# Sistemas de Arquivos Distribuídos

**Fabio Kon**

Dissertação Apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
para a Obtenção do Grau de
Mestre em Matemática Aplicada

Área de Concentração: **Ciência da Computação**

Orientador: **Prof. Dr. Arnaldo Mandel**

*Este trabalho recebeu apoio do CNPq
e da FAPESP (processo No. 93/0603-1)*

– São Paulo, novembro de 1994 –

# Sistemas de Arquivos Distribuídos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Fabio Kon
e aprovada pela comissão julgadora.

São Paulo, 25 de novembro de 1994.

Banca examinadora:

- Prof. Dr. Arnaldo Mandel (orientador) – IME-USP
- Prof. Dr. Markus Endler – IME-USP
- Profa. Dra. Regina Helena Carlucci Santana – ICMSC-USP

*à Paula*

## Resumo

Este trabalho se inicia com uma discussão sobre as principais características dos Sistemas de Arquivos Distribuídos, ou seja, espaço de nomes, localização, cache, consistência, replicação, disponibilidade, escalabilidade, heterogeneidade, tolerância a falhas e segurança. Analisamos diversos sistemas de arquivos existentes quanto a estas características dando especial atenção aos sistemas *NFS, Andrew, Coda, Sprite, Zebra, Harp, Frolic* e *Echo*.

Descrevemos alguns modelos analíticos para o comportamento de sistemas de arquivos distribuídos e, em seguida, apresentamos um novo modelo para sistemas baseados em *leases* – um mecanismo para a garantia de consistência em sistemas distribuídos. Finalmente, descrevemos a nossa implementação do sistema de arquivos distribuído SODA – que utiliza *leases* – apresentando dados sobre o seu desempenho.



## Abstract

We begin with a discussion of some of the main characteristics of Distributed File Systems, namely locality, name space, cache, consistency, replication, availability, scalability, heterogeneity, fault tolerance and security. We analyze several file systems based on these characteristics. We focus mainly on *NFS, Andrew, Coda, Sprite, Zebra, Harp, Frolic,* and *Echo*.

We then describe some analytical models of the behavior of distributed file systems. After that we present a new model of systems based on *leases* – a mechanism to assure consistency on a distributed system. Finally we describe our implementation of the SODA distributed file system – which uses leases – presenting some results on its performance.

# Prelúdio

Escrever uma dissertação sobre sistemas operacionais em português não é uma tarefa fácil. Digo isto porque uma série de termos chaves da teoria de sistemas operacionais carecem de uma tradução satisfatória e consensual na nossa querida língua. Termos como *lock* possuem inúmeras traduções tais como tranca e bloqueio. Já *deadlock*, por sua vez, pode ser traduzido como paralisação, bloqueio morto, impasse. Cada autor possui o seu termo preferido. Oxalá consigamos convergir para um vocabulário comum nos próximos anos.

Neste trabalho, optei por citar o termo correspondente na língua inglesa sempre que introduzo um novo conceito em português. Alguns termos, como *software*, porém, ficaram sem tradução e sem a grafia correta em português (softuer) a fim de não chocá-lo.

Não encontrei nenhuma alternativa satisfatória para a utilização do verbo cachear, que não se encontra no Aurélio. Utilizei, sem perdão, as formas cacheia, cacheiam, cacheado. Quem sabe elas apareçam futuramente em dicionários da língua portuguesa ao lado de cacheada (espécie de dança praticada na festa de reis no nordeste) e cacheado (em forma de cachos). Utilizaremos, também, o neologismo acessar com o mesmo significado do verbo *to access* do inglês.

Conto, portanto, com a sua boa vontade para a assimilação dos termos técnicos aqui apresentados.

Este trabalho não poderia ser realizado de maneira alguma sem a colaboração de duas pessoas às quais devo um agradecimento especial. Ao Arnaldo devo uma boa parte de todo o meu conhecimento em Ciência da Computação. Desde o funcionamento dos famosos computadores a papel até a intersecção de matróides co-gráficos, desde a análise da busca binária até o teorema de Kleene, desde a manipulação de arquivos DOS até as idiossincrasias infindas de uma rede de computadores real. Não bastasse isso, devo ainda agradecê-lo pela sua disposição constante em orientar inteligentemente cada passo da elaboração deste trabalho sem nunca tolher a minha liberdade.

Um agradecimento especialíssimo é endereçado à Professora Dilma Menezes da Silva. Apesar de distante fisicamente, a sua proximidade foi enorme e fundamental. Sinto que a sua contribuição se iniciou mesmo antes da escolha do tema desta dissertação mas foi durante a escolha do tema que a sua participação se tornou clara e, a partir daí, sua cumplicidade se tornou inevitável. A Dilma, assim como o Arnaldo, examinou, em tempo real, cada linha da dissertação e sempre respondeu criticamente oferecendo sugestões valiosas. O seu entusiasmo contagiante não se limitou às questões técnicas; estravasou e serviu de estímulo para que eu não perdesse a vontade de continuar trabalhando.

Aos professores Antonio Galves e Vanderlei da Costa Bueno do Departamento de Estatística do IME e ao professor Isaac Meilijson do Departamento de Matemática da Universidade de Tel-Aviv devo algumas horas que foram investidas na análise do comportamento dos *leases* e que auxiliaram na elaboração do modelo analítico do capítulo 4. Obrigado.

Finalmente, gostaria de agradecer à Paula, à minha mãe, pai, irmãos, cunhados, sobrinhos e amigos que vêm me auxiliando nos aspectos técnicos e não técnicos da vida ☺

# Conteúdo


**1 Introdução** **15**

1.1 Conceitos Básicos . . . 17

1.1.1 Nomes e Localização . . . 17

1.1.2 Cache . . . 18

1.1.3 Disponibilidade . . . 20

1.1.4 Escalabilidade . . . 21

1.1.5 Heterogeneidade . . . 22

1.1.6 Segurança . . . 22

1.1.7 Tolerância a Falhas . . . 23

1.1.8 Operações Atômicas . . . 24

1.1.9 Acesso Concorrente . . . 24

1.1.10 Semântica do Acesso Concorrente . . . 26

1.1.11 Replicação . . . 26

1.1.12 Como tudo isso se relaciona? . . . 27

**2 Estudo de casos** **29**

2.1 Um Pouco de História . . . 29

2.2 Uma Visão Geral . . . 30

2.3 NFS . . . 32

2.3.1 Um Protocolo Livre de Estado . . . 32

2.3.2 Espaço de nomes . . . 33

2.3.3 O Protocolo . . . 35

2.3.4 Segurança . . . 37

2.3.5 Cache Inconsistente . . . 37

2.3.6 Arquitetura . . . 39

2.3.7 Resumo . . . 41

2.4 ANDREW . . . 42

2.4.1 Espaço de Nomes . . . 43

2.4.2 Replicação . . . 45

2.4.3 Arquitetura . . . 45

2.4.4 Segurança . . . 46

2.4.5 Cache . . . 48

2.4.6 Resumo . . . 49

2.5 CODA . . . 50

2.5.1 Replicação . . . 50

2.5.2 Controle da Consistência das Réplicas . . . 51

2.5.3 Os estados do *venus* . . . 51

2.5.4 Tratamento dos Conflitos . . . 54

2.5.5 Desempenho . . . 55

2.5.6 Resumo . . . . . 56
2.6 SPRITE . . . . . 57
2.6.1 Tabelas de Prefixos (Resolução dos *pathnames*) . . . . . 58
2.6.2 Segurança . . . . . 60
2.6.3 Cache . . . . . 60
2.6.4 Disponibilidade . . . . . 63
2.6.5 Analisando o Desempenho . . . . . 65
2.6.6 Comparações com o NFS e o ANDREW . . . . . 70
2.6.7 Resumo . . . . . 71
2.7 ZEBRA - Um Sistema Listrado . . . . . 72
2.7.1 Sistemas de Arquivos Baseados em *log* . . . . . 72
2.7.2 RAID . . . . . 74
2.7.3 As Listras do ZEBRA . . . . . 75
2.7.4 *File Manager* . . . . . 75
2.7.5 Desempenho . . . . . 76
2.7.6 Resumo . . . . . 78
2.8 HARP . . . . . 79
2.8.1 Implementando a replicação . . . . . 80
2.8.2 Modo Normal de Operação . . . . . 81
2.8.3 Desempenho . . . . . 82
2.8.4 Resumo . . . . . 83
2.9 FROLIC . . . . . 84
2.9.1 Replicação Dinâmica . . . . . 86
2.9.2 Semântica . . . . . 87
2.9.3 Desempenho . . . . . 88
2.9.4 Resumo . . . . . 88
2.10 ECHO . . . . . 89
2.10.1 Tolerância a Falhas . . . . . 89
2.10.2 Espaço de Nomes . . . . . 90
2.10.3 Cache . . . . . 90
2.10.4 Desempenho . . . . . 92
2.10.5 Resumo . . . . . 93
2.11 Resumo Comparativo . . . . . 93

**3 Modelos Analíticos** **95**
3.1 Modelo de Borghoff . . . . . 96
3.1.1 IPELA . . . . . 96
3.1.2 CFAP . . . . . 99
3.1.3 Simulação . . . . . 102
3.1.4 Críticas ao Modelo . . . . . 104
3.2 Arquitetura de Cache Remoto . . . . . 106
3.2.1 O modelo . . . . . 106
3.2.2 *Simulated Annealing* . . . . . 108
3.2.3 Simulação . . . . . 109
3.2.4 Críticas ao Modelo . . . . . 111

**4 *Leases*** **113**
4.1 Um Mecanismo para a Consistência do Cache . . . . . 113
4.1.1 A Duração de um *Lease* . . . . . 114
4.1.2 Sincronização de Relógios . . . . . 116
4.2 O modelo de Gray . . . . . 117
4.3 Melhorando o Modelo . . . . . 119
4.3.1 Comparando com o SPRITE . . . . . 121
4.3.2 Estimativas do Modelo . . . . . 122
4.4 SODA . . . . . 125
4.4.1 Implementação . . . . . 125
4.4.2 Análise do Desempenho . . . . . 128
4.4.3 SODA Adaptativo? . . . . . 133
4.4.4 Conclusão . . . . . 133
4.4.5 Trabalho Futuro . . . . . 134

**5 O Futuro** **137**
5.1 Flexibilidade . . . . . 137
5.2 O Sistema de Arquivos Físico Ideal . . . . . 140
5.3 Enfim, o que Esperamos . . . . . 140

**A O Sistema de Arquivos UNIX** **143**

**Bibliografia** **147**

# Lista de Figuras


1.1 Caches em sistemas de arquivos distribuídos . . . . . . . . . . . . . . . . . . . . . 19
1.2 Relações entre os Conceitos Básicos . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.1 Seqüência de comandos *mount* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.2 Procedimentos do protocolo NFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.3 Esboço da ação de cada processo . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.4 *v-nodes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.5 A Arquitetura do NFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.6 O espaço de nomes de um cliente . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
2.7 Identificador de arquivo (*fid*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
2.8 Uma *célula* do AFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
2.9 Arquitetura global do AFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
2.10 Perfil de salvaguarda . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
2.11 Compartilhamento de um arquivo (concorrente×seqüencial) . . . . . . . . . . . . 62
2.12 Desempenho comparativo do NFS, ANDREW e SPRITE . . . . . . . . . . . . 70
2.13 A criação de dois arquivos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
2.14 O listramento . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
2.15 Modelo tradicional de cache nos clientes . . . . . . . . . . . . . . . . . . . . . . 85
2.16 Modelo do FROLIC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
2.17 Replicação de servidores e discos . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
2.18 Replicação no ECHO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

3.1 Uma Rede com 5 *Clusters* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
3.2 *Simulated Annealing* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
3.3 Padrão de acesso aos arquivos . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

4.1 *Leases* em um cliente . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
4.2 Número de mensagens geradas pelo protocolo . . . . . . . . . . . . . . . . . . . 122
4.3 Sensibilidade à relação entre R e W . . . . . . . . . . . . . . . . . . . . . . . . 123
4.4 Escalabilidade . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
4.5 Escalabilidade com $W$ pequeno . . . . . . . . . . . . . . . . . . . . . . . . . . 124
4.6 Registro de um arquivo no cliente . . . . . . . . . . . . . . . . . . . . . . . . . . 126
4.7 Percurso de duas solicitações ao servidor . . . . . . . . . . . . . . . . . . . . . . 128
4.8 Utilização da CPU do servidor . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
4.9 Tempo para efetuar a leitura de 1Kbyte . . . . . . . . . . . . . . . . . . . . . . 131
4.10 Tempo para efetuar uma escrita de 1Kbyte . . . . . . . . . . . . . . . . . . . . 131
4.11 Carga na CPU × duração dos *leases* . . . . . . . . . . . . . . . . . . . . . . . . 132
4.12 Número de mensagens × duração dos *leases* . . . . . . . . . . . . . . . . . . . . 132

5.1 Estrutura do SPRING . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

5.2 Sistema de arquivos fictício construído no SPRING . . . . . . . . . . . . . . . . . 139

A.1 Apontadores para os blocos de um arquivo UNIX . . . . . . . . . . . . . . . . . . 145
A.2 Tabelas descritoras de arquivos . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146

# Lista de Tabelas


1.1 *Deadlock* no protocolo de duas fases . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.1 Servidor livre de estado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.2 Conflitos no AFS em um período de um ano . . . . . . . . . . . . . . . . . . . . . 55
2.3 Tabela de prefixos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
2.4 Atividade dos usuários nos sistemas analisados . . . . . . . . . . . . . . . . . . 65
2.5 Tamanho do cache nos clientes do SPRITE . . . . . . . . . . . . . . . . . . . . . 66
2.6 Fontes de tráfego no cliente . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
2.7 Eficiência do cache dos clientes (%) . . . . . . . . . . . . . . . . . . . . . . . . 67
2.8 Acesso dos processos dos clientes ao sistema de arquivos . . . . . . . . . . . . . 68
2.9 Carga no servidor (%) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
2.10 Acesso a dados desatualizados . . . . . . . . . . . . . . . . . . . . . . . . . . 68
2.11 Tráfego no servidor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
2.12 Tráfego no servidor sem manutenção de consistência . . . . . . . . . . . . . . . 69
2.13 Desempenho do *write-behind* de diretórios . . . . . . . . . . . . . . . . . . . . 92
2.14 Desempenho do ECHO frente ao UNIX local . . . . . . . . . . . . . . . . . . . . 93

3.1 Perfil médio dos programas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.2 Perfil da solicitação de execução de programas . . . . . . . . . . . . . . . . . . 99
3.3 Resultados da simulação do CFAP . . . . . . . . . . . . . . . . . . . . . . . . . 104
3.4 Parâmetros da Rede . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

4.1 Parâmetros do modelo de Gray . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
4.2 Leitura seqüencial sem concorrência . . . . . . . . . . . . . . . . . . . . . . . . 129
4.3 Escrita seqüencial sem concorrência . . . . . . . . . . . . . . . . . . . . . . . . 130

A.1 Principais diretórios do UNIX . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
A.2 Conteúdo de um *i-node* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144

# Capítulo 1

# Introdução

Desde o surgimento dos computadores eletrônicos na década de 40, o tamanho e o preço destas máquinas vêm caindo vertiginosamente. Ao mesmo tempo, o seu poder de computação tem crescido quase que exponencialmente. Este fato possibilitou que, na década de 80, ocorresse uma explosão do número de computadores existentes nas indústrias, estabelecimentos comerciais e nos grandes centros de ensino e pesquisa. Conseqüentemente, veio à tona o desejo de interconectar as diversas máquinas localizadas em um mesmo edifício a fim de compartilhar os recursos disponíveis. Desta forma, surgiram as redes locais e a necessidade do desenvolvimento de software para a administração dos seus recursos.

Inicialmente, foi dada ênfase a problemas simples como a troca de mensagens entre usuários de diferentes máquinas e o compartilhamento de impressoras ou de discos rígidos só para leitura.

Com o passar do tempo, foram elaborados sistemas mais complexos que oferecem ao usuário maior aproveitamento dos recursos distribuídos pela rede. Assim, foram desenvolvidos sistemas onde cada usuário pode ler, criar e alterar arquivos localizados em diferentes pontos da rede. Tudo isto de maneira transparente, isto é, o usuário não toma conhecimento de onde estão armazenados os arquivos que acessa. Independentemente do assento ocupado pelo usuário, a sua visão do sistema de arquivos é sempre a mesma.

Muita pesquisa tem sido feita nos últimos anos com o intuito de criar sistemas operacionais distribuídos onde todas as atividades de um usuário em uma rede sejam distribuídas transparentemente de modo que a eficiência na execução das tarefas seja a maior possível. Já existem em funcionamento sistemas onde o usuário não tem o conhecimento de qual máquina está executando cada tarefa ativada por ele. É o sistema operacional que escolhe, cada vez que uma nova tarefa é iniciada, onde ela será executada. Se apenas um usuário estiver conectado a uma rede com muitas máquinas e ele executar vários comandos de compilação de arquivos grandes, por exemplo, o sistema irá distribuir a tarefa da compilação entre diversas máquinas sem que o usuário se dê conta disso. E mais, se o sistema percebe que alguma das máquinas terminou o seu trabalho e está livre ou com pouca carga, ele executa uma migração de processos, ou seja, ele transfere processos de máquinas que estão sobrecarregadas para máquinas com pouca carga, mais uma vez, de modo transparente ao usuário.

Uma definição mais formal de Sistema Operacional Distribuído é dada por Tanenbaum em [Tan92]:

> "Um Sistema Distribuído é aquele que opera em uma coleção de máquinas que não possuem memória compartilhada mas que se apresentam para os seus usuários como um único computador."

Um sistema distribuído bem implementado poderia ter diversas vantagens sobre um sistema centralizado e sobre conjuntos de máquinas isoladas:

- Em um sistema onde diversos usuários editam arquivos, compilam programas e usam correio eletrônico é muitas vezes mais eficiente e, principalmente, mais barato ter várias estações trabalhando em paralelo do que um computador de grande porte com vários terminais fazendo tudo sozinho.
- Se uma máquina pára, as outras podem continuar o trabalho.
- O compartilhamento de arquivos, bancos de dados ou periféricos, como impressoras ou *plotters*, é simples e natural.
- A distribuição do trabalho pode ser feita eqüitativamente entre os diversos equipamentos disponíveis.
- Em um sistema heterogêneo, cada máquina pode executar tarefas dentro da sua especialidade. Assim, um supercomputador pode ser responsável por executar complicados cálculos matemáticos para a geração de uma animação gráfica enquanto que uma estação gráfica seria responsável pela sua visualização.
- Um sistema distribuído é facilmente expandível em pequenos passos.

Por outro lado, os Sistemas Operacionais Distribuídos estão apenas na sua infância. Atualmente, não existe nenhum sistema com todas as características descritas acima que esteja consolidado com um grande número de usuários. Mas, sem dúvida nenhuma, sistemas operacionais que possibilitem uma boa integração dos computadores de redes locais e de redes de grande área serão os grandes campeões na preferência dos usuários dos sistemas computacionais do futuro.

Neste trabalho, nos concentraremos em apenas um dos aspectos de um sistema operacional distribuído que é o seu sistema de arquivos. A maior parte dos sistemas apresentados a seguir, não possuem todas as características esperadas de um sistema operacional distribuído e, portanto, não podem ser considerados como tais.

Abordaremos, neste capítulo, os principais conceitos básicos dos Sistemas de Arquivos Distribuídos analisando, também, as características desejáveis em tais sistemas bem como as relações entre elas.

No capítulo 2, analisamos os principais Sistemas de Arquivos Distribuídos desenvolvidos nos últimos anos segundo os conceitos definidos no capítulo 1.

No capítulo 3, abordamos a questão da elaboração de modelos analíticos para o desempenho de sistemas distribuídos bem como a implementação de sistemas adaptativos. Para tanto, descrevemos, detalhadamente, dois sistemas adaptativos propostos recentemente.

No capítulo 4, descrevemos o funcionamento dos *leases*, um mecanismo para controle de consistência em sistemas distribuídos. Analisamos um modelo analítico proposto pelo criador do protocolo dos *leases* para a predição do tráfego gerado pelo mesmo. Após indicar os pontos fracos deste modelo, apresentamos o nosso próprio modelo resolvendo os problemas do modelo anterior. Utilizando o nosso modelo, comparamos o desempenho do protocolo dos *leases* com o desempenho do protocolo adotado pelo sistema SPRITE.

Na seção 4.4 apresentamos o sistema SODA, que é a nossa implementação do protocolo dos *leases*. Após a descrição da implementação, apresentamos o resultado de testes avaliando o seu desempenho.

Finalmente, no capítulo 5, relatamos os principais problemas encontrados no desenvolvimento de sistemas distribuídos apresentando possíveis soluções. Concluímos apontando alguns caminhos para os Sistemas de Arquivos Distribuídos do futuro expondo o que deles esperamos.

## 1.1 Conceitos Básicos

Não podemos iniciar nenhuma discussão sobre sistemas de arquivos distribuídos sem definir alguns conceitos fundamentais.

Um **serviço** é um conjunto de facilidades oferecidas aos nós de uma rede por um software que opera em uma ou mais máquinas. Um **servidor** é o software que opera em uma determinada máquina e que trata de oferecer o serviço. Chamaremos, também, de servidor a máquina que executa esse software.

O **cliente** é o software que utiliza o serviço do servidor. Também chamaremos de cliente a máquina que executa o software cliente.

Um **Sistema de Arquivos** é uma parte de um Sistema Operacional que trata de oferecer um repositório de dados de longa duração. Um **Sistema de Arquivos Distribuído** é um Sistema de Arquivos onde vários servidores são responsáveis por oferecer o serviço de arquivos para vários clientes instalados em diferentes máquinas.

Existem sistemas que utilizam a mesma interface dos sistemas de arquivos para outros propósitos além do armazenamento de dados. O UNIX possui certos arquivos especiais que, na realidade, funcionam como canais de comunicação com dispositivos de entrada e saída[1].

Algumas implementações do UNIX (como o LINUX, por exemplo) possuem o diretório */proc* que nada mais é do que uma interface com o núcleo do sistema operacional (*kernel*) que facilita a obtenção de informações sobre o estado do sistema.

O sistema distribuído Plan9 vai ainda mais fundo na utilização deste tipo de interface. Nesse sistema é possível acessar dispositivos de entrada e saída locais ou remotos ou executar funções relativas ao sistema de janelas utilizando interfaces que imitam um sistema de arquivos.

Os sistemas que apenas aproveitam a interface dos sistemas de arquivos não serão objeto de estudo desta dissertação assim como os Sistemas de Arquivos Virtuais[2].

Nas modernas redes de computadores existem diversos tipos de serviços oferecidos pelos servidores: serviço de nomes, serviço de impressão, serviço de correio, e, obviamente, serviço de arquivos. Neste trabalho, nos concentraremos apenas no serviço de arquivos que é oferecido por uma ou mais máquinas às quais estão conectadas unidades de armazenamento de grande quantidades de dados como discos rígidos magnéticos ou ópticos. Os clientes utilizam este serviço enviando mensagens aos servidores solicitando informações sobre os arquivos armazenados nos seus discos ou alterando o seu conteúdo.

No decorrer deste capítulo apresentaremos conceitos de sistemas de arquivos distribuídos que servirão de base para as análises dos capítulos subseqüentes.

### 1.1.1 Nomes e Localização

Já faz muito tempo que a estrutura de árvore de diretórios é utilizada para organizar o acesso dos usuários aos arquivos em disco. Hoje em dia é a estrutura mais difundida embora existam outras possibilidades em estudo[3]. O uso de árvores de diretórios se popularizou com os sistemas UNIX e DOS.

No UNIX (ver apêndice A), cada arquivo é identificado pelo seu nome (uma seqüência de caracteres) e pelo caminho (*path*) até ele. O caminho é uma seqüência de diretórios separados

[1] Um exemplo é o arquivo */dev/console*. Os bytes escritos neste arquivo são ecoados como caracteres no console da máquina.

[2] Um Sistema de Arquivos Virtual (*RAM DISK*) oferece um serviço muito rápido armazenando todas as informações em memória volátil. Por conseguinte, todos os dados se perdem quando acontece uma queda repentina do sistema.

[3] Ver [GJSJ91]

pelo caractere / que indicam a localização lógica do arquivo. Agrupando os arquivos em diretórios e subdiretórios facilita-se a sua manipulação por parte dos usuários. Acrescentando-se o nome do arquivo ao *path* obtém-se o *pathname*.

Dada a localização lógica de um arquivo, isto é, o caminho até ele, é necessário analisar os componentes deste caminho a fim de encontrar a localização física do arquivo. É preciso descobrir em quais blocos de quais discos de quais servidores se encontra o arquivo em questão.

Quando os arquivos de um sistema estão distribuídos entre vários servidores localizados em diferentes máquinas, é desejável que a localização destes arquivos seja **transparente** [Flo89] aos usuários do sistema. Em outras palavras, quando o usuário desejar ter acesso a um determinado arquivo, ele não deve ter que se preocupar com a localização física do arquivo. Assim, não precisaria saber em qual disco de qual servidor ele está guardado.

Da mesma forma, quando um cliente deseja guardar informações em um arquivo em disco, ele deve apenas fornecer o caminho e o nome do arquivo no qual elas devem ser gravadas.

### 1.1.2 Cache

"Metido numa sobrecasaca cor de rapé,
cabelo negro, longo e cacheado"
Várias Histórias – Machado de Assis

Quase a totalidade dos sistemas de arquivos distribuídos seguem o modelo cliente-servidor. Segundo este modelo, os processos que são executados na máquina cliente fazem solicitações de acesso ao sistema de arquivos ao sistema operacional local que, por sua vez, as remete a uma máquina remota (o servidor) através da rede.

No entanto, o caminho por onde as solicitações trafegam apresenta dois pontos em que a taxa de transferência de informações é pequena se comparada à velocidade dos processadores atuais e ao tempo de acesso à memória principal.

O primeiro gargalo (*bottleneck*) é o acesso ao disco do servidor. A fim de acessar os dados armazenados em um disco físico, é necessário mover o braço de leitura e gravação para a trilha apropriada, esperar até que o setor desejado passe em baixo da cabeça e só então iniciar a leitura dos dados armazenados no ritmo determinado pela velocidade de rotação do disco. Todo este processo costuma ser muito mais lento do que o acesso à memória principal.

O outro gargalo é o acesso ao servidor através da rede. Com os grandes avanços na tecnologia de redes de computadores, a sua capacidade de transferência de informações tem crescido enormemente. Mas, em muitos casos, a transferência de dados entre duas máquinas de uma rede é consideravelmente mais demorada do que a transferência de dados dentro da memória principal[4].

Além disso, se uma rede é compartilhada por muitas máquinas produzindo uma carga muito grande, o tempo médio de envio de mensagens aumenta podendo gerar, eventualmente, congestionamentos.

Tudo isso encoraja a utilização da técnica de **cacheamento** que consiste em armazenar os dados mais "importantes" em uma porção da memória de acesso relativamente rápido chamada de cache[5]. Com isso, espera-se atender ao maior número possível de solicitações de acesso aos dados através desta memória de acesso rápido.

O tipo mais comum de cache encontrado na literatura é aquele utilizado pelos processadores quando acessam a memória principal[Smi82]. Os computadores atuais possuem uma pequena porção de sua memória cuja capacidade de transferência de dados é muito grande.

[4]Em uma máquina SPARCserver 1000 executando o sistema operacional SunOS 5.3, o acesso a 1Kbyte demora cerca de 1 milisegundo na memória principal, 10 milisegundos nos discos locais e 90 milisegundos em servidores remotos através de uma rede Ethernet.

[5]O termo cache é derivado do francês *cacher* que significa esconder.

Mantendo os dados mais acessados pelo processador nesta porção da memória, obtém-se um grande aumento no desempenho da máquina. Esta memória de alto desempenho é chamada de memória cache e o único fator limitante para a sua utilização é o seu custo elevado.

No contexto de sistemas de arquivos distribuídos, é possível obter grandes ganhos em desempenho através da utilização de caches no cliente e no servidor. Cada um destes tipos de cache ataca um dos gargalos do sistema de arquivos. A figura 1.1 mostra esquematicamente o ponto de atuação destes caches.

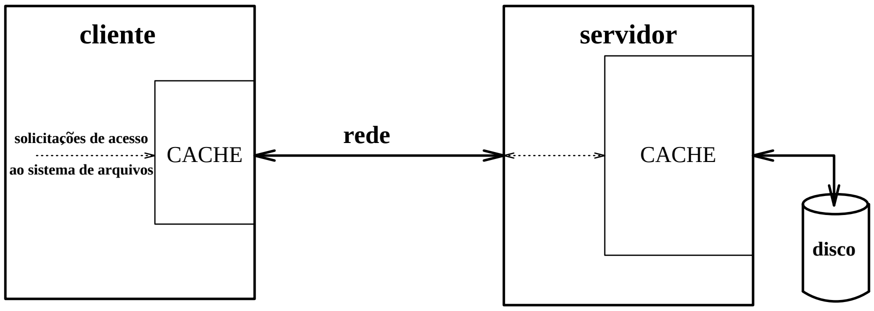



Figura 1.1: Caches em sistemas de arquivos distribuídos

O cache do servidor é armazenado na memória principal e tem como finalidade evitar os acessos ao disco. Já o cache do cliente pode ser armazenado tanto na memória principal quanto em memória secundária (discos locais) e tem como finalidade evitar acessos ao servidor através da rede.

Como os caches não dispõem de espaço suficiente para armazenar todos os arquivos do sistema, é necessário um algoritmo para determinar quais arquivos devem ser cacheados, isto é, permanecer no cache.

O ideal seria manter, no cache, os arquivos que serão acessados em seguida e descartar os arquivos que não serão acessados mais. Mas, como não é possível prever quais serão as futuras solicitações de acesso aos arquivos, é comum a adoção da técnica LRU (*Least Recently Used*) para escolher quais blocos serão cacheados. Segundo esta técnica, sempre que um novo bloco de arquivo que não está no cache é acessado, este bloco é incorporado ao cache. Caso o espaço destinado ao cache termine, o bloco que foi acessado pela última vez há mais tempo é descartado liberando o espaço que ocupava. Caso este bloco tenha sido alterado, é necessário enviar as alterações para o servidor (no caso do cache do cliente) ou para o disco (no caso do cache do servidor) antes de descartá-lo.

### Consistência

A introdução do cache nos clientes traz consigo o perigo da perda da consistência entre as cópias de um mesmo arquivo em diferentes clientes.

Suponha, por exemplo, que um cliente $c1$ solicita a leitura de um arquivo do servidor. Após receber o arquivo, $c1$ armazena em seu cache uma cópia deste a fim de atender futuras leituras. Se, em seguida, um outro cliente $c2$ modifica o arquivo em questão, a cópia armazenada no cache de $c1$ se torna desatualizada. É preciso adotar algum mecanismo de controle da consistência a fim de evitar que uma leitura subseqüente efetuada por um processo de $c1$ acesse os dados desatualizados que estão no seu cache.

Este problema pode ser abordado de várias maneiras. Sistemas como o NFS (ver seção 2.3) e o ANDREW (seção 2.4) limitam mas não evitam a falta de coerência entre os caches dos clientes. Já o objetivo do SPRITE (seção 2.6) e do SODA (seção 4.4) é oferecer a mesma consistência que os sistemas centralizados.

Na seção 1.1.9 discutiremos os problemas que surgem quando diferentes processos acessam concorrentemente os mesmos arquivos. Estudaremos, nos capítulos 2 e 4, as principais técnicas empregadas no controle da consistência.

#### Cachear é realmente necessário?

A manutenção de caches no cliente e no servidor acrescenta, obviamente, complexidade ao sistema. Quando vários clientes acessam o mesmo conjunto de dados, existe a necessidade adicional de manter a consistência entre os seus caches. Será que toda esta sobrecarga acaba fazendo com que os caches deixem de ser um bom negócio? A experiência mostra que não.

Desde meados da década de 80, estudos já demonstravam o grande ganho em desempenho que os caches nos sistemas de arquivos proporcionavam. [Smi85] estudou o comportamento de sistemas acadêmicos centralizados compartilhados por vários usuários e chegou a conclusão de que, se se reservassem 8 Mbytes para o cache do sistema de arquivos, de 80 a 90% dos acessos ao disco poderiam ser evitados oferecendo um grande ganho em desempenho. Segundo [NWO88] cerca de 70% do tráfego entre os clientes e servidores pode ser evitado com caches de 8Mbytes na memória principal dos clientes. Uma boa parte dos arquivos temporários nem chegariam a ser enviados para o servidor. Já [HKM$^+$88] mostra que grandes caches em discos locais dos clientes podem atender a cerca de 80% das solicitações de acesso ao sistema de arquivos[6].

Desde aquela época, o tamanho médio dos arquivos acessados pelos usuários vem aumentando muito. Segundo [Ous90], os progressos na tecnologia dos acionadores de disco não têm acompanhado o crescimento do poder de computação dos processadores principalmente depois da introdução da tecnologia RISC. Desta forma, a diferença entre a velocidade de acesso a dados em discos e a dados no cache só tem aumentado, o que incentiva cada vez mais a utilização de caches sempre que possível.

Atualmente, qualquer sistema de arquivos de uso geral, distribuído ou não, que pretenda ser considerado sério deve, obrigatoriamente, conter um sistema de cache.

### 1.1.3 Disponibilidade

Todo usuário de redes de computadores já teve a infelicidade de perceber que algum serviço oferecido pela sua rede não estava disponível. Falhas de hardware ou de software e falta de energia elétrica podem levar à **queda** (*crash*) de uma máquina, isto é, a uma interrupção momentânea no seu funcionamento. A queda de um nó ou uma falha em um canal de comunicação pode levar a uma **partição** na rede, isto é, a comunicação entre dois pontos da rede é interrompida durante um certo intervalo de tempo.

A grande maioria dos sistemas de arquivos distribuídos em operação são muito sensíveis a partições na rede. Se um cliente não consegue estabelecer contato com alguns dos servidores, os processos que fizeram solicitações ao serviço de arquivos são notificados de que as informações solicitadas não estão disponíveis ou simplesmente são bloqueados até que a conexão se estabeleça.

O método mais utilizado para aumentar a disponibilidade de um serviço de arquivos tem sido a replicação de dados. Os arquivos são armazenados em dois ou mais servidores; se um dos servidores não está disponível, algum outro servidor poderá fornecê-los.

[6] Obviamente, estas taxas de acerto dos caches dependem do tipo de aplicação que é executada sobre o sistema de arquivos. Os artigos citados estudaram o comportamento médio de centros de computação de universidades.

O sistema CODA (ver seção 2.5) levou este método às últimas conseqüências introduzindo o conceito de **Operação Desconectada** como veremos no próximo capítulo. Neste modo de operação, um cliente pode fazer alterações em arquivos mesmo sem estabelecer contato com os servidores. Obviamente, isso pode trazer sérias conseqüências para a consistência dos dados. Dois clientes desconectados podem fazer alterações em um mesmo arquivo simultaneamente. Quando a rede se recupera da partição o sistema é obrigado a descartar alterações realizadas por um dos clientes. Os usuários deste sistema precisam estar conscientes dos perigos que enfrentam para não sofrerem surpresas desagradáveis.

Outros sistemas que replicam os dados como o ECHO e o HARP (seções 2.10 e 2.8 respectivamente) oferecem garantias de consistência dos dados mesmo com partições na rede. Como era de se esperar, nestes sistemas a disponibilidade não é tão grande quanto no CODA.

### 1.1.4 Escalabilidade

A grande maioria dos sistemas de arquivos distribuídos foram projetados para pequenas redes locais com, no máximo, algumas dezenas de máquinas. Mas, com o barateamento do hardware, o número de nós nas redes tem crescido. Paralelamente, tem aumentado o desejo de compartilhar arquivos não só localmente como também entre máquinas de diferentes redes locais que possuam alguma interconexão.

A menos que se tenha este fato em mente no momento do projeto, dificilmente um sistema de arquivos distribuído apresentará um bom comportamento em redes de grande escala com centenas ou milhares de clientes e dezenas de servidores. Em redes com muitos nós, podem aparecer problemas decorrentes do grande número de clientes e do grande número de servidores. Vejamos quais são eles.

Quase sempre, os servidores guardam informações sobre os clientes que acessam os seus arquivos. Tais informações podem ser úteis, por exemplo, para evitar que clientes desautorizados tenham acesso a dados confidenciais ou alterem dados importantes como veremos na seção 1.1.6. Mas, a sua principal função é garantir a consistência dos arquivos cacheados por mais de um cliente como vimos em 1.1.2.

Se um servidor precisa atender a centenas de clientes, o espaço gasto pelas tabelas nas quais ele guarda estas informações pode ficar muito grande comprometendo a operacionalidade do sistema. Além disso, se um arquivo que é cacheado por centenas de clientes sofre uma alteração, o servidor pode gastar muito tempo para invalidar as cópias presentes nos caches de todos os clientes, o que seria indesejável[7].

Outro problema decorrente do fato do servidor guardar muitas informações sobre os seus clientes ocorre quando ele sofre uma queda e perde todas as informações de sua memória volátil. Neste caso, o tempo necessário para o sistema se recuperar da queda pode ser muito grande.

Por outro lado, quando o número de servidores é grande, fica mais difícil descobrir qual é o servidor responsável por um determinado arquivo. Se, cada vez que um cliente precisar abrir um novo arquivo, ele perguntar para cada servidor se ele é o responsável pelo arquivo, certamente haverá um enorme congestionamento na rede. Se uma das máquinas da rede for responsabilizada pela tradução dos *pathnames* nos respectivos endereços físicos, então esta máquina vai ser sobrecarregada.

O sistema ANDREW (seção 2.4) foi o primeiro a oferecer uma solução satisfatória para o problema através da descentralização das informações e da hierarquização da rede. Hoje em

[7] Um bom exemplo de arquivos que são cacheados em muitos clientes e alterados com certa freqüência são os arquivos que compõem o sistema de notícias de redes locais. Novas notícias podem ser introduzidas diariamente pelos administradores da rede e lidas, através do sistema de arquivos, por usuários em todos os clientes.

dia, o ANDREW integra milhares de computadores em dezenas de universidades e centros de pesquisa americanos, europeus e japoneses. No entanto, esta grande escalabilidade foi obtida reduzindo-se a consistência e a transparência de localização.

### 1.1.5 Heterogeneidade

Uma das grandes deficiências dos sistemas de computação tem sido a incompatibilidade tando do hardware quanto do software fabricado por diferentes companhias. Cada vez mais, buscam-se padronizações que permitam que máquinas de diferentes fabricantes se comuniquem sem grandes dificuldades.

Uma rede **heterogênea**, isto é, que possui nós de diferentes tipos, pode ser vantajosa na medida em que cada tipo de tarefa pode ser executada em uma máquina apropriada.

Consideremos, por exemplo, um agência de publicidade. O diretor de arte precisa de uma estação gráfica de alto desempenho para processar as suas imagens, o redator se contenta com uma máquina que ofereça editoração eletrônica. Já o diretor de criação, que é o responsável pelo produto final, precisa de uma máquina que combine o texto com as imagens. Quando estas máquinas se tornam obsoletas, ao invés de irem para a lata do lixo, podem ser úteis para as secretárias digitarem pequenos textos.

Seria muito bom para a eficiência desta empresa que as suas máquinas fossem interligadas oferecendo serviços como correio eletrônico e compartilhamento de arquivos mesmo possuindo hardware e sistemas operacionais distintos.

Sistemas que permitam a utilização de redes heterogêneas facilitam a instalação de redes de grande escala pois, em sistemas deste tipo, qualquer máquina pode utilizar ou oferecer os serviços distribuídos e não apenas um modelo de máquina de um determinado fabricante.

Entre os sistemas de arquivos distribuídos em uso, apenas o NFS (seção 2.3) oferece boas chances de integração em redes heterogêneas. Juntamente com o lançamento do NFS em 1985, a empresa *SUN Microsystems* divulgou a especificação do *protocolo NFS*. Em posse deste protocolo, qualquer um pode desenvolver clientes e servidores NFS que se comuniquem com a implementação da SUN.

Hoje em dia, existem implementações do NFS para dezenas de sistemas. Alguns exemplos são: MS-DOS, MacOS, VAX/VMS além das principais implementações do UNIX.

### 1.1.6 Segurança

Uma das grandes vantagens dos sistemas de arquivos distribuídos é que eles permitem o compartilhamento dos dados de diversos usuários. Mas isso é uma faca de dois gumes pois dados secretos, como o movimento de contas bancárias, informações sobre declarações de renda à receita federal, ou mesmo as provas que um professor aplicará no dia seguinte, são informações que o dono não gostaria de compartilhar com qualquer um. É necessário adotar mecanismos que garantam que pessoas não autorizadas não tenham acesso a tais dados.

O sistema UNIX adota um método baseado em **permissões** para controlar o acesso aos arquivos (ver apêndice A). Cada arquivo possui uma série de bits de permissão que indicam quais usuários podem acessar o arquivo e de que maneira.

Em sistemas distribuídos implementados sobre o UNIX, ao receber uma solicitação de dados de um determinado arquivo, o servidor recebe também a identificação de quem está fazendo essa solicitação. O servidor consulta os bits de permissão para verificar se o cliente poderá, ou não, efetuar a operação solicitada.

Cada máquina UNIX possui um usuário especial chamado *root* que pode alterar qualquer tipo de permissão e, portanto, ter acesso ilimitado a qualquer arquivo. Este sistema funciona bem em sistemas isolados mas, em redes locais, começam a surgir problemas.

Para facilitar o funcionamento de redes locais, costuma-se configurar o sistema de modo que as máquinas confiem no *root* de máquinas vizinhas. Por exemplo, se o *root* de um cliente remoto do sistema de arquivos desejar alterar arquivos pertencentes ao *root* local ele terá essa liberdade. Assim, é preciso tomar muito cuidado pois se um usuário mal intencionado consegue acesso como *root* a uma máquina da rede que é considerada confiável pelas demais máquinas, então este usuário terá acesso ilimitado a todas as máquinas da rede. Um problema semelhante pode ocorrer se um usuário consegue fazer solicitações a um servidor de arquivos fingindo ser outro usuário.

Outra maneira de se controlar o acesso aos arquivos é basear o acesso em **capacidades** (*capabilities*). Nos métodos baseados em capacidade, o cliente precisa apresentar ao servidor uma prova de que ele possui a capacidade de acessar um determinado arquivo. Em geral, o cliente apresenta a sua identificação (e, possivelmente, a identificação do usuário que fez a solicitação) quando pede a abertura do arquivo, recebendo em troca um código que é a prova de que ele possui a capacidade de acessar o arquivo. Nos contatos subseqüentes, o cliente não precisa mais se identificar, bastando apresentar a prova da sua capacidade. É preciso tomar cuidado para que não seja possível forjar provas de capacidade falsas.

A segurança dos dados fica comprometida se, no caminho entre duas máquinas confiáveis, existir uma máquina não confiável. Um intruso pode se apoderar desta máquina e interceptar as mensagens entre as duas máquinas descobrindo informações secretas ou, até mesmo, adulterá-las. A solução mais comum para este problema é criptografar as mensagens antes de transmiti-las. O sistema de arquivos SWALLOW [Svo84], por exemplo, armazena e transmite os arquivos criptografados. O SWALLOW funciona como se fosse um sistema baseado em capacidades onde a chave criptográfica é a prova da capacidade. Uma vantagem deste método é que o servidor não precisa verificar se a prova da capacidade é autêntica, se ela não for correta, o cliente não conseguirá decodificar os dados que recebeu.

Existe ainda um método que se baseia em **listas de controle de acesso**. Inicialmente utilizado pelo MULTICS, este método é hoje utilizado pelo ANDREW como veremos na seção 2.4.4.

Embora possa parecer pouco provável que intrusos atrapalhem a vida de usuários comuns, eles podem estar mais próximos do que imaginamos. Freqüentemente, surgem notícias de que foram detectadas ações de intrusos monitorando o tráfego na rede internacional Internet. Maiores informações sobre segurança em redes de computadores podem ser obtidas em [Sit91].

### 1.1.7 Tolerância a Falhas

Uma boa parte da complexidade de um sistema de arquivos distribuídos decorre do fato de ele ser implementado sobre uma rede de computadores que não é totalmente confiável.

É necessária a utilização de protocolos que possibilitem a detecção de erros na transmissão dos dados. Tais protocolos devem acionar a retransmissão das mensagens que chegarem adulteradas ao seu destino e das mensagens que não forem entregues ao seu destinatário.

Tanto os clientes quanto os servidores de arquivos podem sofrer quedas e romperem a comunicação com os outros nós da rede por intervalos que podem variar de alguns segundos até várias horas. Congestionamentos na rede podem causar atrasos na entrega de mensagens. A rede pode, também, sofrer uma partição, ou seja, por algum motivo a comunicação entre duas máquinas da rede é interrompida durante um determinado período.

Mas não é só a rede que não é imune a falhas. Qualquer mecanismo de armazenamento permanente de dados está sujeito a erros de leitura ou de escrita, sejam eles discos magnéticos rígidos ou flexíveis, discos ópticos, fitas magnéticas, etc.

O sistema deve evitar, sempre que possível, que estas falhas causem uma depreciação muito grande no tempo de resposta aos clientes ou que o serviço seja interrompido, em outras palavras, gostaríamos que a **disponibilidade** do serviço não fosse afetada.

Outro aspecto importante é a **confiabilidade**, isto é, gostaríamos que estas falhas não ameaçassem a integridade dos dados armazenados.

Muitas vezes, essas características são conflitantes. Por exemplo, uma maneira de garantir a integridade dos dados na presença de erros de escrita em disco é manter informações redundantes sobre esses dados. Mas, a manutenção da redundância aumenta a carga no servidor e, portanto, retarda as respostas aos clientes.

Além da replicação, as transações também contribuem para uma melhor tolerância a falhas em troca de um grande aumento na complexidade dos servidores. Discutiremos estes métodos a seguir.

### 1.1.8 Operações Atômicas

Uma operação sobre um arquivo é dita **atômica** quando os passos que compõem esta operação não podem ser observados por nenhum processo exterior a esta operação. Externamente, o arquivo apresenta um estado antes da operação e outro estado depois da operação sem passar por nenhum estado intermediário perceptível externamente. Se alguma parte da operação falha, o arquivo permanece completamente inalterado.

[Ree83] destaca os dois aspectos principais da atomicidade:

- **Atomicidade em relação a concorrência:** dada uma ação fora de uma operação atômica, ou esta ação precede todos os passos da operação ou ela sucede todos os passos.

- **Atomicidade em relação a falhas:** ou todos os passos de uma operação atômica são executados com sucesso ou nenhum o é.

Operações de leitura, escrita, criação ou eliminação de um arquivo são comumente implementadas de modo a apresentar este caráter atômico. As **transações** são mecanismos que permitem que seqüências de operações sejam executadas atomicamente.

A maior parte dos sistemas que oferecem transações dispõem de comandos de *início* e *fim da transação*. Em geral, quando um processo inicia uma transação, ele recebe um identificador da transação que é usado em todas as operações que devem ser executadas dentro dela. Se, dentro de uma transação, um erro é detectado, o processo pode executar o comando *aborta transação* que descarta todas as operações realizadas dentro da transação. Se o processo consegue executar todas as operações com sucesso, ele pode chamar *fim da transação* que efetivamente compromete (*commit*) as alterações realizadas na transação.

Assim, as transações implementam a semântica de tudo ou nada. Ou todas as operações são concluídas com sucesso, ou então, nenhuma. Isso faz das transações um importante mecanismo de tolerância a falhas. Elas evitam que pequenas falhas prejudiquem a integridade do sistema.

Apesar de constituírem uma exigência praticamente obrigatória em sistemas de bancos de dados [BH87], as transações não são implementadas com muita freqüência no núcleo dos sistemas de arquivos. Os sistemas LOCUS [MMP83] e QuickSilver [SW91] são duas exceções a esta regra. O primeiro artigo publicado na revista *ACM Transactions on Computer Systems* [Ree83] apresenta um mecanismo de implementação de transações atômicas em sistemas distribuídos.

### 1.1.9 Acesso Concorrente

Um aspecto fundamental a se considerar nos sistemas de arquivos, principalmente quando distribuídos, é o controle ao acesso concorrente. Quando diversos processos acessam os mesmos dados simultaneamente, é preciso tomar cuidado para que esses processos recebam informações corretas e também para que a consistência do sistema de arquivos não seja afetada.

Considere, por exemplo, a seguinte situação. Um banco guarda em um arquivo os saldos das contas correntes de seus clientes. Um determinado cliente possui um saldo de *$ 1.000 e efetua um saque de *$ 500. Neste mesmo instante, o seu salário, de *$ 10.000 é creditado. Suponha que estas duas operação são realizados por aplicações distintas. O que poderá acontecer se o sistema não for bem implementado?

Suponha que as duas aplicações leiam simultaneamente do arquivo de saldos o valor 1.000 guardando-o em variáveis internas de cada aplicação. A operação de crédito de salário soma 10.000 a esse valor e guarda o novo saldo de *$ 11.000 no arquivo de saldos. Logo depois, a operação de saque subtrai 500 dos 1.000 que estavam registrados na sua variável local e guarda no arquivo o valor 500 (apagando o crédito do salário). O saldo registrado depois das duas operações poderá agradar o dono do banco mas, sem dúvida nenhuma, provocará protestos do cliente. Ao invés de um saldo de *$ 10.500, o seu extrato indicará um saldo de *$ 500!

Para evitar este tipo de problema, as aplicações que acessam simultaneamente os mesmos dados podem agrupar as operações com esses dados em transações e o sistema operacional deve fazer com que estas transações concorrentes tenham uma visão consistente dos dados com que operam. [Tan92] apresenta na seção 11.4.4 três mecanismos de controle de concorrência.

Os **bloqueios** (*locks*) são os mecanismos mais amplamente utilizados para o controle da concorrência. Antes de acessar um determinado arquivo, um processo pode bloquear o acesso a este arquivo através de um comando do sistema operacional. Se outro processo pede um bloqueio para o mesmo arquivo, ele recebe uma mensagem indicando que o acesso àquele arquivo já está bloqueado, ou então, fica pendurado até que o primeiro processo, detentor do bloqueio, execute o comando de liberação de bloqueio[8]. Há ainda a possibilidade de um processo bloquear apenas as escritas em um arquivo deixando livres as consultas ao seu conteúdo.

Através dos bloqueios é possível tornar as transações serializáveis. Isto ocorre quando o resultado de transações ativadas simultaneamente é igual ao resultado que seria obtido se elas fossem executadas em série, uma sendo iniciada após o término da outra. Um protocolo simples para a serialização de transações concorrentes é o protocolo de bloqueio em duas fases (*two-phase locking protocol*). Neste protocolo, quando uma transação começa, todos os arquivos que serão acessados no decorrer da transação são antecipadamente bloqueados (a primeira fase). Em seguida, as operações da transação são executadas. Quando a transação termina o seu trabalho, todos os bloqueios são liberados (segunda fase).

Este protocolo pode gerar um impasse (*deadlock*). Ou seja, uma situação na qual um processo espera pela liberação de um bloqueio por um segundo processo que por sua vez espera pela liberação de um outro bloqueio que de algum modo depende da liberação de bloqueios do primeiro processo.

Vejamos um exemplo bem simples onde ocorre *deadlock*[9]. Suponha que duas transações A e B precisem de acesso exclusivo aos arquivos *arq1* e *arq2* como mostra a tabela 1.1.

Se A e B são executadas simultaneamente, pode ocorrer um impasse no caso de a transação A conseguir bloquear *arq1* e a transação B bloquear *arq2*. Nenhuma das duas transações conseguirá continuar o seu trabalho.

Ao observarmos este exemplo poderíamos ficar com a falsa impressão de que *deadlocks* são causados por falhas na programação que poderiam ser facilmente evitadas. De fato, o impasse

[8] Neste último caso, estamos supondo que apenas dois processos estão pedindo bloqueios para este arquivo. Se vários processos pedem bloqueios para o mesmo arquivo, eles são inseridos em uma fila e vão sendo atendidos à medida em que os bloqueios vão sendo liberados.

[9] [MOO87] contém uma série de exemplos de *deadlocks* em diversos contextos.

| Transação A | Transação B |
|---|---|
| bloqueia *arq1* | bloqueia *arq2* |
| bloqueia *arq2* | bloqueia *arq1* |
| processa arquivos | processa arquivos |
| desbloqueia *arq1* | desbloqueia *arq2* |
| desbloqueia *arq2* | desbloqueia *arq1* |

Tabela 1.1: *Deadlock* no protocolo de duas fases

do exemplo poderia ser evitado se os arquivos fossem bloqueados na mesma ordem pelas duas transações. Mas, como dissemos, este é um exemplo muito simples. A ocorrência de *deadlocks* é um fato real em sistemas complexos. Fica difícil prever como se dará a utilização de todos os recursos do sistema.

Para evitar tais impasses é necessário adotar técnicas automáticas como detecção de *deadlocks*, previsão de possíveis *deadlocks* ou utilização de bloqueios com limite de tempo. Estes mecanismos exigem a capacidade de abortar transações a fim de liberar bloqueios. Uma descrição das técnicas de combate aos *deadlocks* em sistemas distribuídos pode ser encontrada em [Tan92], seção 11.5.

### 1.1.10 Semântica do Acesso Concorrente

Em sistemas de arquivos distribuídos, o problema do acesso concorrente é ainda mais grave pois os clientes podem possuir, em seus caches, cópias locais (possivelmente desatualizadas) de dados remotos. É desejável que o sistema distribuído apresente a mesma consistência dos sistemas centralizados.

Como observamos na seção 1.1.2, mecanismos de controle da consistência do cache devem evitar que clientes acessem dados desatualizados mesmo que os seus processos não estejam utilizando transações ou bloqueios.

