UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E DE ESTATÍSTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PYXIS: UM SISTEMA DE ARQUIVOS DISTRIBUÍDO

por

Antônio Augusto Medeiros Fröhlich

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

Prof. Thadeu Botteri Corso
Orientador

Florianópolis, agosto de 1994.

PYXIS: UM SISTEMA DE ARQUIVOS DISTRIBUÍDO

ANTÔNIO AUGUSTO MEDEIROS FRÖHLICH

ESTA DISSERTAÇÃO FOI JULGADA PARA OBTENÇÃO DO TÍTULO DE

MESTRE EM CIÊNCIA DA COMPUTAÇÃO

ESPECIALIDADE SISTEMAS DE COMPUTAÇÃO E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Prof. Thadeu Botteri Corso, M.Sc.
Orientador

Prof. Hermann Adolf Harry Lücke, Dr. Ing.
Coordenador do Curso

BANCA EXAMINADORA

Prof. Thadeu Botteri Corso, M.Sc. (Presidente)

Prof. Joni da Silva Fraga, Dr.

Prof. Philippe Olivier Alexandre Navaux, Dr.

Prof. Simão Sirineo Toscani, Dr.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Fröhlich, Antônio Augusto Medeiros

PYXIS: UM SISTEMA DE ARQUIVOS DISTRIBUÍDO / Antônio Augusto Medeiros Fröhlich. — Florianópolis: CPGCC da UFSC, 1994.
98 p.: il.

Dissertação (mestrado) — Universidade Federal de Santa Catarina, Curso de Pós-Graduação em Ciência da Computação, Florianópolis, 1994. Orientador: Corso, Thadeu Botteri.

Dissertação: Sistemas de Arquivos Distribuídos.
Sistemas Operacionais, Sistemas Distribuídos, Sistemas de Arquivos, Cliente-Servidor, Servidor de Arquivos, Servidor de Nomes.

À Lu.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este texto apresenta o PYXIS, um sistema de arquivos distribuído portável com alto grau de paralelismo interno, desenhado para ser flexível no que diz respeito ao ambiente sobre o qual seus componentes são distribuídos, possibilitando sua execução em multicomputadores ou em redes de computadores. O projeto foi desenvolvido no Curso de Pós-Graduação em Ciências da Computação da Universidade Federal de Santa Catarina (CPGCC/UFSC) e deverá integrar um projeto coletivo das universidades federais de Santa Catarina (UFSC), do Rio Grande do Sul (UFRGS) e de Santa Maria (UFSM), que visa desenvolver um multicomputador e um ambiente para programação paralela sobre ele.

ABSTRACT

This text presents the PYXIS, a portable, highly parallel distributed file system, designed to be very flexible about the level of distribution of its components, allowing it to run on multicomputers or on computers networks. The project was developed at the Computer Science Department of the Federal University of Santa Catarina (CPGCC/UFSC) and must take part in a larger project, gathering the federal universities of Santa Catarina (UFSC), Rio Grande do Sul (UFRGS) and Santa Maria (UFSM), to develop a multicomputer and a parallel programming environment for it.

Parte I

CONCEITOS BÁSICOS

"...

One love
One blood
One life
You got to do what you should

..."

(Bono)

1 INTRODUÇÃO

Os últimos anos têm registrado o surgimento de uma série de idéias inovadoras na área de sistemas operacionais, muitas delas motivadas pela contínua evolução do *hardware*, outras resultando do amadurecimento natural de conceitos como comunicação, distribuição, paralelismo e heterogeneidade.

Esta dissertação apresenta o PYXIS, uma proposta inovadora de sistema de arquivos baseada numa visão ainda avançada de um ambiente computacional com número de unidades processadoras equiparável ao número de tarefas e onde as entidades comunicantes dispõem de canais de comunicação hoje encontrados apenas em barramentos internos de computadores. A ânsia pela comunicação efetiva de dados, voz e imagens, resultará muito em breve em canais de comunicação de alguns Gbps. Isso, aliado à rápida redução do custo das unidades processadoras, permitirá a construção de sistemas computacionais mais eficientes, onde problemas complexos poderão ser tratados na sua essência, simplificando muitas das estruturas formadoras de um sistema operacional.

Para facilitar a compreensão e melhor organizar os conteúdos abordados, esse texto foi dividido em três partes: conceitos básicos, sistemas de arquivos distribuídos e projeto do PYXIS.

