento, o SAC precisa evitar que dados desatualizados permaneçam no SRB. Por isso, quando o SRB falha, o SAC grava em disco todas as atualizações do sistema de arquivos, que ainda estão pendentes na memória, passando a operar através de gravações síncronas. Além disso, o SRB precisa detectar que falhou e passar por um procedimento de inicialização, que consiste em descartar os dados do *log*, antes de voltar a atender aos pedidos de replicação. O SAC volta a utilizar o serviço de replicação, assim que detecta a inicialização do SRB.

A forma como o SRB detecta que falhou e como o SAC detecta uma inicialização do SRB depende do modo como o SRB é implementado, mais especificamente, do modelo de replicação adotado, da semântica de falha e do protocolo de replicação do serviço. Esses detalhes serão abordados no Capítulo 6, que apresenta um projeto de implementação para o Serviço de Replicação de Buffers.

5.4 Interface do SALIUS

As aplicações podem invocar as operações do SALIUS, usando as primitivas do serviço, que compõem a sua interface. O objetivo dessa interface é possibilitar que o usuário continue realizando as principais operações de alteração, possíveis no sistema de arquivos original, com a funcionalidade complementar de prover estabilidade de forma alternativa à usual, isto é, evitando as gravações síncronas. Conceitualmente, para cada chamada de sistema de alteração do sistema de arquivos original, existe uma primitiva correlata na interface do SALIUS.

Existem duas abordagens possíveis para incorporar um Serviço de Armazenamento Estável a um sistema de arquivos: substituindo todas as primitivas originais pelas primitivas do serviço, ou adicionando novas primitivas à interface do sistema. Na primeira abordagem, o armazenamento estável passa a ser a única forma possível de manipular dados. O serviço é totalmente transparente ao usuário: ao invocar uma primitiva do sistema de arquivos, o usuário grava dados de forma estável, automaticamente. Na segunda alternativa, a transparência cede lugar à flexibilidade, pois o usuário opta entre utilizar o SALIUS, ou continuar usando as primitivas tradicionais.

A especificação do SALIUS prioriza a flexibilidade, porque muitas aplicações não necessitam pagar os custos do armazenamento estável. Além disso, uma mesma aplicação pode requerer a estabilidade de apenas parte dos dados manipulados. Por exemplo, uma aplicação pode desejar gravar seus arquivos de *log* e de auditoria, de forma estável, mas pode prescindir da estabilidade para os demais arquivos. Oferecendo suas primitivas como uma alternativa adicional, para a manipulação de arquivos, o serviço propicia ao usuário o benefício de decidir sobre a conveniência de utilizar o armazenamento estável.

Uma segunda deliberação, muito importante, refere-se à granularidade atribuída à unidade de armazenamento estável. O SALIUS assume que a necessidade de armazenamento estável aplica-se, em geral, a um arquivo inteiro e não a uma fração de arquivo. Por conseguinte, o usuário opta pela utilização do serviço no momento em que abre ou cria um arquivo, indicando explicitamente que o arquivo deve ser manipulado através de primitivas do SALIUS.

Assim, na especificação do SALIUS para UNIX, no nível lógico, o serviço também possui uma nova versão de cada primitiva de alteração do sistema de arquivos original. As primitivas do serviço, descritas a seguir, realizam praticamente as mesmas ações que as primitivas do UNIX, somando a replicação de *buffers*, para garantir a estabilidade de informações armazenadas.

5.4.1 open

A primitiva `open` é utilizada para abrir um arquivo existente, ou criar e abrir um novo arquivo. Através dela, o usuário pode optar pela utilização do SALIUS.

Sintaxe: `fd = open(pathname, flags, mode)`

- *fd* é um descritor de arquivo, que é retornado para a aplicação;
- *pathname* é o nome de caminho do arquivo;
- *flags* é uma combinação de bits, com diretrizes importantes, resumidas na Tabela 5.1;
- *mode* contém as permissões de acesso associadas ao arquivo. Esse parâmetro só é obrigatório para a criação de um arquivo.

A sintaxe do `open` do UNIX foi ligeiramente alterada, adicionando a opção **O_STABLE** no parâmetro *flags*. Ao informar este *flag*, o usuário faz a opção de utilizar o SALIUS na manipulação do arquivo.

O_RDONLY	O arquivo é aberto apenas para leitura.
O_WRONLY	O arquivo é aberto apenas para escrita.
O_RDWR	O arquivo é aberto para leitura e escrita.
O_APPEND	Todas as escritas irão adicionar dados ao final do arquivo.
O_CREAT	Se o arquivo existir, não tem efeito algum. Caso contrário, o arquivo é criado e o parâmetro *mode* deve ser especificado.
O_STABLE	**Todas as alterações do arquivo serão realizadas pelo SALIUS.**
O_EXCL	O arquivo é aberto em modo exclusivo. A operação retorna um erro, se o arquivo existir e o O_CREAT estiver prescrito.
O_TRUNC	Se o arquivo existir, seu tamanho é truncado para zero.
O_SYNC	Indica a gravação síncrona de todas as alterações do arquivo.
O_NOCTTY	O Terminal especificado em *pathname* torna-se o terminal de controle.
O_NDELAY	Afeta todas as leituras e escritas subseqüentes. Se uma operação de leitura, ou escrita, não puder ser realizada, retorna, imediatamente, com erro, sem entrar em uma condição de bloqueio.

Tabela 5.1: *Flags* da operacão *open*, no SALIUS para UNIX.

Semântica: a primitiva `open` abre o arquivo especificado em *pathname*, na forma indicada em *flags*. Se o arquivo não existir e o *flag* O_CREAT for informado, `open` realiza a criação do arquivo e armazena, no *nó-i* associado, as permissões de acesso contidas em *mode*. Normalmente, o arquivo é posicionado no início, mas, se o *flag* O_APPEND for informado, o apontador da posição corrente é inicializado com o tamanho do arquivo.

Se o bit O_TRUNC estiver ligado, `open` ajusta o tamanho do arquivo para zero, liberando todos os blocos de dados. Finalmente, retorna um descritor de arquivos *fd* para o processo, através do qual o processo passa a referenciar o arquivo, nas chamadas de sistema subseqüentes. Caso aconteça algum erro, open retorna o valor '-1' no *fd*.

Quando o O_STABLE é informado, todas as operações de escrita subseqüentes realizam o armazenamento estável, usando a replicação do SALIUS. O_STABLE também afeta o comportamento da primitiva `open`, quando é associado aos *flags* O_CREAT ou O_TRUNC: o `open` só retorna para a aplicação, quando as alterações realizadas estiverem estáveis, isto é, gravadas em disco ou replicadas.

5.4.2 s_creat

A primitiva `s_creat` é a versão estável do `creat` original do UNIX, utilizada para criar um arquivo regular, ou rescrever sobre um existente.

Sintaxe: `fd = s_creat(pathname, mode)`

- *fd* é um descritor de arquivo, que retorna para a aplicação;
- *pathname* é o nome de caminho do arquivo;
- *mode* contém as permissões de acesso associadas ao arquivo.

Semântica: a primitiva cria um novo arquivo, com o nome indicado em *pathname*, e armazena as permissões de acesso, informadas em *mode*, no *nó-i* associado. Se o arquivo existe, `s_creat` trunca o seu tamanho para zero. Após ser criado, com sucesso, o arquivo é aberto exclusivamente para escrita, com a opção O_STABLE, ou seja, indicando que deve ser alterado de forma estável.

A primitiva `s_creat` retorna um descritor do arquivo em *fd*, para uso em outras primitivas, ou retorna o valor '-1', caso ocorra algum erro. A primitiva só retorna, após realizar a replicação, ou gravação em disco dos dados alterados: o superbloco, o *nó-i* do arquivo e do diretório pai e a entrada do arquivo no diretório pai.

5.4.3 write

A primitiva `write` é utilizada para escrever dados em um arquivo.

Sintaxe: `num = write(fd, address, nbytes)`

- *fd* é o descritor do arquivo;
- *address* é o endereço de uma estrutura, na área de memória da aplicação, contendo os dados que devem ser escritos no arquivo;
- *nbytes* indica o número de *bytes* que deverão ser escritos;
- *num* retorna com o número de *bytes* gravados.

Semântica: a primitiva `write` tenta gravar *nbytes* de um *buffer*, apontado por *adderess*, no arquivo associado ao descritor *fd*. Se a operação for bem-sucedida, retorna em *num* o número de *bytes* efetivamente gravados, normalmente igual ao valor de *nbytes*. De outro modo, *num* retorna com o valor '-1', indicando um erro. No retorno, `write` atualiza o apontador da posição corrente do arquivo, adicionando o número de *bytes* gravados.

A primitiva `write` original adota, como padrão, a *gravação assíncrona* de dados. Desse modo, quando `write` retorna, as modificações realizadas no arquivo ficam armazenadas na memória principal, em estruturas como a *buffer cache*, a tabela de *nós-i* e a cópia do superbloco. Posteriormente, essas mudanças são transferidas para o disco, de forma assíncrona. Portanto, o usuário não tem garantias relativas à estabilidade dessas informações, que podem ser apagadas por um *crash* do sistema.

Se o arquivo foi aberto com o bit O_SYNC ligado, `write` realiza a *gravação síncrona* de dados, ou seja, só retorna quando todos os dados e metadados do arquivo estiverem gravados em disco. Nesse caso, a aplicação tem a garantia de que as informações gravadas podem sobreviver a *crashes*.

Se o arquivo for aberto com o *flag* O_STABLE ligado, ou se for criado através da primitiva `s_creat`, `write` realiza o *armazenamento estável* de dados, ou seja, só retorna à aplicação, indicando sucesso, quando ocorre uma das seguintes situações:

- as modificações realizadas no sistema de arquivos estão armazenadas na memória principal e existem cópias dos dados e metadados alterados no Servidor de Replicação de Buffers. Posteriormente, essas mudanças serão transferidas para o disco, como na gravação assíncrona;
- todos os dados e metadados do arquivo, alterados pela operação de escrita, já estão gravados em disco.

5.4.4 s_mkdir

Primitiva utilizada para criar um novo diretório.

Sintaxe: `s_mkdir(pathname, mode)`

- *pathname* é o nome de caminho do novo diretório;
- *mode* contém as permissões de acesso ao novo diretório.

Semântica: a primitiva `s_mkdir` cria um novo diretório, com o nome *pathname*, com as permissões de acesso especificadas em *mode*. O novo diretório criado é vazio, com exceção das duas entradas, "." e "..", apontando, respectivamente, para o próprio diretório e para o diretório pai. A primitiva só retorna quando os dados alterados estiverem estáveis: replicados ou gravados em disco.

5.4.5 s_mknod

A primitiva `s_mknod` é utilizada para criar um arquivo regular, um *pipe* com nome, um diretório ou um arquivo especial.

Sintaxe: `s_mknod(pathname, type-mode, dev)`

- *pathname* é o nome de caminho do *nó-i* a ser criado;
- *type-mode* contém o tipo de arquivo e as permissões de acesso;
- *dev* especifica o maior e o menor número de dispositivo, para os arquivos especiais.

Semântica: a primitiva cria o arquivo especificado em *pathname* e armazena o tipo de arquivo e as permissões de acesso, informados em *type-mode*, no *nó-i* associado. Se o arquivo for especial, de caracter ou blocado, o menor e o maior número de dispositivo informados em *dev* também são armazenados no *nó-i*. A primitiva só retorna após a replicação, ou gravação em disco, dos dados alterados.

5.4.6 s_link

A primitiva `s_link` cria uma nova entrada de diretório para um arquivo existente.

Sintaxe: `s_link(pathname1, pathname2)`

- *pathname1* é o nome de caminho do arquivo existente;
- *pathname2* é o novo nome de caminho associado ao arquivo.

Semântica: a primitiva `s_link` associa o arquivo indicado por *pathname1* ao nome de caminho *pathname2*, na estrutura de diretórios do sistema de arquivos. Para isso, cria uma nova entrada para o *nó-i* do arquivo, no diretório indicado em *pathname2*. A primitiva só retorna à aplicação, quando todas as estruturas alteradas no sistema de arquivos estiverem replicadas, ou gravadas em disco.

5.4.7 s_unlink

A primitiva `s_unlink` é utilizada para remover uma entrada de diretório e/ou excluir um arquivo.

Sintaxe: **`s_unlink(pathname)`**

- *pathname* é o nome de caminho do arquivo existente.

Semântica: a primitiva `s_unlink` remove a entrada de diretório indicada em *pathname*. Quando a última entrada de diretório de um arquivo é removida, o arquivo é excluído do sistema. Para isso, `s_unlink` libera todos os blocos de dados do arquivo e o seu *nó-i*. A primitiva só retorna após a replicação, ou gravação em disco, das alterações realizadas.

5.4.8 s_chown e s_chmode

Permitem modificar, respectivamente, o proprietário e as permissões de acesso de um arquivo.

Sintaxe: **`s_chown(pathname, owner, group)`**
`s_chmode(pathname, mode)`

- *pathname* é o nome de caminho do arquivo;
- *owner* é o novo proprietário de arquivo;
- *group* é o grupo do novo proprietário;
- *mode* contém as permissões de acesso associadas ao arquivo.

Semântica: as primitivas `s_chown` e `s_chmode` alteram informações armazenadas no *nó-i* do arquivo indicado em *pathname*. Elas não afetam os blocos de dados do arquivo. A primitiva `s_chown` passa o direito de propriedade sobre o arquivo para o usuário *owner*, pertencente ao grupo indicado em *group*, enquanto `s_chmode` atribui as permissões de acesso, informadas em *mode*, ao arquivo. Ambas realizam a replicação do *nó-i* do arquivo.

A Tabela 5.2 relaciona as primitivas do SALIUS com as primitivas correspondentes do sistema de arquivos do UNIX e descreve, sucintamente, a função de cada primitiva.

PRIMITIVA NO UNIX	PRIMITIVA NO SERVIÇO	OPERAÇÃO
`open`	`Open`	Criação e/ou abertura de um arquivo.
`write`	`Write`	Escrita de dados em um arquivo.
`creat`	`s_creat`	Criação de um arquivo regular.
`mkdir`	`s_mkdir`	Criação de um diretório.
`mknod`	`s_mknod`	Criação de um arquivo de qualquer tipo: regular, diretório, especial de caracter, especial blocado ou *pipe*.
`link`	`s_link`	Criação de uma nova entrada de diretório para um arquivo existente.
`unlink`	`s_unlink`	Remoção de uma entrada de diretório de um arquivo existente.
`chown`	`s_chown`	Alteração do proprietário de um arquivo.
`chmode`	`s_chmode`	Alteração das permissões de acesso de um arquivo.

Tabela 5.2: Primitivas do SALIUS.

5.5 Servidor de Arquivos Complementar

O diagrama da Figura 5.2 mostra como ocorre a manipulação de arquivos no UNIX, com a adição do SALIUS. A figura mostra que o SALIUS oferece um serviço complementar, implementado por um Servidor de Arquivos Complementar (SAC), o qual é incorporado ao sistema de arquivos, assim como as primitivas do SALIUS são acrescentadas à interface do sistema. Se um programa de usuário quiser atualizar um arquivo, de forma estável, basta invocar uma primitiva do SALIUS.

O SAC executa uma operação solicitada, interagindo com a *buffer cache*, para alterar dados na memória, da mesma forma que o sistema de arquivos do UNIX. A seguir, o SAC interage com os *drivers* de rede para enviar uma mensagem ao SRB, solicitando a replicação dos dados alterados, e receber a confirmação da replicação. O programa de usuário permanece bloqueado, até que a replicação seja confirmada, ou até que o SAC termine de gravar todos dados e metadados alterados no disco, caso ocorra uma falha no serviço de replicação, interagindo diretamente com os *drivers* dos dispositivos de saída.

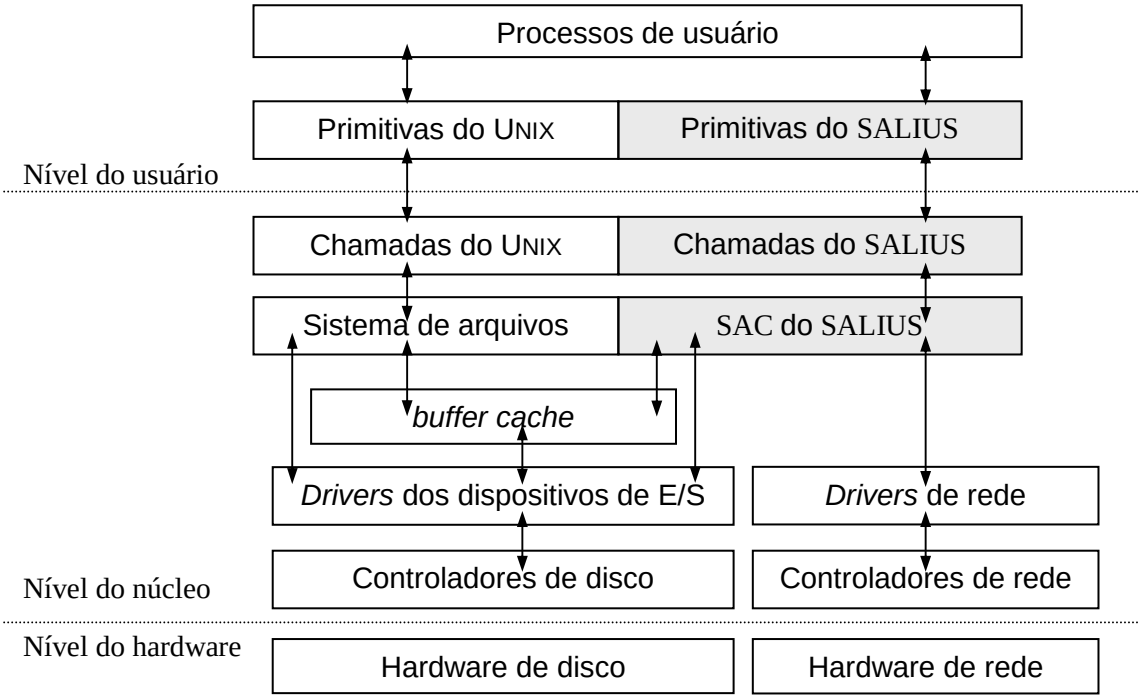

Figura 5.2: Diagrama da manipulação de arquivos no UNIX com SALIUS.