Existem duas principais semânticas de acesso concorrente a arquivos. Elas indicam qual o resultado esperado quando vários clientes acessam os mesmos arquivos simultaneamente:

- **Semântica UNIX**: Qualquer leitura de dados de um arquivo é capaz de observar as alterações realizadas por qualquer escrita que o anteceda independentemente de qual cliente realizou cada operação. O sistema distribuído se comporta como se fosse uma única máquina UNIX.

- **Semântica de Sessão**: Escritas a um arquivo são visíveis imediatamente apenas aos processos sendo executados no mesmo cliente que realizou a escrita. Quando o arquivo é fechado através do comando *close*, as alterações são enviadas para o servidor e passam a ser visíveis para os clientes que abrirem o arquivo a partir daquele momento.

Nos capítulos seguintes, serão descritas e discutidas algumas técnicas para o controle da consistência do cache através da implementação da semântica UNIX, semântica de sessão ou de uma variante.

### 1.1.11 Replicação

A manutenção de cópias de um mesmo arquivo em diversos servidores de um sistema distribuído proporciona importantes vantagens:

- Se um disco é danificado, as informações nele contidas não são perdidas, podendo ser obtidas de outros discos em outros servidores. Em alguns lugares como, por exemplo, a Califórnia, onde o solo não é dos mais estáveis, é inclusive recomendável que os arquivos importantes sejam replicados em servidores localizados a uma grande distância um dos outros.

- Se um servidor está momentaneamente inoperante ou inacessível, os seus arquivos podem ser acessados em servidores alternativos. Em outras palavras, há uma maior disponibilidade do serviço de arquivos.

- Diretórios muito lidos como, por exemplo, o diretório /usr/bin do UNIX ou um banco de dados bibliográfico de uma universidade podem ser oferecidos por vários servidores. Desse modo, a alta demanda é distribuída eqüitativamente entre os diversos servidores aumentando o desempenho global do sistema.

Por outro lado, é claro que a replicação possui um problema intrínseco que é a manutenção da consistência entre as réplicas dos arquivos nos diversos servidores. No capítulo seguinte, veremos como sistemas como o HARP, o ECHO e o CODA resolvem este problema.

Note que o cache também pode ser encarado como uma forma de replicação de arquivos. No entanto, o principal objetivo do cache é aumentar a rapidez do serviço enquanto que o principal objetivo da replicação é oferecer uma maior tolerância a falhas.

### 1.1.12 Como tudo isso se relaciona?

Todos os conceitos apresentados neste capítulo formam uma complexa teia de inter-relações. Seria desejável que um sistema de arquivos distribuídos fosse eficiente sob todos os aspectos descritos mas, infelizmente, isso não é possível. Quando um sistema é elaborado de modo a ter um excelente desempenho em relação a um determinado conceito, ele tende a apresentar um desempenho fraco se analisado sob outros aspectos. É o que se costuma chamar de *tradeoff*.

A figura 1.2 é uma representação gráfica das principais relações entre os conceitos de sistemas de arquivos distribuídos. Uma linha contínua de $x$ para $y$ significa, grosso modo, que se aumentarmos $x$ poderemos esperar um aumento de $y$. Já uma linha descontínua ligando $x$ a $y$ indica que um aumento de $x$ tende a provocar uma diminuição de $y$.

As relações entre os conceitos podem, ainda, variar de acordo com as implementações. Os rótulos nas flechas da figura indicam o nome de um sistema no qual a relação sugerida pela flecha se verifica. Todos estes sistemas serão descritos detalhadamente no capítulo 2.

A relação entre replicação e rapidez é especialmente interessante. No sistema CODA (seção 2.5), a replicação permite o acesso à cópia mais próxima de um determinado arquivo. Além disso, as alterações são propagadas assincronamente possibilitando uma maior rapidez.

Já o HARP (seção 2.8) exige a comunicação síncrona com muitos dos servidores que replicam um determinado arquivo. Conseqüentemente, ele oferece um serviço mais confiável (tolerante a falhas) mas com menor rapidez.

A relação entre heterogeneidade e escalabilidade, por sua vez, deve ser analisada com cautela. Um sistema que permita a integração de equipamentos que diferem tanto em hardware quanto em software facilita a implantação de redes de grande escala simplesmente porque qualquer máquina disponível pode ser agregada ao sistema como ocorre com o NFS (seção 2.3). Isto não quer dizer que o NFS ofereça um bom desempenho em redes de grande escala. Portanto, a heterogeneidade favorece a escalabilidade apenas parcialmente.

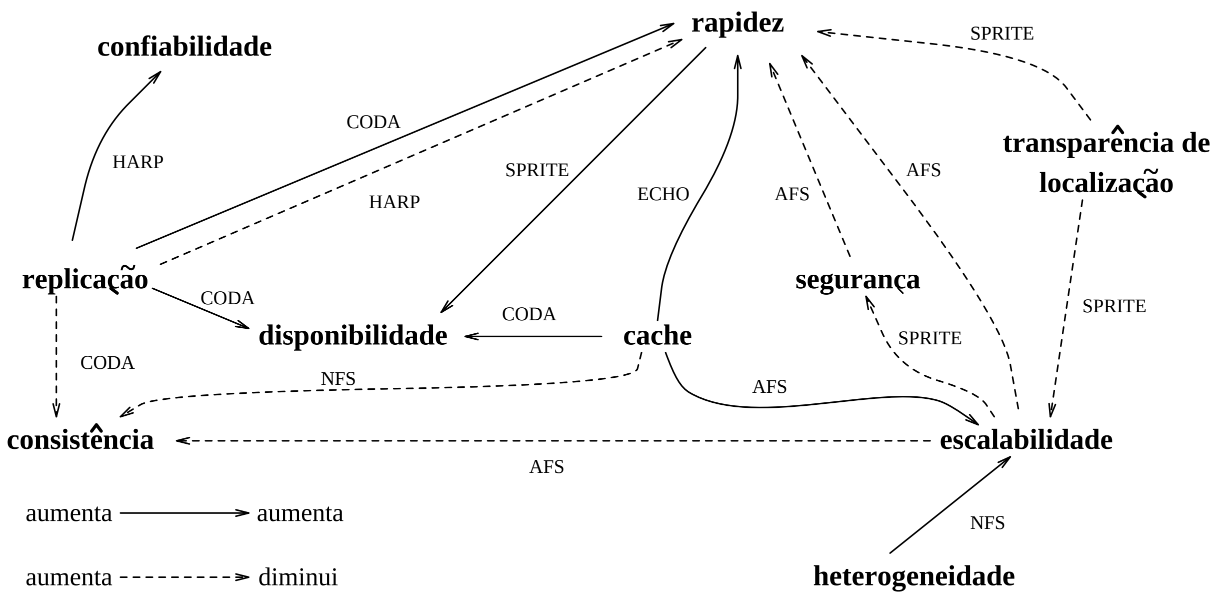



Figura 1.2: Relações entre os Conceitos Básicos

# Capítulo 2

# Estudo de casos

## 2.1 Um Pouco de História

A grande maioria dos trabalhos sobre sistemas de arquivos distribuídos foram escritos a partir da segunda metade da década de 80. Atualmente existem dezenas destes sistemas implementados.

Nos primórdios da computação distribuída destaca-se a *ARPANET*, uma rede construída pelo Departamento de Defesa dos Estados Unidos que entrou em operação em dezembro de 1969. Nesta rede, o *Datacomputer* oferecia um repositório de dados para os computadores que não possuíam capacidade de armazenamento adequada [Svo84]. O Datacomputer, que oferecia um serviço semelhante ao *ftp* atual[1], era executado em um PDP-10 e entrou em operação no fim de 1973.

Dois anos mais tarde, foi a vez do *Interim File Server* (IFS) entrar em operação. Desenvolvido na linguagem BCPL (uma ancestral da C) por pesquisadores do *Xerox Palo Alto Research Center* (PARC), o IFS organizava arquivos privados e compartilhados numa árvore de diretórios. Estações de trabalho clientes recebiam cópias dos arquivos depois de fornecer uma cadeia de caracteres com o nome do arquivo desejado..

Outro sistema experimental desenvolvido no mesmo centro foi o *Woodstock File Server* (WFS) [SMB79] que oferecia a possibilidade de acesso a páginas de arquivos ao invés do arquivo completo. Isto possibilitou a criação de estações de trabalho sem discos locais e a implementação de memória virtual através da rede.

Já em 1977, o PARC começou a utilizar o seu *Xerox Distributed File System* [MJ82] que era um sistema onde vários servidores proviam um serviço muito robusto. A sua finalidade primordial era oferecer uma base para a implementação de sistemas administradores de bancos de dados. Entre as facilidades oferecidas, estavam as transações atômicas envolvendo vários arquivos e servidores, implementadas através do protocolo de duas fases, e o acesso a pequenos trechos de arquivos.

O LOCUS [WPE+83, WJLP85] começou a funcionar em 1980 e já oferecia transparência de localização, transações atômicas aninhadas [MMP83] e replicação automática.

O SWALLOW, projetado no início da década de 80 no MIT, foi o primeiro a implementar uma técnica de controle de acesso concorrente baseado em *timestamps* [Ree83]. No SWALLOW, cada alteração em um arquivo criava uma uma nova **versão** do arquivo. O sistema implementava transações atômicas através do controle de qual das versões será acessada por cada transação. As informações trocadas entre o servidor e os clientes eram criptografadas a fim de garantir a segurança dos arquivos [Svo84].

[1] *ftp* (*File Transfer Protocol*) é um serviço que permite a transferência de arquivos entre máquinas distintas.

Um sistema digno de nota, apesar de suas limitações, é o *Acorn File Server* [Del82]. Desenvolvido pela *Acorn Computers Limited* no início da década de 80 na Inglaterra, tinha como objetivo a implementação de redes de microcomputadores em escolas a um custo muito baixo. Naquela época, julgava-se economicamente viável oferecer teclado, monitor, CPU e uma pequena memória para cada aluno mas acionadores de discos eram muito caros.

Escrito na linguagem de montagem do microprocessador 6502 da *Motorola* (o mesmo do *Apple II*), o *Acorn File Server* gastava menos de 30K de memória e oferecia acesso, através da rede, aos discos flexíveis inseridos em um servidor.

O VICE é um sistema de arquivos distribuído desenvolvido com fundos da IBM e da *Carnegie-Mellon University*. O projeto foi coordenado pelo especialista em sistemas de arquivos distribuídos Mahadev Satyanarayanan e é o ancestral do AFS e do CODA. O VICE começou a funcionar em julho de 1984 apresentando a principal característica dos sistemas desenvolvidos por Satyanarayanan: a grande escalabilidade.

## 2.2 Uma Visão Geral

No decorrer deste capítulo, apresentaremos alguns dos principais sistemas de arquivos distribuídos da atualidade analisando como cada um deles resolve, ou não, os problemas descritos no capítulo anterior.

Vejamos, sucintamente, quais são as principais características de cada um dos sistemas descritos neste capítulo.

1. **NFS**: O *Network File System*, desenvolvido pela *SUN Microsystems*, é o sistema de arquivos distribuídos mais utilizado em ambiente UNIX[2]. Isto ocorre graças à iniciativa da SUN de lançar publicamente a especificação do **protocolo NFS** que permitiu que qualquer fabricante fosse capaz de lançar os seus próprios clientes e servidores NFS. Atualmente, existem implementações do NFS para praticamente todas as plataformas relevantes e todas elas podem compartilhar o sistema de arquivos entre si.

   No entanto, o NFS apresenta uma série de problemas dentre os quais se destacam a sua velocidade relativamente baixa e a falta de consistência entre os caches dos clientes.

2. **ANDREW**: Desenvolvido na Universidade Carnegie-Mellon, o *Andrew File System* foi o primeiro sistema de arquivos a oferecer um serviço de alta escalabilidade possibilitando que dezenas de milhares de clientes compartilhassem os arquivos oferecidos por centenas de servidores. Este era o objetivo principal do projeto e foi alcançado através da adoção da semântica de sessão e da utilização de grandes caches em discos locais dos clientes.

   A fim de limitar a possível falta de segurança em um sistema tão grande, o ANDREW adota uma série de mecanismos como, por exemplo, o *Kerberos Authentication Server* que permite a autenticação mútua de servidores e clientes.

3. **CODA**: Desenvolvido a partir do ANDREW, o CODA oferece uma alta disponibilidade através da adoção de uma técnica otimista de replicação. A grande peculiaridade deste sistema são as ferramentas que permitem o acesso ao sistema de arquivos a partir de computadores portáteis. Existe até a possibilidade de operação desconectada durante a qual clientes podem acessar o sistema de arquivos sem estabelecer contato com os servidores.

[2] O sistema de arquivos distribuído com maior número de usuários no mundo é o NETWARE para ambiente MS-DOS. Apesar disto, não trataremos do NETWARE nesta dissertação pois, além de ser um sistema muito limitado, não existe literatura satisfatória descrevendo a sua implementação ou analisando o seu desempenho.

4. **SPRITE**: O *Sprite Network Operating System*, desenvolvido na Universidade da Califórnia em Berkeley oferece um serviço de arquivos muito rápido além de garantir a consistência das informações cacheadas pelos clientes. Foi o primeiro sistema a adotar uma política na qual o espaço reservadopara o cache varia dinamicamente podendo ocupar toda a memória disponível tanto nos servidores quanto nos clientes.

   Além do sistema de arquivos distribuído, o SPRITE oferece, ainda, a possibilidade de migração de processos entre as máquinas de uma rede local.

5. **ZEBRA**: Descendente do SPRITE, o ZEBRA oferece um serviço muito rápido através da implementação das idéias de matrizes de discos redundantes (RAIDs) e de Sistemas de Arquivos Baseados em *Log* em um sistema distribuído. O ZEBRA, assim como os sistemas seguintes, possui apenas implementações experimentais ainda não utilizadas por um grande número de usuários.

6. **HARP**: Desenvolvido no MIT, o HARP implementa a replicação automática dos arquivos oferecendo um serviço de alta confiabilidade e disponibilidade. A adoção de uma política pessimista de replicação faz com que a rapidez do serviço seja prejudicada.

7. **FROLIC**: Através de um mecanismo de replicação dinâmica, o FROLIC pretende oferecer um serviço altamente escalável possibilitando o acesso concorrente a arquivos por uma grande quantidade de clientes. Está sendo desenvolvido na Universidade de Toronto.

8. **ECHO**: Desenvolvido pela DEC, o ECHO é o resultado de um projeto ambicioso que pretendia oferecer um serviço rápido possibilitando a replicação automática e o acesso transparente a arquivos fisicamente distantes através da Internet. Depois de um ano de funcionamento da primeira versão do sistema, o projeto no qual o ECHO estava inserido foi cancelado interrompendo a implementação completa do sistema.

Vejamos, agora, em detalhes, cada um destes sistemas.

## 2.3 NFS

O *Network File System* é um sistema de arquivos distribuído desenvolvido pela *SUN Microsystems*. Desde 1985, é um produto que acompanha o sistema operacional SunOS permitindo que as estações de trabalho fabricadas pela SUN compartilhem arquivos transparentemente. Foi o primeiro sistema utilizado por um grande número de usuários a oferecer um sistema de acesso transparente a arquivos remotos. É um sistema projetado para redes locais mas também pode ser usado em redes de maior porte.

Após lançar o seu NFS, a SUN tomou a iniciativa pioneira de divulgar publicamente a especificação do *protocolo NFS*. Este protocolo define uma *interface RPC*[3] que utiliza uma representação de dados independente da máquina chamada *External Data Representation* ou, simplesmente, XDR[4]. Desta forma, tornou-se possível a implementação de sistemas compatíveis com o NFS sobre qualquer plataforma de hardware e software.

Atualmente existem implementações do NFS para praticamente todas as implementações do UNIX, para microcomputadores pessoais (sistemas MS-DOS, OS/2 e MacOS) e computadores de grande porte (sistemas IBM/MVS). Todas estas implementações podem se comunicar entre si permitindo o compartilhamento de arquivos em redes heterogêneas. Esta é uma característica muito importante (como observamos na seção 1.1.5) que fez do protocolo NFS um verdadeiro padrão da indústria.

As implementações atualmente em uso seguem a versão 2 do protocolo NFS de março de 1989 [SUN89a]. No fim de 1993, a SUN divulgou a terceira versão deste protocolo [SUN94] mas, até o presente momento, não existem implementações disponíveis.

### 2.3.1 Um Protocolo Livre de Estado

Segundo [SUN89a], a intenção do protocolo NFS é de ser o mais livre de estado (*stateless*) possível. Ou seja, um servidor NFS não precisa manter nenhum tipo de informação sobre o estado da comunicação com os seus clientes para oferecer o seu serviço satisfatoriamente. Em particular, o servidor não possui nenhuma informação sobre os arquivos abertos nos clientes. Esta característica traz duas vantagens imediatas.

A mais significativa é que, quando ocorre uma queda do servidor, ele não perde nenhuma informação sobre o estado da comunicação com os clientes e, portanto, não precisa gastar tempo tentando reconstituir este estado quando se recupera da queda. O trabalho dos clientes também é mais simples. Quando um servidor cai, tudo o que o cliente precisa fazer é repetir as chamadas ao servidor até que ele responda. O cliente não precisa nem saber se o servidor caiu ou não. Tudo o que ele faz é emitir a famosa mensagem "NFS server not responding, still trying..."[5] em seu console.

A outra vantagem é que, não gastando memória para guardar informações sobre os clientes, uma máquina pode oferecer o serviço de arquivos para um número maior de clientes. Mas, a escalabilidade do NFS não é tão boa como pode parecer pois, se muitos pedidos são encaminhados ao servidor, o seu desempenho diminui consideravelmente.

Por outro lado, a incapacidade de perceber o estado do sistema faz com que o NFS não tenha recursos para administrar a consistência dos caches dos clientes eficientemente. Como

[3] Uma interface RPC consiste de uma série de procedimentos que podem ser executados remotamente através de uma *Remote Procedure Call*. Para maiores detalhes, consulte [SUN90] ou [SUN89b].

[4] Ver [SUN89c].

[5] Esta mensagem é emitida quando o diretório acessado foi montado através de um *hard mount* que é o caso normal. Existe uma variante, o *soft mount*, na qual o cliente tenta o contato com o servidor apenas uma vez. Neste caso, se o servidor não responde ao primeiro chamado, o cliente devolve um código de erro para a aplicação que solicitou o acesso remoto.

veremos na seção 2.3.5, a implementação da SUN não garante a consistência do sistema de arquivos.

O NFS também é incapaz de oferecer bloqueios ou transações pois estes são serviços essencialmente ligados ao estado do sistema de arquivos. No SunOS, os bloqueios são oferecidos pelo *Network Lock Manager*[6] que é totalmente independente do NFS. Transações não são oferecidas.

| Vantagens | Desvantagens |
|---|---|
| Recuperação trivial das quedas | Dificuldade em manter cache consistente |
| Economia de memória do servidor | Impossível implementar bloqueios e transações |

Tabela 2.1: Servidor livre de estado

Nas seções seguintes, vamos examinar mais detalhadamente o funcionamento da implementação do protocolo NFS realizada pela SUN para uma rede de máquinas executando o SunOS. A fim de evitar confusões, destacamos que, sempre que utilizarmos o termo "NFS", estaremos nos referindo à implementação da SUN do protocolo NFS. Quando nos referirmos ao protocolo genérico, deixaremos isto explícito.

### 2.3.2 Espaço de nomes

Ao contrário de outros sistemas que possuem servidores dedicados, o NFS não faz distinção entre servidores e clientes. O cliente é implementado dentro do núcleo do SunOS e, portanto, está presente em todas as máquinas da rede. O servidor é implementado através de um ou mais processos *daemon* – que rodam permanentemente nas máquinas escolhidas – atendendo aos RPCs dos clientes.

Quando uma máquina é inicializada, o seu sistema de arquivos é formado unicamente pela raiz /. O administrador do sistema dispõe do comando *mount* para construir o espaço de nomes de cada máquina. Inicialmente, assim como o *mount* do UNIX, ele é utilizado para associar discos locais à raiz e a alguns de seus subdiretórios. Mas, este comando permite, também, que um diretório local seja associado a um diretório presente num sistema de arquivos de outro servidor.

A figura 2.1 mostra o resultado de uma seqüência de três comandos *mount* em um cliente. *mount /dev/hd1a /* associa a raiz ao disco rígido local[7]. *mount farofa:/mailboxes /var/spool/mail* faz com que as mensagens de correio eletrônico que chegarem ao diretório */mailboxes* do servidor *farofa* possam ser lidas pelo cliente através do seu diretório local */var/spool/mail*. Finalmente, *mount brucutu:/executaveis /usr/bin* faz com que o diretório */usr/bin* "contenha" os arquivos do diretório */executaveis* do servidor *brucutu*.

Deste modo, o sistema é configurado para que diversos sistemas de arquivos físicos distintos apareçam como um único sistema lógico para o usuário. Embora não seja obrigatório, costuma-se configurar todos os nós de uma rede local de modo que eles apresentem o mesmo sistema de arquivos lógico permitindo que os usuários utilizem qualquer máquina da rede da mesma maneira sem se preocupar com a localização física dos arquivos.

[6] O *Network Lock Manager* oferecido pela SUN apresenta uma série de *bugs* que prejudicam o seu funcionamento em diversas situações.

[7] Na realidade, a raiz já é montada automaticamente no momento da inicialização do máquina como veremos em seguida.

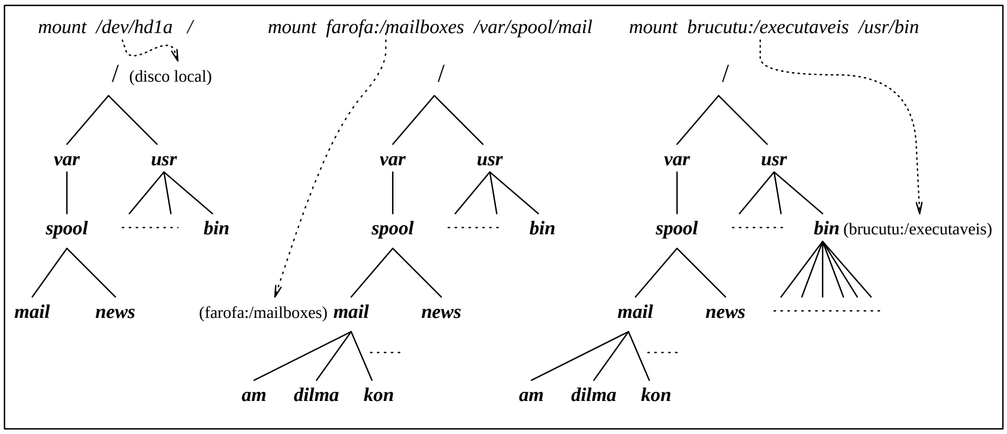



Figura 2.1: Seqüência de comandos *mount*

Assim, podemos dizer que, a menos do processo de montagem de diretórios que é de responsabilidade do administrador do sistema, a localização de arquivos e diretórios no NFS é transparente.

No mesmo documento em que definiu o protocolo NFS [SUN89a], a SUN definiu o protocolo de montagem (*mount protocol*). Em linhas gerais, o que ocorre é o seguinte. O cliente envia ao servidor, via RPC, um pedido de montagem de um diretório. Se tudo correr bem, o servidor devolve um *file handle* daquele diretório. Este *file handle* contém, entre outros, campos que identificam o tipo do sistema de arquivos, o disco físico onde ele se encontra e o *i-node*[8] do diretório. Chamadas subseqüentes de leitura ou alteração do diretório montado são feitas utilizando este *file handle*.

Para automatizar o processo de configuração do espaço de nomes, existe o arquivo */etc/fstab* que contém uma lista de diretórios que são montados automaticamente no momento de inicialização da máquina. O diretório raiz é sempre montado automaticamente segundo a determinação deste arquivo. Já o arquivo */etc/mtab* contém uma lista com informações sobre os diretórios montados que é atualizada toda vez que um novo diretório é montado ou desmontado. Estes arquivos já existiam no UNIX mas a sua sintaxe foi estendida para acomodar as referências aos diretórios remotos.

Além disso, existe a possibilidade de os diretórios serem montados dinamicamente à medida em que surja a necessidade de acessá-los. Isto pode ser feito com um programa chamado *automounter*. Este programa utiliza um mapa que associa diretórios locais a um ou mais diretórios em servidores remotos. Quando um processo tenta acessar um diretório presente em um destes mapas, o *automounter* envia pedidos de montagem de diretório a todos os servidores associados a este diretório. O servidor que responder primeiro é escolhido como o servidor temporário deste diretório. Se o diretório não for acessado em um certo período (cinco minutos por *default*), o *automounter* desmonta o diretório remoto.

O *automounter* é um mecanismo que oferece algumas vantagens. No momento da inicialização (*boot*), a máquina não precisa gastar tempo montando todos os diretórios. Deste modo evita-se a possibilidade de um cliente esperar muito tempo pela resposta de um servidor fora do ar sendo que talvez ele nem precise acessar os diretórios daquele servidor.

Além disso, a SUN espera que o fato de vários servidores serem consultados simultaneamente permita que os diretórios mais acessados sejam replicados em mais de um servidor e

[8] Ver apêndice A.

que a carga seja distribuída eqüitativamente entre eles. Isso seria obtido porque espera-se que o servidor mais livre seja o primeiro a responder ao chamado do *automounter*.

Mas, além de não existirem mecanismos de replicação automática, não há nada que garanta que o primeiro servidor a responder ao *automounter* seja o com menor carga. Logo, a replicação oferecida pelo NFS pode ser considerada muito fraca.

### 2.3.3 O Protocolo

A definição do protocolo é dada como um conjunto de procedimentos definidos na *RPC language* [SUN89b]. A figura 2.2 mostra protótipo de todos os procedimentos do protocolo.

A linguagem RPC é uma extensão da linguagem C que é processada pelo compilador *rpcgen* que tem como saída um programa em C. A primeira e a última linha da figura 2.2 indicam que estamos definindo um programa que se chama `NFS_PROGRAM` e que estará associado à porta 100003 da máquina que o executará. A segunda e a penúltima linha especificam o número da versão (no caso, 2).

As demais linhas especificam três informações para cada procedimento do protocolo: o número do procedimento, o tipo dos parâmetros recebidos pelo procedimento e o tipo do valor retornado. Tais tipos podem ser estruturados e são definidos através do padrão XDR.

Existem procedimentos para ler, escrever, criar e apagar diretórios e arquivos, para a manipulação de *links* e para acessar os atributos dos arquivos. Note a inexistência proposital de operações como *open* e *close* que não tem utilidade para servidores livres de estado.

Vamos ver como um cliente faz uso deste protocolo para acessar um arquivo. Suponha que um processo deseja ler o arquivo */etc/uucp/Permissions* e que o diretório */etc* esteja remotamente montado.

Primeiramente, o cliente efetua a tradução do *pathname* para o *file handle*. Esta tradução é feita componente a componente. No nosso exemplo, o cliente chama[9] o procedimento `NFSPROC_LOOKUP(fhetc,"uucp")` onde `fhetc` é o *file handle* (devolvido pelo mount) do diretório */etc*. Este procedimento devolverá o file handle do subdiretório *uucp* que será utilizado numa nova chamada: `NFSPROC_LOOKUP(fhuucp,"Permissions")`. Esta última chamada devolverá o *file handle* do arquivo */etc/uucp/Permissions* juntamente com os seus atributos.

Agora, o cliente chama `NFSPROC_READ()` passando como parâmetros o *file handle* de */etc/uucp/Permissions*, a posição a partir da qual ele quer ler e o número de bytes que ele deseja. Além dos bytes solicitados, o cliente recebe também uma versão atualizada dos atributos do arquivo.

O cliente agrupa os pedidos de leitura dos seus processos em blocos de 8K a fim de minimizar o tráfego na rede. Se um processo pede os 10 primeiros bytes de um arquivo de 100K, o cliente executa uma RPC pedindo os primeiros 8Kbytes do arquivo. Se este processo pedir os 10 bytes seguintes não haverá necessidade de executar outra RPC pois os dados estarão armazenados no cache (ver seção 2.3.5). Este comportamento é chamado de leitura antecipada (*read-ahead*).

Note que o processo de tradução do *pathname* é algo muito dispendioso pois, a cada componente posterior ao ponto da montagem do diretório remoto, corresponde uma chamada RPC. Seria muito mais eficiente passar o *pathname* completo para o servidor e receber em troca o *file handle* final. Isto não é possível no NFS pois cada cliente possui uma visão própria do sistema de arquivos. O mesmo *pathname* pode ter significados diferentes para um cliente e o seu servidor.

[9] Para chamar este procedimento via RPC, o cliente deve enviar uma mensagem para a porta 100003 do servidor contendo o número da versão do NFS (2), o número do procedimento (4) e, em seguida, os parâmetros apropriados.

```
program NFS_PROGRAM {
        version NFS_VERSION {
                void
                NFSPROC_NULL(void)              = 0;
                attrstat
                NFSPROC_GETATTR(fhandle)        = 1;
                attrstat
                NFSPROC_SETATTR(sattrargs)      = 2;
                void
                NFSPROC_ROOT(void)              = 3;
                diropres
                NFSPROC_LOOKUP(diropargs)       = 4;
                readlinkres
                NFSPROC_READLINK(fhandle)       = 5;
                readres
                NFSPROC_READ(readargs)          = 6;
                void
                NFSPROC_WRITECACHE(void)        = 7;
                attrstat
                NFSPROC_WRITE(writeargs)        = 8;
                diropres
                NFSPROC_CREATE(createargs)      = 9;
                stat
                NFSPROC_REMOVE(diropargs)       = 10;
                stat
                NFSPROC_RENAME(renameargs)      = 11;
                stat
                NFSPROC_LINK(linkargs)          = 12;
                stat
                NFSPROC_SYMLINK(symlinkargs)    = 13;
                diropres
                NFSPROC_MKDIR(createargs)       = 14;
                stat
                NFSPROC_RMDIR(diropargs)        = 15;
                readdirres
                NFSPROC_READDIR(readdirargs)    = 16;
                statfsres
                NFSPROC_STATFS(fhandle)         = 17;
        } = 2;
} = 100003;
```

Figura 2.2: Procedimentos do protocolo NFS

A fim de evitar uma enorme perda de tempo na tradução de *pathnames*, os clientes NFS possuem um cache dedicado unicamente às informações sobre os diretórios.

O protocolo NFS não obriga a implementação do *read-ahead* nem do cache de diretórios mas eles são mecanismos indispensáveis para uma implementação eficiente do protocolo.

### 2.3.4 Segurança

Cada servidor NFS guarda no arquivo */etc/exports* uma tabela indicando qual máquina pode ter acesso a qual diretório do seu sistema de arquivos local. A tabela informa também se este acesso será só para leitura ou para leitura e escrita. Especifica-se também se usuários *root* remotos poderão ter os seus super-privilégios nos diretórios locais.

A partir disso, o NFS faz uso do sistema de segurança dos sistemas de arquivos UNIX. Quando um cliente efetua um RPC pedindo informações, o servidor recebe o *uid* e o *gid*[10] relativos ao processo que fez o pedido. Consultando os bits de permissão do arquivo ou diretório, o servidor decide se fornece a informação ao cliente ou não.

Este método introduz a necessidade de que os *uids* e *gids* nas máquinas de uma certa rede sejam compatíveis. O *Network Information Service* (NIS)[11] oferece um serviço de informações distribuído que é utilizado para fornecer os *uids* e *gids* para todos os nós da rede.

Deste modo, é possível que um intruso que consiga alterar o *kernel* de um cliente forneça um *uid* que não lhe pertença a fim de obter informações confidenciais ou alterar arquivos de outros usuários. Para redes onde não se pode confiar na autenticidade dos clientes, o NFS oferece a possibilidade de uma autenticação mútua entre o cliente e o servidor antes do envio de qualquer informação. Esta autenticação é baseada no método DES de criptografia[12] e as chaves criptográficas públicas são fornecidas pelo NIS. Mas, uma vez que a informação em si não é criptografada, um intruso pode "ouvir" o que está se passando na rede e capturar qualquer informação trocada entre clientes e servidores.

### 2.3.5 Cache Inconsistente

O NFS mantém caches do conteúdo dos arquivos tanto no servidor (para economizar acessos a disco) quanto nos clientes (para economizar chamadas RPC). Os clientes também possuem um cache para os atributos e informações sobre a localização dos arquivos.

Os dados recebidos dos servidores são armazenados nos caches locais na esperança de poderem atender às solicitações subseqüentes sem a necessidade de acessos remotos. Nas primeiras versões do SunOS, o sistema reservava um espaço constante para o cache. Posteriormente, adotou-se a política do SPRITE que possui um cache de tamanho variável como veremos na seção 2.6.3.

*writes* seqüenciais a um determinado arquivo são armazenados no cache local enquanto não formam um bloco de 8K. Este comportamento evita a sobrecarga que seria causada pelo envio de um grande número de mensagens pequenas.

Versões antigas do NFS[LS90], praticavam o *write-on-close*, ou seja, quando um processo fechava um arquivo aberto para escrita, todos os blocos sujos (alterados) do cache eram enviados para o servidor. O *write-on-close* foi abandonado nas versões mais recentes, talvez porque arquivos temporários, gerados por compiladores ou utilitários como o *sort* do UNIX, eram enviados para o servidor. Isto seria um desperdício pois a vida destes arquivos tem-

[10] Ver apêndice A.

[11] Inicialmente, o NIS era chamado de *Yellow Pages* mas este nome foi abandonado após uma disputa judicial pela marca com a companhia telefônica inglesa.

[12] Veja [Den82]

porários não dura mais do que alguns segundos e eles não são acessados por outros clientes não havendo a necessidade de enviá-los ao servidor.

As versões mais atuais praticam o *delayed-write*[13] segundo o qual blocos sujos podem permanecer no cache mesmo após o *close*. Máquinas que não possuem discos locais e que são obrigadas a montar remotamente o diretório de arquivos temporários */tmp* obtém um grande ganho de desempenho com *delayed-writes*. Ao contrário de outros sistemas, como o SPRITE, o NFS mantém a política de retardar as escritas mesmo quando vários clientes estão atualizando o arquivo concorrentemente.

O cache dos clientes pode levar a situações desagradáveis uma vez que alterações realizadas por um cliente não ficam imediatamente visíveis para outros clientes. Como vimos na seção 1.1.2 é preciso adotar alguma política para a manutenção da consistência do cache.

O NFS adota uma política que garante a consistência do cache **quase** sempre. Em outras palavras, o NFS não garante a consistência. Vamos analisar os mecanismos empregados pelo NFS para minimizar as situações de inconsistência.

Quando um processo abre um arquivo presente no cache local, o cliente NFS consulta o servidor para descobrir se a sua versão do arquivo é a mais recente. Ele descobre isso através do campo do *i-node* que guarda o instante no qual o arquivo foi alterado pela última vez. O arquivo só é lido do cache se lá está a versão mais atualizada.

Após a verificação do número da versão no momento do open, outras verificações são feitas. Quando o cliente recebe um bloco de dados de um servidor, ele recebe também uma versão atualizada dos atributos do arquivo e refaz a verificação. No SunOS 4.1.1, a partir do momento no qual um dado é guardado no cache, ele é considerado válido por 3 segundos no caso de arquivos e, segundo o manual do comando *mount*, por até 60 segundos no caso de diretórios[14].

Obviamente, isso não garante nada. Arquivos criados por um certo cliente podem ficar até 60 segundos invisíveis para o resto do sistema. Ou mesmo pior. Considere uma situação com dois clientes *c1* e *c2* e um servidor. Suponha que, no mesmo instante em que *c1* cria o arquivo */usr/lock*, *c2* receba do servidor, uma cópia atualizada do diretório */usr*. Nos 60 segundos seguintes, *c2* consulta os atributos de */usr* sem perceber a existência do arquivo *lock*.

Quando o limite de 60 segundos se esgota, o servidor recebe duas mensagens. Suponha que primeiro chegue a mensagem de *c2* pedindo uma versão atualizada de */usr* e que, logo em seguida, chegue o pedido de criação do arquivo *lock*.

Neste caso extremo, pode acontecer de *c2* só perceber a existência do *lock* dois minutos após a sua criação. Se a intenção era usar este arquivo como um bloqueio (como mostra [Ste90] na seção 3.2), ela não passou nem perto de funcionar.

Para um usuário do NFS é um mistério o momento no qual os seus *writes* a um arquivo serão visíveis para outros clientes que o estão lendo. Se um novo cliente abre este arquivo, ele só poderá receber os dados que já tiverem sido enviados ao servidor.

Com o intuito de observar o comportamento do NFS em ambientes de alto compartilhamento de arquivos para escrita, realizamos alguns testes onde diversos processos localizados em diferentes clientes escreviam em um mesmo arquivo concorrentemente.

A figura 2.3 mostra um esboço do que cada processo fazia.

Além da perda de diversas linhas e de trechos de linhas, o que era esperado, observamos o aparecimento de dezenas de bytes nulos no meio do arquivo.

Concluindo, a semântica do NFS não só não é equivalente à semântica UNIX como também

[13]Quando os *delayed-writes* são executados de forma assíncrona – como neste caso – diz-se que a política adotada é a *write-behind*.

[14]Diferentes prazos de validade podem ser especificados no momento do *mount*.

```
para i de 1 a n execute
  {
   open(arq)
   print(arq, i, identificação do cliente, timestamp)
   flush(arq)
   close(arq)
  }
```

Figura 2.3: Esboço da ação de cada processo

não é equivalente a semântica de sessão ou qualquer outra semântica bem definida (ver seção 1.1.9). Operações idênticas podem gerar resultados diferentes de acordo com condições que estão além da percepção dos usuários.

Apesar disto, é um fato inegável que o NFS é amplamente utilizado com sucesso em diversas plataformas. Pesquisadores responsáveis pelo desenvolvimento do SPRITE apresentaram um interessante trabalho [BHK+91a] onde é realizada uma análise da freqüência com que ocorrem problemas de consistência devido à semântica imprevisível do NFS (ver seção 2.6.5).

O estudo foi feito coletando-se todos os acessos a arquivos efetuados em uma rede com 52 usuários e 40 máquinas durante quatro períodos de 48 horas cada um. A rede executava o sistema operacional distribuído SPRITE. A partir da análise dos dados coletados, foi possível concluir que, se o sistema fosse o NFS, então 20% dos usuários teriam acessado informações inconsistentes no período observado.

Infelizmente, o artigo não deixa claro onde ocorreram estas inconsistências. Esporadicamente, quando se trabalha em uma rede de estações executando o SunOS e o NFS, acontecem erros dos mais variados tipos. O sistema de janelas trava, processos são abortados sem explicações, servidores sofrem *crashes*. Tais erros costumam ser raros o suficiente para permitir que o SunOS seja muito utilizado por milhares de usuários em todo o mundo. Mas, uma pergunta vem à mente: "Quantos desses erros são causados por acessos a dados desatualizados no sistema de arquivos?"[15]. No capítulo 4 apresentamos uma alteração no protocolo NFS que permite manter a consistência entre os caches dos clientes a um baixo custo.

### 2.3.6 Arquitetura

O cliente NFS faz parte do núcleo do SunOS. Ele captura as chamadas ao sistema (*system calls*) relativas a arquivos montados via NFS, executando chamadas a procedimentos remotos e devolvendo os dados enviados pelo servidor.

O servidor, no entanto, é implementado através de um processo *daemon*, denominado *nfsd*, que roda permanentemente atendendo aos chamados dos clientes. Ao se chamar o programa inicializador do *daemon*, */usr/etc/nfsd*, existe a possibilidade de ativar vários processos servidores simultaneamente permitindo que a máquina em questão atenda a várias chamadas RPC em paralelo. Segundo o manual do *nfsd*, *"Eight seems to be a good number"*.

No SunOS 5.3, recentemente lançado, ao invés de vários processos, o servidor é implementado através de um único processo contendo vários *threads*[16].

O número do *i-node* de um arquivo (ou diretório) é uma identificação única deste arquivo

[15] Esta dissertação foi escrita em uma rede de estações SPARC compartilhando arquivos através do NFS implementado no SunOS 4.1.1. Algumas vezes, o console da estação mostrou a mensagem de erro *NFS write error 70 on host feijoada* indicando que um processo tentou escrever em um arquivo do servidor *feijoada* que não mais existia. Não conseguimos descobrir por que isso ocorria mas pode ser um caso semelhante às

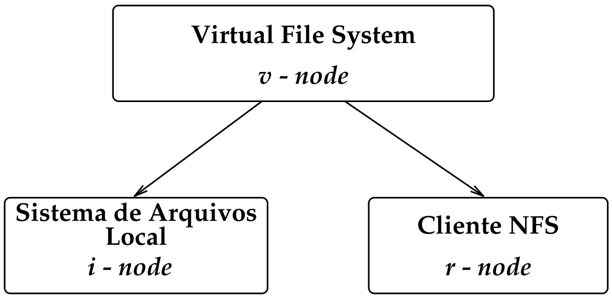



Figura 2.4: *v-nodes*

dentro de um sistema de arquivos UNIX. Mas, arquivos em diferentes sistemas de arquivos UNIX ou, em particular, em diferentes máquinas podem possuir o mesmo número de *i-node*.

Para evitar confusões, as *systems calls* relativas aos sistemas de arquivos são tratadas por uma camada chamada *Virtual File System* (VFS). A função desta camada é oferecer o serviço de arquivos através de *v-nodes*. O número do *v-node* identifica unicamente um arquivo de qualquer ponto da rede.

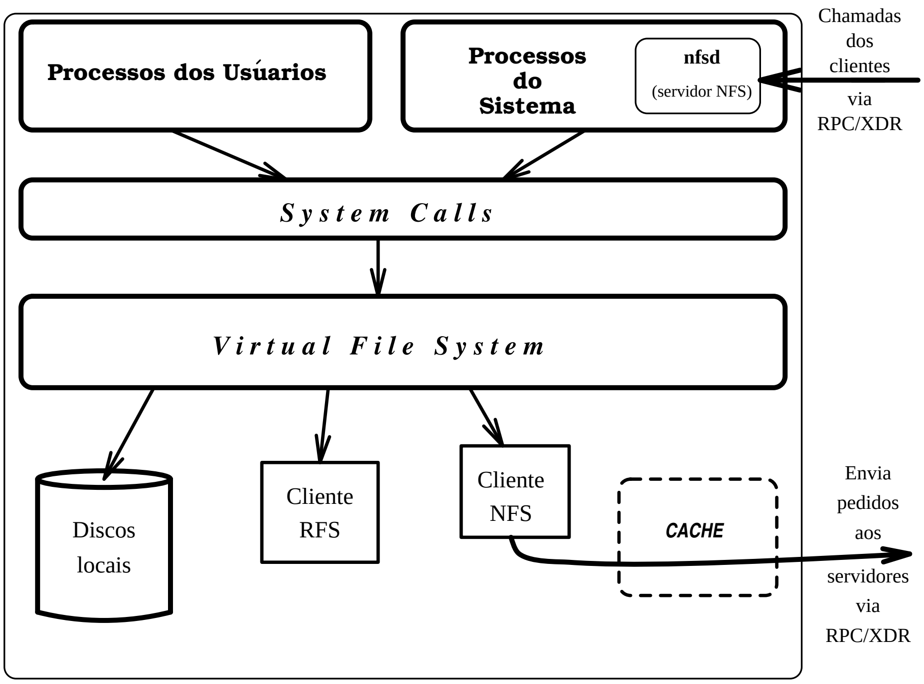



Figura 2.5: A Arquitetura do NFS

Quando o cliente recebe um *file handle* de um arquivo remoto, ele cria um *r-node* contendo, além das informações do *i-node* remoto, informações que identificam o servidor responsável pelo arquivo.

---

inconsistências observadas pela equipe do SPRITE.

[16]Um processo pode possuir vários *threads* que são executados concorrentemente compartilhando o mesmo espaço de dados. Para maiores detalhes sobre *threads* ou *lightweight processes* veja [Tan92], seção 12.1.

Os *v-nodes* da camada VFS por sua vez, apontam ou para um *i-node* local ou para um *r-node* nas tabelas do cliente. Quando os processos abrem um arquivo, eles recebem um descritor que é um apontador para o seu *v-node*.

Quando uma aplicação faz uma solicitação ao serviço de arquivos através das *systems calls open, close, read, write, create, unlink, mkdir, rmdir* e *stat*, esta solicitação é encaminhada para a camada VFS que, de acordo com a localização física do arquivo, a remete para sistemas de arquivos locais ou para o cliente NFS. A figura 2.5 mostra um esquema da arquitetura do NFS.

Esta é uma estrutura muito interessante. Ela oferece uma interface genérica que é independente da implementação. Utilizando esta estrutura é possível implementar uma camada VFS que possibilite acesso transparente a uma série de sistemas de arquivos diferentes como UNIX BSD, MS-DOS, NFS, RFS, etc.

### 2.3.7 Resumo

O NFS é o sistema de arquivos distribuído mais utilizado em ambiente UNIX na atualidade. Isto se deve a dois fatores principais. Ao fato da SUN ter divulgado o protocolo NFS que especifica detalhadamente como se dá a comunicação entre os clientes e servidores através da interface RPC/XDR e à simplicidade do protocolo que facilita a sua implementação.

Os servidores NFS são livres de estado, isto é, não armazenam informações sobre o acesso aos seus arquivos por parte dos clientes fazendo com que informações sobre o estado do sistema não sejam perdidas no caso de quedas dos servidores. Por outro lado, os servidores ficam impossibilitados de controlar a consistência dos arquivos cacheados nos clientes e mesmo de implementar mecanismos como bloqueios ou transações que dependem intrinsecamente de informações sobre o estado do sistema.

O espaço de nomes é construído pelo administrador do sistemas através da manipulação do arquivo */etc/fstab* e do comando *mount* podendo variar de cliente para cliente. O NFS oferece, também, a possibilidade da utilização de clientes sem disco.

Os servidores NFS administram a segurança do sistema através de dois mecanismos. O arquivo */etc/exports* indica quais clientes podem ter acesso (e qual o tipo de acesso) a cada uma das suas sub-árvores de diretórios e os bits de permissão do UNIX indicam quais usuários e grupos de usuários podem acessar (e como) cada arquivo.

No SunOS, as solicitações de acesso ao sistema de arquivos realizadas pelos processos dos usuários são atendidas por um *Virtual File System* que remete as solicitações para vários sistemas de arquivos distintos oferecendo uma interface comum independente do tipo do sistema de arquivos.

A utilização da interface RPC/XDR e a política de cache adotada pelo NFS fazem com que o seu desempenho não seja muito bom se comparado a alguns dos sistemas que analisaremos a seguir.

## 2.4 ANDREW

Em 1983, teve início o ANDREW, um projeto conjunto da IBM e da *Carnegie-Mellon University* que visava o desenvolvimento de um sistema que fosse ideal para o ambiente acadêmico de ensino e pesquisa [HKM+88, Sat90a, Tan92]. O nome foi escolhido em homenagem aos fundadores da universidade, Andrew Carnegie e Andrew Mellon.

O objetivo principal do projeto, coordenado por Mahadev Satyanarayanan era oferecer a cada aluno e professor uma estação de trabalho com um sistema compatível com o UNIX BSD. Os usuários entrariam em qualquer máquina da rede e a sua visão do sistema deveria ser a mesma.

Desejava-se estender esta transparência de localização a uma rede de 5 a 10 mil estações de trabalho. Logo, era essencial tomar o máximo de cuidado com a escalabilidade dos métodos empregados, pois um sistema que funciona bem em uma rede experimental de 50 máquinas pode se mostrar completamente inadequado para uma rede 100 vezes maior. Ao lado da escalabilidade, outro fator essencial é a segurança dos arquivos confidenciais uma vez que seria muito difícil controlar o ataque de intrusos a cada uma das 5 mil máquinas.

Como a idéia era executar os processos localmente, o ANDREW, assim como o NFS, não pode ser considerado como um sistema operacional distribuído. A melhor classificação é de sistema de arquivos distribuído.

Desde que entrou em operação, o sistema foi sofrendo modificações graduais. Para efeito de estudo, costuma-se dividir o ANDREW em três versões [Sat90a].

Entre o fim de 1984 e o fim de 1985, o AFS-1 operou com 100 estações e 6 servidores. O desempenho do sistema era considerado razoável com até 20 usuários por servidor mas, às vezes, o trabalho pesado de alguns usuários degradava o desempenho de maneira intolerável. Para piorar as coisas, o sistema era de difícil manutenção uma vez que os administradores não dispunham de ferramentas adequadas para esta tarefa.

A experiência com a primeira versão e os numerosos testes de desempenho [HKM+88] foram a base para o desenvolvimento do AFS-2. Esta nova versão conseguiu maior eficiência através de um algoritmo melhor para a manutenção da consistência do cache e da melhoria da implementação no que diz respeito à resolução dos nomes, comunicação e estrutura dos servidores. Esteve em operação desde o fim de 1985 até o meio de 1989.

A partir de 1988, trabalhou-se para tornar o ANDREW um sistema comercial. Para isso foi necessário adequar o sistema a alguns padrões da indústria como o *Virtual File System* (VFS) da *SUN Microsystems* que possibilitou a integração do ANDREW a outros sistemas de arquivos. Atualmente, o AFS (ou AFS-3) é comercializado pela *Transarc Corporation*, uma empresa sediada em Pittsburgh e formada por profissionais envolvidos no desenvolvimento do sistema.

Aos poucos, os usuários do ANDREW foram adquirindo a confiança no sistema e se tornaram dependentes do seu serviço de arquivos. A partir deste momento, notou-se a importância da disponibilidade dos dados. No AFS, a queda de um servidor pode interromper o trabalho de muitos usuários por vários minutos. Para resolver este problema, iniciou-se, em 1987, o desenvolvimento de um sistema de arquivos experimental denominado CODA.

Descendente do AFS, o CODA manteve o alto grau de escalabilidade e de segurança do seu ancestral mas passou a oferecer uma disponibilidade muito maior.

A fim de atingir os objetivos propostos, o projeto ANDREW seguiu os seguintes princípios:

1. **As estações tem ciclos para queimar, a rede e os servidores são preciosos.** Sempre que houver a possibilidade de escolher onde uma operação será efetuada, é

melhor efetuá-la nas estações de trabalho clientes. Assim, a escalabilidade é mantida e não haverá a necessidade de aumentar muito o número de servidores à medida em que novos clientes são acrescentados.

2. **Cachear sempre que possível.** Os mecanismos de controle do cache nos clientes do AFS diminuem consideravelmente o tráfego na rede e a carga nos servidores possibilitando uma maior escalabilidade e disponibilidade.

3. **Explorar os tipos de acesso aos arquivos.** Segundo [Sat90a], um terço dos arquivos UNIX são arquivos temporários que dificilmente são compartilhados e, portanto, devem ser mantidos em discos locais. Os arquivos executáveis das ferramentas UNIX são muito lidos e raramente alterados, devem, portanto, ser replicados em diversos servidores para melhorar a disponibilidade e o desempenho do sistema.

4. **Minimizar o conhecimento global em um único ponto e as mudanças globais síncronas.** Se alguma máquina assumir a responsabilidade de armazenar um conhecimento detalhado sobre o estado global da rede, o sistema pode perder em escalabilidade pois o seu crescimento estará limitado pela capacidade desta máquina. Se uma operação depender de mudanças em toda a extensão da rede, ela poderá demorar muito tempo para se concretizar pois a sua propagação pode ser lenta.

5. **Confiar no menor número possível de entidades.** Um sistema cuja segurança dependa da integridade de poucas entidades tem maiores chances de permanecer seguro quando o seu tamanho aumenta.

6. **Agrupar o trabalho sempre que possível.** Realizar uma leitura de 50 Kbytes é muito mais eficiente do que realizar 50 leituras de 1 Kbyte. Agrupando-se as operações diminui-se a sobrecarga (*overhead*) a elas associada. Embora seja bom lembrar que a latência (isto é, o tempo até a chegada dos primeiros resultados) é maior.

Vejamos, agora, como estes seis princípios se manifestaram na implementação.

### 2.4.1 Espaço de Nomes

Ao se conectarem a uma máquina ANDREW qualquer, os usuários têm ao seu dispor um sistema operacional UNIX 4.3 BSD. O espaço de nomes é dividido em duas partes. Uma delas é armazenada em discos locais e guardam basicamente informações temporárias (diretório */tmp*), o cache do cliente (diretório */cache*) e os arquivos necessários para a inicialização da máquina. A outra parte (diretório */afs*) é igual para todas as máquinas da rede e contém os arquivos compartilhados mantidos por servidores dedicados[17]. Na parte compartilhada, são armazenados os diretórios particulares dos usuários e os arquivos do sistema.

A figura 2.6 mostra o espaço de nomes de um cliente. Opcionalmente, arquivos locais podem ser associados a arquivos compartilhados através de *symbolic links*. Esta estruturação do espaço de nomes obedece ao 3º princípio do projeto ANDREW.

A parte compartilhada é dividida em sub-árvores disjuntas chamadas **volumes**. Cada volume é associado a um servidor. Os arquivos são identificados internamente por um código de 96 bits como mostra a figura 2.7.

Este identificador de arquivo (*fid*) é dividido em três grupos de 32 bits. O primeiro grupo determina o volume no qual o arquivo se encontra. O segundo, **número do** *Vnode* é um

[17] Um servidor dedicado é uma máquina que não executa processos dos usuários, a sua única função é auxiliar o fornecimento do serviço de arquivos.

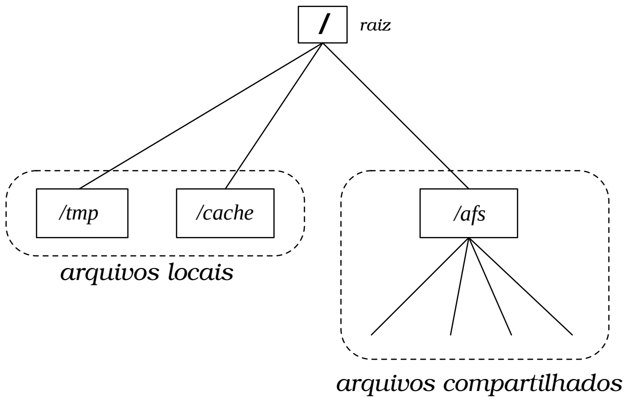



Figura 2.6: O espaço de nomes de um cliente

| 32 bits | 32 bits | 32 bits |
|---|---|---|
| número do volume | número do *Vnode* | número único |

Figura 2.7: Identificador de arquivo (*fid*)

índice para as tabela do sistema que identifica um arquivo dentro do volume. O terceiro grupo, **número único** possibilita a reutilização do número do *Vnode*. Se um arquivo é apagado, o seu *Vnode* pode ser reutilizado por outro arquivo desde que o número único seja alterado evitando, assim, confusões com o arquivo antigo. Estes números não são visíveis para os processos dos usuários.

A comunicação entre o cliente e o servidor é baseada nestes identificadores de arquivos. Quando um processo executa o comando *open*, o cliente verifica se o *pathname* do arquivo começa com */afs* descobrindo se o arquivo é local ou se está na área compartilhada. Se o arquivo for local, o sistema de arquivos UNIX local completa o serviço.

Os servidores mantém uma cópia completa do **banco de dados de localização** que mapeia volumes em servidores. Se o arquivo está na área compartilhada, o cliente executa *lookups* do *pathname*, componente a componente percorrendo vários servidores até encontrar o identificador do arquivo. Em posse do identificador, ele verifica se existe uma cópia válida do arquivo no seu cache e toma as providências necessárias. Veremos como o cache é tratado na seção 2.4.5.

O fato de todos os servidores possuírem uma cópia do banco de dados de localização limita a escalabilidade pois contraria o 4º princípio do projeto do AFS. Porém, configurando os volumes de modo que eles sejam relativamente poucos é possível obter a escalabilidade desejada.

Note que, ao contrário do NFS, onde os clientes mantém tabelas com informações sobre os diretórios remotos, toda a informação sobre os nomes é armazenada nos servidores. Deste modo, a manutenção dos clientes é trivial e a uniformidade do espaço de nomes é uma conseqüência natural da configuração dos servidores (que são poucos se comparados aos clientes). Esta atitude não obedece ao 1º princípio que manda concentrar o trabalho nos clientes sempre que possível. A fim de minimizar esta desobediência, os clientes mantém um cache que armazena informações sobre a localização dos volumes.

Uma grande desvantagem do AFS em relação ao NFS, porém, é a dificuldade de configuração dos servidores. No NFS basta alterar uma linha do arquivo */etc/exports* para determinar que um diretório poderá ser acessado por um conjunto de clientes. Segundo Satyanarayanan [Sat90b], configurar uma máquina para ser um servidor ANDREW é uma tarefa pesada.

### 2.4.2 Replicação

O AFS oferece a possibilidade da migração de volumes de maneira atômica. Inicialmente, o volume é copiado no novo servidor. Após o termino da cópia de todos os arquivos do volume, a cópia antiga é apagada. Durante um certo período, o servidor antigo ainda recebe as solicitações ao volume migrado e os remete para o novo servidor. Isto ocorre enquanto os bancos de dados de localização são paulatinamente atualizados como manda o 4º princípio.

O mesmo mecanismo é utilizado para a replicação de volumes e para a feitura de cópias de segurança (*back-up copies*) suprimindo-se a operação final que apaga o volume original. No caso de volumes replicados, os bancos de dados de localização armazenam a localização de todas as cópias sendo que apenas uma delas é considerada alterável, as demais funcionam apenas para leituras. As cópias de segurança são feitas da seguinte forma. Inicialmente, produz-se um clone do volume. Este clone não é alterado nas 24 horas seguintes. Neste período, um processo do sistema faz uma cópia deste clone para fita magnética. A grande vantagem deste método é que os usuários podem recuperar, sozinhos, arquivos apagados acidentalmente nas últimas 24 horas acessando o clone do volume.

Um cliente AFS sempre acessa a cópia do volume que está mais próxima. Se um servidor não responde aos chamados de um cliente devido a problemas com a rede ou com ele mesmo, o cliente passa automaticamente a acessar os mesmos dados de outra cópia.

Como veremos na seção 2.5, a equipe do ANDREW está agora empenhado no desenvolvimento do sistema CODA que utiliza um esquema otimista de replicação com a finalidade de aumentar a disponibilidade do serviço de arquivos.

### 2.4.3 Arquitetura

Tanto os servidores quanto os clientes executam versões ligeiramente modificadas do UNIX BSD. Acima do *kernel*, clientes e servidores executam softwares totalmente diferentes.

Os clientes executam todo o tipo de aplicação que os usuários podem precisar como sistemas de janelas, editores de texto, os utilitários do UNIX. Já os servidores executam apenas o *vice* que é o programa responsável por atender às solicitações ao serviço de arquivos que chegarem ao servidor. Desde o AFS-2, o *vice* é *multi-threaded* e, desta forma, consegue atender a várias solicitações simultâneas eficientemente. Além das aplicações dos usuários, os clientes executam também o *venus* que é responsável pela interface entre o cliente e o *vice*. Inicialmente implementado como um processo comum, o *venus* foi integrado ao *kernel* dos clientes a fim de melhorar o seu desempenho.

A figura 2.8 mostra a organização de uma **célula** AFS. A idéia do sistema é confinar a maior parte do tráfego dentro dos *clusters* reduzindo, assim, a carga na espinha dorsal. Em alguns casos, isto não é possível. Se um aluno de computação da *Carnegie Mellon* se conecta a uma máquina da Escola de Belas Artes, será necessário transportar alguns de seus arquivos da sua rede local no prédio da Escola de Ciência de Computação para uma rede local no prédio da escola de Belas Artes. Este transporte é feito através da espinha dorsal.

A grande inovação do AFS-3 foi a possibilidade de integração de diversas células (como a da figura 2.8) em um grande sistema de arquivos de dimensões mundiais [ZE88].

Atualmente existem cerca de cem destas células em diversos países como Estados Unidos, Inglaterra, Suécia, Alemanha, Suiça, Austrália e Japão. Todas elas integradas através da Internet (figura 2.9). Cada célula possui autonomia, os administradores locais são responsáveis

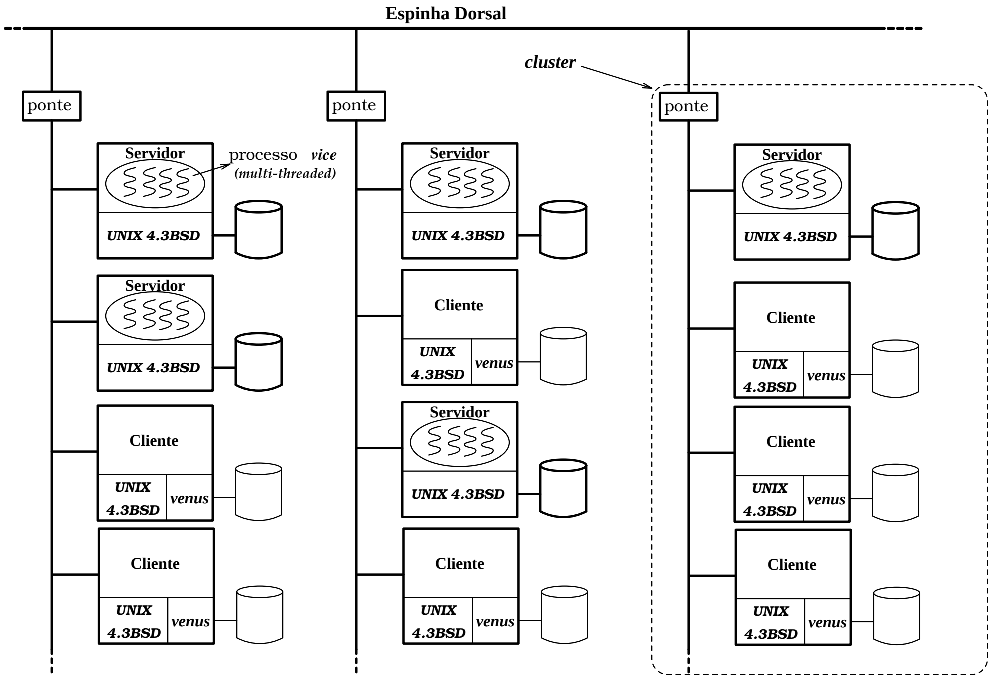



Figura 2.8: Uma *célula* do AFS

pelos servidores e clientes de sua célula. A maior parte do tráfego não sai de dentro da célula que o originou, mas é possível acessar diretórios pertencentes a outras células.

Um usuário que se conecta a um cliente na *Carnegie Mellon University*, por exemplo, terá acesso a diretórios presentes em dezenas de outras células. O diretório */afs* da sua estação de trabalho contém um subdiretório para cada célula acessível através da Internet. Assim, se o usuário listar o diretório */afs*, ele verá o que é chamado de *árvore de diretórios AFS-Internet*. Se ele for do departamento de Ciência da Computação, o diretório contendo os seus arquivos particulares será */afs/cs.cmu.edu/users/<nome.do.usuário>* e poderá ser acessado de diversas partes do mundo.

O acesso a arquivos ou diretórios de células remotas pode ser facilitado através da criação de *symbolic links*, fora isso, o AFS não oferece transparência de localização a nível de células. Para acessar um arquivo é preciso saber em qual célula ele está armazenado. O sistema PROSPERO [Neu92] oferece uma solução alternativa para a organização de redes globais de grande escala possibilitando o acesso de maneira mais transparente.

### 2.4.4 Segurança

Como comentamos no início deste capítulo, um sistema do porte do AFS não pode confiar nas boas intenções dos milhares de usuários que compartilham os mesmos arquivos. Se a política do SPRITE de confiar em todos os clientes[18] fosse adotada a segurança do sistema inteiro estaria comprometida uma vez que seria praticamente impossível controlar a integridade do *venus* nos milhares de clientes.

Obedecendo ao 5º princípio, o AFS confia apenas nos servidores [Sat89] que, por sua vez, aparecem em um número muito menor do que os clientes. As estações clientes podem ser

[18] Ver seção 2.6.2.

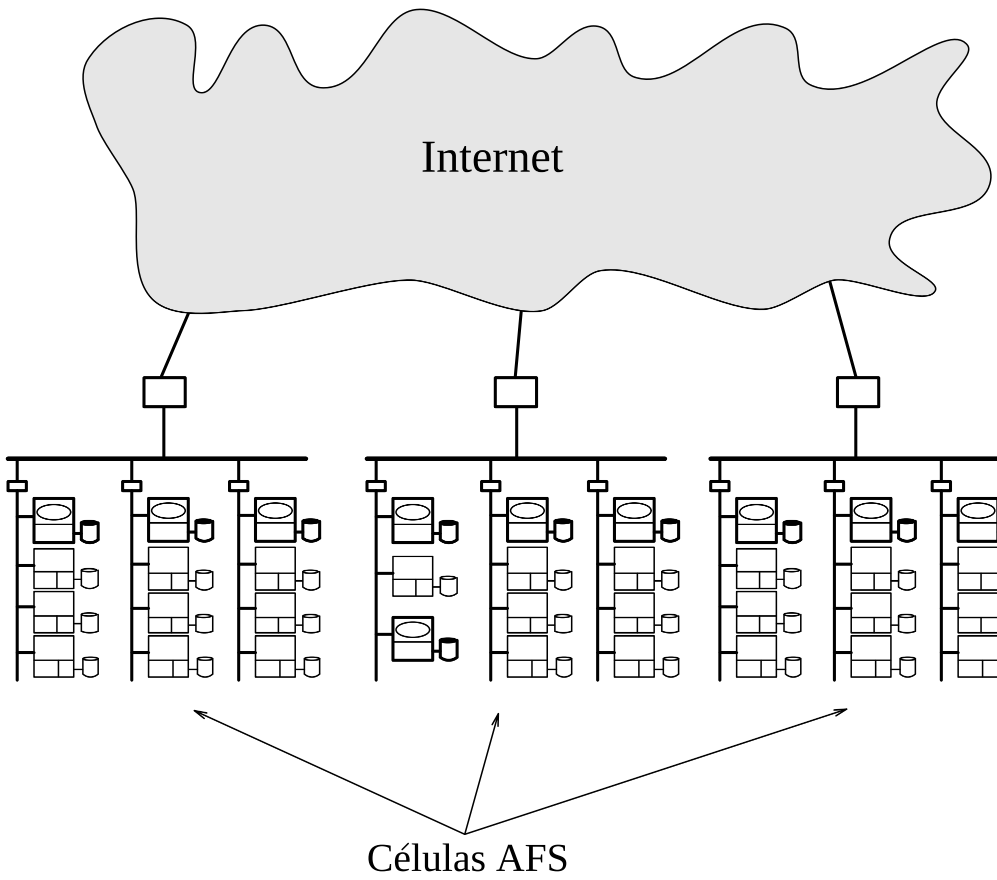



Figura 2.9: Arquitetura global do AFS

públicas ou privadas, seus donos tem a permissão de alterar livremente a sua configuração. Já os servidores, são operados apenas pelos administradores do sistema. Os usuários não podem nem executar processos remotos nestes servidores.

O AFS-3 utiliza o protocolo KERBEROS [SNS88, Jas] para a autenticação mútua entre clientes e servidores. Graças a este protocolo, as senhas dos usuários não trafegam descriptografadas pela rede. Além disso, após uma autenticação, o cliente recebe um bastão (*token*) que é utilizado para acessar o sistema de arquivos nas 24 horas seguintes. Logo, as senhas criptografadas trafegam na rede, no máximo, uma vez por dia.

Para se conectar a um cliente, o usuário deve fornecer a sua senha. O cliente usa esta senha para obter um bastão do *Kerberos Authentication Server* (KAS). O KAS é implementado através de um servidor mestre que é o único capaz de realizar alterações no banco de senhas e pelos demais servidores de arquivos, que possuem cópias, só para leitura, do mesmo banco de senhas. O bastão fornecido pelo KAS é utilizado, nas 24 horas seguintes, para a autenticação das RPCs que implementam o protocolo de acesso aos arquivos.

A proteção a cada arquivo é baseada em listas de controle de acesso que especificam quais grupos de usuários, ou individualmente, quais usuários têm acesso a cada diretório e qual o tipo de acesso permitido. Uma particularidade do ANDREW é a possibilidade de especificar direito negativo de acesso. Pode-se, por exemplo, configurar o sistema de modo que todos usuários menos Antônio Paulo Orestes de Mello tenham acesso a um determinado diretório.

Um grupo especial é o *System:Administrators* que possui acesso irrestrito a qualquer arquivo do sistema. A vantagem da existência deste grupo sobre o usuário *root* do UNIX é que a identidade do usuário que faz uso deste privilégio irrestrito é conhecida e pode ser verificada quando for necessário.

Os bits de permissão do UNIX são inspecionados pelo *venus* mas são ignorados pelo *vice*. Logo, para realmente proteger um arquivo da ação de clientes pouco confiáveis é necessário utilizar as listas de controle de acesso.

### 2.4.5 Cache

A grande particularidade do cache dos clientes é a sua localização. Diferentemente da grande maioria dos sistemas, os clientes do AFS guardam os arquivos cacheados nos seus discos locais, mais especificamente, no diretório */cache*.

Quando um processo abre um arquivo, o *venus* tenta descobrir o seu identificador. Em posse deste, ele verifica se existe uma cópia válida deste arquivo em seu cache. Se tal cópia não existe, o arquivo inteiro é trazido para o disco local obedecendo ao 6º princípio. O *venus* devolve ao processo que executou o *open* um identificador da cópia no disco local.

A partir daí, todas as operações com este arquivo são realizadas com a cópia local. Quando o processo fecha o arquivo (ou quando executa a *system call fsync*), o *venus* verifica se ele foi alterado. Caso tenha sido, ele é enviado para o servidor responsável.

Assim, fica claro que o AFS não pratica a semântica UNIX de acesso concorrente a arquivos, mas sim, a semântica de sessão onde os dados alterados só se tornam visíveis para outros clientes após o *close*. Este tipo de semântica limita o uso do sistema de arquivos como ferramenta de interação entre as máquinas de uma rede mas, ao contrário do comportamento do NFS, é algo bem determinado e que produz resultados previsíveis.

Note que a maior parte do trabalho de implementação do *venus* está na alteração das *system calls open* e *close*. As chamadas *read* e *write* não sofrem grandes modificações em relação às suas implementações no UNIX uma vez que operam nas cópias locais dos arquivos.

O fato dos processos acessarem o sistema de arquivos UNIX local faz com que a semântica entre dois processos de uma mesma máquina seja a semântica UNIX podendo levar a situações estranhas.

Suponha, por exemplo, que um processo $P_1$ abra um arquivo e que, 20 minutos depois, outro processo, $P_2$, abra o mesmo arquivo. Dependendo da máquina na qual os processos são executados é possível que $P_1$ acesse uma versão mais recente do que $P_2$.

A consistência da semântica de sessão entre os clientes é mantida através de um mecanismo chamado *callback*. Quando o *servidor* fornece um arquivo para o *venus*, o *vice* fornece um *callback* que é uma promessa de que o arquivo no cache do cliente é a versão mais recente. *Callbacks* podem ser quebrados por um cliente (quando atualiza a sua cópia local) ou por um servidor (quando recebe uma nova versão do arquivo de algum cliente). Neste último caso, o servidor envia uma mensagem a todos os clientes que possuem *callbacks* para este arquivo quebrando-os. A cópia local de um arquivo pode ser usada várias vezes antes de ser invalidada por determinação do servidor.

Os *callbacks*, introduzidos no AFS-2, evitam que a rede tenha que ser acessada cada vez que um arquivo é aberto como era feito no AFS-1. Graças aos *callbacks* foi também possível implementar um cache de diretórios e resolver *pathnames* localmente. Por segurança, modificações nos diretórios são feitas diretamente no servidor.

No AFS-3, os arquivos deixaram de ser cacheados em sua totalidade e passaram a ser transferidos, conforme a necessidade, em grandes blocos de 64 Kbytes. Assim, reduz-se a latência do *open* e torna-se possível o acesso a arquivos grandes que não cabem no disco local. O CODA ainda mantém a política do AFS-2 de cachear os arquivos inteiros.

A chamada UNIX *flock* é emulada permitindo o controle do acesso concorrente aos arquivos. Os bloqueios criados por esta chamada são administrados pelo servidor responsável pelo arquivo. Se um cliente segura um bloqueio por mais de 30 minutos, ele é automaticamente liberado pelo servidor evitando que a queda de um cliente impossibilite o acesso a um arquivo por muito tempo.

### 2.4.6 Resumo

O AFS oferece um serviço altamente escalável através da adoção da semântica de sessão no acesso concorrente a arquivos e da utilização de grandes caches no disco local dos clientes.

Os clientes devem, necessariamente, possuir discos locais e o espaço de nomes é dividido em uma parte local e uma parte compartilhada na qual o espaço de nomes é uniforme para todas as máquinas do sistema. O espaço de nomes é organizado através de um banco de dados de localização mantido pelos servidores.

O AFS oferece um mecanismo rudimentar para replicação de arquivos que permite a replicação de diretórios só para leitura e a implementação de um serviço de criação automática de cópias de segurança.

Através da Internet, clientes AFS podem acessar arquivos em outras células AFS localizadas em qualquer ponto do globo. Atualmente, existem cerca de 100 células AFS espalhadas pelo globo.

A segurança do sistema é mantida por listas de controle de acesso verificadas pelos servidores e pelos bits de permissão do UNIX verificados pelos clientes. O protocolo KERBEROS é utilizado para permitir a autenticação mútua de clientes e servidores através de técnicas de criptografia.

Veremos, agora, as inovações introduzidas pelo CODA, um sistema desenvolvido a partir do AFS.

## 2.5 CODA

*Operação Desconectada* [KS92] é um modo de operação no qual não há a necessidade de se estabelecer contato imediato com os servidores para ler ou alterar informações do sistema de arquivos. Esta é a grande novidade do sistema CODA [SKK+90, KS92, SKM+93] que pretende com isso oferecer disponibilidade máxima aos seus usuários. Obviamente, este modo de operação só pode ser utilizado quando os dados não são acessados concorrentemente por mais de um cliente, ou então, quando são dados apenas para leitura.

Suponha, por exemplo, que um usuário esteja escrevendo um artigo utilizando um editor de textos, um formatador de textos e um visualizador de textos formatados[19] e que ele possua, em sua sala, uma estação de trabalho com um pequeno disco local mas que os arquivos com o texto do seu artigo estejam em um grande disco do servidor de arquivos. Esta é uma situação muito comum em sistemas como NFS, AFS e SPRITE. Se o servidor sofrer uma queda temporária ou ficar um dia parado para manutenção, o usuário ficará todo este tempo sem poder trabalhar com os seus arquivos. O mesmo acontecerá se algum problema de comunicação impossibilitar o acesso aos servidores.

Um usuário prevenido terá uma cópia da versão mais recente do seu arquivo gravada no seu disco local ou em disquetes. Mas, isto pode não ser suficiente pois dificilmente ele terá uma cópia do editor, do formatador e do visualizador. Resumindo, o usuário possui em sua sala todo o equipamento necessário para realizar o seu trabalho mas não tem acesso às informações que tornariam isto possível.

Como veremos a seguir, esta situação não é um problema para o CODA. E mais, esta situação é incentivada na medida em que o CODA é um sistema ideal para trabalhar com computadores portáteis que só permanecem conectados aos servidores por curtos períodos de tempo. Além da operação desconectada, o CODA utiliza a replicação de arquivos como meio de aumentar a disponibilidade.

Em 1993, o CODA chegou a ser utilizado por 30 usuários através de 25 computadores portáteis e de 15 estações de trabalho fixas. Os computadores portáteis eram, em sua maioria, laptops IBM PS2/L40 baseados no microprocessador 80386. Três servidores eram responsáveis por três réplicas de 150 volumes. O número de usuários tende a crescer muito nos próximos anos pois a procura pelo sistema dentro da *Carnegie Mellon* tem sido grande.

### 2.5.1 Replicação

Ao contrário do AFS, no CODA os volumes podem possuir várias cópias que podem ser tanto lidas quanto alteradas. Cada volume possui um grupo de servidores que o replicam, o *VSG* (*volume storage group*). O subconjunto de servidores do VSG que estão acessíveis por um cliente em um determinado instante é o AVSG (*accessible VSG*) do cliente. A coerência entre as diversas cópias de um arquivo é mantida através dos mesmos *callbacks* do ANDREW. Quando um usuário envia para o servidor um arquivo modificado, o servidor emite uma mensagem quebrando o *callback* de todos os outros clientes que o cacheavam. As alterações são propagadas, em paralelo, para todos os servidores do AVSG e, posteriormente, para os demais servidores do VSG. Para isso, os clientes enviam as alterações através de uma multiRPC [SS90] que são recebidas e tratadas por todos os servidores do AVSG.

Quando um servidor se reintegra à rede, nenhuma providência é tomada para a atualização dos arquivos, o método adotado é o seguinte. Quando um cliente não encontra um arquivo em seu cache, ele envia uma solicitação deste arquivo para o seu **servidor preferencial** e uma mensagem a todos os demais servidores do AVSG solicitando o número da versão do arquivo.

[19] Como, por exemplo, o emacs, LaTeXe Xdvi.

Se o cliente descobre que o seu servidor preferencial está com uma versão antiga do arquivo, o cliente solicita a versão mais recente de quem a possuir. Além disso, se um cliente detecta que algum dos servidores do AVSG está com uma versão desatualizada, ele emite uma mensagem ao AVGS informando onde está o problema.

Em suma, quem toma a iniciativa para a detecção das inconsistências entre as réplicas dos arquivos após uma reintegração são os clientes e não os servidores.

Como vimos em 1.1.11 esta replicação por si só aumenta a disponibilidade do sistema de arquivos pois a probabilidade de um dos servidores do VSG estar acessível é maior do que a probabilidade de um único servidor estar acessível.

Mas, se o CODA permite operações desconectadas, por que haveria a necessidade de replicar arquivos em vários servidores? Isto é necessário porque as cópias dos arquivos nos servidores são mais seguras, publicamente acessíveis, e o espaço disponível nos servidores é bem maior do que o dos clientes. Basta um *laptop* cair no chão para que todas as informações nele contidas sejam perdidas.

O CODA tenta unir as qualidades da replicação em servidores fisicamente seguros aos ganhos em velocidade e disponibilidade que o cache nos clientes possibilita.

Por outro lado, como o CODA não garante que as réplicas estejam sempre sincronizadas, a replicação do CODA não implica em um serviço tão robusto quanto o HARP onde cada byte escrito é replicado imediatamente como veremos na seção 2.8.

### 2.5.2 Controle da Consistência das Réplicas

[DGMS85] discute os mecanismos de manutenção da consistência entre as réplicas de um arquivo quando ocorre particionamento da rede. Os autores dividem as estratégias existentes para atacar este problema em dois grupos:

1. Uma **estratégia pessimista** evita inconsistências limitando o acesso às réplicas. Desabilita-se as alterações aos arquivos replicados ou então limita-se as leituras e escritas a uma das partes da partição.

2. Uma **Estratégia otimista** permite leituras e escritas indiscriminadamente e tenta resolver os possíveis conflitos quando a partição termina.

Obviamente, a estratégia otimista permite uma maior disponibilidade mas o preço desta disponibilidade é a possível perda de informações e o acesso a dados inconsistentes. Se clientes em partições distintas alteram o mesmo arquivo, qual será a cópia válida após a reintegração da rede?

A estratégia pessimista é ideal para ambientes com maior compartilhamento e a otimista para ambientes com pouco compartilhamento.

A escolha do CODA foi a mais otimista possível permitindo as operações desconectadas. Vejamos como este modo de operação foi implementado.

### 2.5.3 Os estados do *venus*

O *venus*, software responsável pelo sistema de arquivos nos clientes, pode estar em um de três estados: **salvaguarda** (*hoarding*), **emulação** ou **reintegração**. O estado de salvaguarda é o seu estado normal. Aqui, a comunicação com os servidores é possível sempre que necessário mas o cliente procura estar sempre preparado para uma eventual perda da comunicação. Quando ocorre uma partição, quer seja ela voluntária (no caso da desconexão de um computador portátil da rede) ou não, o *venus* entra no estado de emulação e passa a fazer o papel de servidor enquanto pelo menos um dos verdadeiros servidores não estiver acessível. Logo que for possível o contato com algum servidor, o *venus* entra no estado de reintegração e passa a fornecer para os servidores responsáveis, os arquivos de seu cache que sofreram alterações. Depois disso, ele volta ao estado inicial de salvaguarda. Um cliente pode estar em diferentes estados em relação a diferentes volumes.

### Salvaguarda

No estado de salvaguarda, o cliente procura manter em seu cache uma cópia de todos os arquivos que ele possa precisar no caso de uma partição. Por outro lado, é preciso manter no cache, também, os arquivos que os processos estão utilizando no momento. A escolha de quais arquivos devem ser cacheados é feita através de um algoritmo misto.

O algoritmo atribui dinamicamente prioridades aos diversos arquivos utilizados pelo cliente. Estas prioridades são calculadas através de uma combinação de uma fonte explícita e outra implícita. A fonte implícita é a história recente de acesso aos arquivos. Arquivos utilizados mais recentemente recebem prioridade maior.

Já a fonte explícita, é um banco de dados (*hoard database - HDB*) mantido por cada estação cliente. O HDB contém os *pathnames* dos objetos a cachear com as suas respectivas prioridades. O CODA fornece algumas ferramentas para a atualização do HDB. É possível registrar, através de um programa chamado *spy*, todos os acessos ao sistema de arquivos durante um período determinado pelo usuário.

O usuário dispõe de comandos para acrescentar ou remover entradas do HDB de sua estação. Uma série destes comandos podem ser especificados em arquivos chamados de perfis de salvaguarda (*hoard profiles*) como mostra a figura 2.10 extraída de [KS92]. O comando `a` no

```
# Personal files
a /coda/usr/jjk d+
a /coda/usr/jjk/papers 100:d+
a /coda/usr/jjk/papers/sosp 1000:d+

# System files
a /usr/bin 100:d+
a /usr/etc 100:d+
a /usr/include 100:d+
a /usr/lib 100:d+
a /usr/local/gnu d+
a /usr/local/rcs d+
a /usr/ucb d+
```

Figura 2.10: Perfil de salvaguarda

início de cada linha adiciona uma nova entrada ao HDB. Se nenhuma prioridade é especificada, assume-se o *default* que é 10 [20]. Tomemos, como exemplo, o segundo comando `a` da figura 2.10. Ele determina que o diretório `/coda/usr/jjk/papers` seja cacheado com prioridade 100. A letra `d` após o sinal `:` indica que este comando se aplica a todos os descendentes deste diretório. O caractere `+`, por sua vez, indica que o comando `a` se aplica também a eventuais descendentes deste diretório que possam surgir no futuro.

Quando o espaço no cache termina e há a necessidade de acessar um arquivo não cacheado, o arquivo com menor prioridade (levando-se em conta os dois aspectos do algoritmo misto) é descartado. Para possibilitar a resolução de *pathnames* no modo desconectado, nenhum arquivo pode ser cacheado sem que todos os diretórios do seu *pathname* também o estejam. O *venus*

[20] Prioridades maiores indicam maior necessidade de cachear o objeto.

resolve este problema atribuindo prioridade infinita aos diretórios que possuem descendentes cacheados.

O cache é considerado em **equilíbrio** quando nenhum objeto não cacheado possui prioridade maior do que um objeto cacheado. A atividade normal de um usuário pode levar o cache ao desequilíbrio. Suponha, por exemplo, que um arquivo que não conste do HDB seja acessado e que, para ele ser cacheado, um arquivo presente no HDB tenha que ser eliminado do cache. Imediatamente a sua prioridade vai ser elevada pois ele acabou de ser usado. Mas, se ele não for acessado novamente, à medida que o tempo vai passando, a sua prioridade vai diminuindo e, após um certo tempo, ficará menor do que a prioridade do arquivo que foi eliminado do cache. Perdeu-se, então, o equilíbrio.

Quebras de *callbacks* também podem gerar o desequilíbrio do cache. O equilíbrio é restabelecido através de um processo (*hoard walk*) que é executado automaticamente de 10 em 10 minutos. Este processo analisa o HDB, consulta o servidor para descobrir se novos descendentes de diretórios foram criados, e, calculando a prioridade de cada objeto, decide o que fazer para restabelecer o equilíbrio. Um *hoard walk* também pode ser ativado por um usuário antes de uma desconexão voluntária.

Concomitantemente, o cliente mantém em sua posse uma certa quantidade de identificadores de arquivos (*fids*) que poderão ser úteis se um processo solicitar a criação de um arquivo quando o cliente estiver no estado de emulação.

### Emulação

Quando o cliente perde o acesso a um determinado volume, se inicia o estado de emulação. Neste estado, o cliente assume algumas das responsabilidades dos servidores como o controle total do acesso aos arquivos e o fornecimento de *fids* para os arquivos novos. No entanto, qualquer alteração ao sistema de arquivos tem que ser confirmada posteriormente por um servidor responsável pelo volume.

O cache é controlado através do mesmo algoritmo utilizado no estado de salvaguarda com a exceção de que arquivos ou diretórios alterados recebem prioridade infinita e não podem ser descartados em hipótese nenhuma. Quando um processo tenta acessar um objeto não cacheado, o *venus* devolve um código de erro. Opcionalmente, pode-se configurá-lo para bloquear o processo até que ele possa ser atendido.

As alterações no sistema de arquivos efetuadas durante o período de emulação são armazenadas em um *replay-log* que permite reconstituir o estado do sistema quando a conexão com um servidor responsável pelo volume for restabelecida.

A operação no modo desconectado é limitada pelo espaço de disco disponível para o cache local. Este problema ainda não foi bem resolvido pelo CODA. Uma possível solução é fazer cópias em disquete de arquivos modificados e apagá-los do disco rígido. Quando o espaço reservado para o *replay-log* termina, o CODA não mais permite alterações a nenhum arquivo. Este problema deve ser minimizado no futuro através de mecanismos que possibilitem a compressão do cache e a cópia de arquivos modificados para disquetes de maneira automática.

### Reintegração

Quando a comunicação com um servidor responsável por um volume anteriormente inacessível é restabelecida, inicia-se o processo de reintegração. Neste período, o *venus* propaga todas as alterações efetuadas no período de emulação e atualiza o seu cache.

A propagação das alterações é feita em duas etapas. Inicialmente, se houve a necessidade do *venus* fornecer *fids* temporários para arquivos criados durante o estado de emulação, envia-se ao servidor um pedido de *fids* permanentes que substituirão os temporários no *replay-log*.

Em seguida, o *replay-log* é enviado, em paralelo, para todos os servidores do AVSG e executado independentemente em cada um deles. Cada servidor executa o *log* dentro de uma única transação[21], se a atualização de um arquivo falha, então toda a transação é cancelada. O *replay-log* é processado em quatro fases:

1. A transação é iniciada e todos os arquivos referenciados no *log* são bloqueados.

2. As operações do *log* são validadas, isto é, verifica-se se há espaço em disco para realizá-las, se não há violação de proteções e se nenhuma operação implica em perda da consistência do sistema de arquivos. Uma operação considerada válida é executada a não ser que ela resulte na transferência de um arquivo. Neste último caso, as meta-informações sobre o arquivo são atualizadas mas a transferência do seu conteúdo é adiada para a fase três.

3. O conteúdo dos arquivos alterados são efetivamente trazidos para o servidor. O adiamento da transferência do conteúdo dos arquivos para esta fase possibilita que as dificuldades na execução do *replay-log* sejam detectadas mais rapidamente.

4. Os bloqueios são liberados e a transação é completada (*commited*).

Se a reintegração falha, o *venus* cria um *replay-file* no seu disco local. Através de ferramentas oferecidas pelo CODA, é possível examinar o *replay-file*, comparar o seu conteúdo com os arquivos do AVSG e determinar qual foram as causas da falha da reintegração. A partir daí é possível executar uma nova tentativa de reintegração eliminando as operações que causaram problemas.

### 2.5.4 Tratamento dos Conflitos

Durante o estado de emulação, podem ser criadas situações de conflito que comprometem a integridade do sistema de arquivos:

1. Clientes distintos podem realizar alterações conflitantes em um mesmo arquivo.

2. Um cliente pode ler dados que foram alterados há várias horas, ou até mesmo dias, por outro cliente.

Mas, segundo [KS92] o único tipo de conflito que interessa aos projetistas do CODA são os do primeiro tipo. Eles argumentam que conflitos do segundo tipo são irrelevantes para o modelo UNIX de sistemas de arquivos pois, no UNIX, não existe a noção de atomicidade além das fronteiras de uma *system call*.

É certo que o UNIX não possui a noção de atomicidade além das fronteiras das *system calls* mas também é verdade que a semântica UNIX pressupõe que um *read* obtém os dados armazenados pelo último *write*. É sabido que o AFS não respeita a semântica UNIX mas sim a semântica de sessão na qual um *read* obtém os dados armazenados no último *close*. Mas, o CODA não respeita nem a semântica de sessão. No modo desconectado, um cliente pode abrir um arquivo, alterá-lo e fechá-lo sem que estas alterações se tornem visíveis para os outros clientes. É o preço que se paga pelo ganho em disponibilidade.

Talvez a melhor saída para o CODA seria a detecção do segundo tipo de conflito durante o processamento do *replay-log*. Os usuários que efetuaram alterações que resultaram em conflitos receberiam uma mensagem de alerta.

[21] As transações são administradas pelo RVM [SMK93] que é um pacote que permite a execução atômica de conjuntos de operações sobre dados armazenados em memória virtual.

O tratamento dos conflitos do primeiro tipo baseia-se no *storeid* que é um registro que identifica unicamente a última alteração de uma cópia de um certo arquivo. Durante a segunda fase do processamento do *replay-log*, o servidor compara o *storeid* dos objetos citados no *log* com os *storeids* no servidor. Se alguma diferença é encontrada, tem-se um conflito[22].

Se um conflito é detectado, toda a transação é abortada e o *venus* cria um *replay-file*.

### 2.5.5 Desempenho

Após a coleta de dados relativos à utilização do CODA em períodos contínuos de um dia e de uma semana foi possível mostrar que discos locais de 100 Mbytes são suficientes para a operação no modo desconectado pelos atuais usuários do CODA por períodos de até uma semana.

O tempo gasto no estado de reintegração se mostrou suportável. Reintegrações após um dia de trabalho desconectado consumiriam cerca de 1 minuto. Após uma semana, menos de 5 minutos. Obviamente, estes valores dependem das operações efetuadas no momento das coletas dos dados e dão apenas uma visão aproximada do comportamento do CODA. Maiores detalhes podem ser obtidos em [SKM+93].

Um estudo anterior [KS92] mostrou que a adoção da estratégia otimista na replicação dos dados pelo CODA não causaria muitos conflitos *write-write*, isto é, conflitos do primeiro tipo. Analisando os acessos ao sistema de arquivo de 400 usuários do AFS em um período de um ano, chegou-se a conclusão de que mais de 99% das alterações aos arquivos são realizadas pelo mesmo usuário que efetuou a alteração anterior ao mesmo arquivo.

A primeira linha da tabela 2.2, por exemplo, mostra os resultados deste estudo no caso de arquivos compartilhados por equipes desenvolvendo projetos. Verificou-se que 0,34% das alterações a este tipo de arquivo são efetuadas por usuários que não foram os últimos a acessarem o mesmo arquivo. Isto nos dá uma idéia da possibilidade de ocorrerem conflitos que abortariam processamento de um *replay-log*.

| Tipo dos volumes | Num. de volumes | Total de alterações | Mesmo usuário | Alterações efetuadas por usuários distintos | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Total | 1 min | 10 min | 1 hora | 1 dia | 1 sem |
| projetos | 108 | 4.437.311 | 99,66% | 0,34% | 0,17% | 0,25% | 0,26% | 0,28% | 0,30% |
| usuários | 529 | 3.287.135 | 99,87% | 0,13% | 0,04% | 0,05% | 0,06% | 0,09% | 0,09% |
| sistema | 398 | 5.526.700 | 99,17% | 0,83% | 0,06% | 0,18% | 0,42% | 0,72% | 0,78% |

Tabela 2.2: Conflitos no AFS em um período de um ano

A última linha da coluna rotulada 1 hora indica que se um administrador do sistema operar no modo desconectado durante uma hora e realizar uma alteração em um arquivo do sistema, a probabilidade de que outro usuário altere o mesmo arquivo neste período é de 0,42%. Grosso modo, se um usuário médio operar desconectadamente durante uma semana sem se preocupar com a consistência das suas alterações em relação às alterações efetuadas pelos demais usuários e realizar 100 alterações em arquivos de usuários, o seu *replay-log* irá abortar com 9% de probabilidade. Estes números parecem toleráveis e só incentivam o uso do modo desconectado.

[SKK+90] apresenta o resultado de comparações entre o desempenho do AFS e do CODA baseados no *Andrew Benchmark*[HKM+88]. O *Andrew Benchmark* é um teste muito usado para a análise do desempenho de sistemas de arquivos. Ele é formado por cinco fases. *MakeDir*

[22] No caso de diretórios, um conflito só é considerado relevante se um nome criado pelo *replay-log* colide com um nome já presente no diretório.

constrói uma sub-árvore de diretórios para acomodar o código-fonte do AFS; *Copy* copia o código-fonte para esta sub-árvore; *ScanDir* percorre todos os arquivos examinando os seus atributos; *ReadAll* lê o conteúdo de todos os arquivos duas vezes; finalmente, a fase *Make* compila e liga todo o código-fonte.

Mesmo nos casos em que os arquivos do CODA eram replicados em 4 servidores, o seu desempenho piorou no máximo 5% em relação ao AFS sem replicação.

Por outro lado, o CODA se mostrou significativamente menos escalável do que o AFS. O tempo necessário para a execução simultânea do *Andrew Benchmark* por 10 clientes de um mesmo servidor foi cerca de 60% maior no CODA com 4 réplicas do que no AFS.

### 2.5.6 Resumo

O CODA apresenta inovações que facilitam a conexão (e desconexão) de computadores portáteis a redes de computadores. O usuário que dispor de um *laptop* pode estabelecer uma rápida conexão com a rede do seu local de trabalho (através de uma ligação física, por linha telefônica ou, até mesmo, por ondas de rádio), ordenar um *hoard walk*, desconectar-se e, em seguida, trabalhar em sua casa como se ainda estivesse conectado. Posteriormente, basta restabelecer a conexão para que todas as alterações realizadas no modo desconectado sejam automaticamente transmitidas para os servidores.

A abordagem otimista da replicação dos dados tanto nos servidores quanto nos clientes (operações desconectadas) aumenta em muito a disponibilidade do serviço de arquivos porém diminui as garantias de consistência no acesso concorrente.

Os testes realizados pelos autores do sistema onde cada arquivo possuía 4 réplicas mostraram uma perda de menos de 5% no desempenho em relação ao AFS sem replicação. Em termos de escalabilidade, no entanto, o CODA mostrou perdas significativas em relação ao AFS.

## 2.6 SPRITE

O *SPRITE Network Operating System* [OCD+88] vem sendo desenvolvido desde a segunda metade da década de 80 na Universidade da California em Berkeley. O projeto, coordenado por John Ousterhout, serviu como base para várias teses de doutorado da divisão de Ciência da Computação daquela universidade. Os principais trabalhos publicados sobre o sprite podem ser encontrados através de FTP anônimo em *sprite.berkeley.edu*.

O núcleo do SPRITE foi escrito desde o início sem aproveitar o código de outros sistemas. A versão inicial, de 1988, era composta por 100.000 linhas em C das quais cerca de metade eram comentários. Apenas alguns trechos críticos foram escritos em linguagem de montagem. A versão atual do SPRITE pode ser obtida por um preço simbólico em CD-ROM podendo rodar em *SPARCstations* ou em *DECstations*.

O núcleo do SPRITE é *multithreaded* o que significa que várias tarefas podem ser executadas concorrentemente dentro do *kernel*. Como conseqüência, o SPRITE pode aproveitar muito bem o hardware de máquinas com vários processadores.

A comunicação entre as máquinas da rede é feita através de rápidas RPCs de *kernel* para *kernel*.

O SPRITE não é apenas uma coleção de serviços de rede, é algo mais próximo daquilo que podemos chamar de Sistema Operacional Distribuído . Além do sistema de arquivos transparentemente distribuído, o SPRITE possibilita migrações de processos também de modo transparente ao usuário a fim de aproveitar os recursos computacionais de máquinas com pouca carga.

As principais inovações introduzidas pelo SPRITE foram:

- Do ponto de vista do serviço oferecido.
  - Semântica UNIX em acessos concorrentes a arquivos distribuídos.
  - Migração de processos transparente ao usuário e ao processo.
- Do ponto de vista da implementação.
  - Grandes caches de tamanho variável.
  - Resolução dos *pathnames* através de tabelas de prefixos.
  - Memória virtual implementada através de arquivos comuns.

No SPRITE, o sistema de arquivos se apresenta como uma única árvore de diretórios UNIX[23]. Esta árvore é a mesma em todas os nós da rede e a localização física dos arquivos é totalmente transparente para o usuário comum. As *system calls* de manipulação de arquivos do UNIX funcionam com a mesma sintaxe.

Os arquivos de *swap* do sistema de memória virtual são implementados como arquivos comuns no mesmo espaço de nomes dos arquivos dos usuários[24].

Como o sistema de arquivos é comum para todos os nós da rede, fica fácil implementar a migração de processos [DO91]. Basta congelar um processo, jogar as suas páginas para a memória virtual no sistema de arquivos, enviar para outra máquina o conteúdo dos registradores e a sua entrada na tabela de processos e descongelar o processo na nova máquina.

[23] A rigor, o sistema de diretórios do UNIX não é uma árvore, mas sim, um grafo que pode conter circuitos construídos através do comando *link*.

[24] Nos outros sistemas, as páginas da memória virtual são armazenadas em uma partição específica dos discos rígidos chamada "partição de *swap*".

É bom lembrar que a migração não pode ser feita entre máquinas que não possuem linguagens de máquina compatíveis. Assim, se a prioridade é distribuir a carga na rede através de migrações, a heterogeneidade do hardware deve ser limitada.

Embora não seja obrigatório, os servidores costumam se dedicar apenas ao serviço de arquivos e a maior parte dos clientes não possuem discos locais.

### 2.6.1 Tabelas de Prefixos (Resolução dos *pathnames*)

A árvore de diretórios do SPRITE é dividida em sub-árvores disjuntas chamadas de **domínios** (o termo do SPRITE para os volumes do ANDREW). Cada servidor é responsável por um ou mais domínios. A tradução dos *pathnames* para a localização física dos arquivos é feita através de um processo apresentado em [WO86][25].

Cada máquina mantém uma **tabela de prefixos** que mapeia domínios em servidores. Os domínios são identificados pelo maior prefixo comum dos *pathnames* dos seus arquivos como veremos a seguir. As tabelas de prefixos começam vazias quando o sistema é inicializado e vão crescendo dinamicamente à medida em que há necessidade de acessar arquivos remotos.

A tabela 2.3 é um pequeno exemplo de tabela de prefixos. A primeira coluna contém o prefixo, isto é, o *pathname* do diretório do domínio que é mais próximo da raiz. A segunda coluna contém o nome do servidor responsável pelo domínio e a terceira coluna contém o índice da raiz deste domínio na tabela de arquivos abertos do servidor.

A última coluna poderia guardar o *pathname* do diretório no servidor mas isto acarretaria um desperdício de tempo pois o mesmo *pathname* teria que ser traduzido várias vezes.

| prefixo | servidor | índice |
|---|---|---|
| */usr* | *s3* | 7 |
| */bin* | *s2* | 2 |
| */usr/alunos* | *s1* | 12 |
| */usr/profs* | *s1* | 27 |

Tabela 2.3: Tabela de prefixos

Quando um processo solicita o acesso a um determinado arquivo fornecendo o seu *pathname*, o cliente procura a linha na sua tabela de prefixos que contém o maior prefixo daquele *pathname*. A seguir, ele envia para o servidor responsável pelo respectivo domínio o restante do *pathname* do arquivo (descontando-se o prefixo encontrado) e o índice presente na terceira coluna da sua tabela.

Em posse destas duas informações, o servidor tenta traduzir o resto do *pathname*. Se ele consegue completar a tradução localmente, então ele envia para o cliente um **designador** para o arquivo (equivalente ao *file handle* do NFS e ao *fid* do ANDREW). Com este designador, o cliente pode solicitar operações no arquivo.

Mas pode acontecer de o domínio do arquivo não estar representado na tabela de prefixos do cliente. Suponha, por exemplo, que um cliente *cl1* possua apenas as duas primeiras entradas da tabela 2.3 e que queira acessar o arquivo */usr/alunos/lista*.

Consultando a sua tabela, o cliente chegará a conclusão de que deverá pedir o diretório *alunos* ao servidor *s3* fornecendo-lhe o número 7 (índice da raiz do domínio */usr* na tabela de arquivos abertos de *s3*).

[25][LS90] também explica muito bem o funcionamento das tabelas de prefixos.

Quando *s3* recebe a solicitação e passa a traduzir *alunos*, ele encontra uma marca chamada *remote link* que indica que o diretório */usr/alunos* não é de sua responsabilidade e, portanto, está armazenado em outro servidor. Neste caso, *s3* responde a *cl1* informando que o arquivo solicitado pertence a um domínio de outro servidor.

Se um servidor chega até um *remote link* então a tabela de prefixos do cliente está incompleta. Há um domínio sem o prefixo correspondente na tabela. Neste caso, o cliente envia um *broadcast*, isto é, uma mensagem para todos os servidores da rede, perguntando quem é o responsável pelo arquivo procurado (no nosso exemplo, o arquivo */usr/alunos/lista*).

O servidor *s1*, responsável pelo domínio */usr/alunos*, que contém o arquivo *lista*, irá responder a *cl1* enviando-lhe os dados necessários para preencher uma linha da tabela de prefixos correspondente ao domínio de *lista*.

Quando um cliente é inicializado, sua tabela de prefixos começa vazia e é feito um *broadcast* para descobrir quem é o responsável pelo domínio raiz. A partir daí, a tabela vai crescendo à medida em que os processos vão solicitando acesso a novos domínios. Domínios não acessados não aparecem na tabela de prefixos.

Se um cliente tenta acessar um domínio presente na sua tabela de prefixos e não obtém resposta, ele invalida a entrada do domínio na tabela e executa um novo *broadcast* para o domínio. Este comportamento facilita a migração de domínios. Se o disco de um servidor fica cheio, o administrador da rede apenas transfere um ou mais de seus domínios para outro servidor. Os clientes se adaptam automaticamente à nova situação sem a necessidade de interferência do administrador.

Ao contrário do esquema do NFS e do AFS de traduzir os componentes do *pathname* um a um, a tabela de prefixos permite o acesso direto ao servidor responsável por um domínio[26]. Além disso, as tabelas de prefixos garantem que, se o servidor responsável por um domínio estiver disponível, então os seus arquivos poderão ser acessados mesmo que os servidores responsáveis pelos domínios do início do *pathname* estejam fora do ar. Explicando melhor através do nosso exemplo: se a tabela de prefixos de *cl1* for igual à tabela 2.3, então, para *cl1* acessar os diretórios */usr/alunos* e */usr/profs* basta que *s1* esteja disponível, *s3* não é necessário.

Máquinas com discos locais podem optar por construir um espaço de nomes ligeiramente diferente do espaço de nomes global da rede. Isto é particularmente útil para o diretório */tmp* de arquivos temporários. Seria desperdício manter este diretório em um servidor remoto pois muitos blocos trafegariam na rede sem necessidade. Neste caso, as máquinas que possuem discos locais colocam uma entrada nas suas tabelas de prefixos indicando que este diretório deve ser tratado localmente.

Um dos servidores da rede é incumbido de responder aos *broadcasts* enviados pelas máquinas sem discos locais solicitando acesso ao */tmp*.

Assim como no NFS e no ANDREW, é possível replicar domínios só para leitura em um conjunto de servidores. Cada servidor é configurado de modo a só responder às solicitações de uma parte dos clientes.

Se a rede possui máquinas com diferentes tipos de hardware, é possível configurar os servidores para fornecerem os arquivos binários executáveis de acordo com o hardware dos clientes. Suponha, por exemplo, que uma rede seja formada por SPARCstations e DECstations. Pode-se determinar que o domínio contendo o diretório */usr/bin* vai ser implementado em um servidor do tipo SPARC que atenderá às solicitações dos clientes SPARC e em um servidor DEC que atenderá somente aos *broadcasts* dos clientes DEC.

Concluindo, as tabelas de prefixos possibilitam um rápido acesso a arquivos mas é preciso ter cuidado pois uma rede mal configurada pode apresentar sérios problemas de escalabilidade

[26] O NFS consegue um efeito semelhante através do cache das informações sobre os diretórios.

devido aos *broadcasts* enviados pelos clientes. Se muitos clientes são inicializados simultaneamente (após uma falha no fornecimento de energia, por exemplo) o excesso de *broadcasts* pode causar um congestionamento na rede e nos servidores.

### 2.6.2 Segurança

Como vimos na seção anterior, quando um cliente deseja acessar um certo arquivo, o SPRITE não realiza necessariamente uma análise de cada componente do *pathname* deste arquivo. Através da consulta à tabela de prefixos, é possível ir diretamente ao servidor responsável.

Ao mesmo tempo em que este processo (sem passar por servidores intermediários) acelera o acesso aos arquivos, ele elimina a possibilidade da verificação das permissões de acesso a cada componente do *pathname*. Em outras palavras, o SPRITE não oferece a mesma segurança do que o UNIX.

No SPRITE, todos os clientes e servidores são considerados confiáveis, não há nenhum tipo de autenticação entre as máquinas nem a possibilidade de criptografar informações. Esta característica deve ser lembrada no momento de expandir uma rede SPRITE para nós que estejam fora do controle dos responsáveis pela rede. É possível obter o código-fonte do núcleo do SPRITE sem maiores dificuldades. Um programador mal intencionado poderia alterar o código de um cliente de modo a se fazer passar por um usuário diferente do seu.

### 2.6.3 Cache

O cache do SPRITE apresenta importantes melhoramentos em relação aos seus antecessores. Há a garantia da consistência dos dados cacheados e o cache, tanto dos servidores quanto dos clientes, possui tamanho variável podendo chegar a ocupar praticamente toda a memória de uma máquina.

Os projetistas do SPRITE nem pensaram em armazenar o cache em discos locais como faz o ANDREW pois, na rede local que serviu de base para o desenvolvimento inicial do SPRITE, o acesso à memória do servidor era mais rápido do que o acesso ao disco local[27]. Como um dos objetivos principais era a rapidez no acesso aos dados, optou-se por cachear os arquivos na memória. Deste modo, permite-se também a existência de clientes sem discos locais.

Os arquivos são cacheados em blocos de tamanho fixo: 4Kbytes. Cada bloco é unicamente determinado por um identificador do arquivo, que é fornecido pelo servidor, e pelo seu número dentro do arquivo. O primeiro bloco agrega os bytes de 0 a 4095, o segundo, de 4096 a 8191 e assim por diante.

Os clientes cacheiam apenas os blocos de dados dos arquivos remotos enquanto que os servidores cacheiam os blocos de dados, mapas dos arquivos no discos físicos e outras informações que auxiliam a manutenção do serviço.