5.5.1 Informações Replicadas pelo SALIUS

Durante a execução de cada primitiva do SALIUS, o SAC replica todos os dados alterados, acrescidos de informações de controle, necessárias ao procedimento de recuperação e ao funcionamento do SRB. Por exemplo, o procedimento de recuperação precisa saber a ordem em que os dados foram gerados, para que possa gravá-los no disco nessa mesma ordem, simulando uma repetição das atualizações realizadas antes de um *crash*. O procedimento de recuperação também precisa saber a posição correta, onde esses dados devem ser gravados no disco. O SRB, por sua vez, precisa estar informado sobre os dados replicados que foram gravados no disco, na máquina do cliente, para que possa descartar a cópia desses dados do seu *log* e liberar espaço na memória principal. O conjunto de informações de controle depende da implementação do serviço.

É possível delimitar os dados e metadados do sistema de arquivos que cada primitiva do serviço pode vir a alterar. A Tabela 5.3 relaciona cada primitiva do serviço com o seu conjunto de dados alteráveis. Na primeira linha, estão agrupadas todas as primitivas do serviço. Na primeira coluna, estão relacionados todos os dados e metadados que podem ser alterados. num sistema de arquivos. Em cada uma das demais colunas, existe um '**X**' nas linhas que contêm dados ou metadados sujeitos a alterações pelas primitivas listadas no cabeçalho da coluna.

Dados Replicados	Primitivas do Serviço de Armazenamento Estável				
	`open` `s_creat`	`write`	`s_mkdir` `s_mknod`	`s_link` `s_unlink`	`s_chown` `s_chmod`
Superbloco	X	X	X	X	
***Nó-i* de arquivo**	X	X	X	X	X
Blocos de dados de arquivo		X	X		
***Nó-i* do diretório pai**	X		X	X	
Bloco de dados do dir. pai	X		X	X	

Tabela 5.3: Dados replicados pelas primitivas do SALIUS.

O superbloco pode ser alterado por quase todas as primitivas do serviço, porque ele contém um sumário de informações sobre o sistema de arquivos. As mudanças no estado do sistema, em geral, modificam essas informações. Por exemplo, qualquer alteração que implique na alocação ou na liberação de blocos de dados, ou de *nós-i*, também altera o superbloco, porque ele contém a lista de *nós-i* livres, o início da lista de blocos livres e contadores com o número de *nós-i* livres e o número de blocos livres.

O *nó-i* de um arquivo pode ser alterado por todas as primitivas do serviço: a adição e remoção de blocos de dados pode alterar a tabela de endereços armazenada no *nó-i*; as operações de criação de arquivos e diretórios alocam um *nó-i* para o novo arquivo; operações de criação e remoção de entradas de diretório alteram o contador de referências do *nó-i*; certos atributos do arquivo, armazenados no *nó-i*, podem ser explicitamente alterados, como o proprietário e as permissões de acesso.

Algumas primitivas alteram blocos de dados de um arquivo. A primitiva `write` sempre altera um ou mais blocos de dados. As primitivas `s_mknod` e `s_mkdir` também alteram blocos de dados, quando estão criando um diretório, pois gravam, no primeiro bloco de dados do diretório, as entradas contendo, respectivamente, o *nó-i* do próprio diretório e o *nó-i* do diretório pai.

Certas primitivas alteram dados do diretório pai de um arquivo, criando ou removendo entradas de diretório para o arquivo. São as primitivas utilizadas para criar um novo arquivo, como `s_creat`, `open`, `s_mkdir` e `s_mknod`, ou primitivas que manipulam diretamente as entradas de diretório, como `s_link` e `s_unlink`. Muitas vezes, essas alterações exigem que novos blocos de dados sejam alocados para o diretório pai, que aumenta de tamanho, alterando, conseqüentemente, o *nó-i* desse diretório.

Nem sempre uma primitiva altera todos os dados contidos no conjunto a ela associado. A primitiva `write`, por exemplo, sempre altera blocos de dados. Porém, somente em algumas situações um `write` precisa alterar o superbloco. A seguir, serão apresentados os dados alterados pelas primitivas `s_creat`, `open` e `write`. O Apêndice A contém uma análise dos dados alterados por todas as primitivas do SALIUS.

5.5.1.1 Informações replicadas pela primitiva s_creat

A primitiva `s_creat` é utilizada para criar um arquivo. Ela aloca um *nó-i* para o novo arquivo. Para isso, retira um *nó-i* da lista de *nós-i* livres. Conseqüentemente, a primitiva altera o superbloco, onde a lista de *nós-i* livres está armazenada e onde existe um contador de *nós-i* livres, que é decrementado quando o *nó-i* é alocado.

A primitiva também cria uma entrada para o *nó-i* do arquivo no diretório pai, alterando um bloco de dados desse diretório. Se o tamanho do diretório pai aumentar, a primitiva ajusta o tamanho de arquivo, armazenado no *nó-i* desse diretório. Se o último bloco de dados do diretório pai estiver cheio, a primitiva `s_creat` deve alocar um novo bloco para armazenar a entrada do arquivo, retirando um bloco da lista de blocos livres e alterando o contador de blocos livres, no superbloco. A primitiva `s_creat` precisa replicar todas as estruturas alteradas durante a sua execução, o que pode consistir de: o superbloco, o *nó-i* do novo arquivo, o *nó-i* do diretório pai e o bloco de dados do diretório pai, com a entrada do novo arquivo.

5.5.1.2 Informações replicadas pela primitiva open

A primitiva `open` abre um arquivo existente e pode criar um novo arquivo, caso o *flag* O_CREAT seja informado. No caso da criação de um arquivo, os dados que devem ser replicados são os mesmos da primitiva `s_creat`.

Quando um arquivo existente é aberto, normalmente, nenhum dado é alterado, com exceção da situação em que o *flag* O_TRUNC é informado. Nesse caso, todos os blocos do arquivo são liberados e a primitiva altera o *nó-i* do arquivo, ajustando o tamanho do arquivo para zero. A operação de truncamento do arquivo também afeta o superbloco, pois os blocos liberados são inseridos na lista de blocos livres e o tamanho dessa lista é incrementado. Assim, a primitiva `open`, quando invocada com a opção O_TRUNC, deve replicar o *nó-i* do arquivo e o superbloco.

5.5.1.3 Informações replicadas pela primitiva write

A primitiva `write` é utilizada para escrever dados em um arquivo. Quando uma aplicação invoca essa primitiva, ela passa, como parâmetro de entrada, um *buffer* de usuário contendo os dados a serem alterados. A primitiva deve replicar os dados informados, bem como a posição no arquivo, onde deve iniciar a gravação desses dados. Essa posição é lida na tabela de arquivos abertos.

Se o arquivo tiver sido previamente aberto com a opção O_APPEND, significa que os dados serão acrescidos ao final do arquivo, aumentando o seu tamanho. Nesse caso, a primitiva ajusta o tamanho do arquivo, no seu respectivo *nó-i*. Quando o tamanho de um arquivo aumenta, novos blocos precisam ser alocados, alterando a lista de blocos livres e o superbloco. Desse modo, a primitiva `write` pode precisar replicar também o superbloco e o *nó-i* do arquivo.

5.6 Procedimento de Recuperação

Essa seção apresenta a especificação do último componente do SALIUS: um procedimento de recuperação de *crash*. O serviço considera a ocorrência de um *crash* toda vez que as estruturas de dados do sistema de arquivos, residentes em memória, são perdidas. A recuperação de *crash* é o ato de restaurar o sistema de arquivos em disco para um estado consistente, de modo que preserve as garantias oferecidas pelo serviço às aplicações: o SALIUS assegura a estabilidade de todos os dados armazenados através de suas primitivas.

De forma genérica, os mecanismos de recuperação podem ser divididos em duas grandes classes: *recuperação para frente* e *recuperação para trás*. Quando ocorre um *crash*, os efeitos de algumas operações podem estar apenas parcialmente refletidos no sistema de arquivos, deixando o mesmo num estado inconsistente. Um procedimento de recuperação tem a função de levar o sistema de arquivos para um estado consistente. Se esse estado for anterior à ocorrência do *crash*, diz-se que o sistema se recuperou para trás. Para isso, os efeitos das operações parcialmente registradas são desfeitos.

Na recuperação para frente, pelo contrário, as operações parcialmente registradas são finalizadas. Os efeitos dessas operações são inteiramente refletidos no sistema de arquivos, levando-o a um estado consistente posterior ao *crash*. A recuperação para frente é conseguida através de um mecanismo de “repetição” das operações executadas até o momento imediatamente anterior ao *crash*: partindo-se do pressuposto de que, inicialmente, o sistema de arquivos está consistente e guardando-se os resultados de todas as operações realizadas durante o funcionamento normal do sistema, para colocar o disco num estado consistente, após um *crash*, basta gravar os resultados das operações no disco, exatamente na ordem em que foram gerados, simulando uma repetição das operações.

O mecanismo de recuperação do SALIUS se enquadra na classe de recuperação para frente: todos os dados e metadados atualizados no sistema de arquivos, que são perdidos em decorrência de um *crash*, permanecem guardados no Servidor de Replicação de Buffers; o mecanismo de recuperação obtém esses dados e providencia a gravá-los no disco, na mesma ordem em que foram gerados. Assim, o mecanismo de recuperação do SALIUS garante a durabilidade dos efeitos de operações realizadas antes do *crash*.

A seguir, serão apresentados, em detalhes, os possíveis estados das estruturas de dados do sistema de arquivos, na ocasião de um *crash*, e as soluções asseguradas pelo mecanismo de recuperação do SALIUS.

5.6.1 Efeitos do Procedimento de Recuperação

As situações descritas neste item ilustram os efeitos causados pelo procedimento de recuperação do SALIUS, sobre as estruturas de dados do sistema de arquivos . Suponha a ocorrência de um *crash* no tempo **t1**, com o sistema de arquivos no estado representado pela Figura 5.3. O *crash* provoca a destruição de todo o conteúdo do sistema de arquivos mantido na memória e ainda não gravado em disco.

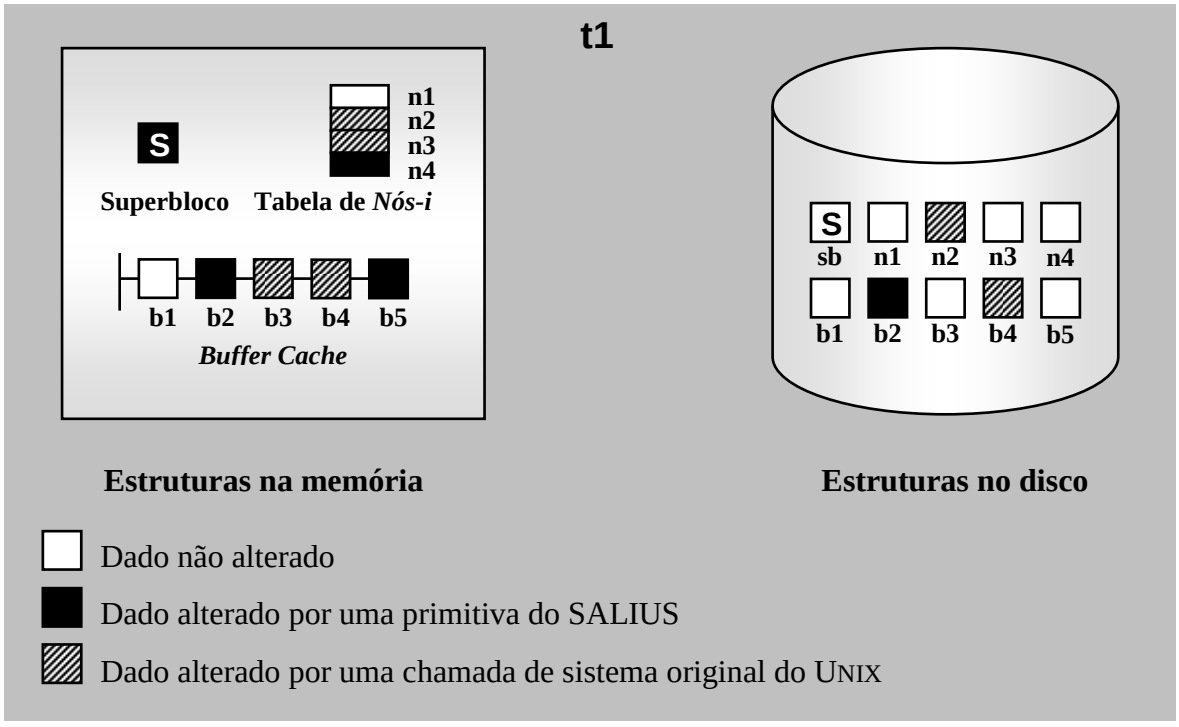

Figura 5.3: Estado de um sistema de arquivos na ocasião de um *crash*.

Nas figuras desta seção, constam os componentes de um sistema de arquivos UNIX, em disco: o superbloco, os *nós-i* e os blocos de dados. O sistema realiza *caching* de tais componentes, com o objetivo de diminuir a quantidade de acessos a disco. Na memória, esses dados ficam armazenados em estruturas auxiliares: o superbloco é copiado para um superbloco auxiliar, o conteúdo dos blocos de dados é carregado na *buffer cache*, enquanto os *nós-i* são carregados na tabela de *nós-i*. Todas as figuras foram configuradas de forma didática: na estrutura real de um sistema de arquivos em disco, normalmente, existe um número bem maior de *nós-i* e de blocos de dados, que não precisam estar armazenados contiguamente, nem de forma ordenada.

As figuras exibem possíveis estados dos dados mantidos em memória: os componentes em branco estão com o mesmo conteúdo lido do disco, significando que não sofreram alterações; os componentes hachurados foram alterados, através de chamadas de sistema originais do sistema de arquivos do UNIX; os componentes em preto foram atualizados por primitivas do SALIUS. Na Figura 5.3, por exemplo, alguns dados modificados já foram gravados em disco, tornando-se, portanto, estáveis: *nó-i* n2 e blocos b2 e b4. Porém, a maioria das alterações foi realizada apenas na memória.

Na hipótese de não acontecer o *crash* e do sistema de arquivos permanecer inacessível para atualizações, durante um certo período, existirá um tempo **t2**, quando todos os dados alterados na memória estarão estáveis. Isso porque os sistemas de arquivos do UNIX gravam todos os dados no disco, periodicamente. O período entre as gravações pode variar de um sistema de arquivos para outro, mas, em geral, não ultrapassa os trinta segundos.

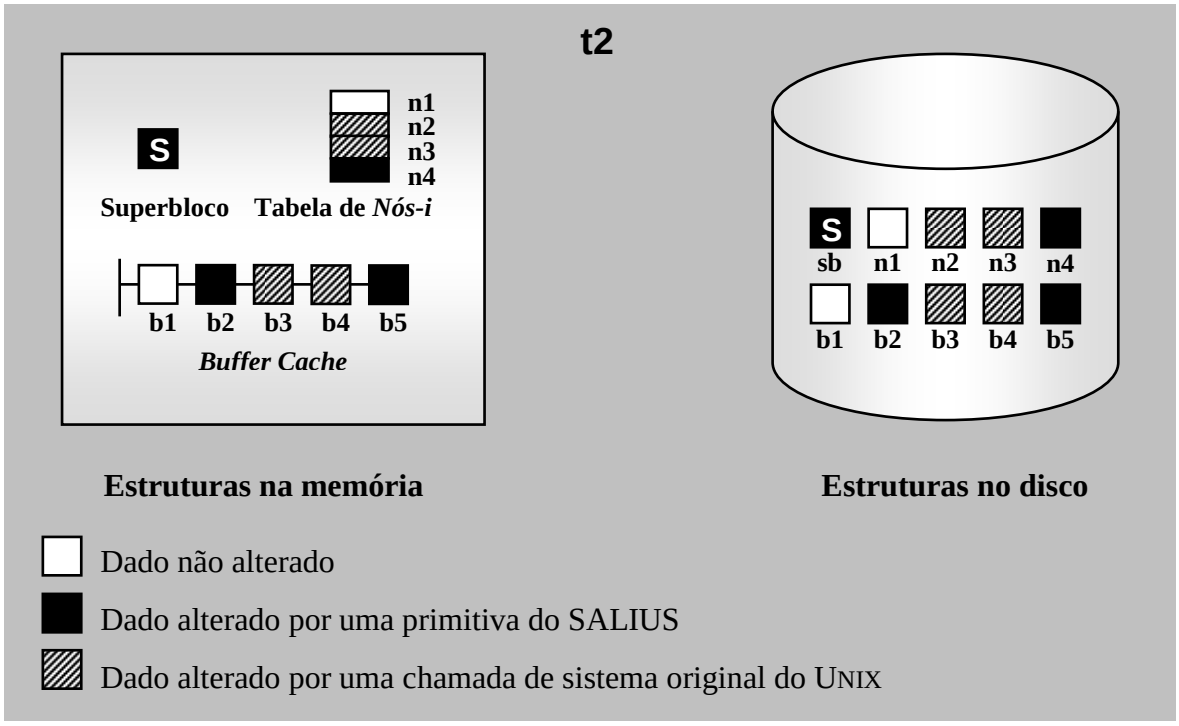

Figura 5.4: Sistema de arquivos da Figura 5.3, após gravação dos dados.

A Figura 5.4 ilustra o estado do sistema de arquivos da Figura 5.3, no tempo **t2**, com todas as alterações já gravadas em disco. Assim, o conteúdo dos componentes de dados na memória é idêntico aos seus respectivos conteúdos no disco.

Na hipótese do *crash* realmente acontecer em **t1**, o SALIUS deve garantir que, num tempo **t3**, após a execução do procedimento de recuperação, o sistema de arquivos contenha todos os dados gravados em disco, antes do *crash*, acrescidos dos dados alterados, em memória, pelas primitivas do serviço.