Baseados em informações sugeridas por [O$^+$85], os projetistas do SPRITE optaram por cachear, também, as escritas nos arquivos. Quando um processo executa um *write*, os dados são escritos apenas nos blocos do arquivo que estão no cache[28].

A cada 30 segundos, todos os blocos sujos do cache que não foram modificados nos últimos 30 segundos, são enviados para o servidor. Um bloco escrito por um cliente é enviado ao servidor em, no máximo, 60 segundos e, do servidor para o disco, também em, no máximo, 60 segundos. Portanto, se um usuário executa o comando *save* no seu editor de textos, o arquivo salvado poderá demorar até 2 minutos para atingir um local seguro (o disco físico). Se o servidor ou o cliente sofrem uma queda neste período, os dados são perdidos. Em casos

[27] Numa rede de maiores proporções isto não é necessariamente satisfeito.

[28] Se um processo escreve dados em apenas uma parte de um bloco que não está cacheado, este bloco é primeiro trazido do servidor e depois alterado.

onde a garantia da integridade dos dados é importante o SPRITE oferece a possibilidade de forçar a gravação dos dados no disco através do comando *fsync* do UNIX.

Esta insegurança é o preço que o SPRITE paga pelo ganho em eficiência decorrente das escritas retardadas (*delayed-writes*). Segundo [$O^+85$], de 20 a 30% dos dados escritos em um sistema de arquivos são apagados em menos de 30 segundos. A carga nos servidores e na rede é significativamente menor se a política das escritas retardadas é adotada pois estes dados de vida curta geram carga apenas nos clientes.

Os processos recebem as respostas aos seus *writes* muito mais rapidamente pois não precisam esperar pelas respostas dos servidores. Por outro lado, se alguma das suas escritas falha por problemas físicos do disco, muitas vezes o processo não pode ser avisado[29] e a coerência do sistema de arquivos pode ficar comprometida.

### Enfim, um Cache Consistente

Um dos principais objetivos do projeto do SPRITE [NWO88], ao lado do alto desempenho, foi oferecer a semântica UNIX de acesso concorrente aos arquivos. Mesmo quando existem cópias de um arquivo no disco e no cache do servidor e nos caches de vários clientes, o SPRITE garante a consistência entre todas estas cópias a não ser que aconteça alguma falha de escrita o que raramente ocorre.

Ao contrário do NFS e do ANDREW, o SPRITE garante que cada byte recebido por um *read* é o resultado da última escrita naquela posição do arquivo mesmo se ele estiver sendo compartilhado por várias máquinas.

Os usuários podem assumir que os dados são consistentes tanto quando o compartilhamento dos arquivos é seqüencial quanto quando ele é concorrente. Deste modo, os projetistas do SPRITE esperavam oferecer um sistema de arquivos que possibilitasse o intercâmbio de informações entre os nós de uma rede local de uma maneira simples e transparente. O sistema de arquivos funciona como se todos os processos da rede estivessem rodando numa mesma máquina.

A consistência de cada arquivo é garantida pelo servidor responsável pelo mesmo. Não há nenhum tipo de comunicação entre os clientes. Os clientes devem avisar o servidor sempre que abrem ou fecham algum arquivo diferentemente do NFS e do AFS que abrem arquivos sem entrar em contato com os servidores através do cache de nomes dos clientes.

Quando um arquivo é apenas lido, não existe nenhum perigo de diferentes clientes o compartilharem. Os problemas aparecem quando pelo menos um dos clientes altera o conteúdo do arquivo. Podemos dividir esta situação em dois casos:

- **Compartilhamento Seqüencial com Escrita**: ocorre quando um arquivo é modificado por um cliente, fechado e, depois, aberto por outro cliente.
- **Compartilhamento Concorrente com Escrita**: ocorre para um arquivo que está aberto em mais de um cliente e que está aberto para escrita em, pelo menos, um cliente.

O SPRITE adota uma política muito simples que evita inconsistências nos dois casos.

Quando, durante um *open*, o servidor detecta que um período de compartilhamento com escrita se iniciará, ele toma duas providências. Primeiramente, se algum cliente possui o arquivo aberto para escrita, ele é chamado a devolver todos os blocos sujos deste arquivo que possuir em seu cache. Apenas um cliente pode estar nesta situação pois, caso contrário, o arquivo já estaria sendo compartilhado concorrentemente com escrita. Em seguida, o servidor

[29] Quando uma escrita em um arquivo no disco falha, pode ser que o processo que solicitou a escrita já tenha finalizado a sua execução como se as suas escritas tivessem sido efetivamente realizadas.

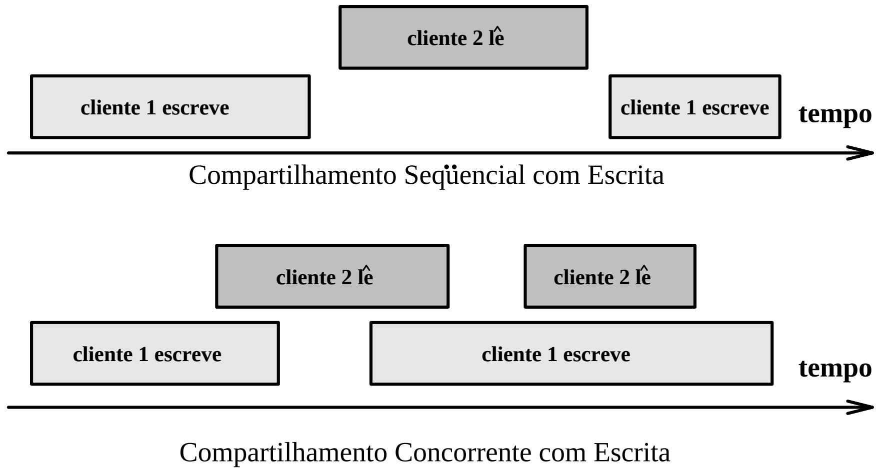



Figura 2.11: Compartilhamento de um arquivo (concorrente$\times$seqüencial)

envia uma mensagem a todos os clientes que possuem o arquivo aberto avisando que ele não é mais cacheável. Ao receber estas mensagens, os clientes invalidam a cópia do arquivo de seu cache e passam a enviar diretamente para o servidor qualquer tentativa de acesso ao arquivo.

Um arquivo pode ser cacheado simultaneamente por vários clientes desde que todos eles estejam apenas lendo do arquivo. Se apenas um cliente possui o arquivo aberto, ele pode usar os blocos do cache para ler e escrever dados neste arquivo.

Quando um arquivo se torna não cacheável, apenas os clientes que o possuem aberto são notificados. Quando o compartilhamento concorrente com escrita deixa de existir, os clientes que já possuem o arquivo aberto não são notificados. O servidor informa o novo estado do arquivo apenas para os clientes que abrem o arquivo após o instante no qual a escrita concorrente terminou.

Já no caso de compartilhamento seqüencial com escrita, dois problemas podem ocorrer. Quando um cliente abre um arquivo, podem existir, no seu cache, blocos desatualizados deste arquivo. Este problema é resolvido através do número da versão de um arquivo. Cada vez que o servidor recebe uma solicitação de abertura de um arquivo para escrita, ele incrementa o número da versão do arquivo. Os clientes guardam o número da versão dos arquivos em seu cache. Quando um arquivo é aberto, o cliente compara o número da versão da cópia local com a cópia do servidor. Se ele é diferente, a cópia local é invalidada da mesma forma como é feito no NFS.

O outro problema ocorre quando a cópia mais atualizada de um arquivo que está sendo aberto não está nem no cliente que quer abrir o arquivo nem no servidor mas sim, no cache de um outro cliente que já fechou este arquivo mas que ainda não enviou os blocos sujos para o servidor. Para resolver este problema o servidor mantém uma tabela que informa, para cada arquivo recentemente aberto, quais são os clientes que podem possuir blocos alterados em seu cache. Quando um cliente abre um arquivo e existe a possibilidade de outro cliente possuir blocos sujos deste arquivo no cache, o servidor envia uma mensagem para este outro cliente ordenando que os blocos alterados sejam enviados para o servidor.

Tudo isso garante que o cliente irá sempre receber os dados mais recentes a menos que ocorram quedas de máquinas ou problemas físicos nos discos. Quando uma máquina cai, os blocos sujos de seu cache (se existirem) são perdidos. A escrita no disco físico também pode falhar devido a algum defeito do equipamento ou, simplesmente, pela falta de espaço disponível no disco.

#### Um Cache Enorme

Outra particularidade do cache do SPRITE [NWO88] é o seu tamanho. Ao invés de ser fixo, o tamanho do cache varia dinamicamente podendo ocupar toda a memória disponível em uma máquina. O cache compete por espaço com o sistema de memória virtual. Deste modo, espera-se que o maior número possível de solicitações dos processos clientes possam ser atendidas sem a necessidade de acesso ao servidor.

Sempre que um novo bloco de um arquivo é lido do servidor, ele é armazenado no cache. Se a memória está completamente utilizada então é necessário liberar alguma página antiga para abrir espaço para o novo bloco.

Tanto a memória virtual quanto o cache utilizam o método LRU (*Least Recently Used*) que consiste em descartar a página de memória usada menos recentemente. Se for uma página da memória virtual, então ela é enviada para o disco responsável pela memória virtual. Se, por outro lado, for uma página que contém um bloco do cache, o sistema verifica se a página foi alterada. Se foi alterada, ela é enviada para o servidor responsável, caso contrário, o bloco é descartado e a página de memória, liberada para uso.

Um processo análogo ocorre se o sistema de memória virtual necessita de uma nova página de memória física. Estudos empíricos indicaram que seria vantajoso se o sistema de memória virtual tivesse prioridade sobre o cache do sistema de arquivos. A fim de implementar isso, o sistema só libera uma página da memória virtual para uso pelo cache se ela não foi acessada há pelo menos 20 minutos [Nel88].

É necessário tomar certos cuidados para que não haja duplicação de páginas na memória uma vez que o sistema de memória virtual é um usuário do sistema de arquivos. Quando páginas da memória virtual são lidas ou enviadas ao servidor, elas não passam pelo cache do cliente[30].

Os projetistas do SPRITE também julgaram importante evitar que páginas de arquivos executáveis estejam presentes simultaneamente no cache e na memória virtual. A solução encontrada foi atribuir um valor "infinitamente" antigo ao último acesso a blocos do cache que correspondam a arquivos executáveis presentes na memória virtual. Deste modo, estas páginas são eliminadas assim que haja a necessidade de mais memória.

O artigo de apresentação do cache do SPRITE [NWO88] conclui recomendando aos usuários que estejam pensando em comprar discos locais que abandonem esta idéia e que, ao invés disso, gastem o mesmo dinheiro comprando memória adicional. Deste modo o desempenho do sistema de arquivos seria maior do que com discos locais e o sistema de memória virtual também seria beneficiado.

### 2.6.4 Disponibilidade

O SPRITE não oferece nenhum tipo de replicação automática de domínios em vários servidores. Logo, quando um servidor está fora do ar, não há como obter cópias dos seus arquivos nem alterá-los. Como diz Mary Baker em [BO91], o SPRITE foi feito para usuários que querem o máximo da velocidade com o menor investimento possível. Tais usuários não se disporiam a gastar dinheiro para a compra de servidores *back-up*.

O SPRITE investe todo o poder computacional da rede para oferecer um serviço rápido e consistente. A única maneira de obter uma boa disponibilidade sem contrariar os objetivos do

[30] Apesar de não serem cacheadas nos clientes, as páginas da memória virtual são cacheadas nos servidores. Assim, a memória dos servidores funciona como uma extensão da memória principal dos clientes.

SPRITE é fazer com que a recuperação das quedas dos servidores aconteça o mais rapidamente possível.

Se um servidor conseguir se recuperar de uma queda, reconstituindo o seu estado em poucos segundos, os clientes perceberiam, no máximo, uma pequena interrupção do serviço o que não causaria nenhum grande prejuízo. Além de não ser necessário possuir hardware adicional, esta abordagem do problema da disponibilidade, simplifica em muito o código do *kernel* uma vez que não é necessário administrar diversas réplicas de um mesmo arquivo.

A maior parte do tempo gasto pelos servidores do SPRITE é aplicado na recuperação do estado e na verificação do sistema de arquivos físico. [BO91] apresenta idéias que visam diminuir o tempo gasto em cada uma destas atividades.

Quando um servidor cai, toda a informação sobre os arquivos abertos nos clientes é perdida. A solução para este problema é manter nos clientes uma cópia destas informações. Cada cliente deve saber quais arquivos ele mantém abertos e em que modo. Quando um cliente detecta que um servidor está em processo de recuperação após uma queda, ele lhe envia as informações sobre os arquivos abertos. Quando o servidor recebe estas informações de todos os clientes que possuem arquivos abertos, o estado do sistema está refeito. Este método é chamado por Mary Baker de "recuperação distribuída de estado".

O maior problema deste método é a escalabilidade. Quando o número de clientes é muito grande, o servidor não consegue atender aos RPCs de todos os clientes que ficam repetindo as chamadas até obterem resposta. Com este algoritmo, o tempo necessário para a recuperação de um servidor com 40 clientes chegava a intoleráveis 15 minutos.

Adicionando-se uma "resposta negativa" aos RPCs dos clientes conseguiu-se diminuir este tempo para cerca 100 segundos. Quando o servidor recebe uma RPC mas não pode tratá-la por excesso de carga, ele responde informando que não pode atender no momento mas que o cliente deve repetir a chamada após um certo intervalo de tempo.

Uma alternativa a este método seria guardar as informações sobre o estado em memória não volátil no servidor[BAD+92]. Assim, o servidor não precisaria gastar tempo com a recuperação do seu estado pois ele estaria a salvo na memória não volátil. O principal inconveniente desta solução é a necessidade de um hardware específico.

Mas, além de diminuir o tempo de recuperação distribuída do estado, é preciso reduzir também o tempo gasto na verificação do sistema de arquivos físico. Quando um servidor é inicializado, é necessário verificar os discos do sistema de arquivos a fim de identificar eventuais inconsistências. Pode acontecer de um bloco não estar marcado como livre e nem pertencer a nenhum arquivo (blocos não referenciados). Também pode ocorrer uma multi-referenciação, isto é, o mesmo bloco ter atribuições distintas e conflitantes.

Procurar e solucionar estas inconsistências é uma tarefa muito demorada. Uma Sun-4 rodando SPRITE demora cerca de 15 minutos para verificar 6 discos totalizando 2 Gigabytes. É um tempo que os clientes não podem esperar. A solução apontada por Baker é a utilização de um sistema de arquivos baseado em log (ver seção 2.7.1).

Em sua tese de doutorado [Bak94], Mary Baker apresenta uma série de técnicas que permitiram diminuir em muito o tempo necessário para a recuperação do servidor. Através da aplicação destas técnicas, uma SPARCstation 2 com 40 clientes consegue recuperar o seu estado em menos de 6 segundos. O tempo total para a reinicialização deste servidor é de menos de 30 segundos incluindo aí a verificação do sistema de arquivos.

Estas técnicas trazem um bom ganho para a disponibilidade no SPRITE mas pouco podem fazer contra interrupções no serviço de arquivos causadas por problemas de hardware. Se um servidor quebra, os seus arquivos não poderão ser acessados até que ele seja consertado ou até que os seus discos sejam transferidos para outro servidor. Se um disco é danificado, a solução é rezar para que exista um *back-up* recente dos seus arquivos.

### 2.6.5 Analisando o Desempenho

[O+85] apresenta um estudo sobre o padrão de acesso ao sistema de arquivos por um grupo de usuários de um centro de computação acadêmico. Os dados coletados serviram como referência para o projeto do SPRITE.

Seis anos mais tarde, com o ambiente de hardware e software totalmente alterado, este estudo foi repetido [BHK+91b] e algumas conclusões puderam ser obtidas. Os dois estudos foram coordenados por John Ousterhout.

O estudo de 1985 foi feito em cima de um VAX-11/780s que era uma máquina (*time-shared*) compartilhada por vários usuários. Todos os acessos ao sistema de arquivos local do UNIX 4.2 BSD em um determinado intervalo foram registrados e, posteriormente, analisados.

Já em 1991, coletaram-se todos os acessos a arquivos efetuados em uma rede com 52 usuários e 40 estações de trabalho fabricadas pela SUN e pela DEC durante quatro períodos de 48 horas cada um. As estações não possuíam discos locais e sua memória principal variava de 24 a 32 Mbytes. O serviço de arquivos era oferecido por quatro servidores sendo que quase todo o trabalho era efetuado por um servidor SUN-4 com 128 Mbytes de memória principal.

A tabela 2.4, extraída de [BHK+91b], traz, lado a lado, resultados dos dois estudos. Os dados coletados foram divididos em intervalos de 10 minutos e de 10 segundos. Intervalos de 10 minutos mostram o comportamento médio do sistema enquanto que os intervalos de 10 segundos mostram o que acontece nos picos de utilização do sistema.

| Atividade dos usuários | | 1985 | 1991 | 1991 (processos migrados) |
|---|---|---|---|---|
| Intervalos de 10 minutos | Número máximo de usuários ativos | 31 | 27 | 5 |
| | Número médio de usuários ativos | 12,6 | 9,1 | 0,91 |
| | Tráfego médio por usuário ativo (Kbytes/segundo) | 0,4 | 8,0 | 50,7 |
| | Tráfego máximo de um usuário em um intervalo (Kb/s) | não medido | 458 | 458 |
| | Tráfego total máximo em um intervalo (Kb/s) | não medido | 681 | 616 |
| Intervalos de 10 segundos | Número máximo de usuários ativos | não m. | 12 | 4 |
| | Número médio de usuários ativos | 2,5 | 1,6 | 0,14 |
| | Tráfego médio por usuário ativo (Kbytes/segundo) | 1,5 | 47,0 | 316 |
| | Tráfego máximo de um usuário em um intervalo (Kb/s) | não medido | 9871 | 9871 |
| | Tráfego total máximo em um intervalo (Kb/s) | não medido | 9977 | 9871 |

Um usuário é considerado ativo se ele acessou o sistema de arquivos dentro do intervalo.

Tabela 2.4: Atividade dos usuários nos sistemas analisados

Podemos ver que há uma diferença significativa na ordem de grandeza da quantidade de bytes acessados por um usuário. No caso dos intervalos de 10 minutos o tráfego médio por usuário cresceu 20 vezes, no caso dos intervalos de 10 segundos o tráfego médio cresceu mais de 20 vezes.

Por outro lado, este crescimento foi muito menor do que o crescimento da capacidade da memória e do processamento das CPUs do sistema (a capacidade de processamento das CPUs cresceu mais de 200 vezes). Seria desejável que se pudesse utilizar este grande poder de computação para diminuir o tempo de espera dos clientes e evitar que o crescimento no tráfego sobrecarregue a rede e os servidores. O SPRITE faz isto através dos seus grandes caches.

Entre os dados coletados em 1991 há o caso interessante de um usuário que, em um intervalo de 10 segundos, acessou quase 100 Mbytes do sistema de arquivos. Tal quantidade seria impensável sem o cache nos clientes uma vez que este tráfego é cerca de 10 vezes superior à capacidade da rede *Ethernet* onde os dados foram coletados.

A tabela 2.4 também mostra que os processos migrados, isto é, que são executados em uma máquina à qual o seu usuário não está conectado, são grandes consumidores do sistema de arquivos. Este fato ocorre principalmente devido à natureza dos processos migrados (em geral, compilações) que utilizam muito o sistema de arquivos.

### Tamanho do Cache

[BHK+91b] analisa também o tamanho ocupado pelos caches do SPRITE. Nos servidores, o cache rapidamente passa a ocupar toda a memória principal disponível. Nos clientes, o cache passa a maior parte do tempo ocupando entre um quarto e um terço da memória total o que equivale a cerca de 7 Mbytes. Este é um valor bem maior do que os 10% normalmente reservados por sistemas cujo tamanho do cache é fixo.

A tabela 2.5 mostra os valores máximo, mínimo e médio do tamanho do cache nos clientes. O desvio padrão é relativo aos valores medidos de 15 em 15 minutos e mostra que o tamanho do cache varia bastante.

| | |
|---|---|
| Mínimo | 195 Kbytes |
| Máximo | 23820 Kbytes |
| Médio | 7110 Kbytes |
| Desvio padrão | 5556 Kbytes |

Tabela 2.5: Tamanho do cache nos clientes do SPRITE

### Eficiência do Cache

O sucesso do cache nos clientes pode ser confirmado observando-se as tabelas seguintes. A tabela 2.6 mostra que 79,8% do tráfego no sistema de arquivos é composto por dados que podem ser cacheados. A maior parte dos dados não cacheáveis dizem respeito à paginação do sistema de memória virtual. Uma parcela menos significativa é composta por arquivos compartilhados com escrita e pelos diretórios.

Mas, mesmo no que diz respeito à paginação, mais da metade dos bytes lidos podem vir do cache. O SPRITE mantém páginas de código na memória mesmo após o término da execução dos processos. Assim, se o mesmo programa é executado novamente, o seu código pode já estar na memória.

Além disso, páginas com dados pré-inicializados de arquivos executáveis, são guardados no cache do sistema de arquivos. Quando um processo acessa uma página de dados pré-inicializada pela primeira vez, ela é copiada para o sistema de memória virtual. Os processos costumam sujar tais páginas. Mas, quando outro processo acessa a mesma página, existe uma cópia limpa no cache do sistema de arquivos.

| Tipo de tráfego | | Bytes lidos (%) | Bytes escritos (%) | Total |
|---|---|---|---|---|
| tráfego cacheável | arquivos | 52,4 | 10,5 | 62,9 |
| | paginação | 16,9 | 0,0 | 16,9 |
| tráfego não cacheável pelo cliente | paginação | 11,3 | 6,7 | 18,0 |
| | arquivos compartilhados | 0,3 | 0,2 | 0,5 |
| | diretórios | 0,5 | 0,0 | 0,5 |
| | outros | 0,3 | 0,9 | 1,2 |
| Total | | 81,7 | 18,3 | 100,0 |

Tabela 2.6: Fontes de tráfego no cliente

Note que a participação dos arquivos compartilhados com escrita (a quarta linha da tabela) é muito pequena e, por conseguinte, o fato de o SPRITE desabilitar o cache para tais arquivos praticamente não altera o desempenho global do sistema de arquivos no ambiente estudado.

As melhorias introduzidas por algoritmos mais eficientes para o tratamento do cache de arquivos compartilhados (ver capítulo 4) só serão percebidas por aplicações específicas que façam uso de arquivos compartilhados com escrita.

A tabela 2.7, que mostra a eficiência do cache em relação ao tráfego cacheável, mostra dados desanimadores frente às previsões de [O$^+$85]. Julgava-se que um cache de 4 Mbytes pudesse oferecer uma taxa de acerto da ordem de 90%[31]. O estudo de 1991 mostrou que, mesmo com um cache maior, ao invés de apenas 10% das leituras não serem resolvidas pelo cache, algo em torno de 40% das leituras são remetidas ao servidor.

| Taxa de acerto em relação ao tráfego cacheável | Total | Processos migrados |
|---|---|---|
| Operações de leitura | 58,6 | 77,8 |
| Tráfego de leitura | 62,9 | 68,3 |
| Tráfego de escrita | 11,6 | não disponível |
| Paginação | 71,3 | 91,2 |

Tabela 2.7: Eficiência do cache dos clientes (%)

41,4% das operações de leitura de dados cacheáveis não podem ser resolvidas pelo cache e são enviadas ao servidor. 37,1% dos bytes lidos vem diretamente do servidor. Provavelmente, o cache não foi tão efetivo quanto se esperava por causa do crescimento do tamanho médio dos arquivos e do *working set*[32] dos usuários nestes seis anos.

Durante a coleta dos dados no SPRITE, um dos usuários da rede estava usando um simulador que lia arquivos que possuíam 20 Mbytes em média. Como o SPRITE não trata diferentemente arquivos pequenos e grandes (ele cacheia ambos), arquivos de 2Mbytes funcionavam como "limpa-caches" apagando blocos do cache que seriam úteis posteriormente.

11,6% dos bytes escritos não são enviados ao servidor em nenhum momento. São bytes que são sobrepostos por outras escritas ou que pertencem a arquivos que são apagados antes de serem enviados ao servidor.

[31] Isto é, que 90% dos *reads* fossem atendidos pelo cache

[32] O *working set* de um usuário é o conjunto de arquivos que ele acessa em uma certa sessão de trabalho.

A grande diferença nas taxas de acerto do cache no caso das leituras e das escritas faz com que, apesar dos processos dos clientes solicitarem quatro vezes mais bytes em leituras do que em escritas (tabela 2.8), esta proporção caia para apenas dois para um nas solicitações recebidas pelo servidor como mostra a tabela 2.9.

| Tipo do *open* | % de acessos | % de bytes |
|---|---|---|
| Apenas leitura | 88 | 80 |
| Apenas escrita | 11 | 19 |
| Leitura e escrita | 1 | 1 |

Tabela 2.8: Acesso dos processos dos clientes ao sistema de arquivos

| Bytes solicitados | Atendidos pelo cache | Não cacheados | Total |
|---|---|---|---|
| Leitura | 41,8 | 24,5 | 66,3 |
| Escrita | 18,3 | 15,4 | 33,7 |

Tabela 2.9: Carga no servidor (%)

**Consistência**

A partir dos dados coletados em 91, também foi possível analisar a importância e o custo do mecanismo do SPRITE para garantia da consistência do cache.

Para medir a importância da consistência, foram realizados cálculos para determinar a freqüência com que dados incorretos seriam acessados se fosse utilizado um mecanismo semelhante ao do NFS para restringir as inconsistências.

Os dados da tabela 2.10 mostram as inconsistências que ocorreriam com um protocolo semelhante ao NFS. Neste protocolo hipotético, os clientes consideram os dados em seus caches válidos por 3 segundos. Se um processo solicita acesso a estes dados após o intervalo de 3 segundos, o cliente se comunica com o servidor para verificar se a cópia em seu cache é atual. Os dados escritos são enviados diretamente para o servidor.

| Medida | Média | Mínimo | Máximo |
|---|---|---|---|
| Erros por hora | 0,59 | 0,12 | 1,8 |
| % de usuários afetados em um dia | 7,1 | 4,5 | 12,0 |
| % de usuários afetados durante todo os períodos | 20,0 | | |
| % de arquivos abertos com erro | 0,011 | $10^{-4}$ | 0,032 |
| % de arquivos abertos com erro por processos migrados | <0,01 | 0,0 | 0,055 |

Tabela 2.10: Acesso a dados desatualizados

A implementação da SUN do NFS é um pouco mais fraca na medida em que os dados no cache dos clientes são considerados válidos por um intervalo que pode variar de 3 a 60 segundos. Além disso, o NFS usa *delayed-writes*, blocos sujos podem permanecer no cache mesmo após o *close*.

Processos migrados podem abrir um arquivo em um certo cliente e, posteriormente, reacessarem o arquivo a partir de outro cliente. Mas, ao contrário do que se poderia imaginar, a tabela 2.10 mostra que os processos migrados acessam uma quantidade menor de dados inconsistentes do que os demais processos. Provavelmente, isto ocorre pois uma grande parte das migrações no SPRITE acontecem antes do processo iniciar a sua execução; tais processos abrem os seus arquivos depois de terem migrado.

O fato de 20% dos usuários acessarem dados desatualizados em algum momento dentro dos 8 períodos de 24 horas parece assustador. Infelizmente, os autores do artigo não dizem qual era a função dos arquivos nos quais estas inconsistências ocorreram. Tal informação seria de extrema importância para uma análise precisa da necessidade da manutenção da consistência. Os autores preferiram apenas argumentar que as inconsistências observadas mostram a necessidade da adoção de mecanismos que garantam a consistência do cache. Mecanismos como, por exemplo, os do SPRITE.

Podemos ter uma idéia do custo da manutenção da consistência no SPRITE examinando uma simulação mais antiga. Utilizando os dados coletados em 1985, [NWO88] apresenta uma simulação de como se daria o uso do cache se cada usuário que usou o sistema centralizado em 85 executasse os mesmos comandos em diferentes clientes do SPRITE. A tabela 2.11, extraída de [NWO88] mostra qual seria a eficiência do cache de acordo com o seu tamanho (supondo que o cache fosse de tamanho fixo).

| Tamanho do cache | Blocos lidos | Blocos escritos | Total | Razão (%) |
|---|---:|---:|---:|---:|
| sem cache | 445.815 | 172.546 | 618.361 | 100 |
| 0,5 Mbyte | 102.469 | 96.866 | 199.335 | 32 |
| 1 Mbyte | 84.017 | 96.796 | 180.813 | 29 |

Tabela 2.11: Tráfego no servidor

Já a tabela 2.12 mostra o que aconteceria se não houvesse a necessidade de manter a consistência entre as várias cópias dos arquivos. Comparando as duas tabelas podemos ter uma idéia de quanto tráfego adicional de blocos de arquivos é necessário para atingir a semântica UNIX. Com cache de 1Mbyte, a simulação aponta um tráfego de 180.813 blocos quando a semântica UNIX é mantida contra apenas 145.635 blocos no caso sem a manutenção de consistência.

| Tamanho do cache | Blocos lidos | Blocos escritos | Total | Razão (%) |
|---|---:|---:|---:|---:|
| sem cache | 445.815 | 172.546 | 618.361 | 100 |
| 0,5 Mbyte | 80.754 | 93.663 | 174.417 | 28 |
| 1 Mbyte | 52.377 | 93.258 | 145.635 | 24 |

Tabela 2.12: Tráfego no servidor sem manutenção de consistência

Mas, é bom lembrar que a tabela 2.12 mostra dados irreais uma vez que um sistema sem nenhum tipo de consistência teria pouco uso prático. Para avaliarmos o custo de oferecer a consistência perfeita em relação ao custo de uma semântica mais fraca, como a do NFS por exemplo, é preciso saber com que freqüência ocorre compartilhamento de arquivos com escrita pois é neste caso que o SPRITE gera mais tráfego do que o NFS.

[BHK$^+$91b] mostra que apenas 0,34% das aberturas de arquivo por parte dos clientes através do comando *open* geraram compartilhamento concorrente com escrita. Além disso, em 1,7% dos comandos *open*, o arquivo aberto continha páginas sujas em outros clientes que tiveram que ser trazidas prematuramente para o servidor.

Estes dados confirmam a hipótese de que a manutenção perfeita da semântica UNIX de acesso concorrente através da desabilitação do cache não afeta o desempenho global do sistema de arquivos. Apenas os processos que compartilham arquivos com processos de outras máquinas sofrem com o custo da manutenção da semântica UNIX.

Obviamente, esta análise só é válida para sistemas que recebam uma carga semelhante à rede onde os dados foram coletados. Aplicações específicas podem gerar um compartilhamento muito maior e, nestes casos, os mecanismos do SPRITE podem se mostrar inadequados.

No capítulo 4, veremos uma alternativa ao protocolo do SPRITE para a emulação da semântica UNIX.

### 2.6.6 Comparações com o NFS e o ANDREW

Poucas comparações entre sistemas de arquivos distribuídos desenvolvidos por diferentes grupos são publicadas. Quase sempre a comparação é feita com o NFS que é o sistema mais difundido para ambientes UNIX. [HKM+88] compara o AFS-2 com o NFS. Utilizando estes dados, realizando outros testes, e normalizando os resultados, [NWO88] chegou aos dados apresentados na figura 2.12.

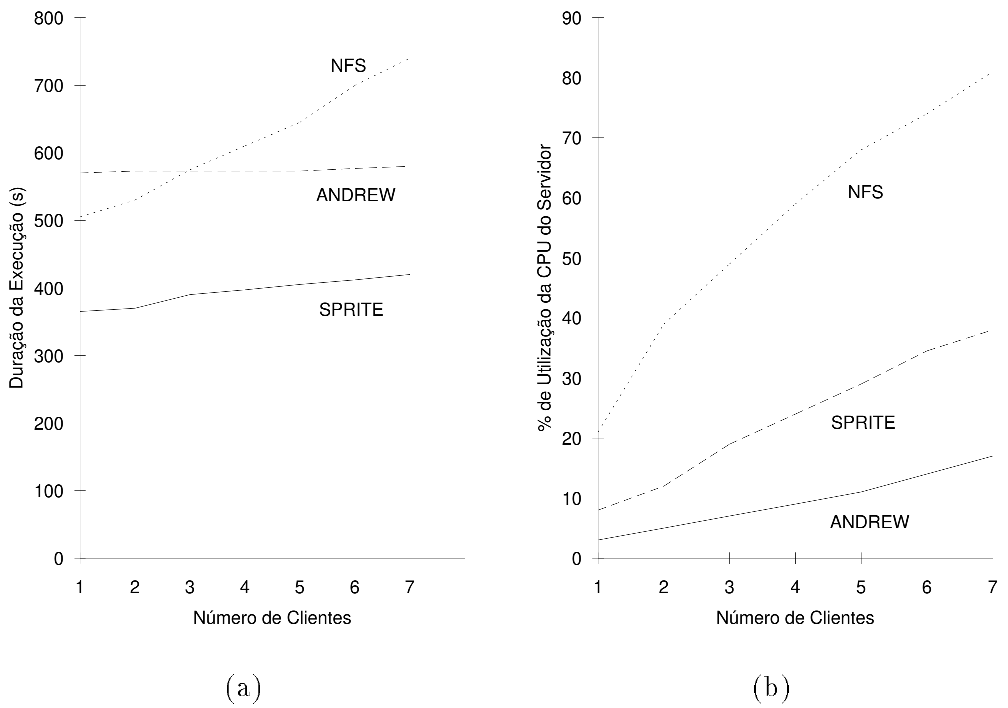



Figura 2.12: Desempenho comparativo do NFS, ANDREW e SPRITE

A figura 2.12a mostra o tempo gasto na execução do *Andrew Benchmark*[33] em clientes

[33] Ver seção 2.5.5.

Sun-3/50[34]. Já a figura 2.12b mostra a porção do tempo da CPU do servidor que é utilizada no oferecimento do serviço de arquivos para a execução do *benchmark*.

Como mostra 2.12a, com apenas um cliente, o SPRITE completa o *benchmark* em um tempo 30% menor do que o tempo gasto pelo NFS e 35% menor do que o tempo gasto pelo ANDREW. À medida em que aumenta o número de clientes executando o *benchmark*, o desempenho do SPRITE e, principalmente, do NFS se degrada enquanto que o desempenho do ANDREW é pouco afetado.

A figura 2.12b mostra a grande sensibilidade do NFS ao aumento da carga no sistema de arquivos fato este que se deve, entre outros fatores, à menor eficiência do cache imposta pelo protocolo NFS. Já o ANDREW apresentou a melhor escalabilidade graças ao mecanismo de *callback* que funciona muito bem no caso analisado onde não há compartilhamento.

Desde a coleta destes dados até os dias de hoje, os três sistemas sofreram várias modificações no que diz respeito às suas implementações. No entanto, os seus protocolos não foram muito alterados o que nos leva a crer que um estudo comparativo entre as versões atuais dos três sistemas não apresentaria resultados muito diferentes daqueles apresentados em [NWO88].

### 2.6.7 Resumo

O SPRITE oferece um serviço de alta velocidade para redes locais ao mesmo tempo em que garante a semântica UNIX no acesso concorrente. É um sistema operacional distribuído completo (não apenas um sistema de arquivos) e toda a comunicação entre clientes e servidores é feita através de rápidas RPCs implementadas no nível do *kernel*.

Os caches dos servidores e clientes disputam espaço com o sistema de memória virtual tentando ocupar a maior porção possível da memória física. O sistema de memória virtual, por sua vez, utiliza arquivos comuns do sistema distribuído para efetuar o *swap* fazendo com que a migração de processos de uma máquina para outra seja facilitada.

A resolução dos *pathnames* é efetuada através de tabelas de prefixos dinâmicas mantidas pelos clientes. O espaço de nomes é comum para todas as máquinas do sistema.

A segurança oferecida pelo SPRITE é deficiente na medida em que todos os clientes e servidores da rede são considerados confiáveis e que, nem sempre, as permissões de acesso a todos os diretórios de um *pathname* são verificadas.

Os autores do SPRITE realizaram testes comparativos do desempenho do SPRITE, NFS e AFS chegando à conclusão de que o SPRITE é o mais rápido dos três sendo menos escalável do que o AFS.

[34] Os dados do SPRITE tiveram que ser normalizados pois foram coletados em clientes Sun-3/75. Os servidores eram do mesmo tipo nos dois testes.

## 2.7 ZEBRA - Um Sistema Listrado

O ZEBRA [HO92, HO93], desenvolvido a partir de 1990 em Berkeley, incorpora idéias dos sistemas de arquivos baseados em *log* e das matrizes de discos redundantes (RAIDs). Sistemas de arquivos baseados em *log*, como o SPRITE LFS [RO91], permitem uma utilização mais eficiente da capacidade de transferência de dados para os discos rígidos através do agrupamento de vários *writes* em uma única escrita seqüencial. Já as RAIDs (*Redundant Arrays of Inexpensive Disks*) conseguem fornecer um serviço de disco de alta disponibilidade e velocidade através da associação de discos de baixo custo. Veremos, a seguir, uma breve descrição destes conceitos e, em seguida, a sua aplicação no ZEBRA.

### 2.7.1 Sistemas de Arquivos Baseados em *log*

Nos últimos anos, a velocidade dos microprocessadores aumentou em um ritmo maior do que a velocidade de acesso aos discos rígidos [Ous90]. Este fato tem feito com que o desempenho de um grande número de aplicações ficasse limitado pela velocidade de acesso aos discos. A implementação de grandes caches nos servidores e clientes é capaz de resolver a maior parte dos acessos ao disco diminuindo o desequilíbrio entre as velocidades das CPUs e dos discos. Mas, a eficiência do cache nas operações de leitura é significativamente superior à eficiência no caso das escritas. Assim, apesar de as leituras serem mais freqüentemente solicitadas pelas aplicações dos usuários, as escritas tendem a dominar o acesso aos discos físicos.

Um dos motivos desta inversão é que, por questões de segurança, é desejável que a maior parte das escritas sejam levadas ao disco. Uma leitura pode ser atendida por um bloco que está no cache há vários dias. Por outro lado, não é nem um pouco recomendável que um bloco recém-escrito permaneça no cache por muito tempo sem ser enviado para o disco. Bastaria uma queda no fornecimento de energia para que os dados se perdessem.

Os sistemas de arquivos baseados em *log* (*Log-Structured File Systems*) agrupam uma série de *writes* em uma única escrita seqüencial em uma estrutura chamada *log*. As novas informações são concatenadas ao fim do *log*. Uma vez armazenado, um dado não é mais alterado. Se o servidor recebe uma solicitação para a atualização de um bloco antigo, os novos dados são concatenados ao *log* e o bloco antigo é marcado como desatualizado. Esta política permite que se elimine quase a totalidade dos *seeks*[35], que são operações muito demoradas em comparação com a transferência de dados aos setores de um cilindro do disco. Como as escritas são sempre seqüenciais, só há necessidade de realizar um *seek* quando o cilindro corrente se esgota ou quando uma leitura é solicitada. Mas, mesmo nestes casos, os *seeks* costumam ser mais curtos do que nos sistemas convencionais.

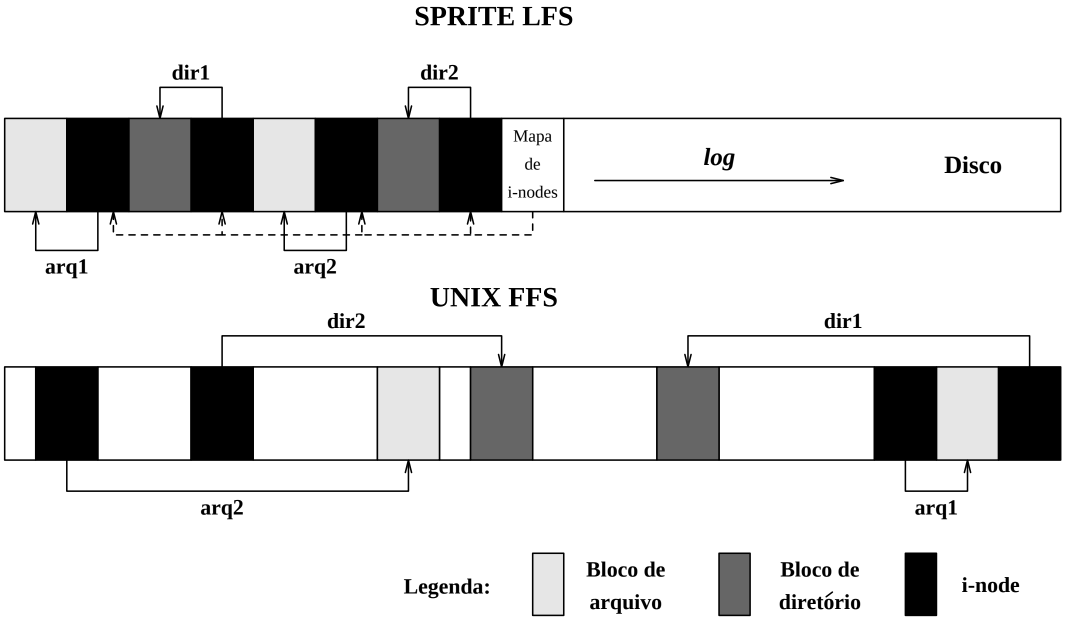
Como o espaço nos discos é finito, é necessário utilizar uma política para liberar espaço extra quando o *log* está prestes a ocupar o disco todo. A solução adotada pelo SPRITE LFS foi dividir o *log* em segmentos de 512Kbytes ou 1Mbyte e executar um processo limpador (*segment cleaner*) que entra em ação sempre que o número de segmentos livres fica menor do que um determinado valor crítico. O limpador de segmentos agrupa diversos segmentos sujos (com blocos desatualizados) em um número menor de segmentos limpos. Desta forma, novos segmentos totalmente vazios são liberados.

A figura 2.13 mostra como o SPRITE LFS e o UNIX FFS [McK84] armazenam os blocos relativos à criação de dois arquivos de um bloco (*dir1/arq1* e *dir2/arq2*). O FFS chega a fazer 10 *seeks* para criar os dois arquivos [36] enquanto que o SPRITE LFS escreve todas as

[35] *Seek* é o movimento que a cabeça de leitura e gravação do acionador de disco faz para se posicionar sobre a trilha do disco onde estão os blocos que serão acessados.

[36] O Unix FFS escreve, não seqüencialmente, 10 blocos: cria dois novos blocos de dados, atualiza os dois blocos que contém o conteúdo dos diretórios e cria um par de *i-nodes* para cada novo arquivo (os *i-nodes* são

informações necessárias em blocos contíguos do disco sem a necessidade de efetuar *seeks*.




No SPRITE LFS, os blocos são armazenados no disco em uma única escrita seqüencial. No UNIX FFS, os blocos são armazenados em diferentes trilhas do disco exigindo que vários *seeks* sejam feitos.

Figura 2.13: A criação de dois arquivos

Experimentos comparando o SPRITE LFS e o SunOS 4.0.3 cujo sistema de arquivos local é baseado no FFS, mostram que o SPRITE LFS apresenta um desempenho superior na maioria dos casos[RO91]. O sistema baseado em *log* só é pior quando um arquivo que foi atualizado de uma ordem aleatória é lido seqüencialmente. Neste caso, é o SPRITE LFS que perde tempo com os *seeks*.

Outra vantagem dos sistemas de arquivos baseados em *log* decorre do fato de que todo servidor precisa fazer uma verificação do seu sistema de arquivos local a fim de detectar possíveis inconsistências causadas por quedas abruptas do sistema. Nos sistemas de arquivos convencionais, esta é uma operação muito demorada que é realizada no momento da reinicialização dos servidores. Quando um servidor se recupera de uma queda, é necessário percorrer todos os blocos do disco em busca de inconsistências. A verificação de um sistema de arquivos de 1100 Mbytes em uma SPARCstation 2, por exemplo, gasta cerca de 6 minutos.

Em sistemas de arquivos baseados em *log*, este tempo pode ser bem menor. Não há necessidade de percorrer todos os blocos de um disco. Basta conferir apenas os segmentos que foram criados após a última verificação do sistema de arquivos uma vez que um bloco que era consistente e não foi alterado dificilmente se tornará inconsistente[37].

Para diminuir ainda mais o tempo gasto com a verificação, pode-se configurar o sistema de modo a realizar uma verificação todas as noites ou então, executá-la permanentemente com uma prioridade muito baixa a fim de não prejudicar o desempenho dos demais processos. Deste modo, quando um servidor se recupera de uma queda, apenas uma pequena parte do seu *log* precisa ser inspecionado.

---

duplicados para facilitar a recuperação de quedas).

[37] Eventualmente, um erro pode fazer com que um bloco antigo deixe de ser referenciado pelas listas do sistema. Estes casos são resolvidos pelo *segment cleaner*

Maiores detalhes sobre o SPRITE LFS podem ser encontrados em [RO90] e [RO91].

### 2.7.2 RAID

A capacidade de armazenamento dos discos rígidos cresceu mais rapidamente do que a sua velocidade de transferência de dados. Como conseqüência, grandes discos não possuem uma velocidade de acesso muito maior do que os discos pequenos e baratos. A partir desta constatação, pesquisadores de Berkeley chegaram à conclusão de que o agrupamento de pequenos discos baratos em uma unidade lógica única seria uma solução para o problema da baixa velocidade dos discos frente aos microprocessadores.

Os RAIDs [PGK88, L+92] possibilitam a divisão da carga no servidor de arquivos entre vários discos relativamente pequenos e baratos. Vários acessos simultâneos ao sistema de arquivos podem ser tratados, em paralelo, pelos diferentes acionadores de disco aumentando consideravelmente a velocidade do fluxo de dados. Através da técnica de listramento (*striping*), que consiste em distribuir os blocos de um arquivo por vários discos, é possível aumentar a velocidade de transferência de arquivos grandes. Ao invés da CPU receber seqüencialmente os blocos de um arquivo tendo que esperar muito tempo até que o último bloco seja transferido, os discos do RAID enviam os blocos em paralelo diminuindo o tempo total da transferência[38]. Enquanto um disco faz o *seek* para alcançar o próximo bloco, outro disco pode transferir os seus dados.

A distribuição dos arquivos por diferentes discos pode comprometer a confiabilidade do sistema. Se qualquer um dos acionadores de disco sofre um dano permanente, todos os arquivos que possuíam blocos armazenados naquele disco são perdidos, ou seja, a confiabilidade de um sistema com listras (blocos de um mesmo arquivo que são divididos entre vários discos) é menor do que a de um sistema convencional. Por isso, surge a necessidade de introduzir algum tipo de redundância que faça com que falhas em um dos discos não prejudiquem o acesso aos arquivos do sistema.

Uma primeira possibilidade de implementação de redundância são os **discos espelho** que contém cópias exatas de outros discos. Se um disco falha, existe outro disponível. Se os dois estiverem em funcionamento, eles podem ser utilizados em paralelo para a transferência de dados. A grande desvantagem é o gasto extra de disco. É necessário adquirir o dobro de espaço do que nos sistema convencionais.

Outra solução é a adoção de um grau menor de redundância. Digamos, por exemplo, que existam 9 discos disponíveis. Neste caso, 8 discos são reservados para os blocos dos arquivos e o nono disco guarda blocos contendo bits de paridade resultantes da operação "ou exclusivo" (XOR) entre os respectivos blocos dos 8 discos que guardam os arquivos. Aqui, as listras são seqüências de 9 blocos, 8 blocos contíguos de um arquivo mais um bloco de paridade.

Se um dos blocos de dados é danificado, é possível reconstituí-lo a partir dos demais blocos de dados e do bloco de paridade da sua listra. Este método gasta muito menos espaço adicional mas, por outro lado, o desempenho, quando um dos discos de dados falha, é menor. Isto ocorre pois, para cada bloco lido, é necessário calcular o XOR dos seus bits com os blocos anteriores da mesma listra a fim de reconstituir os blocos inacessíveis.

Níveis intermediários de redundância podem ser obtidos através da aplicação de técnicas como os códigos de Hamming [Ham86].

---

[38] Por outro lado, o RAID não resolve o problema da latência, isto é, o tempo de espera até a chegada do primeiro bloco não muda muito.

### 2.7.3 As Listras do ZEBRA

A idéia dos RAIDs pode ser estendida para sistemas distribuídos. O SWIFT distribui os blocos das listras pelos diversos servidores de uma rede local [HO92]. Sem necessitar de nenhum hardware adicional (ao contrário dos RAIDs) as listras distribuídas pelos servidores possibilitam uma maior velocidade de acesso aos arquivos e uma distribuição eqüitativa, entre os servidores, das solicitações dos clientes.

Como o SWIFT e os sistema centralizados que utilizam a idéia dos RAIDs constroem listras separadas para cada arquivo, estes sistemas não melhoram o acesso a arquivos pequenos que não chegam a ser distribuídos. Outro problema do método adotado pelo SWIFT é a manutenção dos blocos de paridade. Os RAIDS são conectados a um computador que recebe todos os dados da listra e efetua o cálculo do XOR. Em um sistema distribuído, este computador central não existe. Quem deve ser o responsável pelo cálculo dos blocos de paridade quando um arquivo é acessado simultaneamente por mais de um cliente? Além disso, quando um bloco é atualizado, o bloco de paridade também precisa ser atualizado. Para isso, é necessário acessar o conteúdo antigo do bloco de dados e do bloco de paridade o que implica em vários acessos aos servidores a cada atualização de bloco. Tudo isso tendo que ser realizado atomicamente a fim de evitar a perda da consistência entre os dados e a paridade. A implementação de transações atômicas envolvendo dois servidores introduziria complexidade e sobrecarga ao sistema dificultando ainda mais a distribuição das listras entre os servidores.

Alguns destes problemas são resolvidos pelo ZEBRA através de uma pequena alteração na idéia original das listras. Ao invés de realizar o listramento *por arquivo*, o ZEBRA o faz *por cliente*. Os dados escritos por um cliente em um determinado intervalo de tempo são agrupados em uma mesma listra independente de pertencerem aos mesmos arquivos ou não. Esta listra é, então, dividida em **fragmentos** e cada fragmento é enviado para um servidor distinto.

À medida em que vai enviando os fragmentos, os clientes calculam o segmento de paridade. Quando uma listra termina, o segmento de paridade é enviado para o servidor responsável pela redundância. Este cálculo da paridade em um só nó é possível graças ao fato de que cada listra é gerada por um só cliente. Os acessos aos blocos de dados e de paridade antigos não são necessários porque o ZEBRA utiliza o princípio dos sistemas baseados em *log* segundo o qual os blocos físicos não são alterados. As atualizações são sempre concatenadas ao fim do *log*.

O ZEBRA não atualiza os dados e a paridade de maneira atômica. Um esquema baseado em *timestamps* é utilizado para verificar se os blocos de paridade são recentes. Quando o servidor se recupera de uma queda, ele verifica se todos os blocos de paridade estão atualizados. Caso contrário, é possível atualizá-los.

A figura 2.14 mostra como as listras são formadas em um sistema com 6 servidores. Além da possibilidade da figura onde um dos servidores armazena apenas os blocos de paridade, é possível também distribuir a responsabilidade pelos blocos de paridade entre todos os servidores. Em diferentes listras, diferentes servidores guardam os blocos de redundância.

O arquivo 1 da figura 2.14 tem os seus blocos espalhados por 5 servidores e pode ser lido até 5 vezes mais rápido do que um sem listras dependendo do poder das CPUs. Já o fragmento 5 é um exemplo de agrupamento de escritas em 5 arquivos diferentes em uma única mensagem ao servidor.

### 2.7.4 *File Manager*

Quando o cliente termina de escrever uma listra, ele envia as informações sobre a localização dos blocos da listra e a que arquivos pertencem para um nó especial denominado *file manager*.

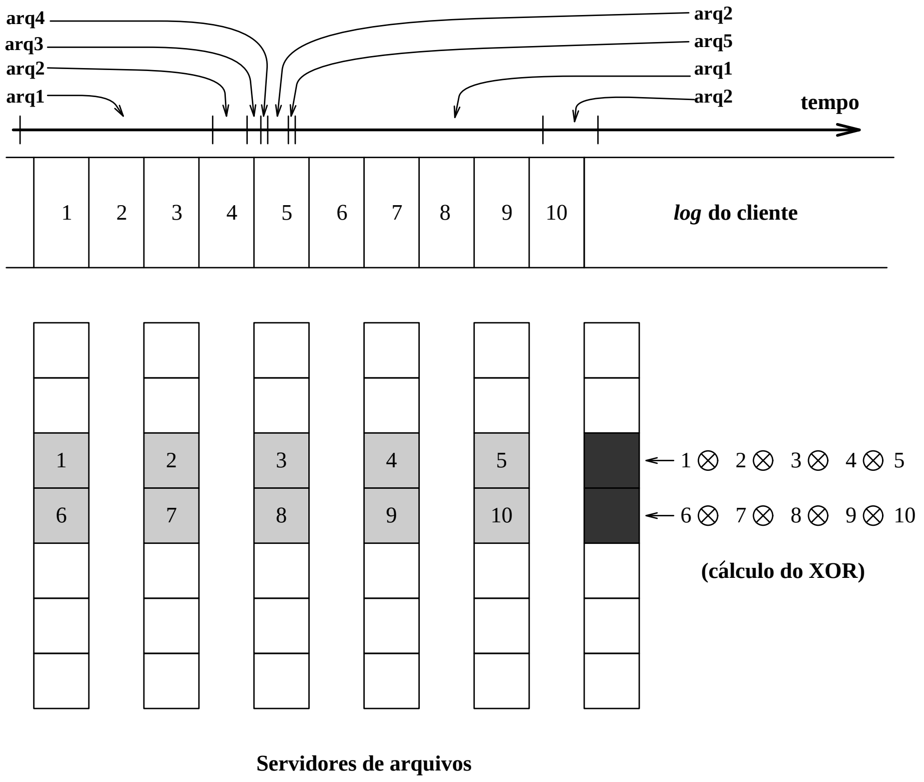



Figura 2.14: O listramento

O *file manager* é um recurso crítico do ZEBRA que administra todas as informações sobre o espaço de nomes do sistema de arquivos. Se, em um certo instante, ele não pode ser acessado, então todo o sistema fica indisponível. Além disso, a escalabilidade fica limitada tanto pela capacidade do *file manager* de armazenar a localização de cada bloco válido do sistema de arquivos quanto pela sua capacidade em atender às solicitações dos clientes o que leva a crer que o ZEBRA seja ainda menos escalável do que o SPRITE.