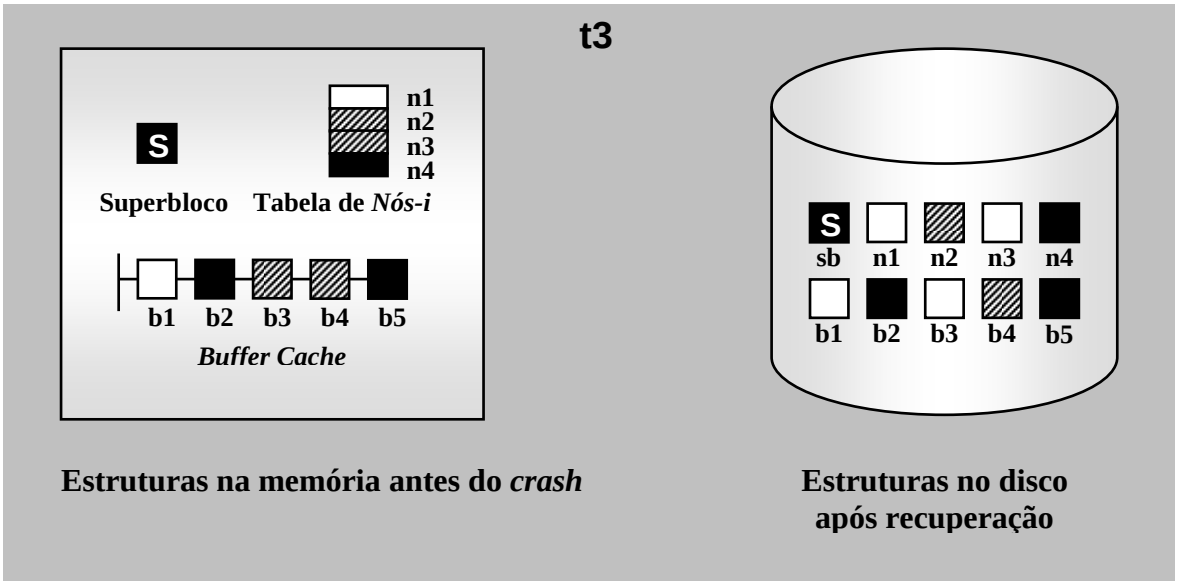

Figura 5.5: Sistema de arquivos da Figura 5.3, após a recuperação.

O sistema de arquivos, que se encontrava no estado apresentado pela Figura 5.3, antes do *crash*, é levado para o estado apresentado na Figura 5.5, pelo mecanismo de recuperação do SALIUS. Note-se que o *nó-i* n2 e os blocos de dados b2 e b4 já se encontravam estáveis na ocasião do *crash*. O procedimento de recuperação do serviço restaura, a partir de uma réplica, os dados em preto, ou seja, armazenados pelas primitivas do serviço: o superbloco, o *nó-i* n4 e os blocos b2 e b5. É importante ressaltar que os dados armazenados pelas primitivas originais do sistema de arquivos do UNIX, e ainda não gravados em disco, são perdidos com o *crash*: o *nó-i* n3 e o bloco b3.

As Figuras 5.3, 5.4 e 5.5, apresentadas, ilustram a idéia básica que norteia o mecanismo de recuperação do SALIUS. Na prática, entretanto, existem situações mais complexas. Durante a permanência de um item de dado na memória, ele pode sofrer várias alterações consecutivas, antes de ser gravado em disco. Algumas alterações podem envolver apenas parte do conteúdo de um item.

As figuras seguintes expõem uma situação mais abrangente, onde aparecem itens cujos conteúdos foram apenas parcialmente alterados, ora por primitivas do SALIUS, ora por chamadas de sistema originais do sistema de arquivos do UNIX.

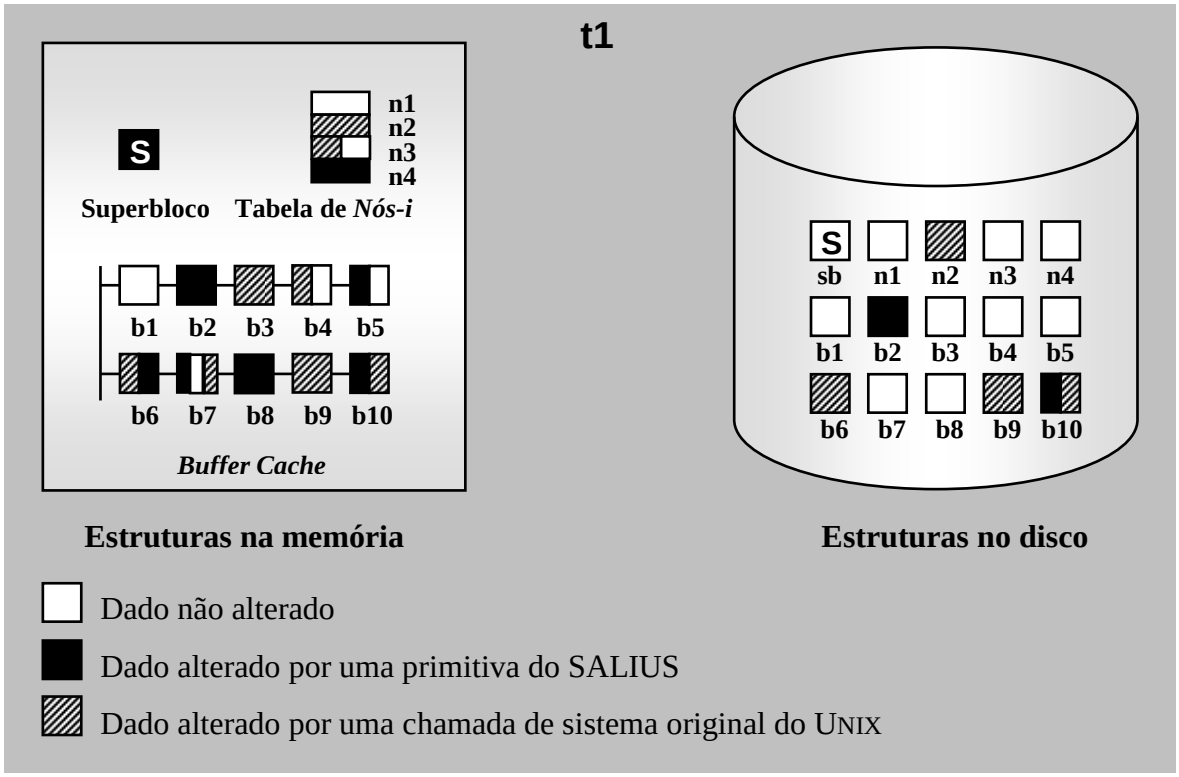

Figura 5.6: Estado de um sistema de arquivos na ocasião de um *crash*.

A Figura 5.6 apresenta um sistema de arquivos na ocasião de um *crash*. Os dados exibidos estão nos seguintes estados:

- o *nó-i* n1 e o bloco b1 foram carregados na memória e permanecem inalterados;
- o superbloco, o *nó-i* n4 e o bloco de dados b8 foram alterados através de primitivas do SALIUS e ainda não foram gravados no disco;
- o bloco b2 foi atualizado por uma primitiva do SALIUS e já foi gravado em disco;
- parte do conteúdo do *nó-i* n3 e o bloco b3 foram atualizados por chamadas de sistema originais do sistema de arquivos do UNIX e essas alterações ainda não foram propagadas para o disco;
- o *nó-i* n2 e o bloco b9 foram atualizados por chamadas de sistema originais do sistema de arquivos do UNIX e já foram gravados em disco;
- apenas uma parte do conteúdo do bloco b4 foi atualizada por uma chamada de sistema original do UNIX. A alteração ainda não foi propagada para o disco;

- apenas uma parte do conteúdo do bloco b5 foi atualizada por uma primitiva do SALIUS. A alteração ainda não foi propagada para o disco;
- o bloco b6 sofreu duas alterações consecutivas. Primeiro, o seu conteúdo foi atualizado por uma chamada de sistema do UNIX e o bloco foi gravado em disco. A seguir, apenas uma parte do bloco foi alterada pelo SALIUS e os efeitos dessa operação ainda não foram refletidos no disco;
- o bloco b7 pode ser dividido em três partes. Uma porção inicial foi atualizada pelo SALIUS. A segunda parte do bloco permaneceu inalterada, enquanto o restante do conteúdo do bloco foi modificado por uma chamada de sistema do UNIX. As alterações ainda não foram propagadas para o disco;
- uma parte do bloco b10 foi atualizada por uma chamada de sistema do UNIX e outra parte, pelo SALIUS. As alterações já foram gravadas em disco.

A Figura 5.7 ilustra a situação do sistema de arquivos apresentado pela Figura 5.6, num estado completamente estável. Esse estado pode ser conseguido num tempo **t2**, após a gravação de todos os dados no disco, caso o sistema permaneça operacional e não sofra alterações no período entre **t1** e **t2**. Nesse caso, todas as alterações são refletidas no sistema de arquivos em disco.

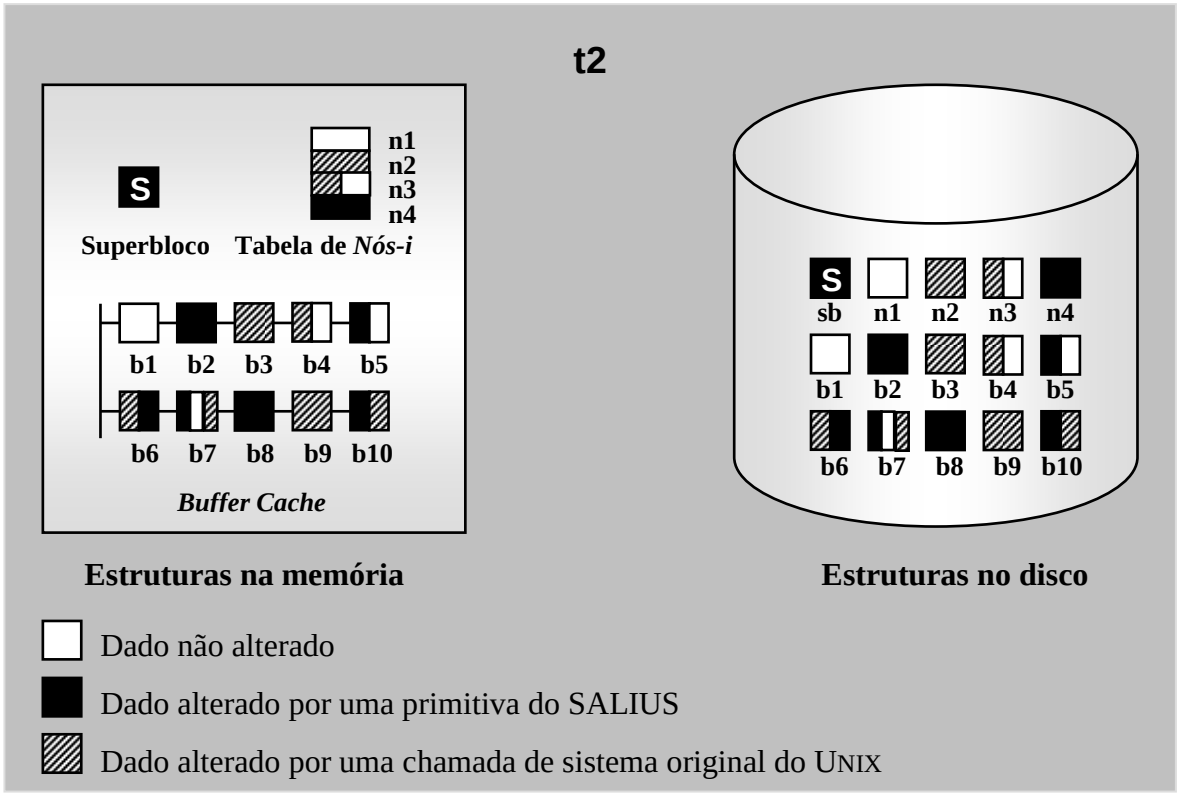

Figura 5.7: Sistema de arquivos da Figura 5.6, após gravação dos dados.

A Figura 5.8 apresenta a situação do sistema de arquivos da Figura 5.6, no tempo **t3**, após a recuperação de um *crash*, ocorrido no tempo **t1**. Note-se que apenas os dados armazenados pelas primitivas do SALIUS foram recuperados.

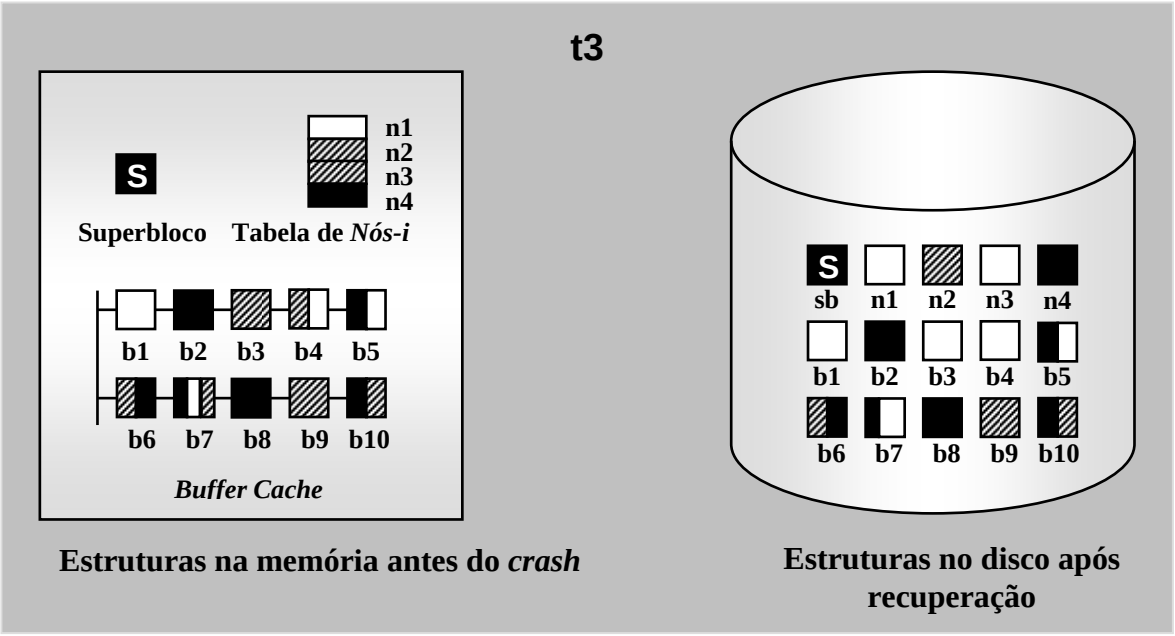

Figura 5.8: Sistema de arquivos da Figura 5.6, após a recuperação.

5.6.2 Restauração do Sistema de Arquivos

O procedimento de recuperação do SALIUS interage com o SRB, solicitando os dados replicados antes do *crash*. Ao obter os dados replicados, o procedimento de recuperação providencia gravá-los no disco, exatamente na ordem em que foram gerados. As mensagens de replicação contêm informações de controle, indicando a ordem em que as atualizações do sistema de arquivos aconteceram. Por exemplo, a implementação do SALIUS pode englobar um *contador de pedidos de replicação*, único para cada sistema de arquivos. Assim, toda vez que o SAC executa uma operação de atualização, o contador é incrementado e o número do pedido é enviado na mensagem de replicação, para o SRB. Quando o procedimento de recuperação obtém as mensagens de replicação de volta, ele ordena os dados recuperados pelo número do pedido, antes de iniciar a gravação em disco.

Como o procedimento de recuperação realiza as atualizações na mesma ordem na qual elas foram originalmente realizadas, a operação de recuperação é *idempotente*. Assim, caso ocorra um *crash* durante a recuperação, seja na réplica utilizada para fornecer os dados da recuperação, seja na máquina onde o sistema de arquivos reside, o procedimento pode ser repetido: mesmo que a recuperação já tenha sido parcialmente realizada, o sistema de arquivos sempre chega ao mesmo estado final.

A Figura 5.9 exibe o mesmo sistema de arquivos da Figura 5.3, no momento da ocorrência de um *crash*, incluindo as informações já replicadas. O *crash* provoca a destruição de todos os dados armazenados na memória principal. Alguns dados já se encontram gravados em disco: o *nó-i* n2 e os blocos b2 e b4. Algumas estruturas foram replicadas: o superbloco, o *nó-i* n4 e os blocos b2 e b5.

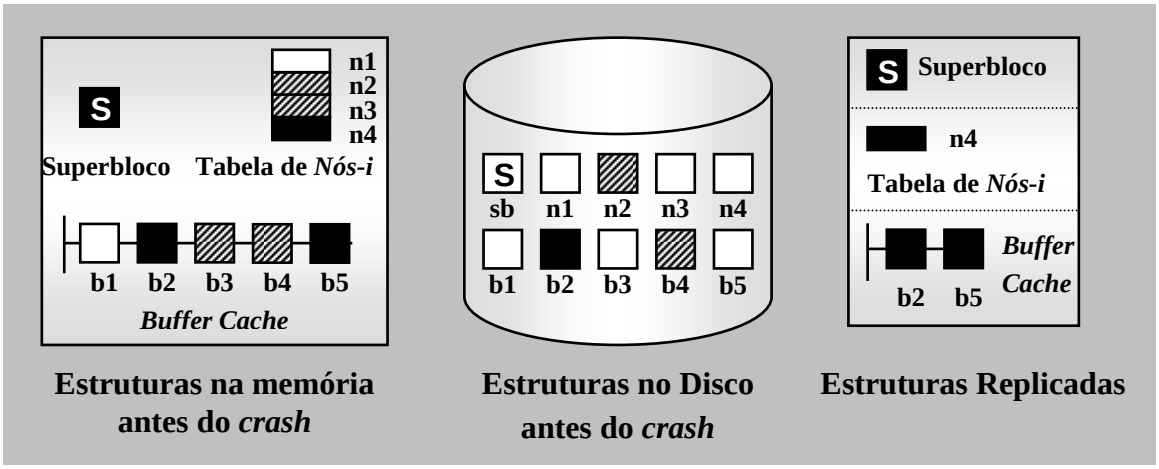

Figura 5.9: Sistema de arquivos antes do *crash*.

A Figura 5.10 mostra como o procedimento de recuperação do SALIUS restaura o sistema de arquivos: lê os dados armazenados na réplica de recuperação e, a seguir, grava os dados recuperados no disco.

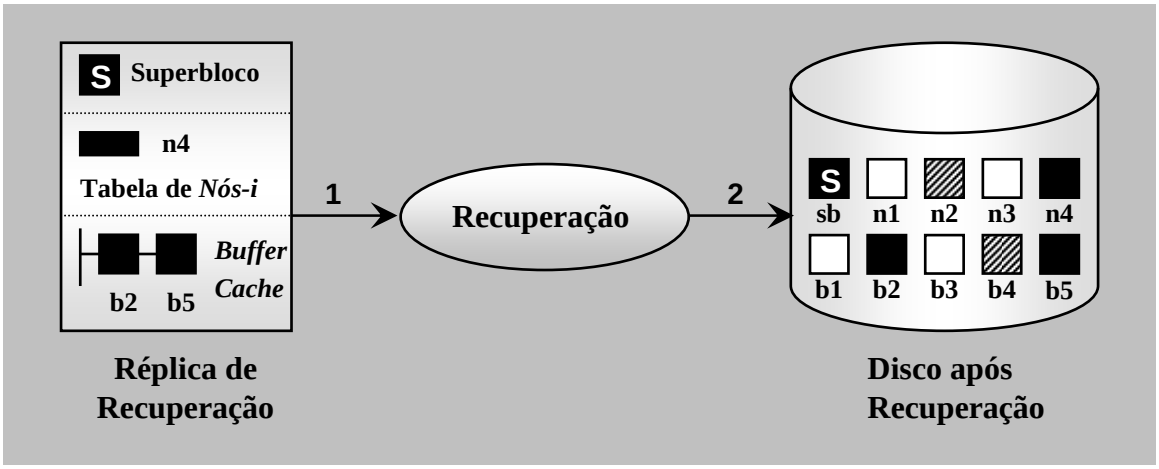

Figura 5.10: Restauração do sistema de arquivos da Figura 5.9.

5.6.3 Situação Atípica

Existe uma situação especial, que merece ser considerada pelo mecanismo de recuperação. Tal situação está ilustrada na Figura 5.11, que foi dividida em cinco partes, cada uma delas apresentando um momento diferente do sistema de arquivos. No primeiro momento **t1**, um bloco de dados foi atualizado e replicado por uma primitiva do SALIUS. No tempo **t2**, uma parte do conteúdo do bloco foi atualizada por uma chamada de sistema original do UNIX (parte hachurada). Em **t3**, o bloco foi gravado no disco e a cópia replicada tornou-se desatualizada. Em **t4**, ocorreu um *crash* e o conteúdo da memória foi perdido. Em **t5**, o procedimento de recuperação do SALIUS foi ativado e gravou no disco o bloco obsoleto, enviado pelo Servidor de Replicação de Buffers, sobrepondo uma atualização mais recente. A situação retratada é rara e pode ser evitada, se a implementação do Servidor de Replicação de Buffers puder impedir a manutenção de blocos obsoletos.

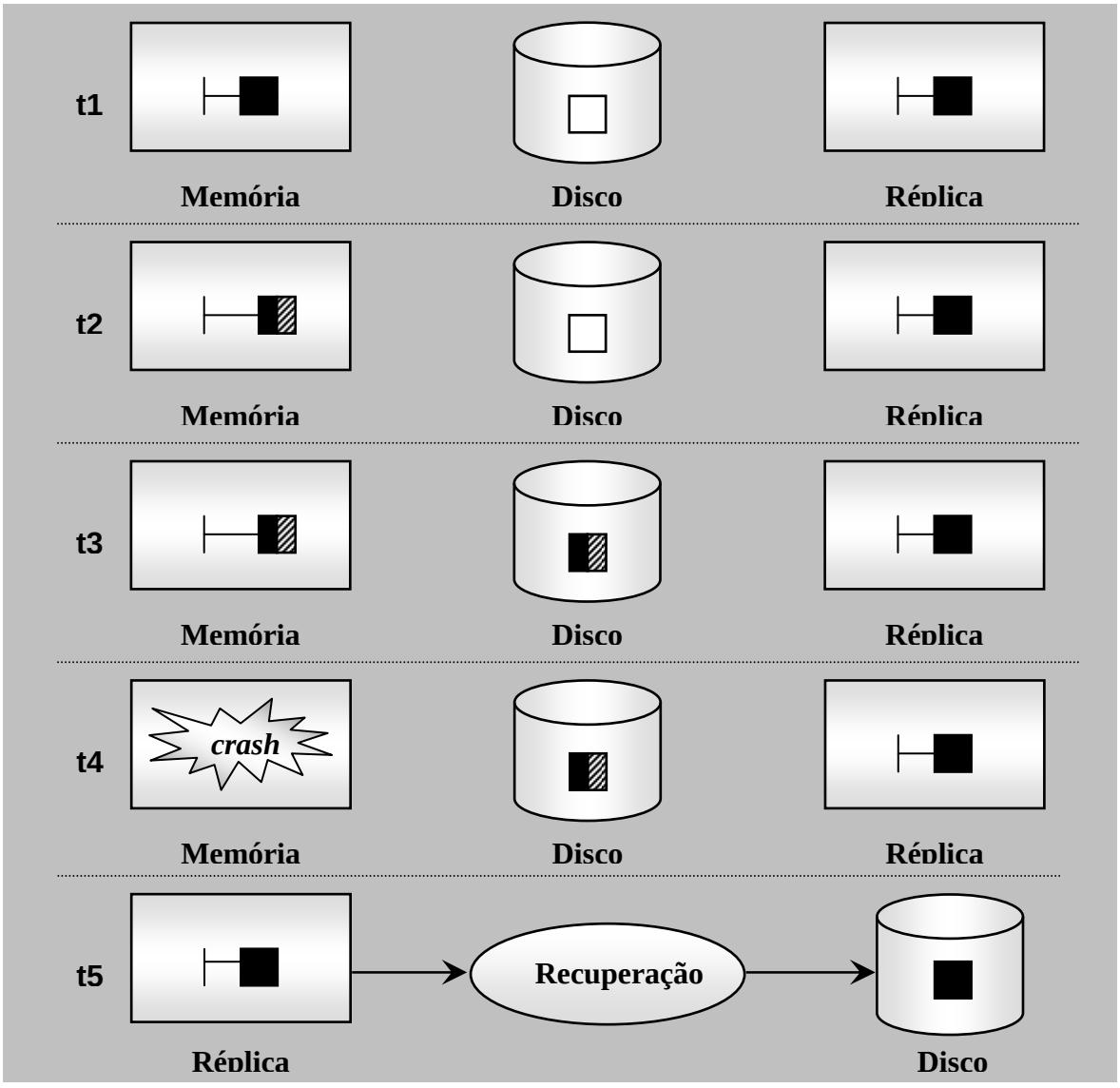

Figura 5.11: Situação especial na recuperação de um *crash*.

5.6.4 Recuperação de Dados Gravados por Primitivas do UNIX

O mecanismo de recuperação do SALIUS assegura, única e exclusivamente, a recuperação de dados armazenados através de primitivas do serviço. Ele não trata os dados armazenados pelas primitivas tradicionais do sistema de arquivos. Para esse subconjunto de dados, devem ser utilizados os procedimentos de recuperação oferecidos pelo próprio sistema de arquivos. Como foi descrito com detalhes, no Capítulo 2, o UNIX utiliza um programa de recuperação, denominado *fsck*, para remover inconsistências do sistema de arquivos. O *fsck* continua sendo necessário, para o subconjunto de dados armazenado com as primitivas originais do UNIX.

5.6.5 Desempenho do Mecanismo de Recuperação do SALIUS

Para o subconjunto de dados armazenados pelo SALIUS, a recuperação de *crash* é potencialmente muito rápida. Não é necessário realizar verificações na estrutura de metadados do sistema, porque todos os metadados afetados numa determinada operação são automaticamente replicados, junto com os dados alterados. Se a implementação do Servidor de Replicação de Buffers puder impedir a situação atípica relatada em 5.6.3, então, também será capaz de fornecer sempre a versão mais atualizada dos dados e metadados do sistema de arquivos. Nesse caso, a recuperação resume-se em obter os dados replicados antes do *crash* e gravá-los no disco.

5.7 Considerações sobre Desempenho e Confiabilidade

Esta seção tem como objetivo demonstrar que o desempenho e a estabilidade, providos pelo SALIUS, dependem fundamentalmente da tecnologia utilizada e da implementação do Serviço de Replicação de Buffers.

5.7.1 Influências da Tecnologia

Todo o funcionamento do SALIUS está diretamente vinculado à tecnologia de rede e à tecnologia de memória. Os principais fatores da tecnologia que influenciam no desempenho e na capacidade de prover estabilidade de dados do SALIUS são:

- **A velocidade de acesso à memória principal.** Como todas as operações providas pelo serviço atualizam dados na memória principal e a replicação transfere dados da memória de uma máquina para outra, quanto maior a velocidade de acesso à memória, melhor será o desempenho do sistema.

- **A confiabilidade da memória principal.** Quanto mais confiável o armazenamento em memória, menores as chances de um dado da memória ser destruído, aumentando a probabilidade de manter as informações estáveis.
- **A quantidade de memória principal disponível.** Quanto maior a capacidade da memória principal das máquinas do sistema, maior a quantidade de dados que podem ser mantidos na *cache* e replicados. A falta de memória pode tornar a replicação impossível, levando o SALIUS a realizar gravações síncronas e degradando, assim, o desempenho do sistema. Com uma grande disponibilidade de memória, o SALIUS pode realizar a replicação com êxito. Adicionalmente, pode-se aumentar o intervalo de tempo no qual os dados alterados permanecem na *cache*, contribuindo para diminuir a quantidade de operações de escrita em disco e melhorando o desempenho do sistema.
- **A taxa de transmissão de dados através da rede.** Quanto maior a taxa de transmissão de dados na rede, mais rápida será a replicação, diminuindo o tempo de resposta das aplicações que usam o serviço e aumentando o desempenho do sistema.
- **A confiabilidade do meio de transmissão de dados.** Quanto mais confiável for o meio de transmissão de dados, menores as chances de mensagens perdidas ou corrompidas, aumentando a probabilidade de manter as informações estáveis. A replicação torna-se mais rápida, porque diminui a quantidade de retransmissões de mensagens, aumentando o desempenho. Adicionalmente, crescem as chances de sucesso da replicação, diminuindo a necessidade de gravações síncronas, o que também contribui para um melhor desempenho do sistema.
- **A banda passante da rede.** Quanto maior a banda passante da rede utilizada pelo serviço, menor a probabilidade do tráfego gerado causar um congestionamento. Um congestionamento pode causar uma falha de tempo no Serviço de Replicação de Buffers, levando o SALIUS a realizar gravações síncronas. Por isso, aumentando a banda passante da rede, crescem as chances de sucesso das operações de replicação, contribuindo para um melhor desempenho do sistema.

É importante frisar que o tráfego na rede é um fator crucial ao desempenho do SALIUS. Toda a operação do SALIUS envolve a troca constante de mensagens através da rede. Um alto tráfego na rede, pode tornar-se um gargalo ao desempenho do serviço, ou chegar a inviabilizar o seu funcionamento.

Portanto, quanto melhor a estrutura de rede e a memória principal utilizadas pelo sistema, melhor o desempenho e a confiabilidade do SALIUS. Quando os componentes de hardware utilizados possibilitam o funcionamento pleno do SALIUS, com a realização bem-sucedida da replicação, a gravação dos dados em disco é realizada de forma assíncrona. Enquanto o sistema está gravando *buffers* da *cache*, a aplicação continua o seu processamento e o SALIUS atende a outras requisições.

O tempo de processamento de uma primitiva do SALIUS depende do tempo de replicação dos dados: a aplicação não espera pela gravação em disco, a menos que ocorra uma falha no Serviço de Replicação de Buffers. Por isso, quando todos os componentes do SALIUS estão funcionando corretamente, a influência do tempo de acesso a disco no desempenho global do sistema é minimizada. O desempenho do sistema passa a depender muito mais da tecnologia de memória e da tecnologia de rede.

Adicionalmente, quanto melhor a estrutura de rede e a tecnologia de memória utilizadas, maiores as chances de sucesso da replicação e maior a probabilidade de garantir a estabilidade das informações, aumentando a confiabilidade do SALIUS.

5.7.2 Influências do Serviço de Replicação de Buffers

O SALIUS não depende apenas da tecnologia para conseguir alcançar os objetivos estabelecidos. O sucesso do SALIUS está subordinado a uma implementação eficiente dos seus componentes, especialmente do Serviço de Replicação de Buffers, o seu componente vital. O processamento de qualquer primitiva do SALIUS utiliza o Serviço de Replicação de Buffers. Ao executar uma operação solicitada, o SAC espera receber uma mensagem de confirmação da replicação, ou detectar uma falha no SRB, antes de retornar o controle ao cliente. Portanto, quanto mais rápida a replicação, melhor o desempenho do sistema. A rapidez de uma operação de replicação depende da forma como o Serviço de Replicação de Buffers está implementado.

Além do desempenho, a confiabilidade do SALIUS também depende da implementação do Serviço de Replicação de Buffers. Quando uma mensagem de confirmação de uma replicação retorna, o SAC assume que os dados já estão sendo mantidos de forma confiável pelo SRB. Por isso, o SAC retorna o controle para a aplicação, garantindo a estabilidade das informações. Porém, a probabilidade do SALIUS conseguir manter, realmente, a estabilidade dos dados armazenados, deriva da probabilidade do Serviço de Replicação de Buffers manter estável uma informação, para a qual uma confirmação foi enviada ao SAC.

Portanto, a implementação do Serviço de Replicação de Buffers influencia tanto no desempenho, quanto na confiabilidade do SALIUS. Normalmente, estes dois requisitos são conflitantes, ou seja, priorizar um implica no detrimento do outro. Por exemplo, uma solução trivial seria não existir um SRB executando. Assim, o cliente nunca receberia uma mensagem de confirmação e realizaria sempre a gravação síncrona das informações. A “vantagem” seria a alta confiabilidade do serviço, ou seja, nunca poderia existir uma mensagem de confirmação de replicação errada. Deste modo, toda vez que o SALIUS confirmasse o sucesso de uma operação para um programa de usuário, os dados alterados estariam, de fato, gravados em disco. Porém, essa solução não atende a um dos principais objetivos do serviço, que é justamente melhorar o seu desempenho em relação às soluções tradicionais, baseadas em escritas síncronas.

Uma segunda implementação bastante simples seria um SRB executando em uma única máquina da rede local, ligada a um *"no-break"*. Nesse caso, o desempenho seria priorizado: a replicação poderia ser muito rápida, pois o cliente só estaria enviando dados e recebendo a confirmação de um único SRB. No entanto, a confiabilidade do serviço seria sacrificada, pelo fato do SRB não ser tolerante a *crashes* no sistema operacional.

Uma variação da segunda solução apresentada seria a implementação de um SRB executando numa única máquina, ligada ao servidor de arquivos por um barramento de alta velocidade. Neste caso, o desempenho seria ainda maior, devido à comunicação veloz provida pelo barramento. Entretanto, a confiabilidade oferecida seria limitada, pelo fato de haver apenas uma réplica de dados.

A implementação de um Serviço de Replicação de Buffers deve balancear, da melhor forma possível, o atendimento aos requisitos de desempenho e confiabilidade. No próximo capítulo, será apresentado um projeto para o Serviço de Replicação de Buffers, que busca um equilíbrio no atendimento a esses requisitos.

Capítulo 6

6 Projeto de um Serviço de Replicação de Buffers

Este capítulo apresenta um projeto de um Serviço de Replicação de Buffers para o SALIUS. Inicialmente, são especificados os requisitos do serviço. A seguir, o projeto é detalhado, com a descrição da utilização de um grupo de réplicas na implementação do serviço e a apresentação do modelo de replicação adotado.

As réplicas que implementam o SRB interagem entre si, e com os clientes, de acordo com um protocolo de replicação. Em tais interações, a comunicação também obedece a um protocolo. Este capítulo descreve o protocolo de comunicação utilizado e detalha as operações e os algoritmos do protocolo de replicação do Serviço de Replicação de Buffers.

6.1 Requisitos do Serviço de Replicação de Buffers

O Serviço de Replicação de Buffers do SALIUS precisa atender a dois requisitos, normalmente conflitantes:

a) enviar a mensagem de confirmação de um pedido de replicação, o mais rapidamente possível, e

b) ter uma alta probabilidade de manter estável uma informação, para a qual uma mensagem de confirmação foi enviada a um cliente.

Para conseguir atender aos requisitos de desempenho, o Serviço de Replicação de Buffers deve utilizar a estrutura de uma rede local: quanto maior a velocidade da rede, melhor será o desempenho do serviço. A garantia da estabilidade das informações armazenadas depende do grau de tolerância a faltas do serviço: a implementação de um serviço, através de um grupo de réplicas, é uma forma de prover tolerância a faltas.

6.2 Grupo de Réplicas

O Serviço de Replicação de Buffers é implementado através de um processo, denominado Servidor de Replicação de Buffers (SRB), executando em várias máquinas do sistema distribuído. Em cada máquina, o SRB realiza o mesmo processamento: recebe e armazena os dados replicados na execução das primitivas do SALIUS e interage com o mecanismo de recuperação, para fornecer os dados necessários à restauração da consistência do sistema de arquivos. Como o serviço oferecido é exatamente o mesmo, em todas as máquinas, os processos servidores são denominados réplicas uns dos outros.

Uma maneira de gerenciar réplicas é considerar cada réplica individual como um membro de um *grupo de réplicas*. A funcionalidade geral de grupos, em sistemas distribuídos, está descrita no projeto ANSA[1] [ANS90] [ANS91] e em artigos técnicos [LCN90] [OOW91]. Também existem previsões de que o serviço de replicação será adicionado às especificações CORBA[2] [OMG95] [Bir96].