O problema da disponibilidade é minimizado pela manutenção de um *file manager backup* que entra em ação quando o titular falha. Uma solução simples que diminuiria consideravelmente a carga no *file manager* seria a implementação do cache de nomes e diretórios nos clientes. Este tipo de cache ainda não foi implementado nem no ZEBRA nem no SPRITE.

A mesma máquina que executa o *file manager* é, em geral, responsável pelo *stripe cleaner* que é um processo semelhante ao limpador de segmentos do SPRITE LFS. A sua função é ler as listras antigas que possuem maior quantidade de blocos desatualizados, escrever os dados válidos destas listras para novas listras e liberar o espaço reservado para as listras antigas.

### 2.7.5 Desempenho

Medições realizadas no SPRITE LFS mostraram que a sobrecarga gerada pelo processo de limpeza é, em média, pequena (entre 2% e 7% das escritas em disco segundo [HO92]). Este esforço extra é compensado por um grande ganho nas leituras e escritas de arquivos grandes e nas escritas dos arquivos pequenos[39].

[39]Os métodos do ZEBRA não afetam a leitura de arquivos pequenos que não chegam a ser listrados.

Para garantir a manutenção da semântica UNIX, o ZEBRA, assim como o SPRITE, desabilita o cache quando existe acesso concorrente com escrita a um certo arquivo. Neste caso, o ZEBRA não espera até que a listra se complete para enviá-la ao servidor. Toda vez que um fragmento de listra recebe um bloco de um arquivo não cacheável, este fragmento é enviado para o servidor mesmo estando incompleto. Posteriormente, o resto do fragmento é enviado. Segundo [BAD+92], em ambientes com utilização massiva de transações, como no caso de bancos de dados por exemplo, até 90% dos segmentos escritos em um sistema de arquivos baseados em *log* são segmentos parciais cuja escrita foi forçada através do comando *fsync*. Em ambientes deste tipo, a eficiência do ZEBRA pode ser seriamente comprometida porque os *writes* deixam de ser agrupados e as escritas no disco podem gerar *seeks*.

[HO93] apresenta o resultado de testes comparativos entre o desempenho do NFS, SPRITE e ZEBRA. Os testes apresentados neste trabalho medem apenas as situações onde era de se esperar que o ZEBRA tivesse o melhor desempenho. O ZEBRA se saiu muito melhor em aplicações que manipulam arquivos de vários Megabytes tanto para leitura quanto para escrita e em aplicações que gravam muitos arquivos pequenos. Não foram apresentadas medições do desempenho do ZEBRA sob a carga normal de uma rede local de uso compartilhado como foi feito em trabalhos anteriores publicados por pesquisadores de Berkeley. Os três testes apresentados foram os seguintes:

1. Cada cliente executa uma única aplicação que cria um arquivo de 12Mb e o envia para o disco. Sem a manutenção dos blocos de paridade, o ZEBRA foi de 2 a 6 vezes mais rápido do que o SPRITE e o NFS de acordo com o número de servidores (entre 1 e 4) utilizados pelas listras. Com a manutenção da redundância o desempenho do ZEBRA foi de 2 a 3 vezes superior. Isto se deve basicamente a dois fatores:
   - O ZEBRA executa RPCs assíncronas permitindo a sobreposição da transferência dos dados ao disco com a transferência dos dados pela rede.
   - Os clientes ZEBRA enviam os dados para os servidores em fragmentos de listras de 512 Kbytes eliminando grande parte da sobrecarga causada pelos pequenos blocos de transferência do NFS e do SPRITE (apenas 8 Kbytes).

2. Cada cliente executa uma única aplicação que lê um arquivo de 12Mb. O ZEBRA chegou a ser 10 vezes mais rápido com 4 servidores. Além das RPCs assíncronas, o principal responsável por este enorme ganho é o paralelismo da transferência dos blocos do arquivo para os clientes. No caso em que um dos servidores não estava disponível e a informação de redundância teve que ser utilizada, o desempenho do ZEBRA foi cerca de 3 vezes superior pouco variando com o aumento do número de servidores.

3. Um cliente cria 2048 arquivos de 1Kbyte e, em seguida, os envia para o servidor (no caso do NFS, cada arquivo foi enviado para o servidor logo após ter sido fechado). Tanto o SPRITE quanto o ZEBRA foram cerca de 3 vezes mais rápidos do que o NFS. Obviamente, este resultado se deve à política de escritas longamente retardadas do ZEBRA e do SPRITE e ao fato de as escritas retardadas do NFS terem sido forçadamente eliminadas com o comando *flush*.

Apesar de parecerem animadores, estes dados devem ser analisados com cautela. Apenas aplicações específicas foram analisadas, esses testes não servem como estimativa do desempenho real do ZEBRA no dia a dia. Os grandes ganhos aparecem nas comparações do desempenho do ZEBRA com 4 servidores com o desempenho do NFS e SPRITE com apenas um servidor. Embora o NFS e o SPRITE não sejam capazes de aproveitar o poder de vários servidores na transferência de um único arquivo, em um sistema real, os vários servidores também são utilizados em paralelo fornecendo os blocos de diferentes arquivos simultaneamente.

### 2.7.6 Resumo

O ZEBRA implementa idéias das matrizes de discos redundantes e dos sistemas de arquivos baseados *log* no contexto de sistemas de arquivos distribuídos. Deste modo, o Zebra distribui eqüitativamente a carga gerada pelos clientes entre todos os servidores do sistema.

Através de um mesmo método, o ZEBRA trata eficientemente tanto os arquivos grandes quanto os pequenos. Tanto a leitura quanto a escrita de arquivos grandes são distribuídas entre vários servidores. As escritas em arquivos pequenos são agrupadas reduzindo a sobrecarga.

A manutenção de blocos de redundância em um ou mais servidores permite que o sistema continue disponível mesmo com a queda de um servidor qualquer.

A disponibilidade e a escalabilidade do sistema são limitadas pela centralização do *file manager*. O problema da disponibilidade, no entanto, é minimizado pela adoção de um *file manager back-up* que assume o controle quando o titular falha.

## 2.8 HARP

Na seção 2.5, vimos que o sistema CODA aplica um método otimista para a replicação de arquivos. Ou seja, no caso de uma partição da rede, todas as cópias dos arquivos podem ser acessados e modificados. Se duas cópias de um mesmo arquivo sofrem modificações incompatíveis, este problema é tratado no momento da reintegração da rede.

Veremos agora, sistemas mais cautelosos que não permitem que tais inconsistências ocorram. O primeiro deles é o HARP, um sistema experimental que vem sendo desenvolvido no MIT desde o início desta década e que tem como principal característica o alto grau de tolerância a falhas.

O HARP (*Highly Available, Reliable, Persistent file system*) utiliza o modelo de cópia primária (*primary copy replication*)[40] para a replicação dos arquivos e, ao contrário de outros sistemas que implementavam este modelo, opera sobre hardware convencional.

As solicitações dos clientes são enviadas para um **servidor primário** que as transmite para **servidores secundários** (ou servidores *back-up*). O servidor primário espera pela resposta dos servidores secundários e, só então, envia ao cliente o resultado de sua solicitação. Quando ocorre alguma falha no acesso ao servidor primário, um dos servidores secundários assume o seu lugar e se torna o servidor titular.

A fim de garantir que o efeito das operações não seja perdido quando o servidor falha, o HARP sempre informa um número suficiente de servidores secundários sobre todas as operações que alteram o estado de seu sistema de arquivos.

Se um cliente que efetua um *write* tivesse que esperar que os dados fossem escritos nos discos de vários servidores, o tempo de resposta desta operação seria muito grande. A solução adotada é semelhante ao cache na memória do servidor adotado pelo NFS e pelo SPRITE. As operações que alteram o sistema de arquivos são concatenadas a um *log* residente na memória principal dos servidores.

Logo que um servidor secundário recebe uma solicitação, a operação é colocada no *log* e uma resposta confirmando o recebimento da solicitação e enviada ao servidor primário. Os servidores completam as operações em *background* retirando da parte crítica do protocolo o lento acesso aos discos. Este tipo de escrita é chamada de *write-behind*.

Uma particularidade do HARP é a utilização de UPSs (*uninterruptible power supplies*) que permitem que os servidores funcionem por mais alguns minutos no caso de uma queda no fornecimento de energia. Este tempo extra é utilizado para aplicar aos discos as operações presentes no *log* evitando a perda de informações quando o fornecimento de energia elétrica é interrompido. Se o período sem fornecimento de energia é curto, o serviço de arquivos não chega a ser interrompido.

A implementação do HARP baseou-se no modelo de sistema de arquivos virtual (ver seção 2.3.6), os clientes o acessam através do protocolo NFS. O HARP acessa o sistema de arquivos local através de *system calls* do UNIX. Para os clientes, não existe nenhuma diferença entre o serviço de arquivos do NFS e do HARP a não ser que o último, sendo tolerante a falhas, consegue oferecer um serviço mais confiável e com menos interrupções.

Para os administradores da rede, o sistema é um pouco mais complexo do que o NFS uma vez que é necessário configurar a replicação e, quando ocorrem falhas mais demoradas, reconfigurá-la.

Finalmente, para os responsáveis pela compra do equipamento, a grande diferença é a necessidade de adquirir as UPSs e os discos extras. Se a carga nos servidores é pesada, pode ser necessária também a aquisição de servidores extras a fim de evitar a degradação do desempenho global da rede.

[40] Veja [Tan92].

Ao contrário do CODA, a coerência entre as várias cópias de um mesmo arquivo em diferentes servidores é mantida. Não se pode dizer que a semântica UNIX seja perfeitamente satisfeita porque o HARP é acessado através de clientes NFS que cacheiam arquivos sem garantir a sua consistência como vimos na seção 2.3.5. O HARP também não ajuda na distribuição da carga entre os servidores pois, a cada instante, apenas um servidor atende às solicitações dos clientes para cada conjunto de arquivos.

### 2.8.1 Implementando a replicação

Um dos métodos mais usados para replicação de dados em sistemas distribuídos é o método da **votação** proposta por Gifford em [Gif79].

Segundo o método da votação, se um arquivo é replicado em $N$ servidores então o cliente precisa conseguir um **quórum para leitura** que corresponde à concordância de pelo menos $N_l$ servidores para que uma leitura a este arquivo seja completada. Analogamente, uma escrita necessita da aprovação de pelo menos $N_e$ servidores (o **quórum para escrita**) para ser completada. Os quóruns estão sujeitos à restrição $N_l + N_e > N$ (1). No caso em que $N_e > N_l$ (2), o método da votação se torna pessimista segundo a classificação da seção 2.5.1. Isto é, quando ocorre uma partição, ou as escritas são desabilitadas ou as leituras e escritas são permitidas em, no máximo, uma das partes. Vejamos como isso ocorre.

Suponha que uma rede tenha sofrido uma partição $\mathcal{P} = \{P_1, P_2, P_3, \ldots\}$ onde cada parte $P_i$ é um conjunto de servidores que conseguem trocar informações entre si mas que perderam contato com os demais servidores. Se $|P_k| \geq N_e$ (onde $|P_k|$ é o número de servidores em $P_k$) então $P_k$ é a única parte que pode efetuar escritas pois (1) e (2) implicam que um quórum para escrita necessita de mais da metade dos servidores.

Se, no entanto, duas partes $P_k$ e $P_t$ possuem o quórum para leitura, então nenhuma delas pode completar escritas. Para verificar isso, suponha que $P_k$ possua o número necessário de servidores para conseguir um quórum para escrita, como $|P_t| \geq N_l$ então $|P_k| + |P_t| \geq N_e + N_l > N$ o que implica que $P_k$ e $P_t$ possuem servidores comuns contradizendo a hipótese de que os servidores de $P_k$ perderam o contato com os servidores de $P_t$.

O HARP segue um caso particular do esquema de votação segundo o qual qualquer operação sobre o sistema de arquivos necessita da aprovação de mais da metade dos servidores para ser completada.

No HARP, cada arquivo é administrado por um grupo de $N$ servidores[41] sendo que um dos servidores é o primário, $\lfloor \frac{N}{2} \rfloor$ servidores são os secundários e o grupo restante de $\lfloor \frac{N}{2} \rfloor$ servidores são as **testemunhas**.

Os servidores primários e secundários armazenam réplicas do arquivo em seus discos e participam das votações. Já as testemunhas, apenas participam das votações sem armazenar o arquivo. Este esquema permite que se acesse o sistema de arquivos mesmo com a queda de até $\lfloor \frac{N}{2} \rfloor$ servidores. Quanto maior for $N$ mais confiável e tolerante a falhas será o sistema.

A melhor maneira de organizar o sistema a fim de distribuir a carga nos servidores eqüitativamente é fazer com que cada servidor desempenhe o papel do primário, secundário e testemunha simultaneamente para diferentes conjuntos de arquivos.

Uma operação sobre o sistema de arquivos exige a aprovação do servidor titular (que é o primário ou um dos secundário no caso do primário estar inacessível) e de mais $\lfloor \frac{N}{2} \rfloor$ servidores secundários ou testemunhas.

Na implementação apresentada em [LGJ+91], $N$ é 3, ou seja, cada grupo de servidores possui um primário, um secundário e uma testemunha. Veremos, a seguir, como é o funcionamento do HARP neste caso.

---

[41] $N$ deve ser ímpar.

### 2.8.2 Modo Normal de Operação

Quando o servidor primário é também o titular, isto é, quando ele pode ser acessado, o HARP está no seu modo normal de operação. Neste modo, o servidor primário mantém um *log* em sua memória contendo registros de todas operações que ainda não estão a salvo nos discos dos servidores secundários e dele mesmo.

As operações no sistema de arquivos são efetuadas em duas fases:

1. O servidor primário cria um registro para a operação no seu *log* e o envia para os servidores secundários que, por sua vez, confirmam o recebimento. Quando as respostas confirmando o recebimento do registro atingem o quórum, o registro é **comprometido**. Um apontador, PC (ponto de comprometimento), indica a posição do último registro comprometido do *log* (os registros são obrigatoriamente comprometidos na mesma ordem em que aparecem no *log*). Os servidores secundários guardam os registros recebidos em seus *logs* na memória principal.

2. Sempre que o primário envia mensagens aos secundários, ele anexa o valor de PC. Deste modo, os secundários são informados do comprometimento dos registros e os enviam, finalmente, para o sistema de arquivos local. Quando um registro comprometido chega ao disco, ele é removido do *log*.

Algumas operações, como a leitura dos atributos de um arquivo, não alteram o estado do sistema de arquivos e podem ser tratadas exclusivamente pelo servidor primário sem a necessidade de efetuar uma votação. Mesmo nestes casos, o servidor primário precisa ter certeza de que uma partição na rede não fez com que um servidor secundário tenha se tornado titular. O primário obtém esta certeza a partir do seguinte mecanismo.

Os servidores de um grupo mantém os seus relógios sincronizados a menos de um erro $\epsilon$. Toda mensagem de um servidor secundário para o primário contém um valor $t$ que é o valor corrente do relógio do servidor secundário + $\delta$, onde $\delta$ é um intervalo de alguns décimos de segundo. $t$ é uma promessa do secundário de que ele não vai assumir a posição de titular nos próximos $\delta$ décimos de segundo. Como o processo de um secundário se tornar titular demora muito mais do que isso, não há perigo desse mecanismo atrasar a promoção de um servidor secundário a titular.

Deste modo, o primário só precisa se comunicar com um secundário a fim de detectar partições quando o valor no seu relógio é maior ou igual a $t - \epsilon$. A sincronização dos relógios é feita através do *Network Time Protocol* [Mil92].

É bom lembrar que a leitura de um arquivo UNIX é uma operação que altera o estado do sistema de arquivos pois um dos atributos de um arquivo é o instante no qual ele foi lido pela última vez. Logo, a operação *read* no HARP tende a ser mais lenta do que no NFS comum, pois há a necessidade de obter a aprovação dos secundários. A fim de amenizar este problema, o HARP pode ser configurado de duas maneiras. Em um dos modos, as leituras são consideradas operações de atualização normais. No outro modo, os registros das leituras são enviados para os secundários mas o primário não espera a formação do quórum, enviando imediatamente para o cliente a última versão comprometida dos dados solicitados. Esta solução pode gerar inconsistências quando ocorrem falhas no sistema mas, como o atributo em questão (instante da última leitura) é raramente acessado pelas aplicações, este método dificilmente causa problemas.

#### Operação no Caso de Partições

Quando ocorre um particionamento ou uma reintegração após um particionamento, o sistema executa uma **mudança de cenário** que consiste na promoção de um servidor secundário a titular ou no restabelecimento do papel do primário no caso de uma reintegração. O estado do sistema após esta reorganização é chamado de **cenário**. Cada cenário é caracterizado por quais servidores desempenham os papéis de titular e de *back-up*.

Os membros de um grupo sempre fazem parte de algum cenário e armazenam, em disco, um número que identifica este cenário. Se há uma partição, diferentes membros de um grupo podem estar em diferentes cenários.

Voltando ao caso apresentado em [LGJ+91] (3 servidores por grupo), cada cenário possui pelo menos dois membros do grupo. Um deles é o titular e o outro, o *back-up*. O primário é o titular sempre que possível, o secundário, normalmente o *back-up*, torna-se titular no caso do primário estar inacessível e a testemunha assume o papel de *back-up* no caso de um dos outros dois servidores estarem inacessíveis.

Uma testemunha que foi promovida a *back-up* é vista pelo titular como um *back-up* normal mas, internamente, o seu funcionamento é diferente. Como a testemunha não possui uma cópia local do sistema de arquivos em questão, ela não pode aplicar as operações do *log*. Ao invés disso, ela mantém todos os registros em um *log* em fita magnética. Quando o servidor que estava inacessível deixa de sê-lo, a testemunha lhe envia o seu *log*, permitindo que o servidor atualize o seu sistema de arquivos convenientemente.

A grande vantagem do HARP é que mesmo com a queda de qualquer um dos três servidores, ainda haverá duas cópias de cada operação comprometida garantindo, assim, a estabilidade do sistema de arquivos. Se dois dos três servidores estão inacessíveis, o serviço não é oferecido[42].

No momento da publicação de [LGJ+91], as mudanças de cenário ainda não estavam implementadas. No entanto, este artigo traz maiores detalhes sobre como implementá-las de maneira eficiente e segura.

### 2.8.3 Desempenho

[LGJ+91] mostra o resultado de alguns testes do desempenho de servidores HARP implementados sobre o UNIX 4.3BSD com modificações da SUN e da Universidade de Wisconsin para o oferecimento do NFS e do VFS. Tanto os servidores quanto os clientes eram MicroVAX 3500. Analisando o tempo de resposta às solicitações dos clientes e o tempo necessário para realização do *Andrew Benchmark*[43] chegou-se a conclusão de que o HARP é ligeiramente superior ao NFS sobre o qual foi implementado. Este melhor desempenho se deve ao fato do HARP substituir o acesso imediato ao disco pela troca de mensagens com o servidor *back-up* no caso das escritas.

No entanto, não podemos dizer que o HARP seja mais rápido do que o NFS implementado nas últimas versões do SunOS. Se comparado a sistemas como o AFS e o SPRITE o desempenho do HARP será bem fraco mas não se deve esquecer que o HARP oferece um serviço tolerante a falhas o que é essencial para muitas aplicações.

O HARP não é tão escalável quanto o CODA pois a responsabilidade de distribuir as réplicas é dos servidores e não dos clientes. Assim, um servidor HARP se satura com menos clientes do que um servidor CODA. Ainda assim, o HARP possui a vantagem de manter um certo número mínimo de cópias consistentes de cada arquivo replicado, coisa que o CODA não faz.

[42] Na realidade, o HARP pode ser configurado para permitir leituras quando apenas um servidor está acessível. Mas, neste modo de operação, os clientes estão sujeitos ao recebimento de dados desatualizados.

[43] Ver seção 2.5.5.

### 2.8.4 Resumo

O HARP oferece um sistema com alta tolerância a falhas através da adoção do modelo de cópia primária para replicação de arquivos. Ao contrário de outros sistemas mais antigos que implementavam a replicação automática, o HARP opera sobre hardware convencional. A única exceção são as UPSs que garantem o funcionamento do sistema por alguns minutos no caso de uma interrupção no fornecimento de energia.

Qualquer operação sobre o sistema de arquivos necessita da aprovação da maioria absoluta das máquinas responsáveis pelo arquivo em questão. Qualquer atualização de dados no sistema de arquivos só é confirmada após ter sido registrada por, pelo menos, dois servidores garantindo, assim, o fornecimento de um serviço de alta confiabilidade.

A adoção desta estratégia pessimista de replicação conduz, porém, a uma inevitável perda na velocidade de acesso aos arquivos e na escalabilidade do sistema.

## 2.9 FROLIC

Tanto o AFS quanto o SPRITE foram projetados a partir da coleta de informações sobre o tipo de acesso ao sistema de arquivos em redes acadêmicas. Em ambos os casos, o compartilhamento de arquivos por diferentes clientes foi considerado pequeno.

Esta constatação serviu para a adoção da política do SPRITE de desabilitar o cache de arquivos compartilhados com escrita, tornando o serviço muito mais lento. Além disso, o protocolo para manutenção da consistência do cache dos clientes (assim como no NFS) obriga os clientes do SPRITE a se comunicarem com o servidor quando da abertura de um arquivo a fim de verificar se a versão no seu cache é a mais recente. Este mecanismo impede o crescimento do número de clientes para algumas centenas pois os servidores ficam rapidamente sobrecarregados.

O AFS, no entanto, utiliza o mecanismo de *callbacks* segundo o qual os clientes não precisam contatar o servidor para abrir um arquivo. A responsabilidade pela manutenção da consistência é dos servidores que avisam os clientes quando um arquivo é alterado. Mas, como o AFS adota a semântica de seção (as alterações só são enviadas para o servidor no momento do *close*), a consistência no acesso concorrente não é garantida. Este não é o único problema. Suponha que um arquivo seja cacheado em 10 células AFS, e, dentro de cada célula, em 10 clientes. Neste caso, quando o arquivo é alterado, o servidor precisa enviar 100 mensagens quebrando os *callbacks* do clientes[44]. Nos minutos seguintes, o servidor receberá dezenas de solicitações de leitura da nova versão do arquivo, como as mensagens não são recebidas simultaneamente, o servidor não pode responder a todas de uma vez, ele terá que introduzir na rede dezenas de mensagens, causando um sobrecarga muito grande. Como a coleta de dados nas redes acadêmicas mostraram que esta é uma possibilidade remota, os projetistas do ANDREW não se incomodaram.

Estudos mais recentes [GZS94] demonstram que um compartilhamento muito maior do que aqueles das redes acadêmicas pode ser comum em redes que possuem outra finalidade[45].

Os dados apresentados em [GZS94] foram coletados em um grande sistema computacional voltado para a produção de software envolvendo milhares de estações em diversas redes locais de uma grande empresa de engenharia. Usuários de um mesmo projeto trabalham em redes locais ou, mais genericamente, *clusters* de 10 a 50 estações e um ou mais servidores. Os *clusters* são interligados por vários tipos de barramentos e linhas seriais. As principais particularidades encontradas foram:

1. Uma grande quantidade de arquivos, incluindo bibliotecas, pequenos executáveis e aplicativos completos que sofrem mudanças diárias, ou, em alguns casos, várias vezes por dia, são compartilhados por centenas de usuários em diferentes *clusters*.

2. A maioria dos arquivos compartilhados possuem até 10 Kbytes, mas algumas exceções chegam a alguns Megabytes.

3. Os arquivos compartilhados são geograficamente distribuídos por *clusters* em diversos pontos da cidade onde a empresa é sediada.

4. A grande quantidade de dados compartilhados freqüentemente causa esperas excessivas devido à sobrecarga na rede e em certos servidores.

[44]Ou, se o protocolo da rede permitir, uma mensagem com 100 destinatários.

[45]Mesmo nas redes acadêmicas, é possível que os diretórios relativos a projetos de grupos de pesquisa apresentassem um compartilhamento muito maior se os pesquisadores não estivessem conscientes dos problemas que o compartilhamento excessivo poderia causar.

5. Entre os dados compartilhados com escrita, a situação mais comum é a existência de um *cluster* produtor e vários *clusters* consumidores. Raramente mais de um *cluster* altera mais de um conjunto de dados.

Em redes com este tipo de carga, os sistemas de arquivos cujos clientes são obrigados a se comunicar diretamente com servidores de outros *clusters* (como mostra a figura 2.15) não funcionam bem. Tanto o AFS quanto o NFS e o SPRITE exigem uma comunicação direta entre o servidor e todos os seus clientes. Os dois últimos foram desenvolvidos para redes locais e não se adaptam a redes de grande escala. O AFS só funciona bem quando os arquivos compartilhados são raramente alterados e o servidor não precisa emitir centenas de mensagens de quebra de *callbacks* a todo instante.

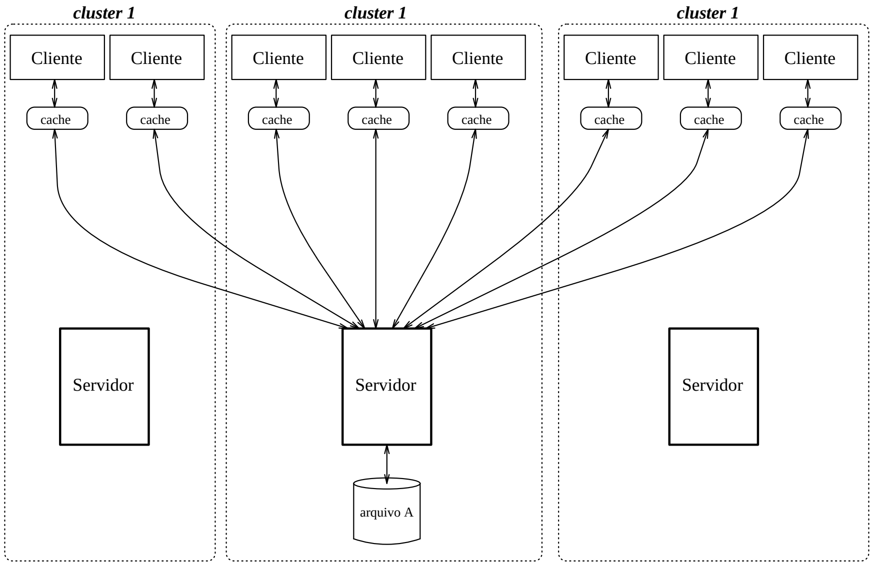



Figura 2.15: Modelo tradicional de cache nos clientes

Por outro lado, o CODA resolve este problema mantendo várias cópias dos arquivos em diferentes servidores. Cada cliente pode acessar a cópia que estiver mais próxima, as alterações são enviadas em paralelo para todos os servidores responsáveis através de multiRPCs. No CODA, os administradores do sistema tem a responsabilidade de escolher quais volumes serão replicados e, para cada cliente, quais serão os servidores preferenciais de cada volume.

Na Universidade de Toronto está sendo desenvolvido o sistema FROLIC que tem como objetivo oferecer um serviço rápido para redes de grande escala através da replicação automática de arquivos em servidores de *clusters* distintos. Um arquivo FROLIC é elegível para replicação em qualquer *cluster* que o acesse. A replicação é determinada dinamicamente pelo acesso ao sistema de arquivos.

A idéia principal do FROLIC é hierarquizar o acesso aos arquivos de modo que cada cliente se comunique apenas com o servidor local. Cada servidor é responsável por providenciar uma cópia dos arquivos de interesse para o seu *cluster*. A consistência entre as réplicas e toda a comunicação *inter-cluster* é de responsabilidade dos servidores (ver figura 2.16). Como os servidores aparecem em um número muito menor do que os clientes, espera-se diminuir a carga nos canais de ligação *inter-cluster* que costumam ser o principal gargalo para as redes de grande escala.

O FROLIC ainda está na fase de desenvolvimento. Uma boa parte dos mecanismos de

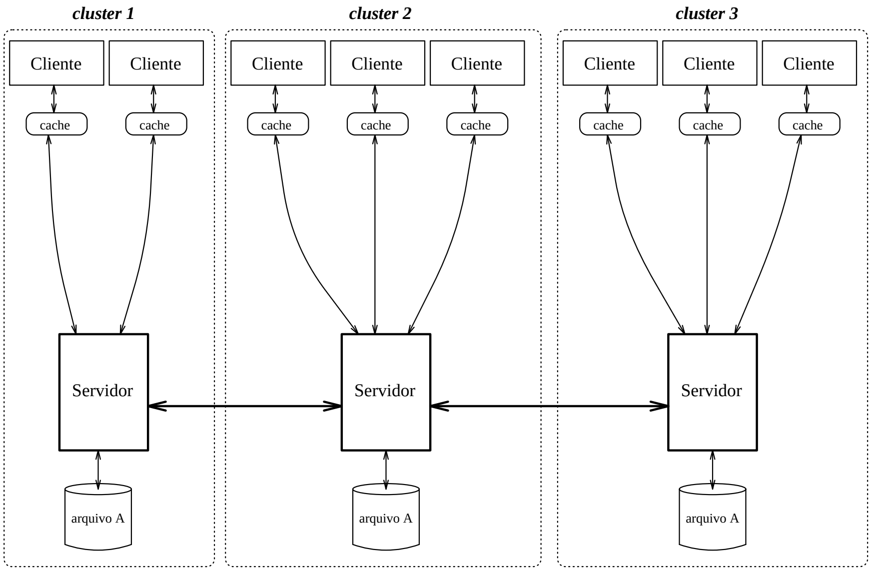



Figura 2.16: Modelo do FROLIC

tratamento da consistência e do processo de localização dos arquivos ainda não foram definidos. [PGZ92] descreve os primeiros passos de sua implementação e mostra o resultado de pequenos testes com dois servidores. [SZ92] discute algumas possibilidades para o tratamento da consistência e do espaço de nomes e apresenta o resultado de simulações em uma rede hipotética de 10 *clusters*. Os conceitos de disponibilidade e de tolerância a falhas ainda não foram abordados.

### 2.9.1 Replicação Dinâmica

Na configuração básica o FROLIC é formado por um conjunto de *clusters* contendo um servidor cada um. Os clientes solicitam o acesso aos arquivos apenas aos servidores locais. Se um servidor recebe a solicitação de abertura de um arquivo que não possui uma réplica local, ele é responsável por se comunicar com os servidores de outros *clusters* e criar uma réplica local.

No protótipo descrito em [PGZ92], as operações *open, close, creat, unlink, mkdir* e *rmdir* são capturadas por uma biblioteca especial que é quem efetivamente implementa o cliente FROLIC. As operações *read* e *write* são tratadas pelo NFS padrão uma vez que elas não ultrapassam os limites de um *cluster*. O processo de resolução dos *pathnames* é feito através de tabelas de prefixos (ver seção 2.6.1) tanto nos clientes quanto nos servidores.

Neste protótipo, os servidores mantém uma lista que pode conter dois tipos de informações para cada arquivo replicado:

- Se a réplica local é a versão mais recente, a lista informa os nomes dos *clusters* que contêm uma cópia válida do arquivo.

- Se a cópia local não corresponde à versão mais recente (é inválida), então a lista guarda o nome de um *cluster* que possui a versão mais recente, ou então, que sabe onde ela está.

Além disso, os *i-nodes* de cada arquivo possuem um bit que indica se o arquivo é compartilhável ou não.

Quando um cliente abre um arquivo, a biblioteca intercepta o *open* e envia a solicitação de abertura para um processo do servidor local. O servidor poderá tomar três atitudes distintas de acordo com o estado da sua cópia do arquivo:

1. Se o *i-node* do arquivo indica que ele não é compartilhável ou se a tabela do servidor indica que a réplica local é válida, então o arquivo é aberto e o cliente recebe um *file handle* que é usado nas chamadas NFS *intra-cluster*.

2. Se, por outro lado, a cópia local está marcada como inválida, o servidor verifica na sua tabela qual foi o *cluster* que enviou uma mensagem de invalidação para esse arquivo. O servidor se comunica com este *cluster* e pergunta se ele possui uma versão atualizada do arquivo. Se ele a possuir, então o servidor solicita uma cópia desta versão. Caso contrário, o servidor do *cluster* remoto indica o nome de um terceiro *cluster* que foi quem invalidou a sua cópia do arquivo. Neste caso o servidor local repete o mesmo processo com este terceiro *cluster* até localizar o paradeiro da cópia mais recente do arquivo.

3. Finalmente, se o arquivo não é encontrado no servidor local e a tabela de prefixos indica que o arquivo está em um diretório remoto, então o servidor envia uma mensagem para todos os *clusters* que possam ter a versão mais recente do arquivo. Os *clusters* que possuem cópias válidas enviam uma resposta afirmativa. O servidor local escolhe um dos *clusters* (possivelmente o que responder primeiro) e solicita uma cópia do arquivo. Uma réplica local é criada e, se necessário, diretórios intermediários também.

Quando um arquivo que foi alterado é fechado, o servidor envia uma mensagem de invalidação para todos os *clusters* que possuírem uma cópia válida deste arquivo. Ao receber uma mensagem de invalidação tudo o que o servidor remoto deve fazer é marcar o arquivo em questão como desatualizado na sua lista de arquivos replicados e gravar o nome do servidor responsável pela invalidação.

Se estes mecanismos forem aplicados a uma rede onde vários *clusters* efetuam modificações em um mesmo arquivo o tempo necessário para a abertura deste arquivo pode ser muito grande pois poderá ser necessário entrar em contato com um grande número de servidores até localizar o detentor da cópia válida.

Na rede estudada em [GZS94], o comportamento normal é que, para cada arquivo, apenas um *cluster* seja o produtor, os outros são apenas consumidores. Neste caso, apenas um *cluster* enviará mensagens invalidando o arquivo e a localização da cópia válida exigirá o contato com apenas um *cluster* remoto.

### 2.9.2 Semântica

O política adotada pela implementação atual é a de sincronizar as várias réplicas apenas no momento do *close*. Logo, a semântica do compartilhamento *inter-cluster* é a de sessão. Os artigos sobre o FROLIC não determinam a política *intra-cluster* de consistência. No protótipo descrito em [PGZ92], o acesso aos arquivos dentro de um *cluster* se faz através do NFS. Assim, esta implementação do FROLIC reune as fraquezas da semântica de sessão e da semântica imprevisível do NFS. Como até o presente momento nenhum mecanismo de votação é adotado, partições podem levar a inconsistências.

O problema da manutenção da consistência entre as réplicas no caso de escritas concorrentes é ignorado pelo protótipo. Já em [SZ92], os autores apresentam uma idéia que consiste em limitar as alterações em um determinado arquivo a um único *cluster* que seria o **dono** momentâneo do arquivo. Se um cliente deseja abrir um arquivo para escrita ou atualização, é necessário que o servidor local seja o dono do arquivo, ou então, que ele entre em contato com o dono corrente e consiga autorização para ser o novo dono. Desta forma, o problema da escrita concorrente *inter-clusters* desaparece.

Os responsáveis pelo FROLIC não mencionam em seus artigos a possibilidade de garantir a consistência perfeita no acesso concorrente. Uma boa solução para este problema seria implementar os mecanismos do SPRITE[46] no acesso *intra-cluster* juntamente com a idéia de limitar as alterações a apenas um *cluster*. Esta seria uma maneira de garantir a consistência das escritas concorrentes dentro de um *cluster* e manter a semântica de sessão entre clientes de *clusters* diferentes.

### 2.9.3 Desempenho

Sendo um sistema ainda em fase de implementação e desenvolvimento, não existem medições do desempenho do FROLIC em ambientes reais. O protótipo descrito em [PGZ92] foi testado em uma pequena rede com dois *clusters* com clientes SPARC IPC servidas localmente através do NFS. Nesta rede, a transferência de um bloco de 8Kbytes gasta 25 ms dentro de um *cluster* e 55 ms entre os dois *clusters*.

O desempenho do FROLIC varia muito de acordo com o padrão de acesso aos arquivos. Se trechos pequenos de arquivos grandes são lidos ou escritos, o FROLIC é ineficiente pois ele primeiro faz uma cópia completa do arquivo no servidor local para depois realizar a operação solicitada pelo cliente.

Podemos tomar como exemplo um caso extremo onde dois clientes em *clusters* distintos escrevem alternadamente algumas dezenas de bytes em um arquivo de alguns Mbytes sendo que o arquivo é aberto e fechado a cada escrita. Neste caso, o NFS (sem replicação) é muito mais rápido do que o FROLIC pois o NFS só transmite os dados novos enquanto que o FROLIC transmite o arquivo inteiro a cada novo acesso.

Se, por outro lado, considerarmos o caso em que um *cluster* atualiza o arquivo e o outro realiza leituras do arquivo inteiro, então o protótipo do FROLIC melhora o seu desempenho.

O protótipo do FROLIC apresenta um desempenho melhor do que o NFS quando os arquivos são acessados por pelo menos quatro clientes de um mesmo *cluster* antes de serem alterados por clientes de outros *clusters*. Quanto maior o número de clientes que acessam os arquivos em um *cluster* antes da sua alteração por clientes externos, melhor é o desempenho do FROLIC. O NFS, no entanto, é insensível à esta característica.

[SZ92] apresenta os resultados obtidos em simulações do comportamento do FROLIC efetuadas com uma ferramenta chamada Csim para uma rede com 10 *clusters*. Os autores observaram que o modelo do FROLIC parece ser mais eficiente para vários padrões de acesso aos arquivos. Resta agora saber se estes resultados serão observados também sob a carga de redes reais e se o modelo do FROLIC é realmente superior ao AFS, o seu concorrente direto.

### 2.9.4 Resumo

O FROLIC utiliza um novo modelo de cache para sistemas de grande escala no qual os clientes se comunicam apenas com o servidores de seu *cluster*. Toda a comunicação *intra-cluster* é realizada entre os servidores que replicam os arquivos dinamicamente conforme a necessidade.

Estudos realizados em uma rede de uma grande empresa de engenharia mostram que a adoção deste novo modelo de cache pode diminuir em muito o tráfego nos canais de comunicação *intra-clusters* beneficiando a escalabilidade do sistema.

[46] No capítulo 4 apresentaremos a técnica dos *leases* que garante a mesma consistência do SPRITE de maneira mais eficiente.

## 2.10 ECHO

O sistema de arquivos distribuído ECHO [BHJ+93] foi desenvolvido a partir de um projeto ambicioso que se iniciou em 1988 no centro de pesquisas da DEC em Palo Alto, California. O projeto reune qualidades do ANDREW (escalabilidade), do SPRITE (consistência do cache), do HARP (tolerância a falhas) além de apresentar características novas como a adoção de escritas retardadas assíncronas (*write-behind*) também no caso de diretórios e atributos de arquivos.

Desenvolvido sobre o sistema operacional experimental TAOS, o ECHO operou de janeiro de 1991 até meados de 1992 com cerca de 60 usuários e 25 Gigabytes de dados que, replicados, ocupavam 50 Gigabytes que eram acessados através de 8 servidores. A complexidade do sistema pode ser constatada a partir do tamanho do seu código fonte. O cliente é formado por 17.400 linhas de código fonte enquanto que o servidor é constituído por nada menos do que 100.400 linhas.

O desenvolvimento do sistema foi interrompido prematuramente no início de 1992 quando o projeto de transferir o TAOS para equipamentos mais modernos foi cancelado. Vejamos as principais características e mecanismos adotados pelo ECHO.

### 2.10.1 Tolerância a Falhas

Além da replicação de informações sobre o estado do sistema na memória principal de dois servidores, o ECHO utiliza simultaneamente dois outros tipos de replicação.

A fim de aumentar a disponibilidade, o ECHO replica servidores. É utilizado um tipo especial de interface com o controlador de discos que permite que dois servidores tenham acesso, um de cada vez, aos mesmos discos como mostra a figura 2.17a. Uma ligação especial entre os servidores denominada *Autonet* permite que se descubra, em menos de um segundo, se um servidor caiu ou não. Este mecanismo evita que dois servidores acessem simultaneamente o disco e que a interrupção no serviço seja grande no caso de quedas.

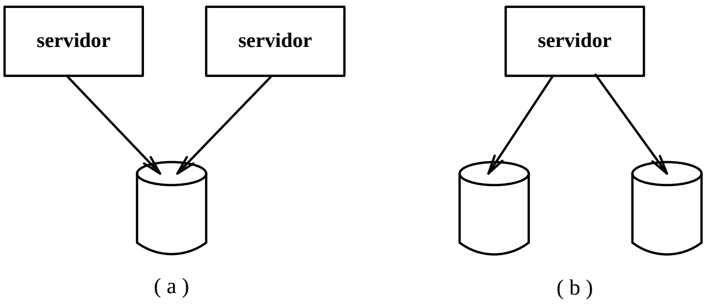



Figura 2.17: Replicação de servidores e discos

A fim de garantir a integridade dos dados armazenados, o ECHO permite que dois ou mais discos sejam associados a um mesmo servidor (figura 2.17b) de modo que as alterações sejam enviadas paralelamente aos dois discos. Isto faz com que defeitos em um dos discos não causem a perda de dados.

A solução empregada pelo ECHO foi unir estas duas técnicas como mostra a figura 2.18.

A replicação é administrada através do modelo de cópia primária com votação e testemunha [SBHM93]. É, portanto, um modelo semelhante ao do HARP (descrito na seção 2.8.1) mas, um pouco mais fraco na medida em que a única função das testemunhas é participar das

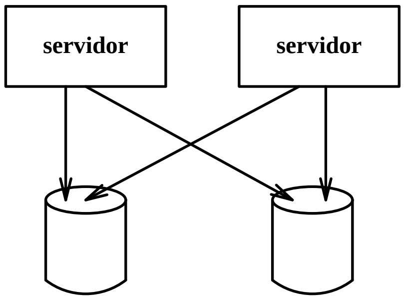



Figura 2.18: Replicação no ECHO

votações. Quando o servidor primário ou o secundário cai, a testemunha não assume o papel de *back-up* como ocorre no HARP.

### 2.10.2 Espaço de Nomes

Assim como o ANDREW, o ECHO foi projetado para atingir dimensões mundiais. A parte esquerda do *pathname* de um arquivo indica o endereço do nó Internet no qual ele se encontra e a parte direita, a sua localização lógica no sistema de arquivos daquele nó.

Considere, por exemplo, o *pathname* */-/com/dec/src/dir/arq*. O símbolo - indica que este é um *pathname* a partir da raiz global do espaço de nomes da Internet. Se este sinal é omitido, o *pathname* é considerado como relativo ao nó Internet local. A porção */com/dec/src* é resolvida por um servidor local de nomes globais que, para isso, se comunica com *Domain Name Service* da Internet. A porção restante */dir/arq* é tratada pelo nó Internet responsável pelo arquivo[47].

O espaço de nomes é idêntico em qualquer cliente. Ao contrário do NFS, onde os diretórios podem ser montados e desmontados de modo diferente em cada cliente, o ECHO constrói o seu espaço de nomes a partir de junções estáticas. Uma **junção** é uma lista de informações que associa um certo diretório do espaço de nomes global a um volume do sistema de arquivos e é implementada como um arquivo especial que contém a lista de informações.

Diferentemente do ANDREW e do SPRITE, podem existir várias classes de volumes com propriedades e semânticas de acesso distintas que são determinadas pelas informações contidas nas junções.

A fim de controlar a segurança em redes com um grande número de clientes, o ECHO adota um mecanismo de autenticação dos clientes por parte dos servidores. O acesso às informações confidenciais é protegido através de listas de controle de acesso.

### 2.10.3 Cache

O ECHO faz tudo o que faz o SPRITE em termos de cache e mais [MBH+93]. Além do cache de arquivos com escritas retardadas e garantia de consistência para acesso concorrente, o ECHO cacheia de modo consistente as leituras e escritas dos diretórios e dos atributos dos arquivos. Além disso, a quantidade de dados cacheados pelos clientes ECHO mostrou-se empiricamente maior do que no SPRITE. A maior parte dos clientes ECHO cacheavam mais de 16 Mbytes contra 7 Mbytes dos clientes SPRITE perdendo, no entanto, para os clientes AFS e CODA que costumam cachear mais de 100 Mbytes em seus discos locais.

[47] O desenvolvimento do ECHO foi interrompido antes que a interface com o *Domain Name Service* fosse efetivamente implementada.

O objetivo do ECHO em termos semânticos era oferecer uma visão do sistema de arquivos que fosse a mais próxima possível de um sistema centralizado. Por isso, foram descartados os métodos otimistas de replicação como os do CODA que permitem que as réplicas divirjam quando ocorrem partições na rede.

Como em um sistema centralizado, o ECHO mantém um arquivo que foi apagado até que todas as aplicações que tenham aberto este arquivo efetuem o *close* correspondente, coisa que o NFS não faz. Em oposição ao NFS e ao ANDREW, um cliente só permite que um bloco seja escrito se existir espaço disponível para ele nos discos do servidor.

Qualquer alteração efetuada no sistema de arquivos é agendada para ser enviada ao servidor 15 segundos após a sua solicitação. Como observamos em seções que tratavam de outros sistemas, espera-se que uma boa parte das alterações sejam logicamente desfeitas neste período. Como o ANDREW e o SPRITE não retardam as escritas de diretórios, a criação e destruição de arquivos no ECHO é bem mais rápida. Aplicações que criam e apagam muitos arquivos remotos apresentam um melhor desempenho. A simples operação de listar o conteúdo de um diretório é efetuada mais rapidamente pelo ECHO do que pelo SPRITE que simplesmente não cacheia diretórios.

A consistência dos arquivos, diretórios e atributos é garantida através de um mecanismo muito eficiente chamado *lease* que estudaremos em detalhes no capítulo 4.

As informações que garantem a consistência dos caches são replicadas na memória principal dos servidores primário e secundário de cada volume. Assim, quando o servidor primário cai, o secundário pode assumir o seu papel quase que imediatamente sem colocar em risco a consistência do sistema de arquivos.

Mas, o grande problema enfrentado pelo ECHO é a ocorrência de falhas que levem a perda de informações que foram escritas pelas aplicações mas que não atingiram o disco dos servidores.

A fim de limitar estas perdas, se algum bloco cuja escrita foi solicitada há mais de 5 minutos ainda não atingiu o disco do servidor, então o cliente bloqueia as escritas subseqüentes ao mesmo volume até que a situação se normalize.

Além disso, se o cliente não obtiver resposta de um servidor, as aplicações que solicitarem acesso a volumes deste servidor são bloqueadas. Se este bloqueio dura mais de dois minutos, as solicitações bloqueadas retornam com um código de erro. Este prazo foi estabelecido pois, no ambiente onde o ECHO foi instalado, dois minutos eram suficientes para que os mecanismos automáticos de tolerância a falhas entrassem em ação. Se a comunicação não era restabelecida em dois minutos, o melhor seria devolver um código de erro pois o problema provavelmente duraria muito tempo.

Estes cuidados limitam mas não evitam a perda de informações. Considere uma aplicação que crie vários arquivos, escreva vários Mbytes de informação e que termine sem receber nenhuma mensagem de erro. Se, logo após, o cliente cair abruptamente ou se a rede sofrer uma partição pode-se perder toda a informação comprometendo a integridade das estruturas de dados mantidas no sistema de arquivos pela aplicação.

Há ainda dois mecanismos que visam minimizar este tipo de perda. O primeiro é o comando *fsync(arq)* que, quando retorna um valor indicando sucesso, garante que todas as alteração efetuadas em *arq* atingiram os discos dos servidores. A desvantagem do *fsync* é que a aplicação tem que esperar que os dados sejam escritos nos discos dos servidores o que costuma ser relativamente demorado.

O outro mecanismo provém do fato de que o ECHO especifica uma ordem parcial nos *writes* e garante que eles atingirão o disco segundo esta ordem parcial. Além disso, através do novo comando *forder(arq)* é possível restringir ainda mais esta ordem parcial. Ao contrário do *fsync*, o *forder* retorna imediatamente mas garante que todas as alterações solicitadas antes da sua execução atingirão o disco antes de qualquer solicitação posterior ao *forder*.

Estas garantias possibilitam o desenvolvimento de aplicações que mantenham a consistência das suas estruturas de dados armazenadas no sistema de arquivos mesmo quando escritas retardadas são perdidas. As aplicações têm condições de garantir que mesmo que ocorram perdas de escritas, as suas estruturas de dados permanecerão consistentes e será possível reconstruir os dados perdidos.

Este método, baseado na ordem de chegada das alterações nos discos, é muito eficiente pois dispensa o acesso síncrono aos discos. Mas, por outro lado, é um método de difícil utilização pois exige do programador um conhecimento detalhado dos mecanismos de ordenação utilizados pelo ECHO. Além disso, é um método existente apenas no ECHO, aplicações desenvolvidas a partir deste método não apresentariam o mesmo ganho em eficiência se fossem executadas em outros sistemas.

### 2.10.4 Desempenho

Apesar de toda a preocupação do projeto do ECHO em oferecer uma boa disponibilidade, a maior queixa dos usuários do ECHO fui justamente a baixa disponibilidade [SBHM93]. Segundo os pesquisadores da DEC, este fato não se deve ao projeto do ECHO mas sim à sua implementação.

A baixa qualidade do hardware no qual o ECHO foi implementado e a fraqueza do código, que muitas vezes provinha de ferramentas que geravam código não muito eficiente, fez com que o sistema se sobrecarregasse facilmente e que os servidor sofressem quedas freqüentes.

Por outro lado, a integridade dos dados foi preservada nos 30 meses durante os quais o ECHO esteve em operação apesar das mais de 10 falhas físicas ocorridas nos discos neste período. Sempre foi possível recuperar as informações a partir do disco *back-up*.

Apesar de não termos estudos comparando o desempenho do ECHO a outros sistemas, podemos ter uma idéia dos ganhos que o *write-behind* de diretórios pode oferecer a algumas aplicações através das tabelas 2.13 e 2.14 extraídas de [MBH+93].

| Política das escritas | Compilação | | Andrew *make* | | Vesta *benchmark* | |
|---|---|---|---|---|---|---|
| | tempo médio | tempo relativo | tempo médio | tempo relativo | tempo médio | tempo relativo |
| ECHO | 240 | 1.00 | 546 | 1.00 | 9.9 | 1.00 |
| SPRITE | 341 | 1.42 | 566 | 1.04 | 24.1 | 2.43 |
| ANDREW | 366 | 1.52 | 592 | 1.08 | 25.1 | 2.54 |

Os tempos (em segundos) se referem à média de cerca de 20 execuções.

Tabela 2.13: Desempenho do *write-behind* de diretórios

A tabela 2.13 mostra a execução de três testes sob três políticas distintas de tratamento das escritas. Na linha rotulada ECHO, o *write-behind* é completo, ou seja, é realizado tanto nos diretórios quanto nos arquivos. Na linha rotulada SPRITE, o cliente ECHO foi modificado para enviar as alterações nos diretórios imediatamente para o servidor (*write-through* de diretórios). Já na linha rotulada ANDREW, além de fazer *write-through* de diretórios, o cliente escreve as alterações nos arquivos no momento do seu fechamento, ou seja, adota a política de *write-back-on-close* para o conteúdo dos arquivos.

O primeiro teste (*benchmark*) consiste na compilação de 100 programas em C contendo uma única linha com a definição de uma função. A única operação de alteração de diretórios neste *benchmark* é a criação dos 100 arquivos objeto resultantes das compilações. A política do ANDREW exigiu um tempo 52% maior do que a política do ECHO para compilar os 100 arquivos. Já o resultado na linha rotulada SPRITE é, na verdade, um limite superior para o desempenho do SPRITE uma vez que, ao contrário do cliente ECHO modificado, os clientes SPRITE não cacheiam os diretórios. Assim, o SPRITE seria, na melhor das hipóteses, 42% mais lento do que o ECHO neste teste.

O segundo teste corresponde a uma das fases do Andrew *benchmark* que consiste em executar o comando *make* para o código fonte de uma das versões do AFS. Neste *benchmark*, o *write-behind* dos diretórios apresentou um ganho muito pequeno que talvez não justificasse a sua implementação.

Já o terceiro *benchmark* mede o tempo necessário para que o VESTA, um administrador de pacotes de software, instale uma nova versão de um programa em um certo repositório. O VESTA é um exemplo de aplicação que foi desenvolvida utilizando pesadamente as garantias de ordenação das escritas retardadas e, como era de se esperar, o seu desempenho sob a política do ECHO é muito superior.

Finalmente, a tabela 2.14 mostra como as operações com diretórios são mais eficientes no ECHO do que em um sistema de arquivos UNIX local.

| Operação | UNIX local | ECHO |
|---|---|---|
| Criar arquivo | 4.95 | 2.94 |
| Apagar arquivo | 1.72 | 3.28 |
| Criar diretório | 18.6 | 2.17 |
| Apagar diretório | 7.21 | 1.53 |

Tempo necessário (em segundos) para a execução de 100 operações.

Tabela 2.14: Desempenho do ECHO frente ao UNIX local

### 2.10.5 Resumo

O ECHO oferece um serviço muito eficiente através da grande utilização do cache nos clientes tanto para leituras quanto para escritas de arquivos e de diretórios. A consistência entre os caches dos clientes é mantida através de um mecanismo eficiente denominado *leases*.