As réplicas do Servidor de Replicação de Buffers são agrupadas por várias razões:

- como uma forma de abstrair as características comuns dos membros do grupo e do serviço de replicação que eles oferecem;
- para encapsular os estados internos e as interações entre os membros do grupo, de forma a prover o usuário de uma interface uniforme, a interface de grupo, contribuindo, assim, para que o serviço atenda ao requisito de transparência;
- com o intuito de delimitar as máquinas que armazenam cópias dos dados de um cliente, racionalizando o uso de recursos do sistema.

Portanto, o Serviço de Replicação de Buffers do SALIUS é provido por um grupo de réplicas. Os clientes desse serviço são de dois tipos: um Servidor de Arquivos Complementar (SAC), que requisita a replicação de informações, e um Procedimento de Recuperação, que requisita as informações necessárias para restaurar o sistema de arquivos a um estado consistente, mantendo estáveis as informações armazenadas através de primitivas do SALIUS.

---

[1] ANSA (Advanced Network Systems Architecture), um projeto britânico, liderado por Andrew Herbert, que constituiu a primeira tentativa sistemática de desenvolver tecnologia para modelar sistemas distribuídos complexos. Consiste de um conjunto de modelos que tratam de vários aspectos de projeto de sistemas distribuídos, com um enfoque de orientação a objetos.

[2] CORBA (Common Object Request Broker Architecture), um projeto posterior ao ANSA, iniciado por um consórcio de fabricantes de computadores, que tem como objetivo definir padrões avançados para conseguir a interoperabilidade de sistemas distribuídos complexos, orientados a objetos, construídos por fabricantes diferentes. CORBA define padrões para um grande conjunto de serviços. CORBA é basicamente um molde para construção de sistemas distribuídos e tornou-se uma tendência.

6.2.1 A Composição do Grupo de Réplicas

O número de réplicas participantes do grupo é definido de acordo com o grau de tolerância a faltas desejado e da semântica de falha do serviço. Quando um sistema adota uma semântica de falha por *crash*, o grupo deve ser formado de **k** réplicas, sendo $\mathbf{k \geq f + 1}$, para que seja possível tolerar **f** faltas [Cri91]. No SALIUS, o cliente também mantém uma cópia dos dados alterados no sistema de arquivos. Por isso, uma réplica é mantida no cliente, enquanto as demais **n** réplicas compõem o grupo do SRB ($\mathbf{n = k - 1 \Rightarrow n \geq f}$ **)**. Assim, se **f** máquinas sofrerem *crashes* simultâneos, restará, pelo menos, uma máquina operacional contendo a cópia dos dados alterados que devem permanecer estáveis.

A composição ideal do grupo de réplicas voltará a ser discutida na seção 6.7, quando serão considerados alguns detalhes da implementação.

Cada pedido de replicação do SAC precisa ser confirmado por **r** réplicas, sendo **r = f**, para que a replicação seja considerada bem-sucedida: se o SAC não receber mensagens de, pelo menos, **r** réplicas distintas, confirmando o recebimento de um pedido de replicação, num tempo máximo predefinido, então o SAC assume que o SRB falhou. Sendo **n** o número de réplicas, **f** o número de faltas tolerado e **r** o número de réplicas que respondem a um pedido, o SRB pode falhar em duas situações distintas:

a) quando $\mathbf{n - freal < f}$, sendo **freal** o número real de replicas que sofreram *crash*; ou

b) quando ocorre um particionamento (real ou virtual) entre o cliente e o grupo de réplicas, ou seja, quando o serviço de comunicação falha, temporariamente, impedindo que o SAC envie pedidos e/ou receba mensagens de confirmação de **r** réplicas.

6.2.2 A Consistência das Réplicas

O Serviço de Replicação de Buffers precisa manter a consistência das réplicas operacionais, isto é, todas as réplicas que respondem aos pedidos de replicação devem permanecer atualizadas, para que seja possível assegurar a estabilidade das informações armazenadas através do SALIUS.

Quando $\mathbf{n > f}$, mesmo que todas as operações de replicação sejam realizadas com sucesso, algumas réplicas podem ficar desatualizadas: uma falha do serviço de comunicação pode impedir que algumas das réplicas recebam um pedido de replicação, muito embora outras **r** réplicas do grupo consigam atender ao pedido, enviando mensagens de confirmação ao cliente. Nesse caso, o SAC recebe **r** mensagens de confirmação para um pedido de replicação, mas algumas réplicas do grupo não contêm os dados daquele pedido.

Se cada pedido de replicação for recebido por um conjunto diferente de réplicas, após um *crash* no cliente, o grupo de réplicas pode estar em situações que dificultam, ou que impossibilitam a recuperação do sistema de arquivos, tais como:

a) nenhuma réplica contém todos os pedidos de replicação, enviados antes do *crash*, ou

b) alguns pedidos foram irremediavelmente perdidos, devido a falhas anteriores, justamente nas únicas réplicas que armazenavam aquele pedido.

Na situação **a**, ainda é possível recuperar o sistema de arquivos: basta que o procedimento de recuperação faça um *merge* das informações armazenadas nas diversas réplicas. Contudo, na situação **b**, a recuperação de todas as informações armazenadas antes do *crash* é impossível.

O serviço precisa garantir que, no mínimo, **r** réplicas, sendo **r = f**, permanecem sempre atualizadas: caso ocorram **f** *crashes* simultâneos, nas máquinas do sistema, envolvendo, inclusive, a máquina que executa o sistema de arquivos, restará, pelo menos, uma réplica com todas as informações necessárias para manter a estabilidade dos dados armazenados antes do *crash*.

Todo o funcionamento de um grupo de réplicas é ordenado através de um protocolo de replicação, inclusive as ações realizadas para manter a consistência das réplicas. Este protocolo especifica como serão as interações entre os clientes e os grupos, e entre os membros que compõem um grupo. O protocolo de replicação depende do modelo de replicação e do protocolo de comunicação utilizados, que serão apresentados a seguir.

6.3 Modelo de Replicação

Existem duas formas básicas de realizar replicação: replicação passiva e replicação ativa [Lit92]. Ambas podem ser usadas para mascarar falhas por *crash* em sistemas distribuídos. No entanto, cada classe de replicação tem uma maneira peculiar de tratar uma réplica individual e um grupo de réplicas.

6.3.1 Replicação Passiva

Na replicação passiva, ilustrada pela Figura 6.1, só existe um membro ativo no grupo de réplicas, denominado *membro primário*, que recebe (fluxo1) e executa as requisições de clientes. Durante o processamento de uma requisição, o membro primário realiza *checkpoints* periódicos, ou seja, propaga as mudanças no seu estado para os membros passivos, preservando a consistência entre os membros do grupo (fluxo 2). Quando o processamento de uma requisição termina, o primário realiza um último *checkpoint* e envia uma resposta ao cliente (fluxo3).

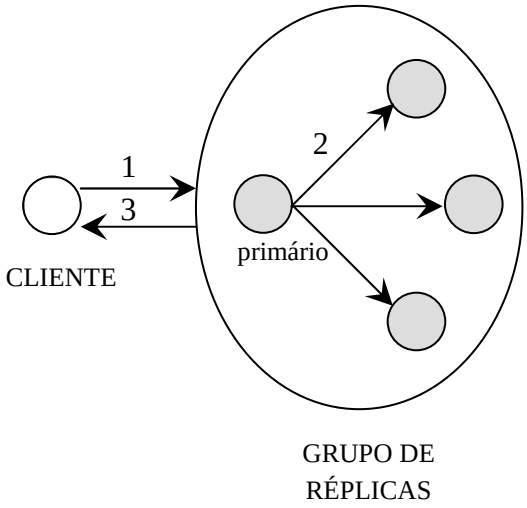

Figura 6.1: Replicação Passiva.

As falhas temporais do membro primário são interpretadas como um *crash*, ou seja, se o primário demorar para responder a um cliente, ou para realizar um *checkpoint*, o cliente e/ou os membros passivos do grupo vão deduzir, por decurso de tempo, que o primário parou de funcionar. Se o primário falha, os membros remanescentes elegem um novo primário, que assume como o único membro ativo do grupo. Durante a eleição, o grupo torna-se indisponível.

6.3.2 Replicação Ativa

No modelo de replicação ativa, mostrado na Figura 6.2, todos os membros do grupo de réplicas recebem, executam e respondem cada requisição de cliente. As réplicas devem ser *deterministas*, ou seja, dado um mesmo estado inicial e um mesmo conjunto de mensagens, recebidas numa mesma ordem, todas as réplicas operacionais devem atingir um mesmo estado final.

A *Máquina de Estado* foi descrita em Sch90 como um modelo de referência para implementar serviços tolerantes a faltas, usando replicação ativa, e coordenar as interações entre clientes e grupos. Uma máquina de estado consiste de variáveis de estado, que armazenam seu estado, e comandos, usados para transformar seu estado. Cada comando é implementado por um programa determinista, cuja execução pode modificar uma variável de estado e produzir alguma saída. A máquina de estado garante o processamento de todas as requisições de um mesmo cliente, na ordem em que foram emitidas, e a ordenação total de requisições de clientes diferentes [Lam78].

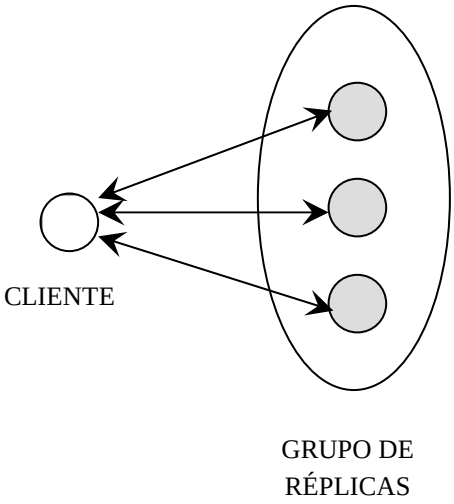

Figura 6.2: Replicação Ativa.

Esse modelo define as condições que devem ser satisfeitas, a fim de garantir a consistência das réplicas:

a) cada membro operacional do grupo deve receber toda requisição de cliente e

b) todos os membros operacionais devem processar as requisições recebidas numa mesma ordem relativa.

6.3.3 Modelo de Replicação do SALIUS

A análise das características de cada modelo levou à escolha de um modelo baseado na replicação ativa, para o projeto do Serviço de Replicação de Buffers do SALIUS.

- Na replicação passiva, o protocolo de replicação precisa garantir que o membro primário propague seu estado para um número suficiente de membros passivos, antes de enviar uma resposta ao cliente, como uma condição indispensável para oferecer um certo grau de tolerância a faltas. Essa operação pode aumentar consideravelmente o tempo de resposta ao cliente, constituindo um fator impeditivo ao atendimento dos requisitos de desempenho do SALIUS.
- Na replicação ativa, a falha de uma réplica não interfere no comportamento das demais e o grupo continua disponível, enquanto na replicação passiva, o serviço fica interrompido, desde que um primário falha, até que um novo primário seja eleito.
- A escolha da replicação ativa, no caso específico do SALIUS, não aumenta expressivamente a quantidade de recursos utilizados.

- O processamento nas réplicas do Servidor de Replicação de Buffers é mínimo: apenas o recebimento da mensagem e atualização de estruturas de dados, na memória, portanto pode ser realizado em todas as réplicas.
- O número de réplicas necessárias, em geral, é pequeno, porque a probabilidade de ocorrerem *crashes* simultâneos, em diversas máquinas do sistema, também é normalmente baixa.
- O protocolo de comunicação utilizado pelo Serviço de Replicação de Buffers pode ser simplificado, mesmo adotando o modelo de replicação ativa. Por exemplo, o protocolo de comunicação não precisa garantir a entrega ordenada de mensagens: nas réplicas, as mensagens que chegam podem ser armazenadas de forma desordenada; a ordem é importante apenas para o procedimento de recuperação, que precisa gravar os dados alterados na mesma ordem em que foram gerados. Como o sistema de arquivos do UNIX é centralizado, as mensagens de replicação são geradas por processos executando numa mesma máquina; as mensagens podem incluir informações que possibilitem uma ordenação posterior, quando um cliente iniciar um procedimento de recuperação.

6.4 Protocolo de Comunicação

O protocolo de comunicação define como será a comunicação entre um cliente e o grupo de réplicas, e entre os membros do grupo. A comunicação ponto a ponto, também conhecida como *unicasting*, acontece quando um transmissor envia uma mensagem para um único receptor [Tan95].

O Serviço de Replicação de Buffers utiliza *unicasting* em algumas de suas interações, como, por exemplo, quando uma réplica envia dados para o procedimento de recuperação do cliente. Porém, na maioria das interações, o serviço utiliza um tipo mais sofisticado de comunicação, denominada multiponto, envolvendo um transmissor e vários receptores: um cliente do serviço de replicação envia mensagens para todo o grupo de réplicas, pois o modelo adotado é de replicação ativa; eventualmente, uma réplica pode precisar interagir com as demais réplicas do grupo, a fim de restabelecer a sua consistência.

Existem várias maneiras de implementar a comunicação multiponto. O transmissor pode, por exemplo, fazer múltiplos *unicasts*, sendo um para cada receptor. Nesse caso, o transmissor deve conhecer todos os membros do grupo, para que possa endereçar as mensagens enviadas. Para cada interação entre um processo transmissor e um grupo de **n** réplicas, circulam **n** mensagens, contribuindo para aumentar o tráfego na rede.

Multicasting é uma técnica mais eficiente de implementar a comunicação multiponto [ANS90] [Lit92]. Um *multicast* é uma mensagem enviada a um grupo e encaminhada, através da rede, para todos os membros do grupo. Em cada interação entre um processo transmissor e um grupo, trafega apenas uma mensagem na rede. *Multicasting* é uma forma transparente de comunicação grupal, pois o transmissor não precisa saber quantos são os membros de um grupo, nem onde eles estão localizados [Tan95].

O esquema de *multicasting* pode ser implementado de várias maneiras e em várias camadas diferentes (enlace, rede, transporte). Em algumas redes, por exemplo, é possível criar um endereço de rede especial, para *multicasting*, como o endereço de Internet de classe D, através do qual várias máquinas podem escutar. Assim, se o transmissor enviar uma mensagem para um endereço de *multicasting*, a mensagem será entregue a todas as máquinas que estiverem escutando nesse endereço [Tan95]. É possível, então, assinalar todos os membros de um grupo de réplicas a um mesmo endereço de *multicasting*.

A escolha de um protocolo de comunicação deve considerar as garantias oferecidas, referentes à entrega de mensagens, e o impacto dessas garantias no desempenho.

- Quanto menos garantias o protocolo de comunicação oferece, menor a carga de processamento introduzida no sistema, contribuindo para um melhor desempenho. Em contraposição, o protocolo de replicação precisa ser mais complexo, para a manutenção da consistência das réplicas.
- Quanto mais garantias o protocolo de comunicação oferece, mais simples pode ser o protocolo de replicação. Porém, maior também é a carga de processamento introduzida pelo protocolo de comunicação, diminuindo o desempenho do sistema.

O Serviço de Replicação de Buffers adota o protocolo de comunicação UDP *Multicast*. Esse protocolo adiciona a facilidade de realizar *multicasting* ao protocolo UDP (*User Datagram Protocol)* [Bir96]. O UDP *Multicast* não garante a entrega confiável de mensagens a todos os membros do grupo, ou seja, uma máquina pode receber uma mensagem que outras perderam. Esse protocolo também não garante a entrega ordenada das mensagens. Além disso, pode ocorrer duplicação de mensagens.

Na maioria dos serviços implementados através de replicação ativa, é necessário garantir a entrega atômica, ordenada e não duplicada de mensagens, para a manutenção da consistência das réplicas. Por isso, a utilização do protocolo UDP *Multicast* nesses sistemas introduziria uma grande complexidade ao protocolo de replicação, a fim de suprir os requisitos de comunicação não atendidos pelo protocolo de comunicação.

Entretanto, O UDP *Multicast* é um protocolo adequado ao Serviço de Replicação de Buffers do SALIUS, porque:

- apesar das limitações citadas, o protocolo UDP *Multicast* tem a vantagem de ser simples e adicionar uma carga de processamento menor ao sistema, se comparado com os protocolos de *multicasting* confiável (que garantem a entrega atômica, ordenada e não duplicada de mensagens), possibilitando uma entrega mais rápida de mensagens;
- o Serviço de Replicação de Buffers está projetado para ser utilizado numa estrutura de rede local, com uma probabilidade pequena de perder pacotes. Assim, o projeto assume uma abordagem otimista, considerando que as mensagens serão entregues, na maioria das vezes;
- as mensagens trafegadas pelo serviço são pequenas (aproximadamente, o equivalente a um bloco do arquivo). Portanto, para a maioria dos sistemas de arquivos UNIX, com tamanho de bloco de 4Kb, cada mensagem no SALIUS pode equivaler a uma única mensagem UDP, cujo tamanho máximo é tipicamente de 8Kb[Bir96].

O Serviço de Replicação de Buffers do SALIUS consegue contornar as limitações do UDP *Multicast*, através de um protocolo de replicação simples, que será descrito a seguir.

6.5 Protocolo de Replicação

Um protocolo de replicação define as possíveis interações entre um cliente e um grupo de réplicas e entre os membros do grupo. A especificação do protocolo de replicação do SALIUS assume que:

- o protocolo de comunicação possui uma semântica não-confiável (não garante a entrega atômica, ordenada e não duplicada de mensagens);
- em todas as máquinas do sistema, só ocorrem falhas por *crash*;
- o sistema é assíncrono, isto é, não existe um tempo máximo para uma mensagem chegar ao seu destino, mas é possível estimar um de tempo de transmissão, dentro do qual a maior parte das mensagens é entregue;
- quando uma réplica não responde a uma requisição, dentro de um intervalo de tempo máximo predefinido, o cliente interpreta que essa réplica falhou.