A tolerância a falhas é garantida através da replicação dos discos de um servidor e da possibilidade de mais de um servidor acessar o mesmo disco.

Um serviço não implementado mas integrante do projeto do ECHO permitiria o compartilhamento de arquivos a longa distância através da Internet. Além disso, o ECHO permitiria a adoção de protocolos diferentes em diretórios diferentes possibilitando uma melhor utilização do sistema.

Depois de pouco mais de um ano de funcionamento, o projeto do ECHO foi abandonado no momento em que o projeto de transportar o sistema operacional TAOS para uma plataforma mais moderna foi cancelado.

## 2.11 Resumo Comparativo

Apresentamos na tabela seguinte um resumo das principais características dos sistemas estudados. Cada linha aborda um aspecto descrito no capítulo 1 e cada coluna um dos sistemas do capítulo 2.

Para maiores detalhes sobre o conteúdo de cada célula da tabela consulte a seção correspondente.

| | SUN NFS | SPRITE | ZEBRA | ANDREW | CODA | FROLIC | ECHO | HARP |
|---|---|---|---|---|---|---|---|---|
| Localização do CACHE dos clientes | **Memória Principal** | **Memória Principal** | **Memória Principal** | **Disco Local** | **Disco Local** | **Memória Principal** | **Memória Principal** | **Memória Principal** |
| REPLICAÇÃO | **FRACA. Somente para diretórios *read-only*.** | **FRACA. Somente para diretórios *read-only*.** | **FRACA. Somente para diretórios *read-only*.** | **FRACA. Somente para diretórios *read-only*.** | **BOA. Parte da sobrecarga é distribuída pelos clientes** | **BOA. Replicação dinâmica baseada em *clusters*.** | **ÓTIMA. Resiste a quedas de servidores e a falhas em discos.** | **BOA. Resiste a quedas de servidores.** |
| CONSISTÊNCIA | **FRACA. Acesso concorrente gera resultados imprevisí-veis.** | **ÓTIMA. Semântica UNIX perfeita.** | **ÓTIMA. Semântica UNIX perfeita.** | **RAZOÁVEL. Semân-tica de sessão.** | **FRACA. Semântica de sessão enfraquecida pela replicação otimista.** | **FRACA. Semântica de sessão no acesso *inter-cluster* e NFS no acesso *intra-cluster*.** | **BOA. Utiliza *leases* para garantir semântica UNIX.** | **FRACA. Utiliza o NFS para a comunicação cliente/servidor.** |
| ESCALABILIDADE | **FRACA.Os servidores se saturam rapidamente.** | **FRACA. O protocolo faz uso de *broadcasts*.** | **RAZOÁVEL. Divide o serviço equilibrada-mente entre os servidores.** | **ÓTIMA. Ideal para redes de grande escala com pouco comparti-lhamento.** | **ÓTIMA. Ideal para redes de grande escala com pouco comparti-lhamento.** | **ÓTIMA. Mesmo em redes de grande escala com alto grau de compartilhamento.** | **RAZOÁVEL. Interage com o serviço de nomes da Internet.** | **FRACA. Os servidores recebem mais carga do que no NFS.** |
| RAPIDEZ | **FRACA. Protocolo pouco eficiente.** | **BOA. Implementação no nível do *kernel*. Escritas retardadas do conteúdo dos arquivos.** | **ÓTIMA. Listramento de arquivos, sistema de arquivos físico baseado em *log*.** | **RAZOÁVEL. Latência grande no acesso a arquivos não cacheados.** | **BOA. Busca a réplica mais próxima de cada diretório.** | **BOA. Os clientes se comunicam apenas com os servidores locais.** | **ÓTIMA. Cache com escritas retardadas para todos os tipos de informação.** | **FRACA.** |
| PERSISTÊNCIA dos dados na ocorrência de falhas | **FRACA.** | **FRACA. Escritas retardadas podem causar perda de informação.** | **BOA. Danos em um dos discos da listra não causam perda de informação.** | **RAZOÁVEL. *Back-up* automático.** | **BOA.** | **BOA.** | **BOA.** | **ÓTIMA.** |
| DISPONIBILIDADE | **FRACA.** | **RAZOÁVEL. Rápida recuperação de quedas.** | **BOA. Tolera quedas isoladas de clientes e de servidores. Recupera-ção de quedas muito rápida.** | **FRACA.** | **ÓTIMA. Replicação otimista. Operação desconectada.** | **BOA.** | **Teoricamente ÓTIMA. A implementação ineficiente trouxe problemas.** | **BOA.** |
| SEGURANÇA | **FRACA.** | **FRACA. Servidores confiam nos clientes.** | **FRACA. Servidores confiam nos clientes.** | **BOA. Listas de controle de acesso. Autenticação mútua criptografada entre clientes e servidores.** | **BOA. Listas de controle de acesso. Autenticação mútua criptografada entre clientes e servidores.** | **FRACA.** | **BOA. Listas de controle de acesso. Autenticação criptografada.** | **FRACA.** |
| PLATAFORMA na qual já foi implementado | **Todas relevantes.** | **SPARCstations e DECstations.** | **DECstation 5000 modelo 200.** | **Estações de trabalho Silicon Graphics, HP, NEXT, DEC, IBM, SUN e VAX.** | **IBM RT, DECstations 3100 e 5000 e *laptops* 386/486.** | **Protótipo desenvolvido sobre o NFS em SPARCstations.** | **FIREFLIES (multiprocessador experimental VAX).** | **Protótipo desenvolvido sobre o NFS em MicroVAX 3500.** |
| Características especiais de HARDWARE | **Permite clientes sem disco.** | **Permite clientes sem disco.** | **Permite clientes sem disco.** | | **Facilita a utilização de computadores portáteis. Replicação exige discos e servidores extras.** | **Replicação exige discos e servidores extras.** | **Utiliza placa especial para replicar servidores de um mesmo disco. Replicação exige discos e servidores extras.** | **Exige UPS. Replicação exige discos e servidores extras.** |

# Capítulo 3

# Modelos Analíticos

Vimos no capítulo anterior que cada sistema de arquivos distribuído funciona melhor em determinada situação. O SPRITE funciona bem quando a memória principal dos clientes e servidores é grande e quando a comunicação entre as máquinas é rápida. A sua escalabilidade não parece ser suficientemente boa para se adaptar a redes com mais de algumas dezenas de servidores e centenas de clientes.

Já o ANDREW foi desenvolvido para redes de grande área. Ele não é tão rápido quanto o SPRITE mas se comporta muito bem quando o número de clientes atinge alguns milhares e o número de servidores, algumas centenas.

O FROLIC foi desenvolvido com o objetivo de diminuir a carga causada por métodos como os do ANDREW em redes de grande porte com alto grau de compartilhamento de arquivos.

As diferenças entre o desempenho dos sistemas em diferentes ambientes dificultam a escolha do melhor sistema para uma determinada rede. Duas possíveis soluções podem ser adotadas para este problema.

A primeira seria o desenvolvimento de sistemas que pudessem ser configurados de acordo com a vontade do seu administrador. Através de alguns comandos, o administrador poderia determinar a utilização do melhor algoritmo para o tratamento de cada grupo de arquivos da sua rede. Além disso, ele poderia fornecer dados como o número de clientes, servidores e usuários, a velocidade das máquinas e das conexões entre elas de modo que o sistema otimizasse o seu desempenho.

A outra possibilidade é mais difícil de ser implementada mas facilitaria o trabalho dos administradores e poderia oferecer uma maior eficiência. Ela consiste no desenvolvimento de sistemas adaptativos que modificariam o seu modo de funcionamento de acordo com a utilização do sistema de arquivos. Tais sistemas monitorariam o acesso aos arquivos e, aplicando os dados coletados a modelos analíticos previamente estabelecidos, modificariam os parâmetros do sistema e, se necessário, utilizariam um novo algoritmo de modo a melhorar o desempenho do sistema. Estes sistemas se adaptariam dinamicamente ao comportamento das redes sempre oferecendo o serviço da maneira mais eficiente possível.

No entanto, alguns fatores impossibilitaram a sua implementação até o presente momento. A necessidade de monitorar a utilização da rede e os cálculos necessários para determinar quais devem ser as modificações no comportamento do sistema implicam em uma sobrecarga que pode causar um efeito contrário ao que se espera. Ou seja, um sistema adaptativo pode apresentar um desempenho mais fraco do que um sistema tradicional devido à alta sobrecarga produzida pelo monitoramento da utilização do serviço.

Há uma grande dificuldade na formulação de modelos analíticos que quantifiquem precisamente o desempenho de um sistema distribuído e na aplicação destes modelos em sistemas adaptativos de modo a efetivamente melhorar o seu desempenho.

Apresentaremos, neste capítulo, duas tentativas de modelagem e otimização do desempenho de sistemas de arquivos replicados. As descrições dos modelos aqui apresentadas divergem ligeiramente das descrições apresentadas nos artigos originais com o intuito de facilitar a sua compreensão.

No capítulo seguinte, apresentaremos um modelo analítico que estima o tráfego de mensagens geradas por um mecanismo que controla a consistência do cache dos clientes em um sistema de arquivos distribuído.

## 3.1 Modelo de Borghoff

Em [Bor92], Uwe Borghoff, da Universidade Técnica de Munique, propõe um sistema adaptativo para redes compostas por *clusters*. Cada arquivo possui cópias em um ou mais *clusters* e a decisão de qual *cluster* deve possuir uma cópia de cada arquivo é tomada a partir de um modelo que visa diminuir o tempo de execução dos programas.

O modelo quantifica o tráfego gerado pela execução dos programas dos usuários em função de em quais *clusters* cada arquivo é replicado. Os servidores e clientes não são tratados individualmente, o modelo não se preocupa com o funcionamento interno dos *clusters*. Deste modo, consegue-se diminuir o tamanho do problema a uma dimensão na qual o modelo pode ser empregado. Não adianta analisar detalhadamente o comportamento de cada máquina se, para isso, for necessário realizar uma quantidade de cálculos tão grande que, na prática, inviabilizaria a sua utilização.

A figura 3.1 mostra o tipo de rede para a qual o modelo foi desenvolvido. O número mínimo de *clusters* onde um arquivo deve ser replicado é um parâmetro a ser definido pelo administrador. O modelo diferencia arquivos de programas de arquivos de dados. A consistência entre as réplicas dos arquivos de dados é mantida através de votações. Cada leitura ou escrita exige a aprovação de 50% + 1 *clusters*. Os arquivos contendo programas são considerados imutáveis e não há a necessidade de obter um quórum para acessá-los.

Dois mecanismos básicos compõem o modelo:

1. O *Individual Program Execution Location Algorithm* (IPELA) determina qual é a melhor localização para a execução de um determinado programa. É consultado sempre que um usuário ativa um novo programa e procura estimar qual seria o tempo gasto com o tráfego *inter-cluster* se ele fosse executado em cada uma dos *clusters* da rede. O *cluster* cuja previsão do tempo gasto for menor é o escolhido.

2. O *Centralized File Allocation Process* (CFAP) é executado periodicamente e determina uma localização das réplicas dos arquivos visando minimizar o tempo gasto na troca de informações entre diferentes *clusters*.

O modelo trabalha com matrizes booleanas que determinam configurações do sistema de arquivos. Suas linhas são indexadas pelos arquivos e as colunas pelos *clusters*; uma entrada vale 1 se o arquivo correspondente à linha possui uma réplica no *cluster* correspondente à coluna e 0 caso contrário.

### 3.1.1 IPELA

Quando um programa é executado, quatro tipos de tráfego *inter-cluster* podem ser gerados:

- Se não há replicas do arquivo executável no *cluster* local, ele deve ser trazido de outro *cluster*.

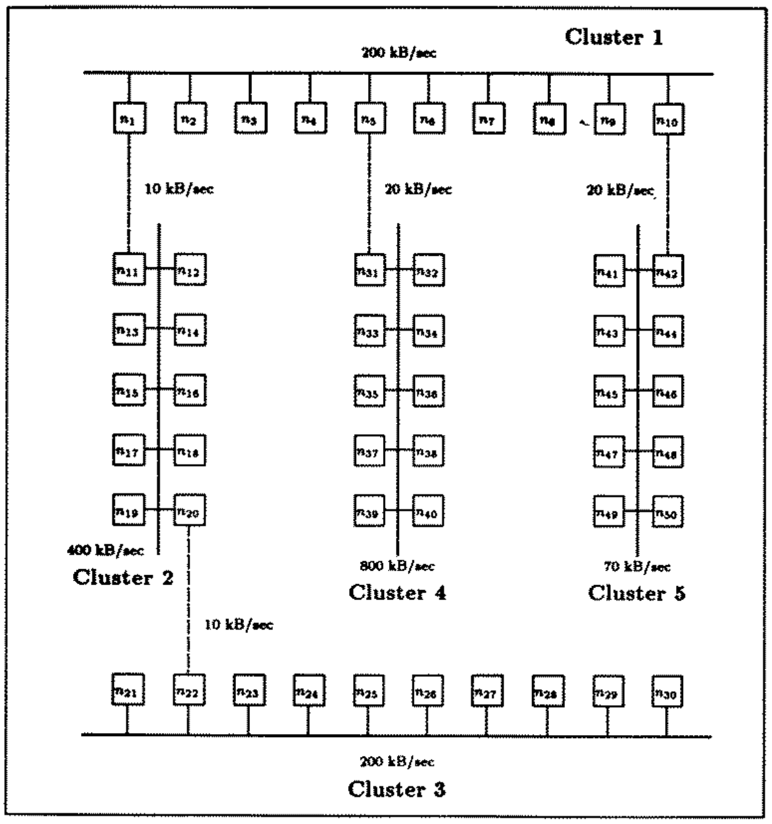



Figura 3.1: Uma Rede com 5 *Clusters*

- Se os arquivos de dados acessados pelo programa possuirem cópias em outros *clusters*, há a necessidade de realizar votações *inter-cluster*. Se a cópia local for obsoleta, é preciso trazer uma versão atualizada.

- Se o programa realizar alterações em um arquivo que possuir cópias em outros *clusters*, eles devem ser avisados destas alterações.

- Se o IPELA determina que um programa interativo deve ser executado em um *cluster* remoto, as informações trocadas entre o usuário e o programa devem trafegar entre os dois *clusters*.

### Monitoramento

Todas as execuções de programas são monitoradas pelo sistema de modo a manter uma estrutura de dados com informações sobre o comportamento de todos os programas. A tabela 3.1 mostra o **perfil médio de programa** de um programa $p$, ou seja, as informações que o sistema mantém sobre este programa.

### Custo da Execução

A $taxa(c_i, c_j)$ é a capacidade do canal de comunicação (em blocos por unidade de tempo) entre os *clusters* $c_i$ e $c_j$.

| |
|---|
| $input(p)$ - número de blocos lidos do console do usuário |
| $output(p)$ - número de blocos escritos no console do usuário |
| $read(p, d)$ - número de blocos lidos do arquivo $d$ |
| $write(p, d)$ - número de blocos escritos no arquivo $d$ |

Estes dados são uma média das ativações recentes do programa. O termo console designa a máquina com a qual o usuário interage fisicamente.

Tabela 3.1: Perfil médio dos programas

O custo da execução de um programa é a soma do custo da sua inicialização, das leituras, das escritas e da comunicação com o usuário:

- Se o arquivo contendo o programa estiver presente no *cluster* local, não há custo na inicialização. Caso contrário, o custo na inicialização de um programa $p$ no *cluster* $c_x$ (*cluster* executor) em um determinado período $t$ é:

  $$INIC_t(p, c_x) = \frac{BP(p) + BP}{taxa(c_x, c_p)}$$

  onde $c_p$ é o *cluster* que pode fornecer o código do programa com a melhor taxa de transferência possível, $BP(p)$ é o número de blocos do arquivo contendo o programa $p$ e $BP$ é o número de blocos extras que devem ser trocados entre os *clusters* $c_x$ e $c_p$.

  O tempo gasto na determinação do melhor *cluster* para efetuar a transferência é $O(|C|)$, onde $C$ é o conjunto de *clusters*.

- Os blocos lidos pelo programa são trazidos do *cluster* com o qual a comunicação for a mais rápida possível ($c_d$). É necessário trocar $BQ$ blocos com outros *clusters* a fim de obter o quórum para leitura. Assim, um limite superior para o custo das leituras efetuadas por uma execução completa de um programa no período $t$ é:

  $$READ_t(p) = \sum_{d \in D} read(p, d) \times \left( \frac{BQ}{taxa(c_x, c_q)} + \frac{BR}{taxa(c_x, c_d)} \right)$$

  onde $D$ é o conjunto de arquivos de dados do sistema, $c_q$ é o *cluster* cuja taxa de comunicação com $c_x$ é a menor possível dentre os *clusters* que participam da votação para o acesso ao arquivo $d$ e BR é o número de blocos necessários para efetuar uma leitura incluindo a sobrecarga do protocolo.

  O modelo utiliza o *cluster* com o qual a comunicação é a mais lenta possível ($c_q$) pois para se conseguir o quórum pode ser necessário esperar pela resposta de todos os *clusters* que possuem uma cópia do arquivo.

  Complexidade do cálculo: $O(|D||C|)$.

- Já o custo das escritas leva em conta que as alterações devem ser enviadas para todos os *clusters* que possuem uma cópia do arquivo; o modelo supõe que isso é feito através de uma mensagem com vários destinatários (*multicast*). Portanto, o custo das escritas é calculado a partir da capacidade de transferência de dados entre $c_x$ e $c_q$:

  $$WRITE_t(p) = \sum_{d \in D} write(p, d) \times \frac{BQ + BW}{taxa(c_x, c_q)}$$

BW é o número de blocos necessários para efetuar uma escrita incluindo a sobrecarga do protocolo.

Complexidade do cálculo: $O(|D||C|)$.

- Se o programa é executado em um *cluster* diferente daquele onde se encontra o console do usuário, então há o custo adicional na transferência *inter-cluster* de dados entre a aplicação e o usuário:

  $$IO_t(p) = \frac{input(p) + output(p)}{taxa(c_i, c_x)}$$

  $c_i$ é o *cluster* que inicia o programa, isto é, o *cluster* de onde o usuário o ativou.

  Complexidade do cálculo: $O(1)$.

Quando a execução de um programa é solicitada por um usuário, o sistema calcula o valor da função

$$IPELA_t(p) = READ_t(p) + WRITE_t(p) + IO_t(p)$$

para todos os possíveis valores de $c_x$. O *cluster* que minimizar esta função é chamado $c_x^*$ e é o *cluster* onde o programa será executado.

A complexidade total envolvida na determinação de $c_x^*$, isto é, a complexidade do IPELA, é

$$O(|D||C|^2).$$

Após a determinação do *cluster* que executará o programa, é necessário escolher qual dos nós deste *cluster* será o responsável pela tarefa. Isto é feito através de um método muito simples. O nó que minimizar a razão $\frac{\text{número de programas ativos sendo executados no nó}}{\text{capacidade de processamento do nó}}$ é o escolhido.

### 3.1.2 CFAP

O papel do CFAP é solucionar um problema clássico da teoria de sistemas distribuídos, o *File Alocation Problem* (FAP) [RW83].

#### Monitoramento

O CFAP monitora as solicitações de execução de programas e armazena, para cada programa do sistema, as informações descritas na tabela 3.2 formando o **perfil da solicitação de execução de programas**.

| |
|---|
| $CS(p)$ - *clusters* que solicitam a execução de $p$ |
| $FREQ(c, p)$ - número de vezes que o *cluster* $c$ solicitou a execução de $p$ |

Dados coletados durante um determinado intervalo de monitoramento.

Tabela 3.2: Perfil da solicitação de execução de programas

Após o término de um período de monitoramento, o CFAP calcula, a partir dos dados coletados no período, uma nova localização das réplicas dos arquivos visando diminuir o tempo gasto com a comunicação entre os *clusters*. Os arquivos são remanejados de modo a obedecer a esta nova localização.

A fim de diminuir o tamanho do problema, o CFAP só trabalha com os programas que foram executados no período anterior e com arquivos de dados que foram acessados no período anterior. Chamaremos de $P_t$ os programas cuja execução foi solicitada durante o período $t$ (isto é, os programas relevantes ao CFAP logo após o período $t$) e de $D_t(p)$ os arquivos de dados acessados por $p$ no período $t$. $D_t$ será

$$\bigcup_{p \in P_t} D_t(p)$$

ou, o conjunto de arquivos relevantes ao CFAP após $t$.

### O custo de uma configuração

O custo de uma dada configuração é igual ao custo de executar todos os programas relevantes nos *clusters* nos quais a sua execução foi solicitada no período anterior e é dado pela função

$$CFAP_t = \sum_{p \in P_t} \sum_{c \in CS(p)} FREQ(c,p) \times IPELA_t(p)^*$$

onde $IPELA_t(p)^*$ é o valor da função $IPELA_t(p)$ no *cluster* ótimo $c_x^*$. Neste caso o IPELA é executado considerando apenas os arquivos relevantes. O cálculo desta função para uma determinada matriz de configuração exige tempo $O(|P_t||D_t||C|^3)$ [1].

O trabalho do CFAP é descobrir uma configuração que minimize $CFAP_t$ mas que obedeça a três restrições:

1. **Número mínimo de réplicas**

   O modelo permite que tanto o administrador quanto os usuários do sistema especifiquem um número mínimo de réplicas ($min_{repl}(arq)$) para cada arquivo de dados e de programa. Para verificar se uma configuração obedece a esta restrição o CFAP gasta um tempo $O(|C|(|P_t| + |D_t|))$.

2. **Disponibilidade**

   O administrador pode ainda especificar uma disponibilidade mínima ($disp_{min}$) para a execução de programas no sistema. Esta disponibilidade é calculada como o produto das probabilidades $PROB_p$ e $PROB_d$ descritas a seguir:

   (a) $PROB_p(p, c_i)$: A probabilidade de conseguir carregar o programa $p$ a ser executado a partir do *cluster* $c_i$. Para isso pelo menos um *cluster* que possua $p$ deve estar acessível[2]

   (b) $PROB_d(p, c_i)$: A probabilidade de conseguir acesso a todos os arquivos de dados lidos ou alterados pelo programa. Para o acesso a um determinado arquivo deve se conseguir um quórum de 50% + 1 dos *clusters* que o possuem.

   A fim de estimar os valores de $PROB_p$ e $PROB_d$, o modelo utiliza as freqüências dos acessos às réplicas de cada um dos arquivos descritas no perfil das solicitações de execuções de programas.

   No entanto, [Bor92] não especifica como a disponibilidade dos *clusters* deve ser calculada. Uma possibilidade é a manutenção de um perfil de disponibilidade. Através de testes

[1] Lembre-se que a complexidade do IPELA é $O(|D||C|^2)$.

[2] O modelo não exige votações para o acesso aos programas.

periódicos da conexão entre os *clusters* poder-se-ia estimar a disponibilidade do acesso a arquivos localizados em outros *clusters*.

Cada *cluster* poderia ser responsável pela manutenção de uma certa disponibilidade dos arquivos que mantém. Se os servidores de um certo *cluster* sofrem quedas freqüentes e duradouras, este *cluster* deveria replicar os arquivos internamente a fim de oferecer uma boa disponibilidade para os outros *clusters*.

A restrição de disponibilidade do CFAP pode ser escrita como

$$\forall p \in P_t \ \ \forall c_i \in CS(p) : DISP_t(p, c_i) > disp_{min}$$

onde $DISP_t(p, c_i)$ é a probabilidade de se conseguir executar, com sucesso, o programa $p$ a partir do *cluster* $c_i$, isto é, $PROB_p(p, c_i) \times PROB_d(p, c_i)$. O seu cálculo exige tempo $O(|D_t||C|)$.

Portanto, o cálculo da restrição de disponibilidade exige tempo $O(|P_t||D_t||C|^2)$.

3. **Capacidade de Armazenamento**

   Obviamente, deve-se tomar cuidado para que a capacidade de armazenamento dos discos dos servidores não seja ultrapassada. Uma solução viável do FAP deve obedecer à restrição de que a soma do número de blocos dos arquivos armazenados em cada *cluster* não deve ultrapassar a capacidade de armazenamento de todos os seus discos somados[3].

   Este é um dos pontos fracos do modelo. A opção de não tratar detalhadamente o que acontece dentro de um *cluster* pode fazer com que o CFAP apresente uma solução impossível de ser posta em prática. Esta restrição é verificada em tempo $O((|P| + |D|)|C|)$.

   Suponha, por exemplo, que um *cluster* contenha 10 servidores e que cada um destes servidores possua 1 Mbyte de espaço livre em disco. Os cálculos do CFAP podem levar a uma configuração que implique na criação de uma réplica de um determinado arquivo de 5Mbytes neste *cluster*. A restrição do modelo vai considerar esta solução viável pois o *cluster* possui 10 Mbytes livres em disco. Mas, a menos que o arquivo seja quebrado em pedaços menores do que 1Mbyte, não haverá servidor que o comporte.

### Complexidade total do CFAP

O número de configurações possíveis para o sistema de arquivos é enorme. Como o FAP é um problema $NP$-completo ([GJ79] problema SR6 e [RW83]), não se conhece nenhum algoritmo determinístico eficiente para o cálculo da melhor configuração. A idéia apresentada em [Bor92] é de restringir ao máximo o número de configurações e, a partir daí, calcular o custo de cada uma delas até que um prazo pré-determinado se esgote. Quando o prazo termina, a melhor configuração até o momento é a solução adotada pelo CFAP que cria ou apaga réplicas de modo que o sistema de arquivos reproduza a solução encontrada.

Para restringir o número de configurações possíveis, o modelo assume inicialmente que o número de réplicas de cada arquivo deve ser igual ao número mínimo de réplicas. Assim, não há a necessidade de verificar a satisfabilidade da primeira restrição. Só no caso em que a restrição de disponibilidade não é satisfeita por nenhuma configuração viável é que o número de réplicas pode ser maior que o mínimo.

Só são consideradas configurações onde há alterações apenas nos arquivos relevantes. Espera-se que $|P_t| \ll |P|$ e que $|D_t| \ll |D|$.

[3] Note a semelhança com o problema tradicional de *bin packing* (problema SR1 de [GJ79]).

Com tudo isso, o número de configurações possíveis após um período $t$ é

$$\prod_{arq \in P_t \cup D_t} \binom{|C|}{min_{repl}(arq)}$$

e a complexidade total do CFAP é:

$$\mathbf{O}\left(\binom{|C|}{REP}^{|P_t \cup D_t|} (|P_t||D_t||C|^3 + (|P| + |D|)|C|)\right)$$

onde

$$REP = \max_{arq \in P_t \cup D_t} min_{repl}(arq).$$

### 3.1.3 Simulação

Apesar do modelo não ter sido implementado em um sistema real, foi construído um simulador para analisar o seu comportamento. Quatro estratégias foram comparadas:

1. Todo programa é executado na máquina que solicita a sua execução. A disposição dos arquivos é fixa.
2. Todo programa é executado no *cluster* que solicita a sua execução. Dentro de um *cluster*, a máquina que estiver com a menor carga é a escolhida para executar o programa. A localização dos arquivos é fixa.
3. Como na estratégia anterior mas o programa é executado no *cluster* ótimo determinado pelo IPELA.
4. Os programas são executados no *cluster* ótimo e a localização dos arquivos é determinada pelo CFAP.

#### Parâmetros da Simulação

A rede, composta por cinco *clusters*, é a representada pela figura 3.1. Cada um dos 50 nós da rede é tanto cliente quanto servidor. Os servidores caem em média uma vez por semana e demoram 10 minutos para restabelecerem o seu serviço. As conexões entre os *clusters* falham uma vez a cada 10 minutos interrompendo a ligação por 10 segundos. As conexões internas de um *cluster* não falham.

A simulação considera ainda a existência de 3 arquivos de dados de 30, 300 e 3000 blocos e de 30 programas separados em três grupos. O primeiro grupo realiza apenas interação com o console do usuário, o segundo apenas acesso a arquivos e o terceiro realiza as duas atividades. Cada um dos 30 programas apresenta um padrão de acesso distinto.

A simulação englobou 30 dias simulados. Na estratégia 4, o CFAP era ativado sempre que $|P_t|$ passava de 10. Cada execução do CFAP analisava até 45.000 configurações gastando, para isso, uma hora da CPU de uma *DEC VAXstation II RC*.

**Resultados**

Foram estudados 4 cenários:

A Cada arquivo possui 3 réplicas.

A.1 Cada programa é executado de 1 hora em 1 hora.

A.2 Cada programa é executado de 15 em 15 minutos.

B Os arquivos não são replicados. Inicialmente, todos os arquivos de dados são armazenados no servidor $n_{21}$ e todos os programas são armazenados em $n_{43}$.

B.1 Cada programa é executado de 1 hora em 1 hora.

B.2 Cada programa é executado de 15 em 15 minutos.

Alguns dos resultados obtidos mostram que é preciso tomar muito cuidado ao implementar sistemas deste tipo. Na situação na qual os arquivos são replicados observou-se que a aplicação das estratégias 3 (IPELA) e 4 (IPELA + CFAP) retarda a execução dos programas do primeiro grupo que só realizam interações com o console do usuário. O tempo gasto com a determinação do melhor *cluster* para executar o programa é maior do que o tempo gasto na execução do programa. Além disso, a sobrecarga gerada pelo CFAP também atrapalha o desempenho dos programas fazendo com que a sua execução gaste, em alguns casos, mais do que o dobro do tempo.

Estas observações sugerem que um sistema que implemente este modelo deve levar em conta dois fatos. Programas cujas operações de entrada e saída são dominadas pela interface com o console do usuário devem ser executados sempre no *cluster* onde está o console. Sempre que possível, a reconfiguração do sistema comandada pelo CFAP deve ser empreendida em momentos de baixa carga na rede. As madrugadas seriam uma boa hora para isso.

Por outro lado, a aplicação do IPELA pela estratégia 3 levou a uma diminuição de cerca de 50% no tempo de execução dos programas do segundo e terceiro grupo no cenário sem replicação (B) e cerca de 8% no cenário com replicação (A). Isso mostra que mesmo a implementação apenas do IPELA (que é muito mais simples do que o CFAP) pode trazer ganhos significativos para o desempenho do sistema. É algo deste tipo que é feito em sistemas operacionais distribuídos como o AMOEBA [Tan92] onde o sistema é o responsável pela determinação das máquinas que executam os processos[4].

Na simulação, as solicitações de execução dos programas foram igualmente distribuídas pelos nós da rede o que fez com que a configuração determinada pelo CFAP no cenário A se estabilizasse após algumas reconfigurações. A tabela 3.3 mostra alguns dados sobre o CFAP.

No cenário A (com replicação), basta verificar as cerca de 1.000 configurações possíveis para se determinar a configuração ótima. Já no cenário B, não é possível analisar todas as possíveis configurações; o sistema coloca em prática a melhor solução encontrada após a verificação de 45.000 configurações o que demora 62 minutos. Após este período, uma média de 6,6 arquivos são transportados para outro *cluster* quando a carga na rede é menor (B.1) e uma média de 14,1 arquivos quando a carga na rede é maior (B.2).

No geral, a aplicação do CFAP foi vantajosa. A principal exceção foram os programas cujas operações de entrada e saída são dominadas pela interface com o console do usuário.

[4] Uma descrição detalhada de mecanismos para a distribuição da carga em sistemas distribuídos pode ser encontrada em [Mil94] que enfoca o sistema MACH.

| Cenário | Número de reconfigurações até a estabilização | Número de configurações testadas | Intervalo médio entre as reconfigurações | Número médio de realocações de arquivos |
|---|---|---|---|---|
| A.1 | 9 | 1.000 | 28 minutos | $\rightarrow 0$ |
| A.2 | 6 | 1.000 | 7 minutos | $\rightarrow 0$ |
| B.1 | não estabilizou | 45.000 | 62 minutos | 6.6 |
| B.2 | não estabilizou | 45.000 | 62 minutos | 14.1 |

Tabela 3.3: Resultados da simulação do CFAP

### 3.1.4 Críticas ao Modelo

Qualquer sistema que implemente este modelo gastará uma grande quantidade de espaço e de tempo para a manutenção das informações exigidas pelo IPELA e pelo CFAP. É necessário monitorar todas as execuções de programas bem como todos os acessos a arquivos e comunicações com os consoles a fim de manter os dados atualizados exigidos pelo modelo. A simulação não estudou o impacto que este monitoramento causa no desempenho do sistema.

A simulação mostra o comportamento da aplicação do modelo em uma rede altamente sub-utilizada. Considerar apenas a execução de 30 programas de 15 em 15 minutos em uma rede com 50 servidores é uma situação irreal. Também é irreal o fato de existirem apenas 3 arquivos de dados.

Um número mais próximo da realidade seria algo da ordem de centenas de milhares de arquivos distribuídos pelos 50 servidores e a execução de alguns programas por segundo.

A aplicação de uma busca exaustiva por todas as possíveis configurações em um sistema dessas proporções é algo impensável[5]. A complexidade exponencial do CFAP faz com que ele só possa ser aplicado em sistemas específicos onde o número de arquivos é pequeno. Mas, nestes sistemas, talvez o melhor fosse adotar uma política como a do FROLIC que cria dinamicamente uma réplica quando um *cluster* acessa um arquivo que não possui cópia local.

Ao exigir que informações sobre a configuração global do sistema estejam disponíveis em cada máquina da rede, limita-se também a sua escalabilidade.

O fato de o modelo tratar diferentemente arquivos de dados de arquivos contendo programas também é algo estranho. Só se considera necessário atingir um quórum mínimo quando se trata de acesso a arquivos de dados. Mas em sistemas de uso geral, não existe esta diferenciação. Um arquivo pode, inclusive, assumir os dois papeis em momentos diferentes. Um arquivo de programa também pode ser visto como um arquivo de dados de um compilador.

O modelo ignora alguns fatos que podem levar a resultados práticos diferentes das previsões teóricas. Além da transferência dos blocos dos arquivos, uma parte dos bytes que trafegam na rede de um sistema de arquivos distribuído se deve à tarefa de resolução de *pathnames*, criação, leitura e remoção de diretórios e outras operações relativas ao espaço de nomes que são ignorados pelo modelo.

O desempenho de um canal de comunicação *inter-cluster* piora quando a carga neste canal aumenta. Este fato não é considerado pelo modelo.

Finalmente, o modelo não considera o fato de que um mesmo programa pode apresentar padrões de acesso diferentes de acordo com o usuário que o executa. Se, por exemplo, um usuário executa um compilador muitas vezes em um determinado *cluster*, o CFAP irá transferir os arquivos de dados deste usuário para lá. Se, logo após, outro usuário solicita a execução

[5]Na seção 3.2.2 apresentaremos um algoritmo probabilístico que, sob certas condições, é capaz de localizar a solução ótima em tempo polinomial.

deste compilador em outro *cluster*, o IPELA determinará que ele seja executado remotamente mesmo que os arquivos que este usuário deseja compilar estejam no *cluster* local.

Veremos, agora, um modelo que utiliza um algoritmo mais eficiente do que a busca exaustiva para a determinação de uma configuração ótima para o sistema de arquivos.

## 3.2 Arquitetura de Cache Remoto

Analisaremos nesta seção o projeto de um sistema que utiliza uma arquitetura de cache diferente das convencionais. Através da aplicação de um modelo analítico espera-se obter uma configuração quase ótima para o sistema de arquivos através de um algoritmo eficiente.

Nas arquiteturas de cache convencionais cada cliente cacheia os arquivos relevantes para as suas aplicações. Assim, os programas mais utilizados em uma rede, costumam ser cacheados em todos os clientes simultaneamente. Em redes locais onde a comunicação entre duas máquinas é muito mais rápida do que o acesso aos discos, existe uma alternativa para estas arquiteturas.

Nos sistemas de cache remoto, a arquitetura não adota exatamente o modelo cliente/servidor tradicional. Quando uma aplicação de um cliente solicita a leitura de um arquivo, este é procurado inicialmente no cache local. Se ele não está cacheado localmente, o sistema envia uma mensagem para todas as máquinas da rede local (*broadcast*)[6] a fim de descobrir se alguma máquina possui esse arquivo em seu cache. Se alguma máquina o possuir, o arquivo é transferido para a máquina solicitante sem a necessidade de realizar os demorados acessos ao disco. A leitura do disco do servidor só é necessária se nenhuma máquina da rede possuir aquele arquivo em seu cache o que é detectado se nenhuma máquina responde ao *broadcast* dentro de um prazo pré-determinado.

Em sistemas como o SPRITE, quando um cliente não localiza um arquivo em seu cache local, há ainda mais uma esperança de conseguir ler o arquivo sem a necessidade de acessar o disco: o cache do servidor. A arquitetura de cache remoto pode ser vista como uma generalização deste procedimento na medida em que os caches de todas as máquinas da rede são consultados e não apenas o do servidor.

Descreveremos o modelo apresentado em [LYW93] por pesquisadores da IBM. Inicialmente, contrariando os sistemas convencionais, analisa-se o caso no qual o cache dos clientes não utiliza a política LRU para determinar quais arquivos devem ser cacheados em cada instante. Os clientes armazenam simplesmente um conjunto fixo de arquivos em seu cache em cada período. Quando um arquivo que não está presente no cache local é carregado, ele só é armazenado no cache se pertencer ao conjunto de arquivos cacheáveis pelo cliente. A soma dos tamanhos dos arquivos cacheáveis deve ser, portanto, menor ou igual ao espaço destinado ao cache do cliente que é fixo.

O que o modelo analítico faz é determinar quais arquivos cada cliente deve cachear a fim de obter o melhor desempenho possível para o sistema de arquivos. O desempenho é considerado bom quando o tempo médio de acesso aos arquivos é pequeno.

Por fim, [LYW93] apresenta uma tentativa de modelagem de uma arquitetura de cache remoto com a aplicação da política LRU para decidir quais arquivos devem deixar o cache quando o seu espaço termina permitindo que o número de arquivos cacheáveis não fique limitado pelo tamanho do cache.

### 3.2.1 O modelo

A fim de diminuir a complexidade do problema, algumas simplificações foram feitas. Os arquivos são considerados como objetos indivisíveis e de tamanho fixo. Todos arquivos possuem o mesmo tamanho e a única operação tratada pelo modelo é a leitura de um arquivo completo. As escritas não são consideradas pois, neste modelo, elas não se beneficiam da arquitetura de cache remoto. A política para a manutenção da consistência entre os caches não é especificada. Os diretórios e operações com o espaço de nomes são ignorados. Finalmente, as

[6] Eventualmente, algum sistema de cache remoto pode utilizar um mecanismo que não o *broadcast* para localizar um arquivo em um cache remoto, mas o sistema aqui descrito utiliza este mecanismo.

máquinas da rede são todas iguais em termos de velocidade de processamento e de capacidade de armazenamento.

Inicialmente, vamos definir os parâmetros do modelo:

- $P_{ij}$ é a probabilidade de uma aplicação na máquina $j$ solicitar acesso ao arquivo $i$ em um determinado período.
- $p_{ij}^{AL}$ é a probabilidade do arquivo $i$ estar presente no cache local quando solicitado por uma aplicação na máquina $j$ (acerto local).
- $p_{ij}^{AR}$ é a probabilidade do arquivo $i$ não estar presente no cache local mas estar presente no cache de alguma máquina remota quando solicitado pela máquina $j$ (acerto remoto).
- $p_{ij}^{NA}$ é a probabilidade do arquivo $i$ não estar em nenhum dos caches da rede quando solicitado pela máquina $j$ (não-acerto).

Da própria definição segue que $p_{ij}^{AL} + p_{ij}^{AR} + p_{ij}^{NA} = 1$.

- $m_l$ tempo gasto para acessar um arquivo no cache local.
- $m_r$ tempo gasto para acessar um arquivo em um cache remoto.
- $disco_i$ tempo gasto para acessar o arquivo $i$ no disco.

O valor básico que se deseja minimizar é o custo de acessar o arquivo $i$ na máquina $j$ que é definido como:

$$c_{ij} = (m_l \times p_{ij}^{AL}) + (m_r \times p_{ij}^{AR}) + (disco_i \times p_{ij}^{NA})$$

Sendo $H_{ij}$ a probabilidade de encontrarmos o arquivo $i$ no cache local de $j$ em um determinado instante e $N$ o número de máquinas da rede, temos que:

$$p_{ij}^{AL} = H_{ij}$$

$$p_{ij}^{AR} = \underbrace{(1 - \overbrace{\prod_{k=1,k\neq j}^{N} (1 - H_{ik})}^{(I)})}_{(II)} \times \overbrace{(1 - H_{ij})}^{(III)}$$

$$p_{ij}^{NA} = \prod_{k=1}^{N} (1 - H_{ik})$$

note que $(I)$ é a probabilidade de $i$ não estar em nenhum cache remoto, $(II)$ é a probabilidade de $i$ estar em algum cache remoto e $(III)$ é a probabilidade de $i$ não estar no cache local.

O que o algoritmo de otimização determina é uma matriz $X$ que indica quais são os arquivos cacheáveis em cada máquina. $X_{ij} = 1$ se o arquivo $i$ pode ser cacheado na máquina $j$ e 0 caso contrário.

No caso em que o espaço ocupado por todos os arquivos cacheáveis por uma máquina deve ser menor ou igual ao espaço reservado para o cache dessa máquina, é óbvio que, após um certo tempo, $H_{ij} = X_{ij}$[7]. Quando esta condição não é satisfeita, a modelagem de $H$ é mais complexa. [DT90] apresenta uma tal modelagem para $H$ em função de $X$ supondo que o

[7] Note que, mesmo neste caso, o número de possíveis configurações para X é exponencial e a determinação da configuração ótima é um problema $NP$-completo.

cache utiliza a política LRU e que o padrão de acesso aos arquivos é constante e determinado por $P_{ij}$.

Os autores mostram que o desempenho da LRU é semelhante ao desempenho obtido por uma estratégia gulosa que consistiria em manter um cache fixo contendo os arquivos lidos com maior freqüência. A única vantagem da LRU seria a sua fácil implementação uma vez que ela não exige nenhum conhecimento prévio sobre a freqüência com que os arquivos são acessados.

O que [DT90] não leva em conta (e, conseqüentemente, o modelo aqui apresentado também não) é o fato de que, em sistemas de arquivos reais, a probabilidade de um arquivo ser acessado em um determinado instante é maior se ele foi acessado há pouco tempo. E, por outro lado, se um arquivo não é acessado há muito tempo, a probabilidade de ele ser acessado é menor.

**O Custo de uma Configuração**

No FAP clássico, o custo de uma configuração do sistema é proporcional ao tempo médio gasto no atendimento de todas as solicitações de acesso aos arquivos em cada máquina em um determinado intervalo de tempo. Assim, temos que

$$CUSTO_{FAP} = \frac{\sum_j \sum_i c_{ij} P_{ij}}{N}$$

(lembrando que $N$ é o número de máquinas da rede).

No entanto, [LYW93] propõe um alternativa melhor. O problema com o FAP clássico é que a sua solução ótima pode não ser uma boa configuração para algumas máquinas isoladamente. Eventualmente, a configuração ótima para o FAP pode resultar em um péssimo desempenho para algumas máquinas que estariam se sacrificando para melhorar o desempenho global do sistema.

A alternativa proposta consiste em garantir que nenhuma máquina adote uma configuração que prejudique o seu próprio desempenho. Uma função custo que garante isso é

$$CUSTO = \max_j \sum_i c_{ij} P_{ij}.$$

Ou seja, deseja-se obter uma configuração que minimize o tempo máximo gasto por uma máquina da rede isoladamente. Com isso, nenhuma máquina é prejudicada. Chamaremos a estratégia de minimizar $CUSTO_{FAP}$ de **estratégia global** e a de minimizar $CUSTO$ de **estratégia consensual**.

### 3.2.2 *Simulated Annealing*

O algoritmo utilizado para a busca da solução ótima é o *Simulated Annealing* [AK89] desenvolvido a partir de uma analogia com um procedimento utilizado em metalurgia denominado *annealing* que, em português, é conhecido como **recozimento**, um termo herdado da culinária.

O processo de recozimento de metais consiste em aquecer o metal até uma temperatura levemente acima da temperatura crítica de homogeneização e, em seguida, resfriá-lo **cuidadosamente** até que as moléculas se arranjem e o metal atinja o estado de equilíbrio termodinâmico. Se o aquecimento e o resfriamento não são feitos da maneira correta, o metal não atinge o estado de equilíbrio termodinâmico que é um estado no qual as suas moléculas formam uma estrutura altamente organizada e a energia do sistema é mínima.

Vejamos agora a descrição do algoritmo já adaptado ao modelo. Em posse dos valores de $P_{ij}$ e demais parâmetros do sistema, aplica-se o algoritmo de modo a encontrar uma configuração $X$ que minimize a função $CUSTO$.

Procedimento *Simulated Annealing*
Início
  $S \leftarrow$ solução inicial escolhida ao acaso
  $T \leftarrow$ 0,1 /* temperatura inicial*/
  $L \leftarrow L_0$
  Enquanto (não se solidificou) faça
    Repita $L$ vezes
      $S' \leftarrow S$
      escolha uma entrada de $S'$ ao acaso e altere o seu valor
      Se $CUSTO(S') < CUSTO(S)$ então $S \leftarrow S'$
      caso contrário $S \leftarrow S'$ com probabilidade $e^{\frac{CUSTO(S)-CUSTO(S')}{T}}$
    $T \leftarrow T \times 0,9$ /* reduz a temperatura */
    $L \leftarrow reduz(L)$
  $X \leftarrow S$, a solução final encontrada pelo algoritmo
Fim.

Uma solução é considerada solidificada quando a temperatura é reduzida 7 vezes em seguida sem que $S$ se altere.

No caso em que número de objetos cacheáveis por cada máquina deve ser menor ou igual ao espaço reservado para o cache, esta restrição deve ser verificada para todas as soluções parciais $S$ e $S'$.

Figura 3.2: *Simulated Annealing*

Como mostra a figura 3.2, a partir de uma solução inicial $S$, o algoritmo percorre soluções vizinhas. Se a solução vizinha $S'$ é melhor do que a solução corrente, ela se torna a solução corrente. Caso contrário, ela se torna a solução corrente com probabilidade

$$e^{\frac{CUSTO(S)-CUSTO(S')}{T}}.$$

Esta condição, chamada **critério de Metropolis**, faz com que o algoritmo não fique restrito a mínimos locais. Quanto menor a diferença entre os custos de $S$ e $S'$, maior é a probabilidade da solução vizinha ser adotada mesmo com um aumento no custo. À medida em que a temperatura $T$ cai, a probabilidade de adotarmos uma solução que aumente o custo decresce fazendo com que o algoritmo se estabilize abaixo de uma certa temperatura.

É possível provar que se as funções $CUSTO$ e $reduz$ obedecerem a determinadas exigências, o algoritmo acima descrito termina a sua execução em tempo polinomial e acha a solução ótima com probabilidade próxima de 1[MRSV86].

### 3.2.3 Simulação

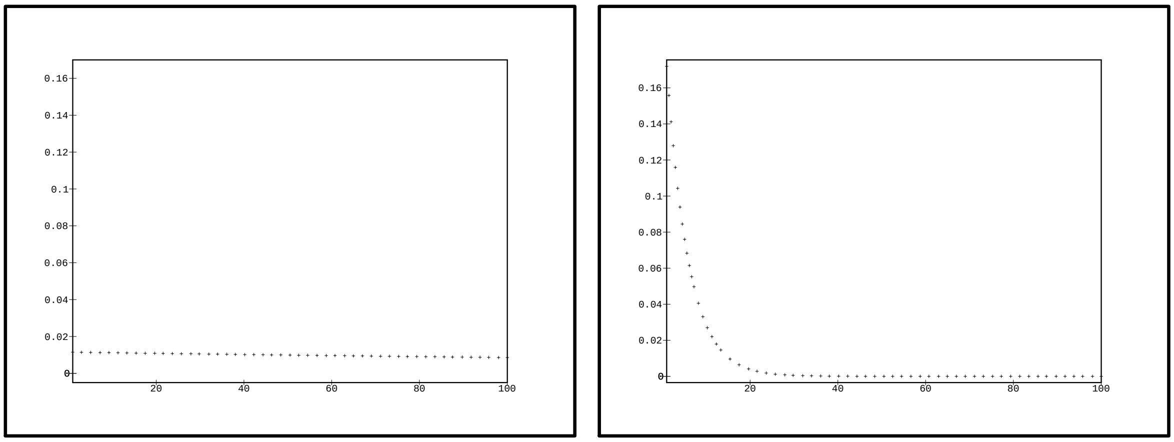
[LYW93] apresenta os resultados de simulações com três estratégias para configuração do cache. Duas delas correspondem à aplicação do *simulated annealing* com as funções $CUSTO_{FAP}$ (estratégia global) e $CUSTO$ (estratégia consensual). A terceira estratégia é a gulosa onde cada máquina cacheia os arquivos mais freqüentemente acessados. A tabela 3.4 mostra informações sobre a rede da simulação.

| | |
|---|---|
| Número de máquinas | 10 |
| Número de arquivos | 100 |
| Tempo de acesso a um arquivo em memória principal local | 1ms |
| Tempo de acesso a um arquivo em memória principal remota | 10ms |
| Tempo de acesso a um arquivo em disco | 59ms |

Tabela 3.4: Parâmetros da Rede

Considerando como medida de desempenho de um sistema, o tempo médio de acesso aos arquivos em um determinado período, a estratégia global é, por definição, a que apresenta os melhores resultados.

Quando a freqüência de acesso aos arquivos $P_{ij}$ se comporta, para toda máquina $j$, como o gráfico da figura 3.3a, o desempenho das duas estratégias é o mesmo. Por outro lado, quando $P_{ij}$ não é tão uniforme (figura 3.3b), a estratégia global apresenta um pequeno ganho de cerca de 10% em relação à estratégia consensual. Mas, analisando-se cada máquina separadamente, observou-se que o desempenho individual varia entre 82% e 118% do desempenho médio na estratégia global e apenas entre 92% e 103% na estratégia consensual. Portanto, pode ser um bom negócio a adoção da estratégia consensual a fim de não penalizar certas máquinas individualmente.




(a) (b)

Os gráficos mostram a probabilidade de acesso a cada um dos 100 arquivos por uma determinada máquina.

Figura 3.3: Padrão de acesso aos arquivos

A estratégia gulosa apresentou quase sempre um desempenho muito inferior às outras duas. No caso extremo no qual todas as máquinas acessam os arquivos com a freqüência indicada na figura 3.3a, a estratégia gulosa é péssima pois apenas uma pequena fração dos arquivos podem ser lidos do cache uma vez que todas as máquinas guardam os mesmos arquivos em seus caches e a arquitetura de cache remoto se torna inútil.

Por outro lado, quando a freqüência com a qual os arquivos são acessados varia de máquina para máquina, o desempenho da estratégia gulosa melhora. No caso extremo – e pouco provável – de não existir nenhuma correlação entre o padrão de acesso aos arquivos nas diferentes máquinas, a estratégia gulosa apresenta um desempenho semelhante ao da estratégia consensual com a vantagem de que sua implementação é bem mais simples.

### 3.2.4 Críticas ao Modelo

Ao comparar a estratégia de otimização consensual com a estratégia gulosa, os autores pretendem mostrar que a aplicação do seu modelo melhoraria o desempenho de sistemas com cache remoto. A estratégia utilizada pelos sistemas convencionais é a LRU; segundo os autores, uma estratégia deste tipo apresentaria o mesmo desempenho que a gulosa e, portanto, o sistema proposto traria benefícios em relação aos sistemas convencionais. Mas, isto só é válido se for considerado o modelo de independência de referenciação adotado em [DT90] que supõe que a probabilidade de um arquivo ser acessado em um determinado instante não depende do fato de ele ter sido acessado há pouco tempo atrás.

No entanto, em ambientes reais é razoável supor que exista uma dependência entre os acessos aos arquivos. A probabilidade de um arquivo ser acessado é maior se ele foi acessado há pouco tempo. Portanto seria interessante comparar o desempenho da estratégia consensual com o LRU tradicional levando em conta a dependência entre os acessos aos arquivos. Deste modo, poderíamos afirmar com maior segurança se vale, ou não, a pena investir no desenvolvimento de um complexo sistema adaptativo como o proposto em [LYW93].

Um importante aspecto não abordado é o de como seria feito o cálculo de $P_{ij}$ para todos os arquivos $i$ e máquinas $j$. Assim como no modelo de Borghoff, a coleta e concentração de tantas informações em apenas um ponto da rede pode prejudicar o desempenho do sistema.

Uma possível solução para estes problemas seria dividir o cache em duas partes. A primeira parte conteria um grupo fixo de arquivos determinados pela estratégia consensual que seria aplicada apenas a um grupo restrito de arquivos determinados pelo administrador do sistema. A outra parte se responsabilizaria pelos demais arquivos e o seu tamanho variaria dinamicamente como acontece no SPRITE (ver seção 2.6.3). Deste modo, os arquivos acessados com muita freqüência seriam tratados eficientemente pela primeira parte do cache e os demais, seriam tratados pela segunda parte que, utilizando a política LRU, garantiria que os arquivos acessados recentemente estariam no cache.

Já a escolha e adaptação do algoritmo *simulated annealing* para a localização da solução ótima foi uma importante contribuição pois permite que o cálculo das novas configurações seja muito mais rápido e que obtenha melhores resultados do que a busca exaustiva utilizada por Borghoff.

# Capítulo 4

# Leases

Estudaremos, neste capítulo, os *leases*, um mecanismo para a manutenção da consistência entre os caches dos clientes em um sistema distribuído. Além de ser tolerante a partições na rede e a quedas dos clientes e servidores, este mecanismo garante uma boa eficiência mesmo quando há compartilhamento de arquivos. É bom lembrar que este é exatamente o ponto fraco do cache do SPRITE que desabilita totalmente o cache quando há acesso concorrente com escrita.

Na seção 4.1, descrevemos o funcionamento e as principais características do método. Apresentamos um modelo analítico que mede o tráfego gerado pelo protocolo dos leases na seção 4.2. Em seguida, (seção 4.3) introduzimos um modelo novo que permite a implementação de um sistema baseado em leases que, adaptando-se à carga gerada pelos clientes, diminui o tráfego gerado para a garantia de consistência. Finalmente, em 4.4 descrevemos a nossa implementação do protocolo e em 4.4.2 apresentamos o resultado de testes sobre o desempenho do sistema.

## 4.1 Um Mecanismo para a Consistência do Cache

Um *lease* é um contrato que garante o direito de propriedade sobre um determinado objeto durante um período limitado. A utilização do termo "lease" no contexto de sistemas de arquivos distribuídos foi introduzida por Cary Gray durante o seu doutoramento na *Stanford University* sob orientação de David Cheriton. Apesar do conceito ter sido apresentado pela primeira vez em [GC89] por Gray, os pesquisadores envolvidos no projeto do sistema ECHO dizem ter desenvolvido o mesmo mecanismo na mesma época de forma independente [MBH+93].

Vejamos como a idéia funciona no contexto de sistemas de arquivos distribuídos. Quando um cliente envia um pedido de leitura de um arquivo para o servidor, ele recebe, juntamente com o arquivo, uma garantia de que o arquivo não será alterado sem o seu consentimento durante um determinado período.

Se uma aplicação de um cliente solicitar uma leitura de um arquivo que está cacheado localmente, o sistema precisa se assegurar de que o prazo de validade do *lease* para este arquivo ainda não terminou. Se o *lease* ainda é válido, a aplicação pode receber o arquivo sem que o cliente entre em contato com o servidor.

Por outro lado, se o prazo de validade do *lease* já expirou, o cliente precisa entrar em contato com o servidor para se certificar de que a versão presente no cache local é a mais atual. Se a versão local está desatualizada, é necessário trazer a nova versão do servidor.

Quando um cliente solicita ao servidor uma alteração em um arquivo, o servidor precisa obter a concordância de todos os clientes que possuem um *lease* válido para este arquivo antes de executar a alteração e responder ao cliente que a efetuou. O servidor só pode confirmar a alteração após o consentimento de todos os clientes com *lease* válido ou após a expiração dos *leases* dos clientes que não responderem.

Quando o servidor solicita a concordância de um cliente para a atualização de um arquivo, o cliente marca o seu *lease* para este arquivo como expirado. Se, em seguida, uma aplicação deste cliente solicita a leitura do arquivo, a nova versão tem de ser trazida do servidor.

Os *leases* servem não apenas para garantir a consistência do conteúdo dos arquivos cacheados mas também das "meta-informações". Isto é, ao cachear os atributos de arquivos e o conteúdo dos diretórios, os clientes podem manter *leases* a fim de garantir a consistência destas informações.

Note que este mecanismo garante a consistência entre os caches dos clientes desde que se adote o *write-through*, isto é, que os clientes enviem imediatamente para os servidores as escritas solicitadas pelas suas aplicações e que esperem pela confirmação do servidor antes de continuar a sua execução.

É também possível manter a consistência via *leases* quando se adota a política de *write-behind* mas, neste caso, o protocolo se torna mais complexo. O sistema ECHO [MBH+93] adotava o *write-behind* e garantia a consistência do cache através de *leases* tanto para os arquivos quanto para diretórios e atributos.

Para usar *leases* juntamente com *write-behind*, é necessário controlar o acesso aos arquivos através de dois tipos de *leases*. Um *lease* para leitura funcionaria como o *lease* descrito anteriormente. Já um *lease* para escrita retardada garantiria o direito de um cliente a efetuar escritas em seu cache sem mandá-las imediatamente ao servidor.

Antes de conceder um *lease* para escrita retardada a algum cliente, o servidor deve se certificar que nenhum outro cliente possua *leases* para o mesmo arquivo sejam eles para leitura ou escrita.

Por outro lado, antes de conceder um *lease* para leitura, o servidor deve verificar se algum cliente possui um *lease* para escrita retardada e, se este for o caso, enviar uma mensagem solicitando que os blocos sujos deste cliente lhe sejam enviados. Somente após receber todos os blocos sujos do cliente ou após a expiração do *lease* é que o servidor pode conceder o *lease* para leitura.

Em determinadas situações o tráfego gerado pela manutenção destes dois tipos de *leases* pode ser muito grande fazendo com que a simples desabilitação do cache das escritas apresente um melhor desempenho.

No decorrer deste capítulo estudaremos apenas o método mais simples que adota o *write-through*. Em sistemas que adotam esta política é importante assegurar que o maior número possível de arquivos temporários sejam tratados localmente uma vez que eles são os maiores beneficiados pelo *write-behind*. Uma prática simples e eficaz é fazer com que o diretório */tmp* (o diretório de arquivos temporários do UNIX) seja armazenado localmente e incentivar os programadores a utilizar este diretório para os arquivos temporários que não precisem ser compartilhados por mais de um cliente.

### 4.1.1 A Duração de um *Lease*

Um fator fundamental no desempenho dos *leases* é a determinação do seu prazo de validade.

Podemos destacar três importantes vantagens na adoção de *leases* de curta duração:

- Quando um servidor perde o contato com um de seus clientes, seja por uma partição na rede ou devido à queda do cliente, o servidor precisa esperar pelo término do prazo dos *leases* daquele cliente antes de efetuar atualizações nos respectivos arquivos.

  Se o prazo de validade do *leases* é curto, o tempo de espera, no caso de falhas, é menor.

- Quando um servidor cai, as informações sobre os *leases* são perdidas. Se, quando o servidor se recupera de uma queda, todos os *leases* dos clientes já tiverem expirado, então o servidor não precisa se preocupar com o que aconteceu antes da sua queda. Ele pode efetuar escritas e conceder novos *leases* sem comprometer a integridade do sistema.

  Portanto adotar um prazo de validade menor do que o tempo de reinicialização do servidor tende a ser uma boa escolha.

- Se os *leases* duram muito tempo, aumenta a probabilidade de acontecer aquilo que é chamado de **falso compartilhamento**. O falso compartilhamento ocorre quando um cliente executa uma escrita em um arquivo que é coberto por um *lease* de outro cliente que não vai mais acessar o arquivo durante o prazo de validade do seu *lease*. Neste caso, o servidor solicita o consentimento do cliente que possui o *lease* sem que houvesse necessidade.

  *Leases* curtos diminuem a ocorrência de falsos compartilhamentos.

Por outro lado, a adoção de *leases* longos também pode ser vantajosa. Quanto mais curto é o prazo de validade, maior é a sobrecarga gerada pela renovação dos *leases*.

No caso de arquivos que não são compartilhados com escrita por mais de um cliente ou que raramente o são, é recomendável a adoção de prazos de validade mais longos. Se os *leases* para um arquivo valem por muito tempo, então os clientes que possuem um *lease* para este arquivo podem usar a cópia cacheada muitas vezes antes de precisar entrar em contato com o servidor.

*Leases* longos são particularmente recomendáveis para arquivos instalados pelo administrador do sistema e utilizados publicamente como aqueles dos diretórios */bin, /man, /lib, /usr/bin/, /usr/include*. Estes arquivos são alterados raramente porém são lidos com freqüência.

Pode-se até mesmo adotar um prazo de validade infinito. Se isto for feito, os *leases* se comportam de maneira idêntica aos *callbacks* do ANDREW. Quando a duração dos *leases* é longa, é preciso adotar algum mecanismo para que o servidor consiga reconstituir o seu estado após uma eventual queda. Isto pode ser feito de várias maneiras:

- O servidor pode armazenar as informações sobre os *leases* em memória RAM não-volátil que sobrevive a quedas [BAD+92].

- Uma cópia das informações sobre os *leases* pode ser mantida em um *log* em fita ou disco magnético.

- Após uma queda, o servidor pode aplicar o método de recuperação distribuída de estado adotado pelo SPRITE (ver seção 2.6.4).

Se o prazo de validade é 0, o sistema se comporta como o SPRITE quando há acesso concorrente com escrita[1], ou seja, todas as leituras e escritas são enviadas diretamente para o servidor.

### Tolerância a Falhas

Um dos pontos fortes do protocolo dos *leases* é a sua tolerância a falhas. Ao contrário do ANDREW e do SPRITE, o protocolo dos *leases* é, por si só, tolerante a falhas[2]. Adotando

[1] A definição de acesso concorrente com escrita pode ser encontrada na seção 2.6.3

[2] Estamos considerando aqui apenas falhas como quedas de servidores e clientes e partições da rede. Não estamos considerando "falhas bizantinas" como a adulteração de mensagens

um prazo de validade menor do que o tempo de reinicialização do servidor após uma queda, o protocolo dispensa mecanismos extras de tolerância a falhas e garante a coerência das informações contidas nos caches dos clientes como veremos a seguir.

Se, por um motivo qualquer, uma parte dos clientes não possa ser acessada pelo servidor, nenhuma atitude especial deve ser tomada. No caso do servidor receber uma solicitação de escrita, tudo o que ele faz é obedecer ao protocolo, isto é, ele retarda o cumprimento desta solicitação até o instante da expiração do último *lease* em posse de um cliente inacessível.

A disponibilidade do cache em um sistema baseado em *leases* também é maior do que no SPRITE. Se um cliente SPRITE sofre uma queda enquanto possuía um arquivo aberto para escrita, nenhum cliente poderá cachear este arquivo até que o cliente volte a funcionar. No caso dos *leases* este problema não existe.

Já no caso em que o servidor não pode ser acessado por um cliente, também não há nada de especial a ser feito. O cliente utiliza os arquivos para os quais possui *lease* válido sem que a consistência do sistema de arquivos corra perigo.

Finalmente, as quedas dos servidores também não comprometem a coerência dos caches desde que o prazo de validade dos *leases* seja menor do que o tempo necessário para o servidor se recuperar de uma queda. Ao contrário dos servidores ANDREW e SPRITE que perdem uma grande quantidade de informações importantes quando sofrem quedas, os servidores que utilizam *leases* não perdem nenhuma informação relevante. Isso se dá pois os *leases* perdidos não são mais válidos no momento em que o servidor volta a oferecer o seu serviço.

### 4.1.2 Sincronização de Relógios

"Num relógio é quatro e vinte,
No outro, quatro e meia.
É qui dum relógio pro outro,
As hora vareia."
Tocar na Banda (1965) – Adoniran Barbosa

A utilização de prazos de validade para os *leases* exige que os clientes mantenham os seus relógios sincronizados com o servidor. Ao enviar um *lease* para um cliente, o servidor envia o instante exato, segundo o seu relógio, no qual o *lease* será expirado.

Se o relógio do cliente não estiver sincronizado com o relógio do servidor, o horário de expiração do *lease* não fará sentido. Pode acontecer de um *lease* já chegar expirado ou então do cliente ler dados de seu cache mesmo depois da expiração do *lease* de acordo com o relógio do servidor.

A dificuldade encontrada na sincronização de relógios de máquinas distintas se deve ao fato de que não se sabe exatamente qual é o tempo que uma mensagem gasta para ir de uma máquina para outra em um determinado momento. Este tempo depende basicamente da distância entre as duas máquinas, da capacidade do canal de comunicação entre elas e da carga presente no canal no momento da sincronização. Eventualmente, mensagens podem ser perdidas ou o seu conteúdo pode ser alterado dificultando ainda mais a sincronização.

Além disso, não existem relógios perfeitos. Sempre há alguma diferença entre a velocidade de dois relógios. Um mesmo relógio pode, também, adiantar e atrasar em situações distintas.

A fim de minimizar este problema, existem vários mecanismos para a sincronização de relógios. Executando um processo de sincronização de tempos em tempos, é possível manter os relógios de uma rede sincronizados a menos de um erro $\epsilon$ que seja aceitável para as aplicações executadas no sistema.

Na nossa implementação, por exemplo, a duração dos *leases* é da ordem de alguns segundos. Assim, um erro menor do que um décimo de segundo (facilmente obtido em uma rede local) é algo aceitável. Tudo o que se tem que fazer é retirar um décimo de segundo de cada *lease* quando ele é recebido pelo cliente.

[YM93] é uma compilação de artigos sobre sincronização de relógios. Uma introdução didática ao tema pode ser encontrada em [Tan92].

Nas seções seguintes discutiremos maneiras de determinar o melhor prazo de validade para os *leases*.

## 4.2 O modelo de Gray

Como observamos na seção anterior, quanto mais curta é a duração de um *lease*, maior é a sobrecarga gerada pela sua renovação. Quanto mais longo é um *lease*, maior é a sobrecarga gerada pelo falso compartilhamento.

Gray apresenta, em [GC89], um modelo que mede a carga no servidor gerada pelo protocolo dos *leases* em função do seu prazo de validade. O modelo analisa a sobrecarga gerada para um único arquivo de um determinado servidor. A tabela 4.1 mostra os principais parâmetros do modelo.

| | |
|---|---|
| $N$ | número de clientes que acessam o arquivo |
| $R$ | taxa de leituras por cliente |
| $W$ | taxa de escritas por cliente |
| $m_{prop}$ | tempo para a propagação de uma mensagem |
| $m_{proc}$ | tempo para o processamento de uma mensagem |
| $\epsilon$ | discordância máxima entre os relógios |
| $t_s$ | duração do *lease* no servidor |
| $t_c$ | duração do *lease* no cliente |
| $S$ | número de clientes que compartilham o arquivo em um dado instante |

Tabela 4.1: Parâmetros do modelo de Gray

Cada um dos $N$ clientes efetua, em média, $R$ *reads* e $W$ *writes* por unidade de tempo no arquivo em questão segundo um processo de Poisson. Quando o arquivo recebe uma escrita, ele é compartilhado, em média, por $S$ clientes. Isto é, $S$ clientes possuem *leases* válidos para este arquivo. $m_{prop}$ é o tempo de propagação de uma mensagem do servidor para um cliente ou vice versa e $m_{proc}$ é o tempo gasto no processamento (envio ou recebimento) de uma mensagem tanto pelo servidor quanto pelos clientes. Estes valores já devem levar em conta a necessidade de retransmissão das mensagens quando ocorrem falhas na comunicação; atrasos causados por congestionamentos e esperas nas filas de mensagens são ignorados pelo modelo.

Portanto o tempo total para o envio e recebimento de uma mensagem é $m_{prop}+2m_{proc}$. Se a rede permite o envio de uma mensagem com vários destinatários (*multicast* [Dee89, AFM92]), o servidor gasta pelo menos

$$2m_{prop} + (n+3)m_{proc} \tag{4.1}$$

para invalidar o *lease* de $n$ clientes[3].

[3] A mensagem gasta $1m_{prop}$ para ir e $1m_{prop}$ para voltar; o servidor gasta $1m_{proc}$ para enviar o *multicast* e $nm_{proc}$ para processar as $n$ respostas; finalmente, cada cliente gasta $2m_{proc}$ para processar a mensagem recebida do servidor e a sua resposta (os clientes trabalham em paralelo).

Quando um *lease* é recebido por um cliente, já se passaram $m_{prop}+2m_{proc}$ desde o momento da sua criação pelo servidor. Portanto, se a duração de um *lease* no servidor é $t_s$, a sua duração no cliente é de apenas

$$t_c = t_s - (m_{prop} + 2m_{proc}) - \epsilon.$$

Uma situação indesejável é aquela na qual $t_s > 0$ mas $t_c \leq 0$. Neste caso, as leituras não ganham nada com os *leases* – porque eles sempre chegam expirados nos clientes – mas as escritas precisam ser retardadas até o término do prazo de validade dos *leases*. Se $t_s$ for menor do que $m_{prop} + 2m_{proc} + \epsilon$, então é melhor adotar $t_s = 0$ a fim de não retardar as escritas.

Portanto quanto maior é a discordância entre os relógios e quanto mais lenta é a rede, maior deve ser o prazo de validade dos *leases* para que a sua utilização traga algum benefício.

Em seu modelo, Gray supõe que $W$ é muito pequeno sendo possível ignorar o fato de que os *leases* podem ser invalidados e durarem menos do que $t_c$. Em todo o seu trabalho, a duração dos *leases* nos clientes é considerada fixa e independente de $W$.

### Custo dos *leases* nas leituras

Durante a validade de um *lease*, um cliente atende $Rt_c$ solicitações de leituras sem contar o *read* que resultou na solicitação do *lease*. Portanto o custo da concessão de um *lease* (2 mensagens) é amortizado entre $1 + Rt_c$ *reads*. Logo, a taxa de mensagens relativas a concessões de *leases* gerada pelos $N$ clientes (ou o custo dos *leases* nas leituras) é

$$C_R = \frac{2NR}{1 + Rt_c}. \tag{4.2}$$

O atraso nas leituras causado pela concessão de *leases* é, em média,

$$\frac{2(m_{prop} + 2m_{proc})}{1 + Rt_c}.$$

### Custo dos *leases* nas escritas

O modelo supõe que, sempre que o servidor recebe uma solicitação de escrita, ele deve invalidar o *lease* de $S - 1$ clientes [4].

Para obter a aprovação para uma escrita, o servidor envia um *multicast* recebendo $S - 1$ respostas dos clientes. De 4.1 segue que o tempo exigido para a aprovação é $t_a = 2m_{prop} + (S + 2)m_{proc}$ se $S > 1$ (se $S = 1$ não há necessidade de obter aprovação).

Se, por outro lado, a rede não oferecer *multicast*, o atraso para cada escrita sobe para pelo menos $2m_{prop} + 2Sm_{proc}$.

O número de mensagens relativas à consistência tratadas pelo servidor no caso de escritas é

$$C_W = \begin{cases} SNW & \text{com multicast} \\ 2(S-1)NW & \text{sem multicast} \end{cases}. \tag{4.3}$$

### Custo total do protocolo

Dos resultados anteriores segue que, para $S > 1$ e $t_s > 0$, o servidor envia ou recebe

$$\frac{2NR}{1 + Rt_c} + C_W \tag{4.4}$$

[4] Um dos clientes seria o mesmo que solicitou a escrita e, portanto, não precisaria ser notificado.

mensagens por unidade de tempo relativas à garantia da consistência dos caches.

O principal ponto fraco do modelo de Gray é que o grau de compartilhamento ($S$) é considerado constante. Os estudos de Gray analisam o desempenho do protocolo em função do prazo de validade dos *leases* para diferentes valores fixos de $S$. Mas, se analisarmos o problema com cuidado, notamos que o grau de compartilhamento não é independente do prazo de validade dos *leases* ($t_c$).

$S$ representa o número de clientes que possuem um *lease* válido para o arquivo. Mas, quanto menor for a duração dos *leases* menor será a probabilidade de um cliente qualquer possuir um *lease* em um determinado instante. Assim, quanto menor for o prazo de validade dos *leases* menor será $S$.

De acordo com o comportamento de $S$ em função de $t_c$, pode acontecer da carga relativa a consistência aumentar a partir de um determinado prazo de validade. Um comportamento deste tipo indicaria que a adoção de *leases* mais longos só trariam prejuízos ao desempenho do sistema. Na seção seguinte, discutiremos alterações no modelo de Gray de modo a refletir mais fielmente o comportamento de $S$.

## 4.3 Melhorando o Modelo

A fim de estimar o número médio de clientes possuidores de *leases* para um mesmo arquivo em cada instante consideremos, inicialmente, o acesso de um único cliente ao arquivo. A figura 4.1 mostra esquematicamente períodos nos quais o cliente possui um *lease* (indicados pela letra L) e períodos nos quais o cliente não possui (indicados pela letra T).

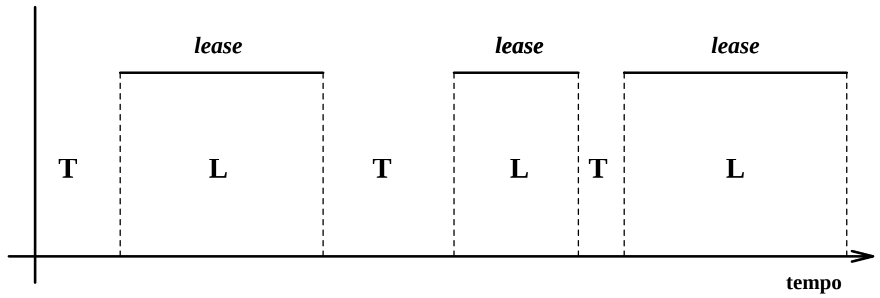



Figura 4.1: *Leases* em um cliente

O que queremos, agora, é determinar qual é a porção do tempo na qual o cliente possui um *lease*. Um resultado conhecido da Teoria da Disponibilidade[5] garante que a porção do tempo na qual o cliente possui um *lease* é, em média,

$$\frac{E(L)}{E(L)+E(T)}$$

onde $E()$ denota a duração média dos períodos.

O período T começa quando o *lease* se expira e termina no momento da próxima leitura. Processos de Poisson não possuem memória, ou seja, o comportamento futuro do processo

[5] Ver [BP81], seção 7.2.

independe do passado; assim, o tempo esperado até a próxima leitura é sempre o mesmo, o inverso da taxa de leituras:

$$E(T) = \frac{1}{R}.$$

Já o período L começa quando o cliente recebe um *lease* e termina na próxima escrita ou após $t_c$ unidades de tempo.

A fim de calcular $E(L)$ podemos imaginar que, nos períodos L, ocorre a superposição de dois processos de Poisson. O primeiro, com taxa $NW$, representa que o *lease* pode ser cancelado por uma escrita de um dos $N$ clientes. O segundo, com taxa $\frac{1}{t_c}$ representa que o *lease* é expirado após $t_c$ unidades de tempo.

O processo resultante desta superposição tem como taxa a soma das taxas dos processos que o compõe. Assim, chegamos ao valor médio do período L que é o inverso da taxa:

$$E(L) = \frac{1}{NW + \frac{1}{t_c}}.$$

Logo, a porção de tempo na qual o cliente possui um *lease* é

$$\frac{\frac{1}{NW+\frac{1}{t_c}}}{\frac{1}{NW+\frac{1}{t_c}} + \frac{1}{R}} = \frac{\frac{1}{NW+\frac{1}{t_c}}}{\frac{1}{NW+\frac{1}{t_c}} + \frac{1}{R}} \times (\frac{NW + \frac{1}{t_c}}{NW + \frac{1}{t_c}}) = \frac{Rt_c}{1 + Rt_c + NWt_c}.$$

Supondo que os $N$ clientes acessem o arquivo independentemente chegamos à nossa estimativa para o compartilhamento:

$$S = \frac{NRt_c}{1 + Rt_c + NWt_c}. \tag{4.5}$$

#### Custo dos *leases* nas leituras

$E(L)$ é uma estimativa mais precisa do tempo médio de duração de um *lease* do que $t_c$. Assim, uma estimativa do custo dos *leases* nas leituras melhor do que a dada pela fórmula 4.2 é a que se segue:

$$C_R = \frac{2NR}{1 + RE(L)} = \frac{2NR}{1 + R\frac{1}{NW+\frac{1}{t_c}}} = \frac{2NR(1 + NWt_c)}{1 + Rt_c + NWt_c}. \tag{4.6}$$

#### Custo dos *leases* nas escritas

Quando o servidor recebe uma escrita, há a necessidade de invalidar os *leases* de

$$S - \frac{S}{N} = \frac{S(N-1)}{N}$$

clientes[6]. Estamos subtraindo de $S$ a probabilidade do cliente que solicitou a escrita possuir um *lease* válido para o arquivo. Gray supunha que esta probabilidade era 1.

Assim, a estimativa para o custo da invalidação dos *leases* nas escritas é

$$C_W = \begin{cases} (1 + S\frac{N-1}{N})NW = NW + S(N-1)W & \text{com multicast} \\ \\ 2S\frac{N-1}{N}NW = 2S(N-1)W & \text{sem multicast} \end{cases} \tag{4.7}$$

que é um valor maior do que aquele previsto por Gray.

[6] E não de $S - 1$ clientes como supunha Gray.

**Custo total do protocolo**

De 4.6 e 4.7, temos que o custo total do protocolo com a garantia da consistência é

$$C_{total} = \begin{cases} \frac{2NR(1+NWt_c)}{1+Rt_c+NWt_c} + NW + S(N-1)W & \text{com multicast} \\ \\ \frac{2NR(1+NWt_c)}{1+Rt_c+NWt_c} + 2S(N-1)W & \text{sem multicast} \end{cases}. \quad (4.8)$$

### 4.3.1 Comparando com o SPRITE

Trataremos aqui apenas o caso no qual há acesso concorrente com escrita. Neste caso, o protocolo adotado pelo SPRITE verifica o número da versão do arquivo a cada leitura gerando um tráfego de $2NR$ mensagens para a garantia da consistência[7].

Assim, a fórmula 4.8 nos diz que, quando há acesso concorrente com escrita, o protocolo dos *leases* é mais eficiente do que o protocolo do SPRITE se e somente se

$$\frac{2NR(1+NWt_c)}{1+Rt_c+NWt_c} + NW + S(N-1)W < 2NR \quad (4.9)$$

com *multicast* e se e somente se

$$\frac{2NR(1+NWt_c)}{1+Rt_c+NWt_c} + 2S(N-1)W < 2NR \quad (4.10)$$

sem *multicast*.

Aplicando o valor de $S$ dado por 4.5 à inequação 4.10 obtemos a seguinte condição para a superioridade dos *leases* no caso sem *multicast*.

$$R > W(N-1) \quad (4.11)$$

Podemos observar então que, quando o número de leituras é suficientemente maior do que o número de escritas, isto é, quando 4.11 é satisfeita, o protocolo dos *leases* é uma boa escolha. Por outro lado, quando o número de escritas é relativamente grande, o melhor é desabilitar o cache como faz o SPRITE.

No caso com *multicast* é possível provar que o protocolo dos *leases* apresenta um desempenho melhor se e somente se

$$R > \frac{NWt_c + \sqrt{N^2W^2t_c^2 + 8(NW^2t_c^2 + Wt_c)}}{4t_c}.$$

Estas observações levam à idéia da implantação de um sistema adaptativo que monitoraria o acesso aos arquivos decidindo quais arquivos devem ser cacheados no momento do acesso concorrente com escrita e quais não. Voltaremos a discutir este ponto na seção 4.4.3.

A seguir apresentamos as estimativas dadas pelo nosso modelo para diferentes valores de $N, R, W$ e $t_c$.

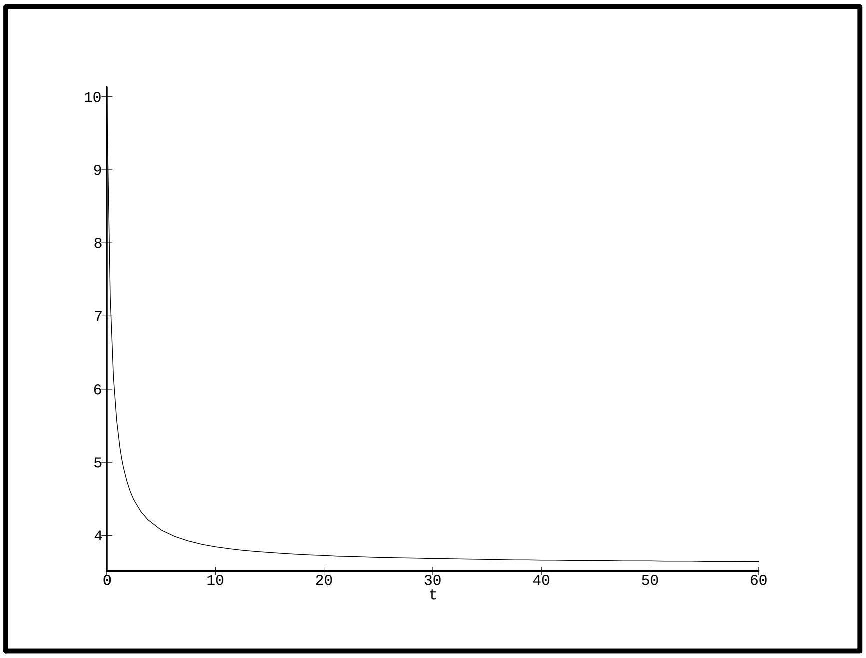
[7] Um programador em ambiente SPRITE pode evitar o estado de acesso concorrente com escrita fechando o arquivo após um grupo de leituras ou escritas e abrindo-o imediatamente antes do acesso seguinte. No entanto, como no SPRITE as operações *open* e *close* exigem a troca de mensagens com o servidor, este procedimento pode não resolver o problema e aumentar ainda mais o tráfego na rede.

### 4.3.2 Estimativas do Modelo

A figura 4.2 apresenta o número de mensagens geradas pelo protocolo dos *leases* em função da duração dos *leases*. O gráfico foi construído considerando 5 clientes realizando, em média, uma escrita a cada 10 segundos e duas leituras por segundo cada um. As estimativas desta seção tratam do caso sem *multicast*.

Podemos observar que, ao contrário do que podíamos esperar, não existe um prazo de validade ótimo para os *leases*. Quanto maior a duração do *lease*, menor é a carga gerada pelo protocolo dos *leases*. Portanto, em uma situação como essa, o melhor é adotar o maior *lease* possível. As duas únicas limitações são que *leases* muito longos podem fazer com que o servidor tenha que adotar algum mecanismo especial para que as suas informações sobrevivam às quedas e que *leases* muito longos podem fazer com que as escritas em um arquivo fiquem suspensas por um período grande quando ocorre a queda de um cliente que possui um *lease*.

Por outro lado, podemos observar que, na situação da figura, *leases* com 60 segundos ou mais não oferecem um ganho muito maior do que *leases* de 20 ou 30 segundos, por exemplo. Assim, uma duração desta ordem seria a mais indicada.




$N = 5, R = 2$ e $W = 0,1$

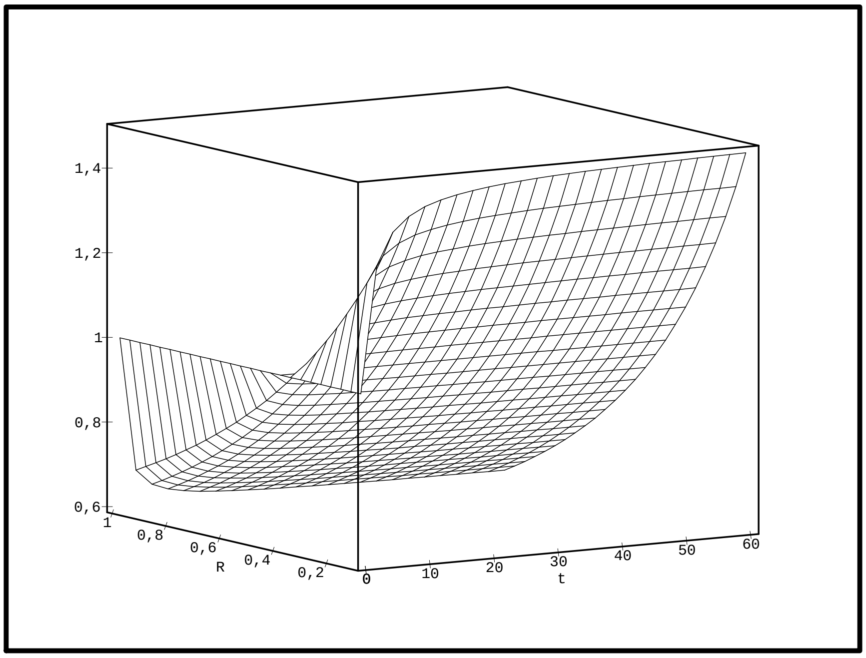
Figura 4.2: Número de mensagens geradas pelo protocolo

As figuras seguintes apresentam a razão entre o número de mensagens geradas pelos *leases* e o número de mensagens geradas pelo SPRITE.

A figura 4.3 indica que os *leases* são uma boa escolha somente se a condição 4.11 é satisfeita, isto é, $R > W(N-1)$. Se a taxa de leituras é menor do que este valor, quanto maior a duração do *lease* maior é o número de mensagens geradas pelo protocolo. Este é o ponto no qual se deve passar do protocolo dos *leases* para o do SPRITE.

Como podemos observar através da figura 4.4, o mesmo fenômeno ocorre quando aumenta o número de clientes. Quando $N > \frac{R+W}{W}$, o protocolo dos *leases* passa a ser menos vantajoso.

No entanto, o modelo supõe que todos os clientes realizam escritas com a mesma taxa.



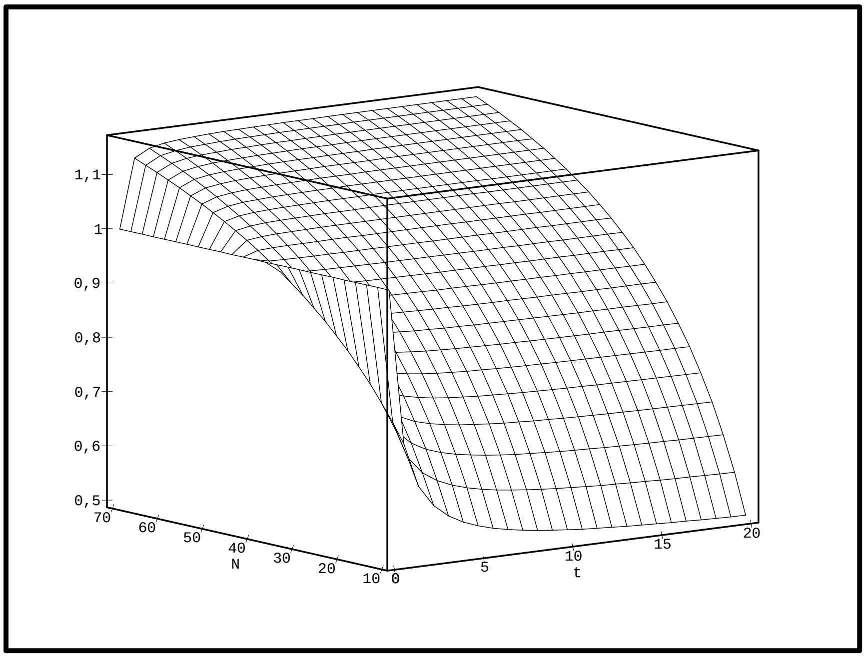

$N = 5$ e $W = 0,1$

Relação entre o tráfego gerado pelos *leases* e o tráfego gerado pelo protocolo do SPRITE com $W$ constante e $R$ crescente. Valores menores no eixo vertical indicam que o número de mensagens geradas pelo protocolo dos *leases* é menor.

Figura 4.3: Sensibilidade à relação entre R e W

São raras as situações nas quais dezenas de clientes efetuam uma escrita a cada 20 segundos em um mesmo arquivo como mostra a figura 4.4. Portanto podemos considerar como boa a escalabilidade do protocolo.

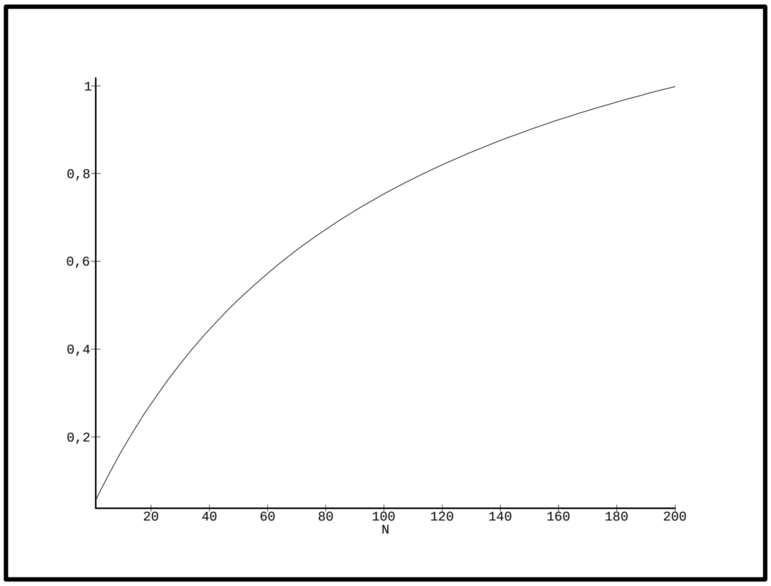
Quando $R$ é muito maior do que $W$, a vantagem do protocolo dos *leases* é, também, muito maior como podemos observar na figura 4.5. Esta figura descreve a relação entre o número de mensagens geradas pelos dois protocolos quando $R = 1, W = 0,01$ e a duração do *lease* é de 20 segundos.




$R = 1$ e $W = 0,05$

Sensibilidade dos *leases* ao aumento do número de clientes. O gráfico apresenta a relação entre o tráfego gerado pelos *leases* e o tráfego gerado pelo protocolo do SPRITE. Valores menores no eixo vertical indicam que o número de mensagens geradas pelo protocolo dos *leases* é menor.

Figura 4.4: Escalabilidade




$t_c = 20, R = 1$ e $W = 0,01$

Sensibilidade dos *leases* ao aumento do número de clientes em um ambiente com poucas escritas. O gráfico apresenta a relação entre o tráfego gerado pelos *leases* e o tráfego gerado pelo protocolo do SPRITE.

Figura 4.5: Escalabilidade com $W$ pequeno

## 4.4 SODA

A fim de estudar o comportamento de um sistema de arquivos distribuído baseado em *leases* desenvolvemos, no primeiro semestre de 1994, o SODA, um Sistema para Operação Distribuída de Arquivos para o sistema operacional LINUX. O LINUX é um clone do UNIX cujo desenvolvimento se iniciou com o estudante Linus Torvalds na Finlândia e, hoje, ocupa dezenas de programadores em todo o mundo. A coordenação dos trabalhos é feita pelo próprio Linus através da rede Internet.

Escolhemos o LINUX pois é um sistema que pode ser livremente copiado e cujo código fonte é amplamente distribuído. Além disso, o LINUX utiliza computadores pessoais da linha PC 386/486 como plataforma básica permitindo que milhões de usuários em todo o mundo tenham acesso a um bom sistema operacional a um custo muito baixo.

O SODA foi desenvolvido a partir do código do NFS implementado no LINUX por Rick Sladkey. Foi necessário realizar uma série de alterações no NFS do LINUX para implementar corretamente o protocolo dos *leases*. As principais alterações foram as seguintes:

1. O NFS do LINUX não efetua cache nos clientes, apenas nos servidores. Portanto foi necessário criar as estruturas de dados e as funções responsáveis pela manipulação do cache no núcleo do sistema operacional dos clientes.

2. Servidores NFS são livres de estado. Como o protocolo dos *leases* exigem que o servidor guarde informações sobre o acesso dos clientes aos seus arquivos, foi necessário acrescentar estruturas de dados e funções para a sua manipulação no processo servidor (*nfs daemon*).

3. Ao contrário do NFS, o protocolo dos *leases* exige que os servidores enviem mensagens para os clientes e esperem pela sua resposta. Assim, foi necessário introduzir um processo chamado *sodad* em cada máquina cliente a fim de receber, tratar e responder às mensagens dos servidores.

Veremos agora, mais detalhadamente, como esta implementação se deu.

### 4.4.1 Implementação

#### O Cache nos Clientes

Os blocos lidos através da RPC `NFSPROC_READ()` são armazenados em porções da memória virtual alocadas dinamicamente conforme a necessidade. Devido a limitações do LINUX, o NFS, e por conseguinte, o SODA trabalha com blocos de no máximo 1K. Atualmente, esta limitação está sendo superada e o SODA poderá trabalhar com blocos de até 8K oferecendo um serviço mais eficiente graças à diminuição da sobrecarga[8].

Cada arquivo cacheado por um cliente possui um registro como o da figura 4.6 contendo a dupla (número do *i-node*, nome do servidor) que identifica univocamente um arquivo, o prazo de expiração do seu *lease*, o instante da última alteração ao arquivo e um apontador para o início de uma lista linear duplamente ligada contendo os blocos do arquivo.

Para tornar o acesso aos registros dos arquivos cacheados mais rápido, eles são armazenados em uma tabela de *hash* onde o número do *i-node* é a chave para a função de *hash*[9]. Quando

[8] Embora os dados sejam transferidos em blocos de tamanho fixo, os *leases* do SODA valem para o arquivo todo.

[9] Eventualmente, dois arquivos diferentes podem possuir o mesmo número de *i-node* se estiverem em servidores distintos. Estes conflitos são resolvidos através do campo do registro que guarda o nome do servidor. Os servidores fornecem números de *i-node* distintos para cada um de seus arquivos.

```
struct cached_file {
  unsigned long inode_number;
  char *hostname;
  unsigned long lease_expiration;
  unsigned int seconds,useconds;   /* last modification time */
  struct list_node *block_list;
  struct cached_file *colision_next,
                     *colision_prev;
};
```

Figura 4.6: Registro de um arquivo no cliente

um processo solicita a leitura de um arquivo, é possível determinar se existe uma cópia válida no cache local muito rapidamente. Eventuais colisões na tabela de *hash* são tratadas através dos dois últimos campos do registro.

Suponha, por exemplo, que um processo solicite a leitura de uma porção de um arquivo cujo registro na tabela de hash indique que os seus blocos cacheados sejam válidos. Neste caso, é preciso percorrer a lista de blocos cacheados deste arquivo a fim de descobrir se os dados solicitados estão no cache ou não. Pode acontecer de apenas uma parte dos dados solicitados estarem cacheados. Neste caso, o cliente envia uma solicitação de leitura ao servidor apenas dos bytes que não estejam cacheados localmente.

Além das listas ligadas que guardam os blocos de um mesmo arquivo existe outra lista duplamente ligada que indica a ordem na qual os blocos foram acessados pela última vez. Sempre que um bloco é acessado, ele é inserido no início da lista. Assim, quando a memória termina, o bloco no final da lista é descartado. Na versão atual do SODA (0.2), o sistema reserva 2 Megabytes para o cache. Em versões futuras, pretendemos implementar uma política na qual o tamanho do cache varie dinamicamente.

No SODA 0.2, adotamos a política de *write-through* devido a sua simplicidade em relação a política de escritas retardadas. Assim, quando um processo efetua uma escrita em um arquivo remoto, ela é imediatamente enviada para o servidor e a execução do processo que solicitou a escrita só é restabelecida quando o cliente recebe do servidor o resultado de sua escrita. Portanto, é desejável que os diretórios contendo arquivos temporários sejam locais até que implementemos uma política de escritas retardadas em alguma versão futura do SODA.

Na versão 2.0 do NFS do LINUX, os clientes não cacheiam o conteúdo dos arquivo mas mantém um cache de diretórios. A versão atual do SODA utiliza o cache de diretórios implementado no NFS 2.0 e, portanto, não garante consistência perfeita no que diz respeito à atualização de diretórios. Um arquivo recém-criado por um cliente pode ficar invisível por alguns segundos para outros clientes. Este problema será resolvido em versões futuras do SODA quando o mecanismo de controle da consistência implementado para o conteúdo dos arquivos for estendido para os diretórios.

### O Servidor SODA

Um servidor SODA precisa armazenar informações sobre todos os seus arquivos que possuam uma cópia com *lease* válido em algum cliente. Isso é armazenado em uma tabela de hash análoga à tabela mantida pelos clientes. Cada registro desta tabela contém o número do *i-node*, uma lista dos clientes que possuem um *lease* para este arquivo bem como o instante da expiração destes *leases*.

Em posse destas informações, o servidor tem condições de manter a consistência entre a visão que os clientes têm do conteúdo dos arquivos.

**O Protocolo**

Realizamos duas alterações principais no protocolo NFS. A primeira delas diz respeito à função `NFSPROC_READ()`. Se um processo efetua uma solicitação de leitura de um bloco que está cacheado mas cujo *lease* está expirado, se faz necessário verificar, junto ao servidor, se a versão cacheada ainda é a mais recente. Se a versão local estiver desatualizada, é necessário trazer uma nova versão do servidor. No NFS da SUN, isto é feito em dois passos: uma chamada a `NFSPROC_GETATTR()` (lê os atributos de um arquivo) verifica se a versão no cache local esta atualizada; caso não esteja, uma outra chamada, agora a `NFSPROC_READ()`, é efetuada. No SODA, eliminamos a chamada desnecessária a `NFSPROC_GETATTR()`. Acrescentamos aos parâmetros do `NFSPROC_READ()` o instante da última atualização da cópia do arquivo armazenado no cache do cliente. Se a versão do servidor foi alterada pela última vez no mesmo instante, então a versão do cliente está atualizada e, ao invés de devolver todos os bytes da solicitação de leitura, o servidor devolve apenas uma código indicando que a versão do cliente é a mais recente.

A segunda alteração diz respeito à invalidação dos *leases*. Quando o servidor recebe uma solicitação de escrita, ele percorre as suas estruturas de dados a fim de descobrir quais clientes possuem um *lease* válido para o arquivo e envia mensagens de invalidação para estes clientes.

Se utilizássemos RPCs também para a invalidação dos *leases*, estaríamos perdendo muito tempo desnecessariamente pois teríamos que esperar pela resposta de um cliente antes de enviar uma RPC para o cliente seguinte. Esta é uma limitação da própria semântica das RPCs.

A solução encontrada foi a utilização de soquetes (*sockets*) TCP/IP na modalidade datagrama[10]. O servidor mantém abertos soquetes para os clientes que possuem leases válidos para os seus arquivos. Quando há a necessidade de invalidar os *leases* de um arquivo, o servidor envia uma mensagem para cada cliente que possui *lease* válido e passa a esperar pelas respostas.

Como o LINUX ainda não oferece a possibilidade de envio de *multicasts*, fomos obrigados a implementar o protocolo na sua versão sem *multicast*. Enviamos uma mensagem distinta para cada cliente possuidor de *lease*.

A *system call select()* do UNIX permite o recebimento das respostas dos vários clientes por um soquete único. O *select()* permite, também, que se especifique um limite de tempo máximo pelo qual o servidor irá esperar. Este limite máximo é importante para garantir que, se um cliente estiver inacessível, o servidor não ficará esperando pelas suas mensagens indefinidamente. O limite de tempo estipulado pelo servidor é exatamente o maior prazo de expiração dentre os *leases* dos clientes que ainda não responderam à mensagem de invalidação.

As mensagens enviadas pelo servidor são recebidas pelo processo *sodad* que é executado nos clientes. Este processo executa a *system call invalidate_lease()* que é o canal de comunicação entre o *sodad* – que é um processo executado a nível de usuário – e o cliente SODA – que é executado dentro do núcleo do sistema operacional. Após a conclusão da *system call*, o *sodad* envia uma mensagem ao servidor confirmando a invalidação do *lease*.

A figura 4.7 descreve o caminho percorrido por uma solicitação de leitura não atendida pelo cache seguida de uma solicitação de escrita efetuada por outro cliente. Vejamos como o SODA se comporta neste exemplo.

[10] Veja [Ste90].

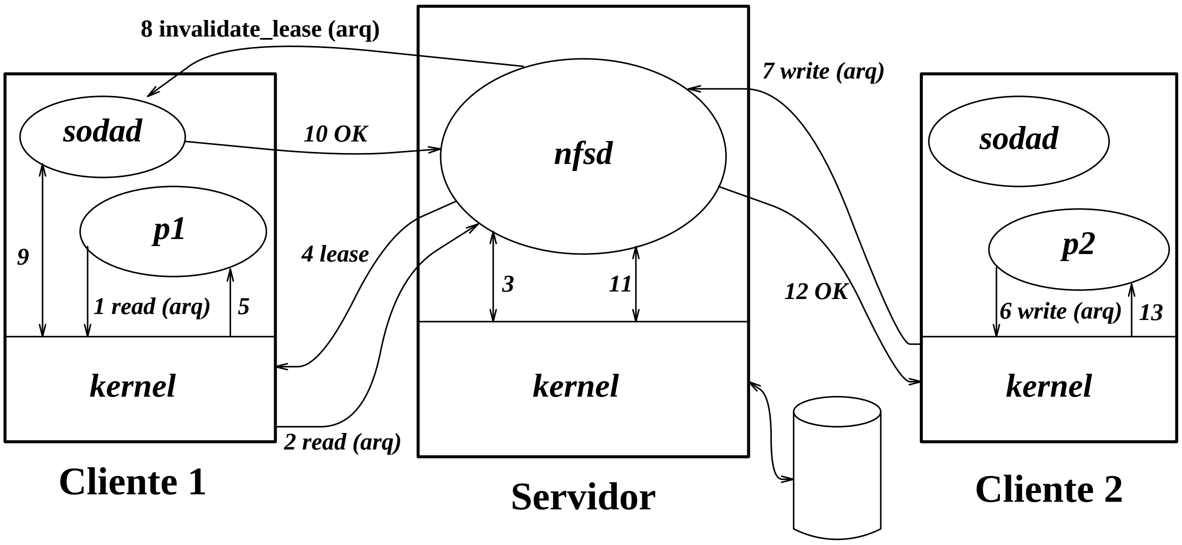



Figura 4.7: Percurso de duas solicitações ao servidor

Inicialmente, o processo *p1* executa a *system call read()* para ler alguns bytes de um certo arquivo (1). Após perceber que o arquivo solicitado é de responsabilidade de um servidor remoto, o *kernel* do cliente 1 envia a solicitação de leitura ao servidor apropriado através de uma RPC (2). Esta RPC é recebida pelo processo *nfsd* do servidor que remete a solicitação de leitura ao *kernel* local (3) que, por sua vez, acessa o disco se necessário.

Em posse dos bytes solicitados, o *nfsd* retorna a RPC devolvendo, além dos bytes solicitados, um novo *lease* para o arquivo em questão (4). O *kernel* do cliente 1 faz uma cópia dos dados recebidos em seu cache, atualiza a tabela de *leases* e retorna a *system call* efetuada por *p1* (5).

Se, enquanto o *lease* de *p1* para este arquivo ainda é válido, um processo *p2* em outro cliente efetua uma solicitação de escrita (6), então ocorre o seguinte.

Ao receber a solicitação de escrita em um arquivo remoto, o *kernel* do cliente 2 envia uma RPC ao processo *nfsd* do servidor apropriado (7). Após receber a solicitação de escrita, o *nfsd* consulta a sua tabela de *leases* concedidos e descobre que o cliente 1 possui um *lease* válido para o mesmo arquivo. Neste instante, *nfsd* envia uma mensagem de invalidação do *lease* para o processo *sodad* (8).

Logo que recebe a solicitação do *nfsd* do servidor, *sodad* executa a *system call invalidate_lease()* (9) fazendo com que o *kernel* invalide o *lease* para este arquivo. Assim que a *system call* é completada, *sodad* envia uma mensagem para o servidor confirmando a invalidação (10).

Somente neste instante, *nfsd* efetua a escrita junto ao seu *kernel* (11) e retorna a RPC devolvendo o resultado da escrita (12). Finalmente, ao receber o resultado da RPC, o *kernel* do cliente 2 conclui a *system call* efetuada pelo processo *p2* devolvendo o resultado da escrita (13).

### 4.4.2 Análise do Desempenho

A fim de analisar o desempenho do SODA realizamos dois tipos de testes. O primeiro deles mede a eficiência da implementação do cache do SODA em relação ao NFS do LINUX.

A tabela 4.2 mostra o tempo necessário para a leitura seqüencial, sem concorrência, de arquivos de diversos tamanhos em 5 situações: LINUX NFS com o cache do servidor vazio, SODA com caches vazios, LINUX NFS com o cache do servidor cheio, SODA com cache do

| Número de bytes do arquivo | Linux NFS cache vazio | SODA caches vazios | Linux NFS cache cheio | SODA cache do cliente vazio | SODA caches cheios |
|---|---|---|---|---|---|
| 512 | 38 (027) | 40 (029) | 16 (0000) | 18 (002) | 1,4 (00,4) |
| 1024 | 39 (009) | 41 (003) | 18 (0000) | 36 (023) | 1,6 (00,5) |
| 2048 | 53 (009) | 62 (004) | 39 (0011) | 53 (033) | 1,8 (00,4) |
| 4096 | 104 (002) | 140 (056) | 82 (0034) | 91 (024) | 2,8 (00,4) |
| 8192 | 193 (033) | 198 (020) | 148 (0042) | 173 (016) | 3,8 (00,4) |
| 16384 | 330 (024) | 360 (008) | 273 (0040) | 363 (044) | 6,0 (00,0) |
| 32768 | 662 (096) | 684 (068) | 625 (0067) | 709 (194) | 10,6 (00,5) |
| 65536 | 1431 (132) | 1543 (291) | 1128 (0235) | 1483 (057) | 19,0 (00,0) |
| 1Mega | 20175 (886) | 21539 (540) | 17860 (1135) | 18409 (475) | 1037,3 (42,2) |

Tempo em milisegundos

Tabela 4.2: Leitura seqüencial sem concorrência

servidor cheio e cache do cliente vazio e, finalmente, SODA com cache do cliente cheio.

Estes valores foram coletados em um cliente PC-486 acessando arquivos de um servidor PC-386 através de uma rede Ethernet. Cada valor da tabela é a média dos resultados obtidos em cinco ativações do teste. Os números entre parênteses representam o desvio padrão encontrado.

Comparando as duas primeiras colunas da tabela 4.2 podemos perceber que a perda introduzida pela maior complexidade do SODA é pequena. O tempo gasto com a administração do cache faz com que o SODA apresente, em média, um desempenho 10% pior do que o NFS quando o cache do cliente não contém os dados solicitados.

Comparando as colunas três e quatro percebemos que a sobrecarga introduzida pela administração do cache chega a algo em torno de 20% em média.