6.5.1 Informações de Controle de um Pedido de Replicação

Cada pedido de replicação que um cliente envia ao grupo de réplicas contém dados e metadados alterados no sistema de arquivos e um cabeçalho com as seguintes informações de controle adicionais:

6.5.1.1 Informações Relacionadas com a Entrega das Mensagens

São informações utilizadas em ações previstas no protocolo de replicação, com o objetivo de compensar as limitações do protocolo de comunicação UDP *Multicast*. Por exemplo, as mensagens de replicação precisam conter um identificador de pedido, único e global, para que uma réplica consiga reconhecer e descartar os pedidos duplicados. Além disso, a identificação de um pedido deve indicar a ordem na qual ele foi gerado, para que uma réplica consiga detectar que deixou de receber algum pedido, iniciando um procedimento de regeneração, junto às demais réplicas, para recuperar as mensagens perdidas.

Como o sistema de arquivos do UNIX é centralizado, é possível implementar um contador de pedidos, único para cada sistema de arquivos e armazenado numa região crítica da memória. Assim, antes de enviar uma mensagem de replicação, um cliente obtém um número de pedido e monta o identificador do pedido formado da associação das seguintes informações: um identificador da máquina onde o sistema de arquivos reside, o identificador do sistema de arquivos e o número do pedido.

6.5.1.2 Informações para a Limpeza dos *Logs* de Replicação

As réplicas precisam descartar todos os dados replicados, que tenham sido gravados no sistema de arquivos original. Um pedido de replicação deve conter o número de pedidos anteriores, cujos dados tenham sido gravados em disco.

6.5.1.3 Informações para o Procedimento de recuperação

O identificador de pedido também é necessário ao procedimento de recuperação: como o número do pedido é obtido durante a execução de uma operação de atualização e é único para cada sistema de arquivos, esse número reflete a ordem na qual as atualizações foram originalmente realizadas; então, o procedimento de recuperação utiliza o identificador de pedido, para ordenar os dados recuperados, antes de gravá-los no disco, simulando uma repetição das operações de atualização.

Além do identificador de pedido, a mensagem de replicação deve incluir outras informações de controle, que indicam a posição, dentro do sistema de arquivos, onde os dados devem ser gravados. Um pedido de replicação pode conter diferentes tipos de dados, tais como: bloco de dados, superbloco e *nó-i*. Um mesmo pedido pode conter mais de um item de dado. Para cada item de um pedido, é preciso informar: o tipo e o tamanho da informação e sua posição no sistema de arquivos.

6.5.2 Operações do Protocolo de Replicação

O protocolo de replicação permite que as seguintes operações sejam realizadas:

- ***send_replica (p, flag_init, i_srb)***: utilizada por um cliente, para enviar um pedido de replicação para o grupo de réplicas. O parâmetro *p* é o identificador do pedido; o parâmetro *flag_init* é um *flag* indicando que o SRB precisa ser inicializado; o parâmetro *i_srb* informa a encarnação atual do SRB;
- ***receive_request (p, flag_init, i_srb)***: utilizada por um membro do grupo de réplicas, para receber uma requisição de um cliente. Existem dois tipos diferentes de requisições: um pedido de replicação enviado por um SAC; ou um pedido de recuperação enviado por um processo de recuperação do SALIUS, ou por uma réplica em regeneração. O tipo de pedido é indicado no cabeçalho da mensagem e orienta o servidor quanto ao processamento que deve ser executado;
- ***send_reply (p, i)***: utilizada por um membro do grupo de réplicas, para enviar uma mensagem ao cliente, confirmando a realização de um pedido de replicação *(p)* e informando a encarnação atual da réplica *(i)*;
- ***receive_reply (p, i)***: utilizada por um cliente, para receber uma resposta de um membro do grupo de réplicas, confirmando a realização de um pedido de replicação e informando a encarnação da réplica em *(i)*;
- ***init_recovery***: utilizada por um cliente, ao retornar de um *crash*, para informar ao grupo de réplicas o início do procedimento de recuperação. Também utilizada por uma réplica, após um *crash*, ou ao perceber que se encontra desatualizada, para informar aos demais membros do grupo o início de um procedimento de regeneração;
- ***send_status (p, i)***: utilizada por um membro do grupo de réplicas, para informar o último pedido de replicação atendido *(p)* e a sua encarnação corrente *(i);*
- ***receive_status (p, i)***: utilizada por um cliente, ou uma réplica em regeneração, para receber o estado de atualização de uma réplica, ou seja, a encarnação e o último pedido de replicação atendido;
- ***request_data_recovery (p)***: utilizada por um cliente, ou por uma réplica em regeneração, para requisitar a outra réplica do grupo o envio de dados do *log* de replicação, referentes a pedidos maiores que um pedido informado *(p)*;
- ***send_data_recovery (l)***: utilizada por um membro do grupo de réplicas, para enviar a um cliente, ou a uma réplica em regeneração, uma mensagem contendo os dados *log* de replicação *(l)* previamente solicitados;
- ***receive_data_recovery (l)***: utilizada por um cliente em recuperação, ou por uma réplica em regeneração, para receber uma mensagem contendo dados do *log* de replicação *(l)*.

6.5.3 Algoritmo de Replicação

O algoritmo de replicação define o comportamento de um cliente e de um grupo de réplicas, para realizar a replicação durante o processamento das primitivas do SALIUS. Após realizar as alterações de dados e metadados de um sistema de arquivos, na memória principal, o cliente (SAC) tenta replicar todos os dados alterados, antes de retornar o controle para a aplicação. Para isso, o cliente envia um pedido de replicação ao grupo de réplicas do SRB (uma mensagem *multicast* para todas as réplicas).

Além de dados e metadados alterados, uma mensagem de replicação inclui um cabeçalho com informações de controle. Para montar o cabeçalho da mensagem, o cliente adquire um número de pedido, gerado através de um contador de pedidos, armazenado numa região crítica da memória. Esse número de pedido é associado com um identificador do sistema de arquivos e um identificador da máquina onde o sistema de arquivos reside, para compor o identificador do pedido.

O cabeçalho de uma mensagem de replicação também contém um *flag (flag_init)*, cuja função é informar ao grupo de réplicas se o SRB precisa ser inicializado. A inicialização do SRB faz parte do tratamento de falhas, discutido no item 5.3.3. O cliente guarda o estado do SRB numa variável de estado (*srb_inicializado)*, para cada sistema de arquivos. Toda vez que um cliente vai replicar dados, ele consulta essa variável: se o SRB não estiver inicializado, o cliente ativa o *flag_init*, para ordenar uma inicialização ao grupo de réplicas. Caso contrário, o cliente envia o pedido de replicação com o *flag_init* desligado. A operação que monta um sistema de arquivos torna a variável *srb_inicializado* falsa, forçando uma inicialização, ao primeiro pedido de replicação de um sistema de arquivos.

O cliente também inclui o número da encarnação corrente do SRB (*i_srb*) no cabeçalho do pedido. O esquema de encarnações é utilizado para que uma réplica do SRB possa verificar se deixou de receber algum pedido, no qual o cliente solicita uma inicialização ao grupo de réplicas. Se uma réplica parar de receber pedidos de clientes por um certo tempo, devido a falhas no serviço de comunicação, ela pode deixar realizar uma inicialização. Quando a réplica volta a receber pedidos, ela verifica se a sua encarnação está atualizada.

O esquema de encarnações funciona da seguinte maneira: toda vez que uma réplica passa por um processo de inicialização, ela incrementa a sua encarnação; quando uma réplica envia uma mensagem de confirmação a um cliente, ela informa a sua encarnação atual; ao receber as mensagens de confirmação, um cliente guarda o maior número de encarnação recebido numa variável de estado (*i_srb*); todo pedido de replicação contém o valor armazenado na variével *i_srb*. Assim, quando uma réplica recebe um pedido de replicação, ela compara a sua encarnação com a encarnação corrente do SRB: se a encarnação da réplica for anterior à encarnação do SRB, ela detecta que deixou de fazer alguma inicialização e entra num processo de regeneração, mantendo-se atualizada.

Quando termina de montar a mensagem, o cliente tenta realizar a replicação. Para isso, envia o pedido ao grupo de réplicas e espera pela confirmação da replicação, durante um intervalo de tempo predefinido. Se, nesse tempo, chegarem mensagens de confirmação de, pelo menos, **r** réplicas distintas do servidor**,** todas da encarnação mais recente, o cliente assume que a replicação foi bem-sucedida. Caso contrário, o cliente retransmite o pedido e aguarda novamente por respostas. Isso acontece um número finito de vezes (*max_retry*, configurado pelo administrador do sistema), antes do cliente interpretar uma falha no serviço de replicação.

Se a replicação for bem-sucedida, o cliente ajusta a encarnação corrente do SRB, guardando na variável *i_srb* o maior número de encarnação recebido nas mensagens de confirmação. Se o SRB não estava inicializado, o cliente também atualiza a variável *srb_inicializado*, indicando que o SRB foi inicializado com sucesso. A seguir, o cliente retorna o controle para a aplicação, informando o sucesso da operação solicitada.

Quando um cliente detecta uma falha no SRB, durante uma tentativa de replicar dados, suas ações dependem do estado do SRB, indicado pela variável *srb_inicializado*:

a) se o SRB estiver inicializado, significa que essa é a primeira tentativa de replicar dados que falha, desde a última inicialização: o cliente torna falsa a variável *srb_inicializado* e executa o comando *sync*, gravando no disco todas as atualizações pendentes do sistema de arquivos, inclusive os dados da operação em curso;

b) se o SRB não estiver inicializado, significa que o comando *sync* já foi executado anteriormente, não havendo atualizações pendentes na memória: o cliente realiza a gravação síncrona apenas dos dados e metadados alterados pela operação em curso.

CLIENTE:

```
altera dados em memória;
adquire contador (p);                   /* adquire o contador de pedidos */
incrementa contador (p);                /* incrementa o contador de pedidos */

if  (srb_inicializado) then             /* preenche flag de inicialização do SRB */
        flag_init = false;
else    flag_init = true;

retry = 0;                              /* inicia o contador de tentativas */
do {
        retry = retry + 1;
        send_replica (p, flag_init, i_srb);
        start_timeout;                  /* inicia a contagem do tempo */
        num_replies = 0;                /* inicia o número de respostas da tentativa */
        i_max = 0;
        do {
                receive_reply (p, i);
                if (i > i_max) then {
                        i_max = i;
                        num_replies = 1;        /* reinicia a contagem de respostas */
                } else  incrementa num_replies;
        } while not (timeout) and (num_replies < r);
} while (num_replies < r) and (retry < max_retry);

if (num_replies ≥ r)  then {                    /* sucesso na replicação */
        libera contador (p);
        if not (srb_inicializado) then {
                srb_inicializado = true;
                i_srb = i_max;                  /* guarda a encarnação do SRB */
        }
} else {                                        /* SRB falhou */
        decrementa contador (p);
        libera contador (p);
        if (srb_inicializado) then {
                srb_inicializado = false;
                sync ();
        } else  gravação síncrona;
}
return ok;
```

Ao receber um pedido de um cliente, o Servidor de Replicação de Buffers, primeiro, verifica se esse pedido já foi atendido, descartando as mensagens duplicadas. Se for um novo pedido, o servidor cria um processo filho para processar o pedido e volta a aguardar novas requisições, possibilitando o atendimento de pedidos de outros clientes. O processo filho, então, realiza o processamento adequado ao tipo de pedido.

Quando se trata de um pedido de replicação, o servidor verifica se o cliente solicitou o procedimento de inicialização, através do *flag_init*. Se o *flag* estiver ligado, o servidor realiza o procedimento de inicialização: descarta todos os dados do *log* de replicação, relacionados com o sistema de arquivos informado no pedido. Ao inicializar, o servidor também atualiza a sua encarnação para uma encarnação imediatamente seguinte à encarnação atual do SRB, informada no pedido do cliente (*i_srb*).

Durante uma inicialização do SRB, uma réplica pode receber vários pedidos do cliente, solicitando a inicialização: o cliente só considera o SRB inicializado quando recebe a confirmação de **r** réplicas; enquanto essa condição não é satisfeita, o cliente continua enviando pedidos com o *flag_init* ligado. Cada réplica do servidor só deve incrementar a sua encarnação uma única vez, a cada inicialização do SRB. Por isso, o servidor mantém uma variável de controle, *em_inicialização*. Se a variável estiver falsa, significa que esse é o primeiro pedido de inicialização recebido pela encarnação atual: nesse caso, o servidor atualiza o número da sua encarnação e torna a variável *em_inicialização* verdadeira. Ao receber o primeiro pedido de replicação com o *flag_init* desligado, significa que a inicialização do SRB foi concluída com sucesso, então o servidor torna a variável *em_inicialização* novamente falsa.

Quando um servidor recebe um pedido de replicação com o *flag_init* desligado, ele verifica o número do pedido e a encarnação atual do SRB: se o número do pedido for maior que o pedido esperado, significa que o servidor deixou de receber algum pedido; se o número da encarnação informada for maior que o da encarnação atual do servidor, significa que o servidor deixou de passar por algum processo de inicialização. Nos dois casos, a réplica encontra-se desatualizada. Portanto, o servidor executa um procedimento de regeneração, antes de voltar a atender aos pedidos de replicação. Ao reiniciar de um *crash*, uma réplica atribui zero a sua encarnação, para que ela passe, necessariamente, por um procedimento de regeneração.

Finalmente, o servidor envia uma mensagem de confirmação ao cliente, informando a sua encarnação corrente, e armazena os dados recebidos num *log* de replicação, na memória principal.

SERVIDOR:

```
for (; ;) {
        receive_request (p, flag_init, i_srb);
        verifica duplicidade;
        if novo_pedido then {
                cria processo filho;
        }
}
```

PROCESSO FILHO:

```
if replication then {                          /* atende ao pedido de replicação */

        if (flag_init) then {                  /* inicialização */
                reinicia log;                  /* descarta log */
                if not (em_inicialização) then {   / * primeiro pedido de inicialização */
                        i_replica = i_srb + 1;     /* atualiza a encarnação da réplica*/
                        em_inicialização = true;
                }
        } else {                               /* cliente não detectou falha no SRB */
                if ((i_srb > i_replica) or (p > pedido_esperado)) then {
                        proc_regeneration;
                if  (em_inicialização) then
                        em_inicialização = false;
        }

        send_reply (p, i_replica);
        pedido_esperado = p + 1;
        armazena dados no log;

} else if recovery then proc_serve_recovery;   /* atende ao pedido de recuperação */
```

6.5.4 Algoritmo de Regeneração de uma Réplica

Quando uma réplica reinicia, depois de sofrer um *crash*, ou verifica que se encontra desatualizada, ela realiza um procedimento de regeneração, para tornar-se consistente, antes de voltar a atender às requisições de clientes. Para isso, executa um processo que envia uma mensagem ao grupo de réplicas, informando o início da regeneração.

Nas demais réplicas, o servidor executa o procedimento *proc_serve_recovery* (o mesmo procedimento utilizado para atender a um pedido de recuperação de um sistema de arquivos), para atender ao pedido de regeneração. Ao ser informado do início de um procedimento de regeneração, o servidor envia uma resposta à réplica em regeneração, informando o seu estado de atualização, ou seja, a sua encarnação atual e o último pedido de replicação atendido. É preciso observar que, através do algoritmo de replicação, uma réplica não tem como perceber que deixou de receber a última atualização de um sistema de arquivos e, ainda assim, considera-se atualizada. Por isso, é importante que o processo de regeneração receba respostas de várias réplicas, para que possa eleger uma réplica atualizada como a provedora dos dados necessários à regeneração.

O processo de regeneração aguarda a chegada de respostas, durante um tempo máximo e predefinido. Se, nesse tempo, não chegarem as repostas de, pelo menos, **u** réplicas distintas, todas na encarnação mais recente, o processo de regeneração repete a tentativa, até obter sucesso. O número de réplicas que devem responder a um pedido de regeneração é um valor conhecido e configurado pelo administrador do sistema, estando situado no intervalo $\mathbf{n - f < u \leq n}$ (o conjunto de réplicas possivelmente ativas).

A seguir, o processo de regeneração elege uma réplica atualizada (com a maior encarnação e o maior pedido). O processo envia uma mensagem à réplica eleita, solicitando os dados do *log* de replicação referentes aos pedidos de replicação maiores que um pedido informado **p**. O valor de **p** depende da situação da réplica em regeneração:

a) se a encarnação da réplica em regeneração for anterior à maior encarnação recebida (se ela estiver se recuperando de um *crash*, a sua encarnação estará com valor zero), o processo de regeneração atribui o valor zero a **p**, requisitando todo o *log* à réplica eleita. Ao receber o *log* de replicação, o processo de regeneração armazena esse *log* em sua memória, descartando qualquer *log* antigo existente;

b) se a réplica em regeneração estiver na mesma encarnação da réplica eleita, significa que ela apenas deixou de receber alguns pedidos, mas não precisa substituir seu *log* de replicação. Assim, o processo de regeneração informa em **p** o número do último pedido de replicação existente no *log* local. Ao receber os dados da réplica eleita, o processo de regeneração completa o *log* com dados dos pedidos que faltavam.

PROCEDIMENTO DE REGENERAÇÃO:

```
proc_regeneration {
        do {
                init_recovery;                  /* informa o início da regeneração */
                start_timeout;                  /* inicia a contagem do tempo */
                num_replies = 0;                /* inicia o número de respostas  */
                i_max = 0;
                do {
                        receive_status (p, i);
                        if (i > i_max) then {
                                i_max = i;
                                num_replies = 1;
                        } else   incrementa num_replies;
                } while not (timeout) and (num_replies < u);
        } while (num_replies  < u);

        elege uma réplica;
        if (i_replica < i_max) then
                p = 0;
        else
                p = maior pedido do log;

        request_data_recovery (p);
        receive_data_recovery (l);
        armazena dados no log;
}
```

SERVIDOR:

```
proc_serve_recovery; {
        if init_recovery then
                send_status (p, i);
         else {
                ler dados do log (p);
                send_data_recovery (l);
        }
}
```

6.5.5 Algoritmo de Recuperação

Quando um cliente reinicia, após um *crash*, o procedimento de recuperação deve ser executado, para restaurar o sistema de arquivos. Para isso, o procedimento de recuperação interage com o grupo de réplicas, a fim de obter os dados replicados antes do *crash*.

O algoritmo de recuperação é muito semelhante ao algoritmo de regeneração.

CLIENTE:

```
retry = 0;
do {
        retry = retry + 1;
        init_recovery;                  /* informa o início da recuperação */
        start_timeout;                  /* inicia a contagem do tempo */
        num_replies = 0;                /* inicia o número de respostas  */
        i_max = 0;
        do {
                receive_status (p, i);
                if (i > i_max) then {
                        i_max = i;
                        num_replies = 1;
                } else   incrementa num_replies;
        } while not (timeout) and (num_replies < u);
} while (num_replies  < u) and (retry < max_retry);

if (num_replies ≥ u) then {
        elege uma réplica;
        request_data_recovery (0);      /* requisita todo o log */
        receive_data_recovery (l);      /* recebe o log */
        ordena pedidos do log;
        grava dados no disco;
} else fail;
```

O cliente envia uma mensagem para o grupo de réplicas, informando o início da recuperação. O servidor executa o procedimento *proc_serve_recovery*, para atender a um pedido de recuperação de um cliente (o mesmo procedimento que atende um pedido de uma réplica em regeneração). Cada réplica operacional envia uma resposta ao cliente, informando o seu estado de atualização, ou seja, a sua encarnação e o último pedido de replicação atendido.

O cliente espera, por um tempo máximo e predefinido, pela resposta de, pelo menos, **u** réplicas (**n - f < u ≤ n**), sendo **u** um parâmetro informado ao procedimento de recuperação. Caso não cheguem as **u** respostas esperadas, o cliente retransmite o pedido e aguarda novamente por respostas. Isso acontece um número finito de vezes (*max_retry*). Quando o número de tentativas se esgota, caso o cliente não tenha recebido as respostas esperadas, o processo de recuperação falha. O administrador do sistema pode repetir o processo de recuperação, atribuindo um valor menor para **u**.

O restante do procedimento de recuperação acontece da seguinte maneira: com base nas informações recebidas, o cliente elege uma réplica atualizada e requisita à mesma o *log* de replicação; a réplica eleita envia o *log* de replicação para o cliente; de posse do *log*, o cliente grava os dados de cada pedido de replicação no sistema de arquivos em disco, obedecendo à ordem na qual os pedidos foram originalmente gerados.

6.6 Processamento nas Réplicas

O processamento executado pelo Servidor de Replicação de Buffers é muito simples. Após receber um pedido de replicação, o servidor armazena essa mensagem num *log*, mantido inicialmente na memória principal. Para agilizar o procedimento de recuperação, o servidor mantém um *log* para cada sistema de arquivos. Assim, no momento da recuperação, as mensagens de cada sistema de arquivos já estarão agrupadas.

6.6.1 Limpeza dos Logs de Replicação

À medida que os pedidos de replicação vão sendo processados, os *logs* aumentam de tamanho. Para liberar espaço na memória principal, o servidor realiza um procedimento de limpeza, descartando do *log* todos os dados já estabilizados no sistema de arquivos de origem, ou seja, os dados já gravados em disco.

Para realizar a limpeza dos *logs*, o servidor precisa saber quais os dados que podem ser descartados. A cada pedido de replicação, o cliente informa ao servidor o número do maior pedido já estabilizado. O servidor, então, descarta o pedido informado e todos os pedidos anteriores.

O cliente mantém uma tabela de pedidos pendentes em sua memória, para controlar os pedidos que já foram estabilizados. Essa tabela mantém, para cada pedido, uma lista invertida, com os números de todos os blocos de dados replicados pelo pedido. Quando um bloco é gravado no disco, ele é retirado das listas invertidas. Assim, quando a lista de blocos de um pedido torna-se vazia, significa que aquele pedido foi estabilizado. Ao enviar um pedido de replicação, o cliente consulta a tabela de pedidos pendentes, para enviar ao grupo de réplicas o número do maior pedido já estabilizado.

A informação adicional enviada em cada pedido de replicação é pequena (apenas o número do maior pedido já estabilizado). Portanto, não influencia significativamente no tempo da replicação. Adicionalmente, essa solução apresenta uma alta probabilidade de evitar a Situação Atípica da Recuperação, apresentada no item 5.6.3. Contudo, o processamento adicional no cliente pode atrapalhar o desempenho do sistema.

Existe uma forma mais simples de realizar a limpeza dos *logs*: periodicamente, o servidor descarta do *log* todos os pedidos armazenados por um período superior a **2t**, sendo **t** o tempo máximo que um bloco permanece na *cache* de arquivos do cliente, antes de ser gravado em disco. Caso essa ação não libere espaço suficiente na memória, o procedimento de limpeza grava uma parte de cada *log* no disco, começando pelos *logs* maiores, até reduzir o tamanho de todos os *logs* para um percentual predefinido pelo administrador do sistema. Essa solução apresenta o inconveniente de vincular o comportamento do servidor ao comportamento do cliente: se um cliente aumentar o seu tempo **t**, o servidor precisará ajustar o tempo máximo que um pedido de replicação permanece no *log*. Na prática, entretanto, existe um valor padrão para **t** (de trinta segundos), adotado pela maioria dos sistemas de arquivos do UNIX. Assim, embora a primeira solução seja mais genérica e elegante, essa solução é uma forma prática de reduzir o custo da operação de limpeza dos *logs*.

6.7 Considerações Finais

O projeto de um Serviço de Replicação de Buffers apresentado visa atender, de forma equilibrada, aos requisitos de desempenho e confiabilidade. O balanceamento entre tais requisitos é obtido através da escolha dos valores de **n** (número de réplicas do grupo) e **r** (número de réplicas que precisam confirmar um pedido de replicação).

Quanto maior **r**, maior é a probabilidade da informação poder ser recuperada, caso o sistema de arquivos sofra um *crash*; por outro lado, quanto menor **r**, mais rapidamente a confirmação será recebida pelo cliente.

O item 6.2.1 apresentou as razões para que o SALIUS seja implementado por **k** réplicas, com $\mathbf{k \geq f + 1}$, sendo **f** o número de faltas que o SALIUS deseja tolerar, e demonstrou que o SRB deve ser composto de **n** réplicas, sendo $\mathbf{n \geq f}$. O item 6.2.2 demonstrou a necessidade de, pelo menos, **r** réplicas responderem a um pedido ($\mathbf{r = f}$), para que uma replicação seja considerada bem-sucedida. Portanto, se $\mathbf{n = f}$, então, basta que uma réplica falhe, para que o SRB deixe de atender aos pedidos de replicação e o SALIUS passe a utilizar a gravação síncrona, deixando de ser vantajoso, em relação às técnicas de armazenamento estável existentes. Assim, quanto maior $\mathbf{n - f}$, maior a probabilidade de, pelo menos, **f** réplicas não falharem no atendimento a um pedido de replicação, diminuindo a ocorrência de gravações síncronas.

Sendo **freal** o número de réplicas que realmente falham (**freal ≤ f**), o SRB só atende a um pedido de replicação, quando **n - freal ≥ f**. Como o **freal** máximo é igual a **f** (número máximo de faltas suportado pelo SALIUS), fazendo **n ≥ 2f**, a única possibilidade de falha do SRB será devida a uma falha na comunicação entre o cliente e as réplicas.

Num cenário prático, pode-se adotar uma abordagem otimista, assumindo que falhas na comunicação são raras e que durante um intervalo de tempo pequeno (suficiente para que o sistema de arquivos do UNIX grave as informações "estáveis" no disco) a probabilidade de duas réplicas falharem é desprezível. Assim, pode-se estabelecer: **n = 2** e **r = f = 1**. Obviamente, muitos outros cenários são possíveis, mas somente uma experimentação, com uma implementação do serviço, poderia indicar qual o melhor cenário para uma determinada plataforma.

Capítulo 7

Conclusões

7.1 Sumário da Dissertação

Garantir a estabilidade de informações armazenadas é um objetivo dos sistemas de arquivos. Um armazenamento só pode ser considerado totalmente estável, se não existir chance alguma do seu conteúdo ser destruído. Na prática, nenhum sistema de arquivos é capaz de cumprir inteiramente tal condição, porque sempre haverá a possibilidade de falhas de componentes do sistema provocarem a destruição de dados armazenados. Porém, quando o sistema sofre especificamente um *crash*, os dados mantidos na memória volátil são apagados, enquanto os dados gravados em disco são preservados. Assim, assumindo uma semântica de falha por *crash*, um sistema de arquivos pode garantir a estabilidade de informações armazenadas, se puder gravá-las em disco.

A estabilidade de dados não é o único requisito importante para as aplicações atuais. O surgimento de novos tipos de aplicações, como os serviços de comércio eletrônico e os serviços de banco a domicílio, exige que os servidores de arquivos sejam capazes de garantir um certo nível de desempenho, além de prover o armazenamento estável.

Os sistemas de arquivos tradicionais realizam a gravação síncrona em disco, como um meio de garantir a estabilidade de dados armazenados pelas aplicações. No entanto, à medida que a tecnologia evolui, a gravação síncrona vai deixando de ser uma técnica de armazenamento estável recomendável, porque ela atrela o desempenho do sistema ao desempenho dos discos. Atualmente, os demais componentes de hardware, que integram um sistema de arquivos, são superiores aos discos em velocidade. As tendências da tecnologia apontam para uma acentuação dessa diferença. Realizando gravações síncronas, um sistema de arquivos não consegue aproveitar os avanços tecnológicos para oferecer um melhor desempenho aos usuários, devido às limitações impostas pela tecnologia de disco.

As tendências da tecnologia e as necessidades das novas aplicações motivam o desenvolvimento de novas técnicas de armazenamento estável, capazes de desvincular o desempenho dos sistemas de arquivos do desempenho dos discos.

Esta dissertação apresentou a especificação de uma nova técnica de armazenamento estável, denominada SALIUS — SERVIÇO DE ARMAZENAMENTO ESTÁVEL COM RECUPERAÇÃO PARA FRENTE BASEADO NA REPLICAÇÃO REMOTA DE BUFFERS —, como uma alternativa às soluções baseadas em gravação síncrona. A técnica proposta baseia-se na replicação remota de *buffers* alterados na *cache* de arquivos. Assim, após a ocorrência de um *crash*, os dados apagados da memória principal podem ser recuperados, a partir de uma cópia mantida na memória principal de uma máquina remota, para que sejam, finalmente, gravados em disco, restabelecendo a consistência do sistema de arquivos e mantendo as informações estáveis.

O SALIUS é constituído de quatro componentes: uma interface, formada por um conjunto de primitivas, que são utilizadas pelas aplicações, para solicitar que o SALIUS realize operações sobre arquivos; um Servidor de Arquivos Complementar, que executa as operações requisitadas ao SALIUS; um Serviço de Replicação de Buffers, responsável pela replicação confiável dos dados alterados na execução de primitivas do SALIUS e pelo fornecimento de informações necessárias à recuperação do sistema de arquivos, após um *crash*, e, finalmente, um Procedimento de Recuperação, cuja função é ler os dados alterados, a partir de uma réplica, e restaurar a consistência do sistema de arquivos, garantindo a estabilidade de informações gravadas através das primitivas do SALIUS.

Uma aplicação pode escolher entre utilizar as primitivas do SALIUS, ou as primitivas do UNIX, de acordo com as suas necessidades. Quando o Servidor de Arquivos Complementar executa uma primitiva do SALIUS, solicita ao Servidor de Replicação de Buffers a replicação de todos os dados e metadados alterados. O SRB recebe os pedidos de replicação e mantém os dados replicados em um *log* de replicação.

O procedimento de recuperação do SALIUS garante a recuperação de todos os dados armazenados através das primitivas do serviço. Ele solicita ao Servidor de Replicação de Buffers o fornecimento de todos os dados replicados antes do *crash*. De posse desses dados, o procedimento de recuperação providencia gravá-los em disco, refletindo no sistema de arquivos o efeito de todas as operações que tenham sido realizadas com o uso do serviço, antes do *crash*. As informações são gravadas em disco, exatamente na mesma ordem na qual aconteceram originalmente, simulando uma “repetição” das atualizações. O procedimento de recuperação do UNIX continua sendo necessário, porque a alteração de dados e metadados, através da utilização de primitivas originais do UNIX, pode levar o sistema de arquivos a um estado inconsistente.

Este trabalho evidenciou que a probabilidade do SALIUS conseguir alcançar os objetivos estabelecidos, de prover armazenamento estável e de oferecer um desempenho superior às soluções baseadas na gravação síncrona, depende da tecnologia utilizada e da forma como o Serviço de Replicação de Buffers é implementado. Foi apresentado um projeto de implementação para o Serviço de Replicação de Buffers do SALIUS, através de um grupo de réplicas de um processo Servidor de Replicação de Buffers, executando em máquinas remotas.

O projeto sugerido para o Serviço de Replicação de Buffers adota o modelo de replicação ativa e utiliza o protocolo de comunicação UDP *Multicast*. O projeto também especifica um protocolo de replicação para o serviço, detalhando as operações possíveis e os algoritmos utilizados na replicação de dados, na recuperação de um sistema de arquivos e na regeneração de uma réplica.

O trabalho, também, demonstrou que o Serviço de Replicação de Buffers pode balancear o atendimento aos requisitos de desempenho e confiabilidade, bastando ajustar o número de réplicas do grupo e o número de réplicas que precisam enviar uma mensagem de confirmação, para que a replicação seja considerada bem-sucedida.

7.2 Análise do Trabalho

A especificação do SALIUS, proposta nesta dissertação, consegue atender aos requisitos previstos e detém potencialidades, para que uma implementação eficiente possa atingir os principais objetivos:

- o SALIUS adota realmente uma semântica de falha por *crash*, pois prevê a replicação dos dados alterados na memória principal, durante as operações que atualizam o sistema de arquivos, prevenindo a perda dessas dados na ocasião de um *crash*;
- o SALIUS é transparente quanto à localização e quanto à replicação, pois as ações para realizar a replicação de dados e manter a consistência das réplicas são abstraídas da aplicação. O SALIUS também consegue mascarar falhas no grupo de réplicas, porque continua realizando as operações solicitadas, até mesmo quando todas as réplicas falham simultaneamente e o Serviço de Replicação de Buffers é interrompido;
- a administração do SALIUS é muito simples, cabendo ao administrador apenas definir o grupo de réplicas e reconfigurá-lo; quando necessário, executar o procedimento de recuperação e definir alguns parâmetros, tais como: o número de faltas que precisam ser toleradas; o número de réplicas necessárias à recuperação do sistema; o número de tentativas de replicação realizadas pelo SAC, antes de assumir que o SRB falhou; o tempo máximo que um cliente deve esperar por respostas do grupo de réplicas; o tempo máximo que um pedido de replicação permanece no *log*, antes de ser descartado;

- o SALIUS mantém a semântica de compartilhamento do UNIX, porque preserva a ordem das operações de escrita, quando um arquivo está sendo compartilhado. Para isso, o SALIUS utiliza um contador único de pedidos de replicação, para cada sistema de arquivos. Quando um processo está atualizando o sistema de arquivos, adquire um número de pedido, o qual é enviado no cabeçalho da mensagem de replicação. O procedimento de recuperação utiliza esse número de pedido para ordenar a gravação dos dados do *log* de replicação no disco;
- a especificação do SALIUS não requer a utilização de hardware especial para a sua implementação. Assim, o objetivo de não acrescentar custos ao sistema consegue ser atendido;
- o procedimento de recuperação do SALIUS torna possível a recuperação do sistema de arquivos após um *crash*, garantindo a estabilidade das informações armazenadas com o uso do serviço;
- o SALIUS vincula o desempenho das operações de escrita ao desempenho da rede, da memória e dos processadores utilizados na implementação, diminuindo a influência da tecnologia de disco. Como as tendências da tecnologia apontam para um grande aumento no potencial de processamento, na velocidade de acesso à memória e nas taxas de transmissão de dados através de redes, o desempenho oferecido pelo SALIUS também tende a crescer;
- a confiabilidade do SALIUS também tende a aumentar com os avanços tecnológicos, à medida em que forem surgindo memórias e redes mais confiáveis, pois diminui o risco de dados serem perdidos ou corrompidos enquanto estiverem armazenados na memória, ou enquanto estiverem trafegando pela rede;
- finalmente, uma implementação eficiente do Serviço de Replicação de Buffers é fundamental para que os objetivos de desempenho e confiabilidade do SALIUS sejam alcançados.

A Tabela 7.1 é uma extensão da Tabela 4.1, apresentada no Capítulo 4, e relaciona as características de vários sistemas de arquivos robustos e do SALIUS, num quadro comparativo. Cada característica apresentada refere-se a um dos objetivos do SALIUS, descritos no Capítulo 5. As informações associadas ao SALIUS consideram a probabilidade de uma implementação eficiente poder atender a cada um desses objetivos.

Assim, o SALIUS tem um grande potencial para constituir um serviço de armazenamento estável de propósito geral, capaz de oferecer um melhor desempenho que os sistemas convencionais (baseados na gravação síncrona em disco), dotado de um mecanismo de recuperação para frente, que garante a estabilidade de dados de cache replicados, e a um custo acessível, porque não depende de um hardware especial.

SISTEMAS	ARMAZENAMENTO ESTÁVEL	MELHORIA NO DESEMPENHO	HARDWARE ESPECIAL	MECANISMO DE RECUPERAÇÃO	PROPÓSITO GERAL
BASEADOS EM LOG	NÃO	SIM	NÃO	SIM	SIM
CACHE DE NVRAM	SIM	SIM	SIM	NÃO	SIM
RIO FILE CAHE	SIM	SIM	SIM	SIM	SIM
SISTEMA ENVY	SIM	SIM	SIM	NÃO	NÃO
SALIUS	SIM	SIM	NÃO	SIM	SIM

Tabela 7.1: Quadro comparativo do SALIUS com trabalhos relacionados.

O SALIUS apresenta a desvantagem de aumentar o consumo da banda passante do meio de comunicação, pois o funcionamento do serviço exige a troca constante de mensagens entre os clientes e o SRB. Teoricamente, a utilização de redes de alta velocidade diminuiria a probabilidade de um congestionamento inviabilizar a replicação de dados. Porém, somente a experimentação, através de uma implementação do serviço, permitirá uma avaliação mais precisa dos efeitos do aumento do tráfego de dados na rede, causados pelo SALIUS.