No entanto, esta sobrecarga é totalmente compensada com o ganho obtido na próxima leitura ao mesmo arquivo pois, se compararmos as colunas três e cinco, percebemos que a leitura de dados cacheados pelos clientes chega a ser várias dezenas de vezes mais rápida no SODA do que no NFS do LINUX que não possui cache nos clientes. A leitura de 1 Megabyte, por exemplo, gasta 18 segundos no NFS do LINUX e apenas 1 segundo em um cliente SODA com o cache cheio.

Já a tabela 4.3 mostra o tempo gasto em escritas seqüenciais sem concorrência no NFS do LINUX e no SODA. Não detectamos nenhuma perda significativa no desempenho do SODA.

Para o segundo teste, escrevemos dois programas que realizam leituras e escritas em um determinado arquivo obedecendo a uma distribuição de Poisson com taxa de leitura $R$ e taxa de escrita $W$, respectivamente.

A fim de comparar o desempenho do nosso sistema com o do SPRITE, construímos, a partir do SODA, um sistema que adota, em todas as situações, o protocolo adotado pelo SPRITE quando há compartilhamento com escrita[11].

A figura 4.8 mostra o desempenho de um servidor PC-486 atendendo a leituras e escritas de três clientes (dois PC-486 e um PC-386). A taxa de leituras nos três clientes é de um arquivo de 1Kbyte por segundo (obedecendo a uma distribuição de Poisson) e a taxa de escritas varia de 0 a 0,7 como indica a abcissa do gráfico. A curva rotulada SODA descreve a carga gerada pelo processo servidor de arquivos do SODA com *leases* durando 20 segundos.

[11] Devido a indisponibilidade do equipamento necessário, não pudemos realizar testes com o SPRITE real. Portanto as comparações aqui apresentadas são entre o protocolo do SODA e o protocolo do SPRITE e não entre as implementações destes sistemas.

| Número de bytes do arquivo | Linux NFS | SODA |
|---|---|---|
| 512 | 18 (0004) | 17 (0000) |
| 1024 | 18 (0000) | 25 (0011) |
| 2048 | 45 (0020) | 35 (0000) |
| 4096 | 66 (0000) | 67 (0000) |
| 8192 | 132 (0004) | 131 (0001) |
| 16384 | 257 (0002) | 261 (0002) |
| 32768 | 507 (0002) | 682 (0121) |
| 65536 | 1.122 (0173) | 1.057 (0023) |
| 1Mega | 18.232 (1328) | 18.617 (1771) |

Tempo em milisegundos

Tabela 4.3: Escrita seqüencial sem concorrência

A curva rotulada sSPRITE apresenta o mesmo tipo de carga para o SPRITE simulado[12].

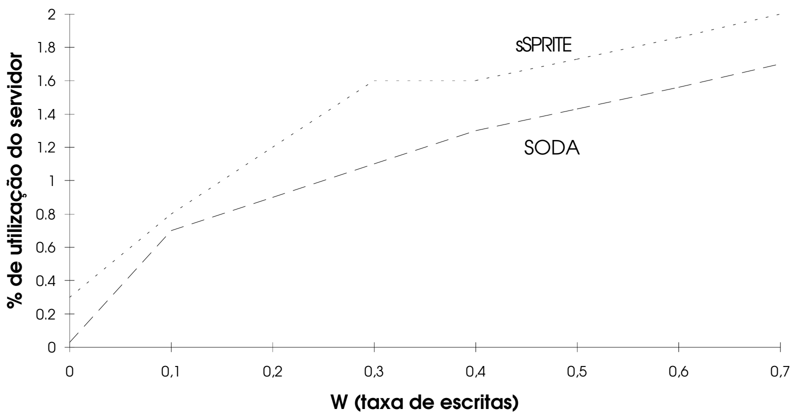



Figura 4.8: Utilização da CPU do servidor

Como podemos notar através da figura 4.8, mesmo com $R = 1$ e $W = 0,7$, a carga no servidor foi menor no SODA do que no SPRITE simulado. Estes valores não satisfazem a inequação 4.11 da seção 4.3.1 e, portanto, o nosso modelo analítico indica que o servidor SODA atende um número maior de mensagens do que o SPRITE.

Esta aparente contradição – ou seja, o modelo indica um número maior de mensagens mas a prática mostra uma carga menor – reside no fato de que a invalidação dos *leases* foi implementada através de soquetes do tipo datagrama que geram menos carga no servidor do que as RPCs utilizadas para a verificação da validade do arquivo cacheado.

Assim, mesmo atendendo a um número maior de mensagens, o servidor do SODA é menos exigido. Por outro lado, se a taxa de escritas é muito maior do que a taxa de leituras, o servidor SODA é mais exigido do que o SPRITE simulado.

Executamos um experimento com três clientes acessando 10 arquivos de 1Kbyte cada e

[12]Não estamos considerando aqui a carga gerada pelo processo *top* que é utilizado para examinar a carga na CPU.

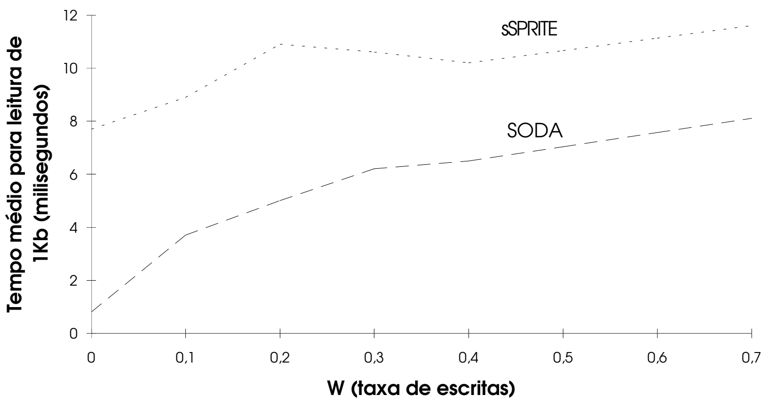



Figura 4.9: Tempo para efetuar a leitura de 1Kbyte

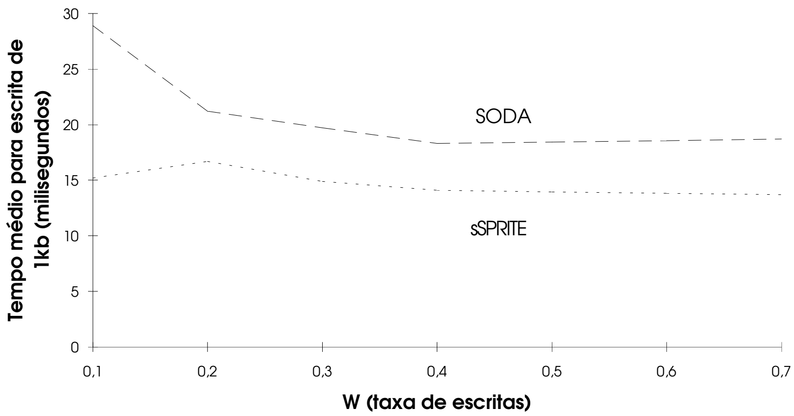



Figura 4.10: Tempo para efetuar uma escrita de 1Kbyte

com $R = 1$ e $W = 2$ para cada arquivo. O servidor SODA ocupou, em média, 38,3% da CPU enquanto que o SPRITE simulado ocupou apenas 33,8%, ou seja, uma carga 13% menor.

Já as figuras 4.9 e 4.10 mostram o tempo médio necessário para efetuar leituras e escritas de 1Kbyte nos dois sistemas sob a mesma carga da figura 4.8. Podemos notar que as leituras no SODA conseguem ser muito mais rápidas do que no SPRITE simulado principalmente quando a taxa de escritas é pequena. Neste caso, uma grande parte das leituras é completada sem a necessidade de comunicação com o servidor.

No entanto, as escritas no sSPRITE são, por definição, mais rápidas do que no SODA pois não existe a necessidade de invalidar *leases*. Desta forma, as escritas são imediatamente completadas. A figura 4.10 mostra o custo adicional destas invalidações.

Finalmente, a figura 4.11 mostra a influência da duração dos *leases* na carga gerada pelo processo servidor. Esta figura apresenta a carga gerada por processos sendo executados em três clientes distintos. Cada cliente executa 10 pares de processos; cada par de processos efetua leituras e escritas com taxas 2 e $0,01$, respectivamente, em um arquivo diferente. O servidor trata de 10 arquivos, cada um sendo acessado concorrentemente pelos três clientes.

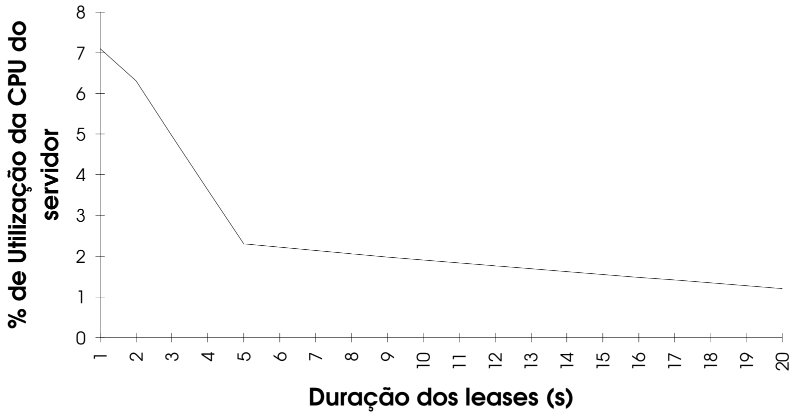
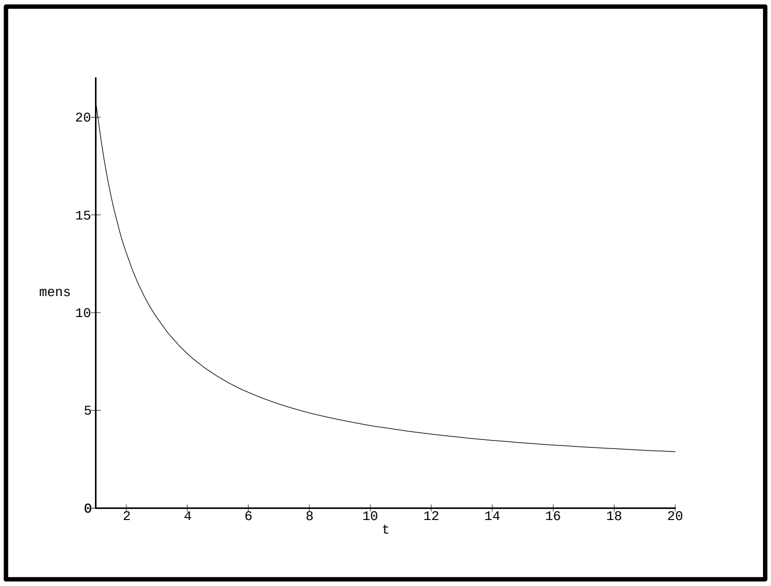



Figura 4.11: Carga na CPU × duração dos *leases*

Já a figura 4.12, por sua vez, apresenta a estimativa do nosso modelo analítico para o número de mensagens tratadas pelo servidor para a carga descrita. É possível notar a semelhança entre o formato dos dois gráficos.




Estimativa do modelo analítico para o número de mensagens tratadas pelo servidor para a carga descrita.

Figura 4.12: Número de mensagens × duração dos *leases*

### 4.4.3 SODA Adaptativo?

As conclusões extraídas do modelo analítico apresentado na seção 4.3 nos levavam à idéia da implementação de um sistema adaptativo. Em posse das taxas de leituras e de escritas de cada arquivo, o sistema decidiria, baseado na inequação 4.11, se o protocolo dos *leases* deveria ser aplicado ou não. Ou seja, se $R > W(N-1)$ aplicar-se-ia o protocolo dos *leases*; caso contrário, o cache para este arquivo seria desabilitado quando houvesse compartilhamento com escrita.

No entanto, o fato de termos utilizado soquetes ao invés de RPCs para a invalidação dos *leases* fez com que o SODA apresentasse um desempenho superior ao previsto pelo modelo analítico. Desta forma, ao adicionarmos o esquema adaptativo ao SODA, devemos levar em conta a diferença entre a carga gerada por uma RPC e por uma mensagem datagrama. Considerando esta diferença, a utilização dos *leases* seria ainda mas recomendável.

Um SODA adaptativo teria que coletar informações sobre o acesso ao sistema de arquivos a fim de estimar os valores de $R$ e $W$ e armazenar estas informações em disco ou na memória do servidor. Esta sobrecarga (gerada pela manutenção das informações sobre o acesso aos arquivos) tende a depreciar o desempenho do sistema e só valeria a pena em ambientes específicos onde uma boa parte dos arquivos tivesse uma taxa de escritas grande em relação à taxa de leituras.

Como a taxa de escritas costuma ser muito menor do que a taxa de leituras, optamos, no momento, por não implementar a versão adaptativa do SODA ficando livre da sobrecarga que ela geraria. No entanto, em ambientes especiais, uma versão adaptativa do SODA poderia apresentar resultados positivos.

Apesar de não termos utilizado o modelo analítico para implementar um SODA adaptativo, ele é útil para sabermos como se comporta o SODA sob diferentes cargas e em redes de diferentes escalas. O modelo ainda nos permite inferir algumas situações nas quais a implementação de um SODA adaptativo poderia trazer bons resultados.

### 4.4.4 Conclusão

Dentre os sistemas de arquivos distribuídos em operação, o SPRITE (e seus descendentes) é um dos que oferecem o serviço mais veloz. No entanto, um dos pontos fracos do protocolo do SPRITE é o fato dele desabilitar o cache nos clientes quando um arquivo é aberto por mais de um cliente e aberto para escrita em pelo menos um deles.

Neste capítulo, mostramos que o protocolo dos *leases* pode ser uma boa alternativa para melhorar o desempenho do sistema de arquivos em ambientes onde o acesso concorrente com escrita é freqüente. Em ambientes onde tal tipo de acesso é raro, ambos os protocolos se saem bem. Vimos que, além de uma melhor utilização do cache, os *leases* oferecem ainda uma maior tolerância a falhas.

Tanto o modelo analítico da seção 4.3 quanto os dados coletados no SODA indicam um ganho significativo no desempenho em diversas situações.

Outra desvantagem do sistema de arquivos do SPRITE é o fato da comunicação cliente/servidor ter sido implementada em um nível muito baixo o que dificulta o desenvolvimento de clientes e servidores que possam se comunicar com máquinas SPRITE. Como vimos na seção 2.3, o principal motivo da disseminação do NFS foi a adoção de um protocolo simples e aberto para a comunicação cliente/servidor.

A implementação do protocolo dos *leases* no SODA foi realizada alterando o protocolo NFS implementado no LINUX. O desenvolvimento de clientes e servidores capazes de se comunicar com as máquinas SODA é uma tarefa relativamente simples desde que se tenha disponíveis RPCs e soquetes datagrama TCP/IP. Caso seja possível partir de uma implementação do NFS, o trabalho é facilitado.

Durante o desenvolvimento do SODA enfrentamos vários obstáculos que nos mostraram algumas das dificuldades encontradas no desenvolvimento de sistemas distribuídos. O primeiro obstáculo foi a baixa qualidade da documentação do núcleo do LINUX. Gastamos algumas semanas na compreensão das diversas camadas que compõem o cliente NFS, o que foi dificultado, também, pela má modularização do sistema.

Depois de compreender como o sistema antigo funcionava escrevemos o código do novo sistema procurando não cometer os mesmos erros. Procuramos estruturar as novas funções de forma modular e documentamos tanto o código novo quanto uma parte do código do NFS do LINUX.

No entanto, foi na fase de depuração que o caráter distribuído do sistema trouxe maiores problemas. Não tivemos acesso a um depurador de núcleo (*kernel debugger*) e tínhamos que rastrear a execução do sistema em até quatro máquinas simultaneamente. Quando encontrávamos um erro, era necessário corrigi-lo, recompilar o núcleo, instalar a nova versão em todos os clientes e reinicializá-los. Todo este ciclo seria muito mais rápido e simples se estivéssemos trabalhando em um sistema centralizado.

Esta dificuldade nos despertou para a importância da elaboração de ambientes de desenvolvimento de sistemas distribuídos que pudessem facilitar este trabalho. Abordaremos esta questão novamente na seção 5.1.

### 4.4.5 Trabalho Futuro

Existem alguns pontos importantes tanto no modelo analítico quanto na implementação do SODA que ainda podem ser melhorados.

O modelo analítico apresentado na seção 4.3 analisa o SPRITE apenas em instantes de acesso concorrente com escrita. A fim de representar mais fielmente o comportamento do SPRITE em todas as situações, teríamos que incluir, no nosso modelo, as ocorrências dos comandos *open* e *close* que determinam a política de cache adotada pelo SPRITE. Esta alteração aumentaria em muito a complexidade do modelo mas seria um bom caminho para pesquisas futuras na medida em que possibilitaria uma compreensão mais profunda do comportamento do SPRITE.

Em relação ao SODA, muita coisa ainda pode ser melhorada. As primeiras modificações seriam as seguintes:

- Efetuar a transferência de dados através de **blocos de 8Kbytes** ao invés de blocos de 1Kb. Isto já está sendo implementado no NFS do LINUX e poderá ser transportado ao SODA de maneira trivial.
- Utilização da técnica de **read-ahead**[13]. Também está sendo implementado no NFS do LINUX e poderá ser transportado ao SODA de maneira trivial.
- Utilização de *leases* para garantir a **consistência dos diretórios e atributos**.
- Implementação de **write-behind** assim como descrevemos em 4.1.
- Quando o protocolo da rede permitir, deveremos implementar a invalidação dos *leases* através de **multicasts**.
- Estender o protocolo a fim de tratar consistentemente as escritas em arquivos abertos no **modo append** (o NFS não oferece este tipo de recurso).
- Permitir ao cache a utilização de toda a memória disponível.

[13]Ver seção 2.3.3.

- Implementação da **versão adaptativa**.
- Criação e divulgação de um pacote de distribuição do SODA de modo que qualquer usuário possa instalá-lo no sistema LINUX.

# Capítulo 5

# O Futuro

## 5.1 Flexibilidade

A maior dificuldade para o desenvolvimento de sistemas operacionais e, em particular, sistemas de arquivos distribuídos é a implementação dos algoritmos teóricos nas redes de computadores reais. Uma idéia muito simples como os *leases* exigiu o trabalho de um programador durante mais de dois meses para a implementação de um protótipo satisfatório. A implementação de um sistema operacional distribuído completo como o SPRITE exigiu anos de trabalho de uma grande equipe de estudantes de pós-graduação de Berkeley.

Se o esforço necessário para a implementação de novos mecanismos fosse menor, poder-se-ia investir mais tempo no desenvolvimento de novas idéias. Se fosse possível avaliar o desempenho de um algoritmo logo após a sua elaboração, o desenvolvimento dos sistemas operacionais seria muito mais rápido e, hoje, teríamos sistemas muito mais eficientes.

Esta é uma possibilidade que está sendo buscada através da aplicação de metodologias avançadas de desenvolvimento de software como, por exemplo, o paradigma de objetos. O SPRING é um Sistema Operacional Distribuído Orientado a Objetos que está sendo desenvolvido nos laboratórios da *SUN Microsystems*.

O SPRING é um sistema *multi-threaded* que tem como principal objetivo ser facilmente expandível. O sistema é baseado em uma estrutura denominada **objeto** que é composta pelo seu estado (conjunto de dados estruturados) e por métodos (funções) que manipulam o seu estado.

Juntamente com o núcleo, o administrador de memória virtual (VMM) oferece a base para todos os serviços oferecidos pelo sistema distribuído (figura 5.1). O núcleo do SPRING é um *MicroKernel* que gerencia a implementação de objetos baseado em *threads* e na sua interface com o VMM.

Os serviços são implementados de maneira essencialmente modular; toda a comunicação com outros serviços ou com aplicações SPRING é feita através de uma interface muito bem definida através da **linguagem de definição de interfaces**.

Utilizando este modelo, deseja-se construir um sistema operacional que tenha como principal característica a sua flexibilidade. A implementação de um novo serviço ou a alteração de um serviço existente seria muito mais simples uma vez que o SPRING é completamente modularizado e as interfaces entre os módulos são definidas claramente.

Sobre a base do SPRING já foi implementado um sistema que emula um subconjunto do UNIX [KN92] possibilitando que uma grande quantidade de aplicativos UNIX sejam executados e que processos UNIX interajam com processos SPRING.

[NKM93] descreve a construção do SPRING FILE SYSTEM que não traz nenhuma novidade em termos algorítmicos mas que mostra como os diversos aspectos de um sistema de

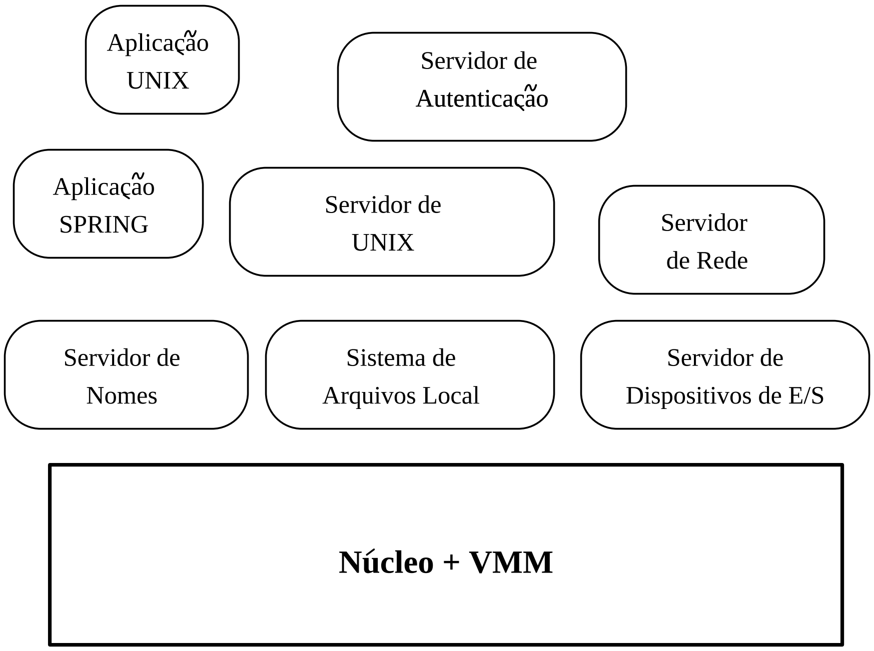



Figura 5.1: Estrutura do SPRING

arquivos podem ser tratados separadamente por módulos independentes. O sistema é construído, a partir de um administrador de memória virtual, por um módulo de manipulação de arquivos locais, um administrador do cache, um administrador de nomes, um administrador de réplicas. As funções são distribuídas entre os módulos que interagem através de interfaces explicitamente definidas. Através de "subcontratos" [HPM93] é obtida uma certa flexibilidade na interação entre os módulos.

Já o artigo [KN93] mostra como os programadores podem facilmente estender o sistema de arquivos do SPRING para desempenhar outras funções. Em posse da especificação precisa da interface de cada módulo que forma o SPRING FILE SYSTEM, o programador pode construir novos módulos que funcionariam como uma nova camada sobre as camadas fornecidas pela configuração básica do SPRING.

A figura 5.2 mostra como seria possível implementar vários sistemas de arquivos "empilhados". O SPRING permite que este empilhamento se faça tanto de maneira estática (determinada no momento da compilação do sistema) quanto dinâmica, isto é, uma nova camada pode ser criada em função da execução de uma determinada aplicação e, se for desejável, desaparecer quando esta aplicação é encerrada.

Nesta figura, vemos um esquema da implementação de um sistema de arquivos distribuído fictício. O módulo *Replicador* replica os arquivos em dois sistemas, o *Compressor* que cuida da compressão e descompressão dos arquivos do *UNIX FS* e o *Berkeley FFS* que acessa o disco diretamente. O *Replicador* envia as solicitações também para o *log* que as registra em fita magnética.

Sobre o replicador existe a camada do *Distribuidor* que é responsável por oferecer o serviço de arquivos aos clientes remotos. A arquitetura do SPRING permite que estes módulos sejam implementados em máquinas distintas e que a comunicação entre elas seja transparente para os módulos.

Um sistema deste tipo facilitaria enormemente o desenvolvimento de sistemas operacionais

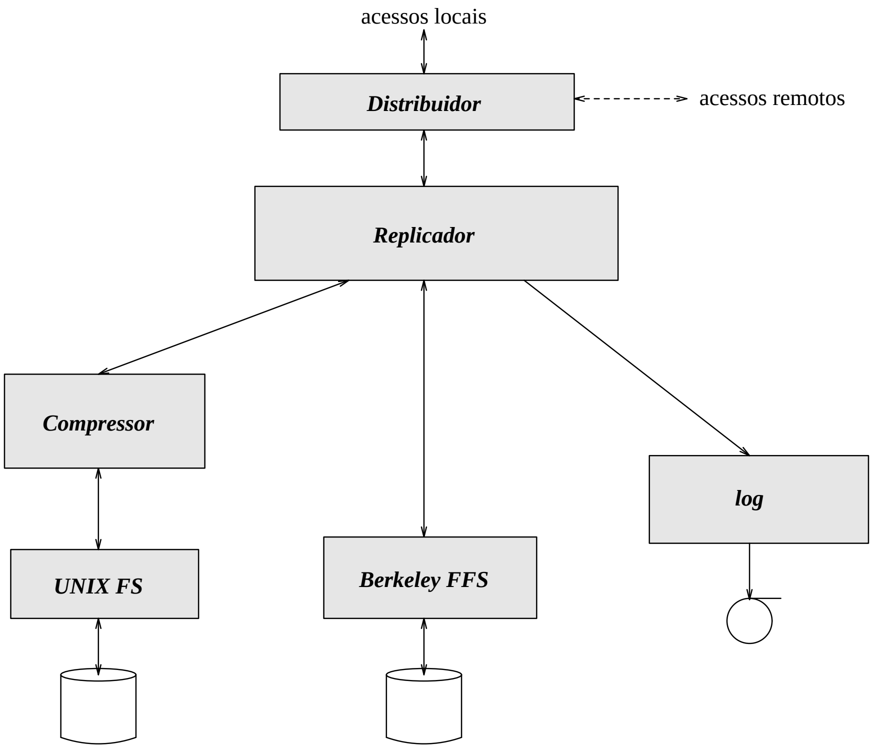



Figura 5.2: Sistema de arquivos fictício construído no SPRING

distribuídos. Mas, ao mesmo tempo em que facilita o trabalho do programador, o alto grau de modularização prejudica a eficiência do sistema. Se cada módulo da figura 5.2 mantivesse um cache próprio, haveria uma enorme redundância de informações na memória principal dos servidores. Por outro lado, se os vários módulos compartilhassem o mesmo cache, a interface entre os módulos tornar-se-ia mais complexa ou o sistema teria que perder a modularidade. Em ambos os casos é também necessário administrar a consistência entre as visões que diferentes módulos têm do cache.

O sistema operacional é um recurso crítico para o desempenho de qualquer coisa que se queira fazer com um computador ou com uma rede de computadores. Por isso, ele deve ser escrito de modo a ser o mais eficiente possível pois uma rotina mal escrita pode implicar em um fraco desempenho de todas as aplicações executadas no sistema.

Sistemas modulares desenvolvidos a partir de conceitos de engenharia de software freqüentemente apresentam um desempenho mais baixo do que sistemas desenvolvidos através de técnicas convencionas. O *SPRING File System*, por exemplo, apresenta um desempenho de 2 a 7 vezes pior do que o sistema de arquivos do SunOS 4.1.3. Os projetistas do SPRING argumentam que isto é compreensível uma vez que o SunOS é um produto comercial muito bem configurado enquanto que o *SPRING File System* é apenas um protótipo em fase de desenvolvimento [NKM93].

Agora, resta esperar pelos próximos resultados nesta área para descobrir se é ou não possível o desenvolvimento de sistemas tão críticos quanto os sistemas operacionais através de metodologias como a programação orientada a objetos de modo a construir sistemas modulares e facilmente extensíveis.

## 5.2 O Sistema de Arquivos Físico Ideal

[MP91] apresenta um esquema muito interessante para a implementação de um servidor de arquivos muito eficiente. A idéia consiste em um sistema de arquivos multi-estruturado onde as operações com o sistema de arquivos são divididas em três classes e cada classe é tratada diferentemente.

Os diretórios são armazenadas em uma matriz de discos pequenos (RAID) cada um contendo uma réplica de todos os diretórios. Desta forma é possível atender a várias solicitações de diretórios simultaneamente.

Os arquivos são listrados em outra matriz de discos pequenos a fim de permitir a transferência dos arquivos em paralelo. Todas as operações que modificam o sistema de arquivos são imediatamente transferidas para um sistema de arquivos baseado em *log* e, só posteriormente, para as matrizes redundantes.

Este sistema, desenvolvido na Universidade da California em San Diego, consegue uma rapidez muito grande no acesso aos arquivos e pode ser utilizado por qualquer sistema de arquivos distribuído. O grande inconveniente é a alta complexidade do servidor e a necessidade de hardware específico.

Uma outra alternativa para o aumento da eficiência dos servidores de arquivos, seria a utilização de memória RAM não-volátil (NVRAM) ao invés de discos magnéticos. Mas o preço destas memórias ainda são muito altos.

## 5.3 Enfim, o que Esperamos

Após a análise dos principais avanços na pesquisa em sistemas de arquivos nos últimos anos, podemos ressaltar alguns pontos que deveriam fazer parte de um sistema de arquivos distribuído ideal. Apresentamos, aqui, um resumo das características desejáveis independentemente das dificuldades encontradas na sua implementação.

- **Transparência**: o sistema de arquivos deve facilitar a interação dos usuários e programadores escondendo aspectos como a localização dos dados, tipo de rede, tipo de sistema operacional, ocorrência de falhas.

- **Escala Mundial**: sistemas como o AFS facilitam o compartilhamento de arquivos em diferentes partes do globo. Futuramente, a interconexão de todos os computadores do universo conhecido poderá ser uma realidade e os sistemas de arquivos poderão ser uma importante ferramenta para o compartilhamento de informações entre eles.

- **Semântica UNIX**: a semântica UNIX permite que o sistema de arquivos seja utilizado como um instrumento muito versátil para o compartilhamento consistente de informações entre diferentes processos. No entanto, oferecer semântica UNIX e escala mundial simultaneamente é uma tarefa extremamente difícil. Um sistema que oferecesse estes dois recursos seria muito ineficiente.

  Uma possível solução para este impasse seria possibilitar a convivência de diferentes semânticas de acesso concorrente. Alguns diretórios de uso local obedeceriam à semântica UNIX enquanto que diretórios mundialmente compartilhados obedeceriam a semânticas menos restritivas.

- **Replicação Automática**: além de aumentar a disponibilidade e a confiabilidade do sistema, permite que os clientes acessem as cópias mais próximas.

- **Listramento**: distribui eqüitativamente a carga entre os servidores e permite que os servidores transfiram os dados concorrentemente.

- **Sistemas Baseados em Log**: Eliminam a demorada verificação no momento da reinicialização e o tempo para o *seek* nas escritas.

- **Modularização**: Facilita o desenvolvimento dos sistemas e permite a confecção de sistemas flexíveis.

- **Compressão Automática**: Com o aumento da capacidade de processamento dos microprocessadores em relação aos discos torna-se vantajoso compactar os dados antes de enviá-los aos discos. Diminui-se o espaço ocupado e a transferência leva menos tempo [Dou93].

Como observamos no caso da semântica UNIX e da escala mundial, muitas destas características são contraditórias o que torna a sua reunião em um único sistema uma tarefa dificílima e não necessariamente possível. Devemos trabalhar, meditar e aguardar.

# Apêndice A

# O Sistema de Arquivos UNIX

No sistema UNIX, um **arquivo** nada mais é do que uma seqüência de zero ou mais bytes não interessando, para o sistema, qual o significado e a estrutura do seu conteúdo. Textos, programas, dados numéricos ou qualquer outro tipo de informação são armazenados da mesma maneira.

A fim de permitir a organização do sistema de arquivos por parte dos usuários, o UNIX permite que os arquivos sejam armazenados em diretórios ou em diretórios dentro de diretórios, os subdiretórios.

O nome dos arquivos e diretórios são compostos por seqüências de até 14 ou 255 caracteres dependendo da versão do UNIX. Os diretórios são implementados como arquivos comuns; um bit determina se um arquivo é um diretório ou não.

O *pathname* de um arquivo é uma seqüência de nomes de diretórios separados pelo caractere / e finalizada pelo nome do arquivo. A sua finalidade é identificar a localização lógica de um arquivo dentro do espaço de nomes criado pelos administradores e usuários.

Ao contrário de outros sistemas onde os diretórios sempre formam uma árvore, o UNIX oferece a possibilidade de um diretório ter como seu filho um de seus ancestrais. Isto é feito através do comando *link* que associa um novo arquivo (ou diretório) a um arquivo (ou diretório) existente.

O UNIX adota um método baseado em permissões para controlar o acesso aos arquivos. Cada usuário possui um código, *uid*, que o identifica univocamente. Existe também o conceito de grupo de usuários, que também são identificados por um código, *gid*. Entre as informações que o sistema guarda sobre cada arquivo se encontram os códigos do grupo e do usuário ao qual o arquivo pertence. Além disso, o sistema armazena, para cada arquivo, uma seqüência de 9 bits que informa quem tem permissão para fazer o que com aquele arquivo. Esses bits são divididos em três grupos de três bits cada.

Os três primeiros bits indicam se o dono do arquivo tem permissão para ler, escrever ou executar o arquivo, respectivamente. O segundo grupo de bits guarda as mesmas permissões de leitura, escrita e execução para os usuários que pertençam ao grupo do arquivo. Os três últimos bits indicam quais são as permissões para os demais usuários[1].

Embora cada sistema de arquivos possa ser configurado de acordo com a vontade do administrador, existe um padrão para a localização lógica dos arquivos de uso coletivo. O diretório */bin*, por exemplo, guarda os arquivos executáveis correspondentes a utilitários do UNIX como *cat, cc, cp, date, diff, find, grep, ls, make, pwd, rm, sh, sort, tail, wc*. Na tabela A.1 vemos alguns dos diretórios mais importantes do UNIX.

Os arquivos são divididos em blocos que são armazenados em posições não necessariamente

[1] Se o arquivo corresponder a um diretório, então os bits de permissão de execução são interpretados como permissão de entrada no diretório.

| | |
|---|---|
| /bin | utilitários do UNIX |
| /lib | bibliotecas |
| /tmp | arquivos temporários |
| /usr | arquivos dos usuários |
| /usr/bin | outros arquivos executáveis de uso público |
| /usr/include | *header files* das bibliotecas |
| /usr/man | manuais |
| /usr/src | arquivos contendo programas fonte |
| /usr/spool | arquivos temporários das filas de impressão, dos sistemas de correio, de notícias |
| /dev | arquivos especiais: interfaces para os dispositivos de entrada e saída |

Tabela A.1: Principais diretórios do UNIX

contíguas do disco. Um bloco pode conter 256, 512, 1024 bytes, dependendo da versão do UNIX. Além dos blocos com o seu conteúdo, cada arquivo possui um bloco chamado nó índice (*i-node*) que contém uma série de informações sobre o arquivo como mostra a seguinte tabela.

| |
|---|
| bits de permissão |
| outros bits |
| *uid* |
| *gid* |
| tamanho |
| número de *links* |
| instante da criação |
| instante do último acesso |
| instante da última alteração |
| endereço dos 10 primeiros blocos |
| apontador indireto para blocos |
| apontador duplamente indireto |
| apontador triplamente indireto |

Tabela A.2: Conteúdo de um *i-node*

A figura A.1 mostra como é o funcionamento dos apontadores citados na tabela A.2. No próprio *i-node* há espaço para indicar a localização, no disco, dos 10 primeiros blocos do arquivo. Se o arquivo possuir mais de 10 blocos, então um apontador indireto indicará o endereço de um bloco contendo uma lista de apontadores para os blocos seguintes do arquivo.

Se, por exemplo, forem necessários 4 bytes para endereçar um bloco do disco e cada bloco possuir 1 Kbyte, então um bloco poderá guardar até 256 apontadores para os outros blocos do arquivo. Se um arquivo possuir mais de 266 Kbytes, o apontador duplamente indireto apontará para um bloco contendo uma lista de até 256 apontadores para listas de apontadores indiretos. Os apontadores triplamente indiretos funcionam de maneira análoga

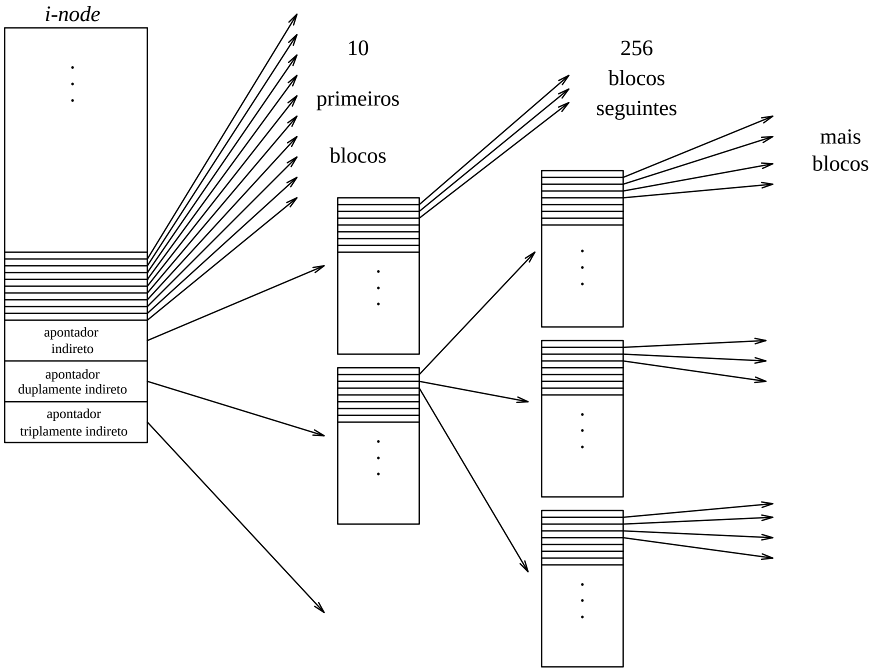



Figura A.1: Apontadores para os blocos de um arquivo UNIX

permitindo que um arquivo UNIX possa ser formado por até $10 + 256 + 256^2 + 256^3$ blocos que é um número muito maior do que o número de blocos dos discos atuais.

Uma tabela de arquivos abertos é mantida na memória principal de toda máquina UNIX guardando informações sobre o acesso a cada arquivo aberto. Ela guarda o modo no qual o arquivo foi aberto (leitura, escrita, ambos ou concatenação), o índice da próxima posição do arquivo a ser acessada e um apontador para uma cópia do i-node que é mantida na memória principal. Cada processo possui uma tabela de descritores de arquivos que nada mais é do que uma lista de apontadores para a tabela de arquivos abertos como mostra a figura A.2.

Os descendentes de um processo (criados através da chamada *fork()* compartilham a mesma entrada na tabela de arquivos abertos e, portanto, o mesmo apontador para a próxima posição do arquivo a ser acessada.

Os diretórios são implementados através de arquivos contendo uma série de pares *(nome_do_arquivo, número_do_i-node)*.

A fim de diminuir o tempo de acesso ao sistema de arquivos UNIX, cada vez que um bloco é lido do disco, ele é armazenado no *buffer cache* mantido na memória principal. Se um bloco cuja leitura foi solicitada estiver no *buffer cache* ele é devolvido para a aplicação sem a necessidade de se acessar o disco. As escritas não são enviadas para o disco imediatamente. Em geral, permanecem no *buffer cache* por um período de 20 segundos e, só então, são enviados para o disco. Opcionalmente, pode-se ordenar que os blocos ainda não gravados sejam enviados imediatamente para o disco através do comando *sync*. O *buffer cache* armazena tanto blocos de dados quanto *i-nodes* e diretórios.

Para maiores detalhes sobre o funcionamento e a implementação não só do sistema de arquivos UNIX, mas também os seus outros componentes, consulte [Bac86].

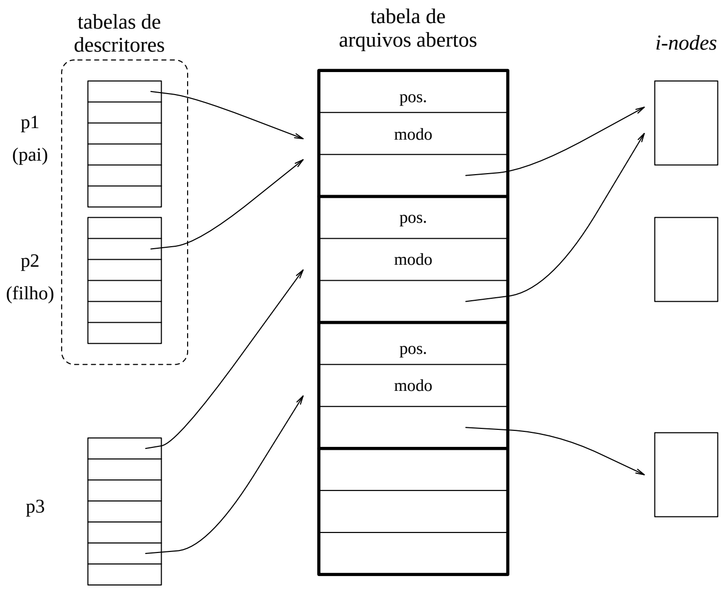



Figura A.2: Tabelas descritoras de arquivos

# Bibliografia


[AFM92] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. RFC 1301, Network Information Center, SRI International, February 1992.

[AK89] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons Ltd., 1989.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, 1986.

[BAD+92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[Bak94] Mary Baker. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California, Berkeley, CA 94720, January 1994. Technical Report UCB/CSD 94/787.

[BH87] Philip A. Bernstein and Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, 1987.

[BHJ+93] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The echo distributed file system. Technical Report #111, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, September 1993.

[BHK+91a] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991.

[BHK+91b] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212, Pacific Grove, CA, October 1991.

[BO91] Mary Baker and John Ousterhout. Availability in the Sprite distributed file system. *ACM Operating Systems Review*, 25(2):95–98, April 1991. Also appeared in the Fourth ACM SIGOPS European Workshop – Fault Tolerance Support in Distributed Systems.

[Bor92] Uwe M. Borghoff. Design of optimal distributed file systems: A framework for research. *ACM Operating Systems Review*, 26(4):30–61, 1992.

[BP81] Richard E. Barlow and Frank Proschan. *Statistical Theory of Reliability and Life Test - Probabilistic Models.* TO BEGIN WITH, Silver Spring, MD, 1981.

[Dee89] Steve Deering. Host extension for IP multicasting. RFC 1112, Network Information Center, SRI International, August 1989.

[Del82] Carl N. R. Dellar. A file server for a network of low cost personal microcomputers. *Software - Practice and Experience*, 12(11):1051–68, November 1982.

[Den82] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[DGMS85] S. B. Davidson, H. Garcia-Molia, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice & Experience*, 21(8), August 1991.

[Dou93] Fred Douglis. On the role of compression in distributed systems. *ACM Operating Systems Review*, 27(2):88–93, 1993.

[DT90] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer schemes. *ACM SIGMETRICS - Performance Evaluation Review*, 18(1):143–52, May 1990.

[Flo89] Richard Allen Floyd. *Transparency in Distributed File Systems.* PhD thesis, University of Rochester - Dept. of Computer Science, January 1989. Technical Report 272.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 202–210, December 1989.

[Gif79] David K. Gifford. Weighted voting for replicated data. *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 150–62, December 1979.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic File Systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 16–25, 1991.

[GZS94] Deepinder S. Gill, Songnian Zhou, and Harjinder S. Sandhu. A case study of file system workload in a large-scale distributed environment. Technical Report 296, Computer Science Research Institute - University of Toronto, 1994.

[Ham86] Richard Wesley Hamming. *Coding and Information Theory.* Prentice-Hall, Englewood Cliffs, 2 edition, 1986.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HO92] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. In *Proceedings of the USENIX File Systems Workshop*, pages 71–78, Ann Arbor, Michigan, May 1992. USENIX.

[HO93] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[HPM93] Graham Hamilton, Michael Powelol, and James Mitchell. Subcontract: A flexible base for distributed computing. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 69–79, December 1993.

[Jas] Barry Jaspan. Kerberos Users' Frequently Asked Questions. Disponível por FTP anônimo em rtfm.mit.edu, diretório /pub/usenet/news.answers/kerberos-faq.

[KN92] Yousef A. Khalidi and Michael N. Nelson. An implementation of UNIX on an object-oriented operating system. Technical Report SMLI TR-92-03, Sun Microsystems Laboratories, Inc, December 1992.

[KN93] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 1–13. ACM PRESS, Dec 1993.

[KS92] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda filesystem. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[L+92] Edward K. Lee et al. RAID-II: A scalable storage architecture for high bandwidth network file service. February 1992.

[LGJ+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 226–38, 1991.

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[LYW93] Avraham Leff, Pkilip S. Yu, and Joel L. Wolf. Performance issues in object replication for a remote caching architecture. *Computer Systems Science & Engineering*, 8(1):40–51, 1993.

[MBH+93] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. Technical Report #103, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, June 1993.

[McK84] Marshall K. McKusick. A fast file system for unix. *ACM Transactions on Computer Systems*, 2(3):181–97, 1984.

[Mil92] David L. Mills. Network time protocol (version 3) - specification, implementation and analysis. RFC 1305, Network Information Center, SRI International, March 1992.

[Mil94] Dejan S. Milojicic. *Load Distribution, Implementation for the Mach Microkernel.* Vieweg, 1994.

[MJ82] J. G. Mitchell and J.Dion. A comparision of two network-based file servers. *Communications of the ACM*, 25(4):233–45, 1982.

[MMP83] Erik T. Mueller, Johanna D. Moore, and Gerald Popek. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 71–89, October 1983.

[MOO87] Mamoru Maekawa, Arthur E. Oldehoeft, and Rodney R. Oldehoef. *Operating Systems: Advanced Concepts*. Menlo Park, Benjamin.Cummings, 1987.

[MP91] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 56–67, 1991.

[MRSV86] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vicentinelli. Convergence and finite-time behavior of simulated annealing. *Advanced Applied Probability*, 18:747–771, 1986.

[Nel88] M. N. Nelson. *Physical Memory Management in a Network Operating System*. PhD thesis, University of California, Berkeley, CA 94720, November 1988. Technical Report UCB/CSD 88/471.

[Neu92] B. Clifford Neuman. Prospero: A tool for organizing internet resources. *Eletronic Networking: Research, Applications, and Policy*, 2(1), 1992.

[NKM93] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. The Spring file system. Technical Report SMLI TR-93-10, Sun Microsystems Laboratories, Inc, February 1993.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[O+85] J. Ousterhout et al. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer USENIX '90*, pages 247–256, Anaheim, CA, June 1990.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). *ACM SIGMOD Record*, 17(3):109–16, June 1988.

[PGS93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.

[PGZ92] James Y.C. Pang, Deepinder S. Gill, and Songnian Zhou. Implementation and performance of cluster-based file replication in large-scale distributed systems. Technical report, Computer Science Research Institute - University of Toronto, August 1992.

[Ree83] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[RO90] Mendel Rosenblum and John Ousterhout. The LFS storage manager. In *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

[RO91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[RW83] C. V. Ramamoorthy and B.W. Wah. The isomorphism of simple file allocation. *IEEE Transactions on Computers*, 32(3), 1983.

[Sat89] Mahadev Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–80, August 1989.

[Sat90a] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, pages 9–21, May 1990.

[Sat90b] Mahadev Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4:73–104, 1990.

[SBHM93] Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann. Availability in the echo file system. Technical Report #112, DIGITAL Equipment Corporation Systems Research Center, Palo Alto, CA, September 1993.

[Sit91] Site Security Policy Handbook Working Group. Site security handbook. RFC 1244, Network Information Center, SRI International, 1991.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasai, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–59, April 1990.

[SKM+93] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*, June 1993. Cambridge, MA.

[SMB79] Daniel Swinehart, Gene McDaniel, and David Boggs. WFS: A simple shared file system for a distributed envoronment. In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 9–17, December 1979.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi85] Alan Jay Smith. Disk cache - miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[SMK93] M. Satyanarayanan, Henry H. Mashburn, and Puneet Kumar. Lightweight recoverable virtual memory. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. Disponível por FTP anônimo em athena-dist.mit.edu, arquivo /pub/kerberos/doc/usenix.txt,PS, March 1988.

[SS90] M. Satyanarayanan and Ellen H. Siegel. Parallel communication in a large distributed environment. *IEEE Transactions on Computers*, 39:328–48, 1990.

[Ste90] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[SUN89a] SUN Microsystems, Inc. NFS: Network file system protocol specification. RFC1094, Network Information Center, SRI International, March 1989.

[SUN89b] SUN Microsystems, Inc. RPC: Remote procedure call protocol specification. RFC 1057, Network Information Center, SRI International, 1989.

[SUN89c] SUN Microsystems, Inc. XDR: External data representation standard. RFC 1014, Network Information Center, SRI International, 1989.

[SUN90] SUN Microsystems, Inc. *Network Programing Guide*. 1990.

[SUN94] SUN Microsystems, Inc. NFS: Network file system version 3 protocol specification. Disponível por FTP anônimo em ftp.uu.net, diretório /networking/ip/nfs-/NFS3.spec.ps.Z, February 1994.

[Svo84] Liba Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–398, December 1984.

[SW91] Frank Schmuck and Jim Wyllie. Experience with transactions in quicksilver. *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 239–53, 1991.

[SZ92] Harjinder S. Sandhu and Songnian Zhou. Cluster-based file replication in large-scale distributed systems. *ACM SIGMETRICS - Performance Evaluation Review*, 20(1):91–102, June 1992.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[WJLP85] M. J. Weinstein, T. W. Page Jr., B.K. Livezey, and G.J Popek. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.

[WO86] B. B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed filesystem. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 184–189, Boston, Mass., May 1986. IEEE.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 49–70, October 1983.

[YM93] Zhonghua Yang and T. Anthony Marsland. Annotated bibliography on global states and times in distributed systems. *ACM Operating Systems Review*, 27(3), 1993.

[ZE88] Edward R. Zayas and Craig F. Everhart. Design and specification of the cellular andrew environment. Technical Report CMU-ITC-88-070, Information Technology Center - Carnegie Mellon University, August 1988.

# Índice


AFS, 42–49
AMOEBA, 103
ANDREW, 42
  *Benchmark*, 55
ARPANET, 29
árvore de diretórios, 17
*automounter*, 34–35

bloqueio, 25

cache, 18–20
  do NFS, 37
  memória cache, 19
  no AFS, 48
  no cliente, 19
  no ECHO, 90
  no servidor, 19
  no SODA, 125
  no SPRITE, 60
  remoto, arquitetura de, 106
*callback*, 48
CFAP, 99
cliente, 17–18
CODA, 42, 50–56
compressão, 141
confiabilidade, 24
consistência, 19–21, 25–27, 68, 113
  no AFS, 48
  no CODA, 51
  no FROLIC, 87
  no NFS, 37
  no SODA, 126
  no SPRITE, 61

*deadlock*, 25
*delayed-write*, 38
disponibilidade, 20–21, 27
  no CODA, 50
  no SPRITE, 63
DOS, 17, 30

ECHO, 89–93
escalabilidade, 21–22
  do AFS, 42
  no NFS, 32
  no SPRITE, 60

flexibilidade, 137
FROLIC, 84–88

HARP, 79–83
heterogeneidade, 22, 32

*i-node*, 39, 144
IFS, 29
impasse, 25
IPELA, 96

KERBEROS, 47

lease, 91, 113–123
  duração, 114–115, 131
LINUX, 17, 125
listras, 74
  do ZEBRA, 75
*lock*, 25
LOCUS, 24
LOTUS, 29
LRU, 19

memória cache, 19
modelo
  analítico, 95
  de Borghoff, 96
  de Gray, 117
  melhorado para leases, 119
  para cache remoto, 106
*mount*, 33–34
*multicast*, 117
multiRPC, 50

NETWARE, 30
NFS, 22, 32–41
  protocolo, 35
NIS, 37

nomes e localização, 17
- no AFS, 43
- no ECHO, 90
- no NFS, 33
- no SPRITE, 58

NVRAM, 140

operação desconectada, 50
operações atômicas, 24

partição, 20
*pathname*, 18, 35
Plan9, 17
PROSPERO, 46

QuickSilver, 24

RAID, 74
RAM DISK, 17
*read-ahead*, 35
replicação, 26
- no AFS, 45
- no CODA, 50
- no ECHO, 89
- no FROLIC, 86
- no HARP, 80
- no NFS, 34

Resumo Comparativo, 93
RPC, 32
- linguagem, 35

segurança, 22–23
- no AFS, 46
- no NFS, 37
- no SPRITE, 60

semântica
- de sessão, 26
- UNIX, 26

serviço, 17
servidor, 17, 18
*simulated annealing*, 108
sincronização de relógios, 116
sistema de arquivos, 17
- baseado em *log*, 72
- distribuído, 17
- físico, 18, 140
- UNIX, 143
- virtual, 17

sistema operacional distribuído, 15, 57
SODA, 125–135
soquete, 127
SPRING, 137–139
SPRITE, 57–71
SPRITE LFS, 72–74
SWALLOW, 29
SWIFT, 75

tabelas de prefixos, 58
TAOS, 89
*thread*, 39
tolerância a falhas, 23
- no ECHO, 89
- nos leases, 115

transação, 24
- serializável, 25

transparência, 18
*two-phase locking*, 25

UNIX, 17, 22, 143–146
- semântica, 26

VFS, 40–42
VICE, 30
votação, 80

WFS, 29
*write*
- *behind*, 38–89, 114
- *on-close*, 37

XDR, 32, 35

ZEBRA, 72–78