7.3 Trabalhos Futuros

As idéias discutidas nesta dissertação abrem caminho para pesquisas futuras.

- O principal trabalho a ser desenvolvido é a implementação do SALIUS. Embora a especificação apresentada indique uma alta probabilidade de sucesso, a técnica proposta só estará validada depois de implementada e utilizada como um sistema de armazenamento, num ambiente real de produção.
- O serviço proposto pode ser estendido para o ambiente de sistema de arquivos em rede, como o NFS, ou para um sistema de arquivos distribuído. Para isso, é preciso prever soluções para o acesso concorrente a arquivos e para a manutenção da consistência das réplicas, num ambiente distribuído.
- O SALIUS pode, também, ser adaptado para o ambiente do Windows NT. A dificuldade de criar uma especificação, uma implementação e um projeto para cada ambiente específico pode ser solucionada com a utilização de um *framework*.

- Outro trabalho a ser desenvolvido é o estudo do número ideal de réplicas do Servidor de Replicação de Buffers (**n**) e o número de réplicas que devem responder aos pedidos de replicação de um cliente (**r**), de modo a otimizar o atendimento aos requisitos de desempenho e confiabilidade. Este estudo pode ser realizado através de experiências com uma implementação do SALIUS, para diferentes valores de **n** e **r**.
- Um trabalho possível é a flexibilização do tempo de permanência dos dados na *cache* de arquivos, para os dados armazenados com o uso do SALIUS, conforme a disponibilidade de memória principal. Assim, havendo memória suficiente, esses dados poderiam ser mantidos na memória por um tempo maior. Quando o espaço em memória se tornasse escasso, o tempo de permanência dos dados na *cache* também diminuiria, dinamicamente. O efeito dessa estratégia para a otimização do desempenho do sistema deve ser testado na prática.
- A associação do SALIUS com outras técnicas pode vir a otimizar o serviço de armazenamento estável oferecido: o uso de novas tecnologias de armazenamento secundário, como a tecnologia RAID, pode contribuir para uma melhoria no desempenho do serviço, enquanto a confiabilidade pode ser aprimorada, através de um mecanismo de proteção de memória, capaz de evitar que a memória seja sobregravada acidentalmente pelo sistema.

Bibliografia

[ABC+97] G. Alverson, P. Briggs, S. Coatney, S. Kahan and R. Korry. Tera Hardware-Software Cooperation. In *Proceedings of the ACM/IEEE SC97*, pp. 15-21, November 1997.

[ACP95] T. Anderson, D. Culler and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro* 15(1): 54-56, February 1995. Also http:\\www.now.cs.berkeley.edu.

[AD98] G. Abandah, E. Davidson. Effects of Architectural and Technological Advances on the HP/Convex Exemplar's Memory and Communication Performance. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 318-329, June 1998.

[ADN+95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th Symp. on Operating Systems Principles*, 109-126, December 1995. Also *ACM Trans. On Computing Systems*, 14(1): 41-79, February 1996.

[Amd67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, April, 1967.

[AMD99] Advanced Micro Devices, Inc. Technology Background AMDDL160 and DL320 Series Flash: New Densities, New Features. Publication 22271, May 1999.

[AMD99a] Advanced Micro Devices, Inc. Technology Background 3.0 Volt-only Page Mode Flash Memory Technology. Publication 22249, May 1999.

[ANS90] ANSA. A Model for Interface Groups. *ANSA, ISA Project,* APM/RC 093.00, May 1990.

[ANS91] ANSA. An Abstract Model for Groups. *ANSA, ISA Project,* APM/RC 259.01, June 1991.

[AT89] AT&T. ***UNIX System V, Release 3.2. System Administrator's Guide***. Prencice Hall, Inc. 1989.

[Bac86] M. Bach. ***The Design of the UNIX Operating System***. Prentice Hall P T R, 1990.

[BAD+92] M. Baker, S. Asami, E. Deprit, J. Ousterhout and Margo Seltzer. Non-Volatile Memory for Fast Reliable File Systems. *In Fifth International Conference on Architectural Support for Programming Languages and Operating Systema (ASPLOS-V),* pp. 10-22, October 1992.

[BBD+97] M. Beck, H. Bõhme, M. Dziadzka, U. Kunitz, R. Magnus and D. Verworner. ***LINUX Kernel Internals***. Second edition. Addison-Wesley, 1997.

[BGI97] BGI Datentechnik GmbH. Solid State Disk Frequently Asked Questions. In www.silicondisk.com/SSD_tech_faq.html, October 1997.

[BHK+91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th Symp. on Operating Systems Principles,* pp. 198-212, October 1991.

[Bir96] K. P. Birman. ***Building Secure and Reliable Network Applications***. Manning Publications Co., 1996.

[BMR+91] M. Benatre, G. Muller, B. Rochat and P. Sanchez. Design decisions for the FTM: a general purpose fault tolerant machine. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, pp. 71-78, June 1991.

[Cas98] C. Cassidy. Understanding the Performance of Quantum Solid State Disks. In www.quantum.com/src/whitepapers/wp_ssdperformance.htm, Quantum Corporation's whitepaper, 1998.

[CLG+94] P. M. Chen, E. Lee, G. Gibson, R. Katz and D. Petterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2): 145-188, June 1994.

[CNR+96] P. M. Chen, Wee Teck Ng, G. Rajamani, C. M. Aycock. The Rio File *Cache*: Surviving Operating Systems Crashes. *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, SIGPLAN Notices, 31(9): 74-83, September 1996.

[Cri91] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *ACM Communications*, 34(2): 56-78, February 1991.

[CS91] D. Comer and D. Stevens. ***Internetworking with TCP/IP,Principles, Protocols and Architecture.*** Vol. I. Prentice Hall International, Inc., 1991.

[CS93] D. Comer and D. Stevens. ***Internetworking with TCP/IP, Client/Server Programming and Applications BSD Socket Version.*** Vol. III. Prentice Hall International, Inc., 1993.

[Cus94] H. Custer. ***Inside the Windows NT File System***. Microsoft Press, 1994.

[Dah+94] M. Dahlin et al. A Quantitative Analysis of *Cache* Policies for Scalable Network File Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 150-160, May 1994.

[DEC94] DEC 3000 300/400/500/600/700/800/900 AXP Models System Programmer's Manual. *Technical Report, Digital Equipament Corporation*, July 1994.

[ECG+92] T. von Eicken, D. Culler, S. Goldstein and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th ISCA*, pp. 256-266, May 1992.

[HO95] J. Hartman, and J. Ousterhout. The Zebra Stripped Network File System. *ACM Trans. On Computer Systems*, 13(3): 274-310, August 1995.

[IEE98] IEEE (*Institute of Electrical and Eletronics Engineers*) Standards. IEEE 802.3-1998 Edition. June 1998.

[Int94] Intel Corporation. Flash Memory, 1994.

[Int97] Intel Corporation. *Intel Architecture Software Developer's Manual*, Vol.1: Basic Architecture, Order Number 243190, 1997.

[Jal94] P. Jalote. ***Fault Tolerance in Distributed Systems***. Prentice Hall, 1994.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *ACM Communications*, 21(7): 558-565, July 1978.

[LCN90] L. Liang, S. T. Chanson, and G. W. Neufeld. Process Groups and Groups Communications: Classifications ans Requirements. *IEEE Computer*, February 1990.

[LEK91] R. LaRowe, C. Ellis and L Kaplan. The Robustness os NUMA Memory Management. In *Proceedings os Thirteenth Symposium on Operating Systems Principles*, ACM, pp. 137-151, 1991.

[Lit92] M. C. Little. Object Replication in a Distributed System. Ph.D. Thesis, University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, Nº 376, February 1992.

[LS90] E. Levy, and A. Silberschatz. Distributed File System: Concepts and Examples. *Computing Surveys*, 22(1): 321-374, December 1990.

[McK96] M. McKusick. Secondary Storage and Filesystems. *ACM Computing Surveyes*, 28(1), March 1996.

[MCP+98] P. Messina, D. Culler, W. Pfeiffer, W. Martin, J. Oden and G. Smith. Architecture. *Communications of the ACM*, 41(11): 36-44, November 1998.

[MH88] J.Menon and M. Hartung. The IBM 3990 Disk *Cache*. In *Proceedings of the COMPCON*, pp. 146-151, June 1988.

[MJL+84] M. McKusick, W. Joy, S. Leffler and S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3): 181-197, August 1984.

[MMP94] D. Major, G. Minshall and K. Powell. An Overview of the NetWare Operating System. In *Proceedings of the 1994 Winter USENIX*, pp. 355-372, January 1994.

[MSC+90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs and B. Lyon. Breaking Through the NFS Performance Barrier. In *Proceedings of the EUUG Spring*, pp. 199-206, April 1990.

[Mud+96] T. Mudge et al. Strategic Directions in Computer Architecture. *ACM Computing Surveyes*, 28(4): 671-678, December 1996.

[Nor95] P. Norton. ***Peter Norton's Inside the PC***. Sams Publishing, 1995.

[NWO88] M. Nelson, B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1): 134-154, February 1988.

[OCD+88] J. K. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2): 23-36, February 1988.

[OCH+85] J. K. Ousterhout, Herve Da Costa, D. Harrison, J. Kunze, M. Kupfer and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 1985 Symposium on Operating System Principles*, pp. 15-24, December 1985.

[OD88] J. K. Ousterhout, and F. Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Technical Report, UCB/CSD 88/467, Computer Science Division, UC Berkeley, October 1988.

[OMG95] Object Management Group. CORBA Services: Common Object Services Specification. Reference OMG 1995, http://www@omg.org.

[OOW91] M. H. Olsen, E. Oskiewicz, and J.P. Warne. A Model for Interface Groups. *Proceedings of SRDS-10*. October 1991.

[PLC95] S. Pakin, M. Lauria and A. Chein. High Performance Messagins on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[RO92] M. Rosenblum, and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computing Systems*, 10(1): 26-52, February 1992.

[RT78] D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Thecnical Journal*, 57(6): 1905-1930, July 1978.

[Sat93] M. Satyanarayanan. ***Distributed System***. ACM Press, 1993.

[SBM+93] M. Seltzer K. Bostic, M. McKusick and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the Winter 1993 USENIX Conference*, pp. 307-326, January 1993.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using State Machine Approach: A Tutorial. *ACM Computing* Surveys, 22(4): 299-319, December 1990.

[SCO90] M. Seltzer, P. Chen and J. Ousterhout. Disk Scheduling Revisted. In *Proceedings of the Winter 1990 USENIX Techinical Conference*. January 1990.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings. of the Summer 1985 USENIX*, pp. 119-130, June 1985.

[SSA+95] C. Stunkel, D. Shea, B. Abali, M. Atkins, C. Bender, D. Grice, P. Hochschild, D. Joseph, B. Nathanson, R. Swetz, R. Stucke, M. Tsao and P. Varker. The SP2 High-Performance Switch. *IBM System J*, 34(2): 185-204, 1995.

[Sta97] T. Stabell-Kulo. Security and Log Structured File Systems. *Operating Systems Review*, 31(2): 9-10, April 1997.

[Ste91] H. Stern. ***Managing NFS and NIS***. O'Reilly & Associates, Inc, 1991, ISBN 0-937175-75-7.

[Ste92] W. Stevens. ***Advanced Programming in the UNIX Environment***. Addison Wesley, 1992.

[STH+99] S.Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi and Y. Ishikawa. The Design and Evalution of High Performance Communication using a Gigabit Ethernet. In *Proceedings of the International Conference on Supercomputing*, pp. 260-267, 1999.

[Tan92] Tanenbaum, A. S. ***Modern Operating Systems***. Prentice Hall, Inc, 1992.

[Tan95] Tanenbaum, A. S. ***Distributed Operating Systems***. Prentice Hall, Inc, 1995.

[TE95] TIA (*Telecommunications Industries Association*) / EIA (*Eletronics Industries Alliance*) Standards. TIA/EIA-568-A. Commercial Building Telecommunications Cable Standard. Comittee TR-41.8.1, October 1995.

[WZ94] Michael Wu and Willy Zwaenepoel. eNVy: Non-Volatile, Main Memory Storage Syatem. In *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1994.

Apêndice A

Dados Replicados pelas Primitivas do SALIUS

Cada primitiva do SALIUS está associada a um conjunto de dados, os quais ela possivelmente altera e replica. Esse apêndice analisa, para cada primitiva do serviço, os dados que podem ser alterados e em que situações tais alterações acontecem.

a) s_creat

A primitiva `s_creat` é utilizada para criar um arquivo. Ela aloca um *nó-i* para o novo arquivo. Para isso, retira um *nó-i* da lista de *nós-i* livres. Conseqüentemente, a primitiva altera o superbloco, onde a lista de *nós-i* livres está armazenada e onde existe um contador de *nós-i* livres, que é decrementado quando o *nó-i* é alocado.

A primitiva também cria uma entrada para o *nó-i* do arquivo no diretório pai, alterando um bloco de dados desse diretório. Se o tamanho do diretório pai aumentar, a primitiva ajusta o tamanho de arquivo, armazenado no *nó-i* desse diretório. Se o último bloco de dados do diretório pai estiver cheio, a primitiva `s_creat` deve alocar um novo bloco para armazenar a entrada do arquivo, retirando um bloco da lista de blocos livres e alterando o contador de blocos livres, no superbloco. A primitiva `s_creat` precisa replicar todas as estruturas alteradas durante a sua execução, o que pode consistir de: o superbloco, o *nó-i* do novo arquivo, o *nó-i* do diretório pai e o bloco de dados do diretório pai, com a entrada do novo arquivo.

b) open

A primitiva `open` abre um arquivo existente e pode criar um novo arquivo, caso o *flag* O_CREAT seja informado. No caso da criação de um arquivo, os dados que devem ser replicados são os mesmos da primitiva `s_creat`.

Quando um arquivo existente é aberto, normalmente, nenhum dado é alterado, com exceção da situação em que o *flag* O_TRUNC é informado. Nesse caso, todos os blocos do arquivo são liberados e a primitiva altera o *nó-i* do arquivo, ajustando o tamanho do arquivo para zero. A operação de truncamento do arquivo também afeta o superbloco, pois os blocos liberados são inseridos na lista de blocos livres e o tamanho dessa lista é incrementado. Assim, a primitiva `open`, quando invocada com a opção O_TRUNC, deve replicar o *nó-i* do arquivo e o superbloco.

c) write

A primitiva `write` é utilizada para escrever dados em um arquivo. Quando uma aplicação invoca essa primitiva, ela passa, como parâmetro de entrada, um *buffer* de usuário contendo os dados a serem alterados. A primitiva deve replicar os dados informados, bem como a posição no arquivo, onde deve iniciar a gravação desses dados. Essa posição é lida na tabela de arquivos abertos.

Se o arquivo tiver sido previamente aberto com a opção O_APPEND, significa que os dados serão acrescidos ao final do arquivo, aumentando o seu tamanho. Nesse caso, a primitiva ajusta o tamanho do arquivo, no seu respectivo *nó-i*. Quando o tamanho de um arquivo aumenta, novos blocos precisam ser alocados, alterando a lista de blocos livres e o superbloco. Desse modo, a primitiva `write` pode precisar replicar também o superbloco e o *nó-i* do arquivo.

d) s_mkdir

A primitiva `s_mkdir` cria um novo diretório. Assim, deve replicar os mesmos dados que a primitiva `s_creat`, com uma pequena diferença: ao invés do *nó-i* de um arquivo, a primitiva vai replicar o *nó-i* do novo diretório. Adicionalmente a primitiva `s_mkdir` deve replicar o primeiro bloco de dados do diretório, contendo as entradas “.” (do próprio diretório) e “..” (do diretório pai).

e) s_mknod

A primitiva `s_mknod` é utilizada para criar qualquer tipo de arquivo. Assim, deve replicar os mesmos dados que a primitiva `s_mkdir`, no caso da criação de um diretório, ou os mesmos dados replicados pela primitiva `s_creat`, nos demais tipos de arquivos.

f) s_link

A primitiva `s_link` é utilizada para criar uma nova entrada para um arquivo existente, num diretório informado. Para isso, altera um bloco de dados desse diretório, acrescentando a entrada para o arquivo. O diretório aumenta de tamanho e a primitiva incrementa o tamanho do diretório, armazenado no seu respectivo *nó-i*. Se o último bloco de dados do diretório não comportar a nova entrada, a primitiva `s_link` precisa alocar um novo bloco de dados, alterando, portanto, a lista de blocos livres e o superbloco. Finalmente, a primitiva altera o *nó-i* do arquivo, incrementando o contador de ligações. Assim, a primitiva `s_link` sempre precisa replicar o *nó-i* do arquivo, o *nó-i* do diretório o e o bloco de dados do diretório contendo a nova entrada. Eventualmente, a primitiva pode necessitar alterar e replicar o superbloco.

g) s_unlink

A primitiva `s_unlink` é utilizada para remover uma entrada de diretório. Portanto, o bloco de dados do diretório, onde a entrada está armazenada, é alterado. Além disso, a primitiva ajusta o tamanho do diretório, que diminui, no seu respectivo *nó-i*. Finalmente, a primitiva altera o *nó-i* do arquivo, decrementando o número de ligações existentes. Assim, a primitiva precisa replicar o bloco de dados de onde a entrada foi excluída, o *nó-i* do diretório e o *nó-i* do arquivo.

Quando a última entrada de diretório de um arquivo é removida, o arquivo é excluído do sistema. Para isso, a primitiva `s_unlink` libera todos os blocos de dados do arquivo e o *nó-i* associado. O superbloco também é alterado, porque os blocos de dados liberados são inseridos na lista de blocos livres, enquanto o *nó-i* do arquivo é inserido na lista de *nós-i* livres. Nesse caso, a primitiva replica o superbloco e não replica o *nó-i* do arquivo excluído, obviamente.

h) s_chown e s_chmod

As primitivas `s_chown` e `s_chmode` só alteram o *nó-i* de um arquivo, ou de um diretório, mudando, respectivamente, os atributos proprietário e permissões de acesso. Assim, tais primitivas só precisam replicar o *nó-i* do arquivo, ou do diretório que está sendo alterado.