A primeira parte tem por objetivo introduzir uma série de conceitos fundamentais ao entendimento do resto do texto. Ela começa com a descrição do modelo cliente-servidor, no qual o PYXIS está baseado, juntamente com uma proposta simplificada de núcleo de sistema operacional para suportar a implementação do referido modelo e de um mecanismo de chamada de procedimento remoto. Após são introduzidos os conceitos de arquivo e diretório, bem como uma série de outros conceitos a eles associados.

A segunda parte apresenta as principais características de um conjunto de sistemas de arquivos distribuídos, visando dar subsídios para que o leitor possa compará-los com o PYXIS. Os sistemas de arquivos descritos são: NFS, DOMAIN, ANDREW e AMOEBA. Encerrando essa parte do texto, tem-se uma análise comparativa das principais características dos sistemas de arquivos descritos.

A terceira e última parte é dedicada à descrição do projeto e da implementação da versão preliminar do PYXIS, apresentando principalmente os aspectos inovadores, como a divisão do sistema de arquivos em um conjunto de servidores, o mecanismo original de localização de servidores, a transparência de localidade alcançada pela extensão do mecanismo de *link*, entre outros. Por fim são apresentadas as conclusões, juntamente com sugestões para a evolução do PYXIS.

2 MODELO CLIENTE-SERVIDOR

Durante bastante tempo, a baixa capacidade de processamento dos sistemas computacionais impôs sistemas operacionais monolíticos, onde todo o sistema era compilado em um único módulo executável. A dificuldade de desenvolvimento e manutenção desses sistemas levou a definição de modelos hierárquicos, organizados em camadas. A evolução das unidades processadoras e dos sistemas de interconexão das mesmas acarretou no desenvolvimento do modelo cliente-servidor, mais apropriado a arquitetura dos sistemas computacionais atuais.

A principal idéia por trás do modelo cliente-servidor é estruturar o sistema operacional como um conjunto de processos cooperativos (servidores) que oferecem serviços aos processos de usuário (clientes). Para suportar a execução dos servidores, os nodos participantes executam um pequeno núcleo em modo supervisor. Esse núcleo implementa apenas os mecanismos necessários para a execução, em modo usuário, dos processos clientes e servidores.

No modelo cliente-servidor, a comunicação entre processos se dá exclusivamente através da troca de mensagens. Isso torna o modelo adequado à implementação de sistemas distribuídos, uma vez que os mecanismos de comunicação podem ser desenhados para permitir a troca de mensagens entre processos executando em nodos distintos. O modelo cliente-servidor permite ainda que componentes do sistema operacional sejam modificadas enquanto este se encontra em execução.

Geralmente, o modelo cliente-servidor faz uso de protocolos de comunicação simples, do tipo requisição/resposta. A fim de obter um serviço, um cliente envia uma requisição ao servidor. Este, por sua vez, executa as operações associadas ao serviço e envia uma resposta ao cliente, contendo da-

dos ou um código de erro caso o serviço não possa ser executado. A figura 2.1 mostra o esquema básico do modelo cliente-servidor. Embora as mensagens sejam trocadas através do canal de comunicação por intermédio dos núcleos, para simplificar a apresentação, a figura apresenta as mensagens sendo trocadas diretamente entre os processos.

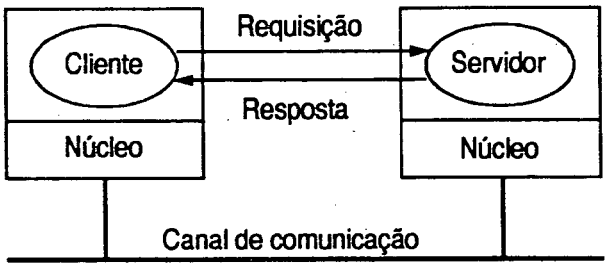

Figura 2.1: O modelo cliente-servidor

A principal característica do protocolo requisição/resposta é a simplicidade, pois não se gasta tempo com o estabelecimento e liberação de conexões. Além disso, a confirmação de mensagens fica simplificada, uma vez que uma resposta confirma uma requisição. Para evitar que um cliente espere indefinidamente por uma resposta do servidor, pode-se optar pelo uso de tempo de expiração das requisições. Desta forma, uma requisição sem resposta por mais de um determinado tempo é considerada perdida e um procedimento de falha pode ser ativado, como por exemplo, o reenvio da requisição.

2.1 Núcleo

A adoção do modelo cliente-servidor para um sistema operacional implica em servidores cooperando para atender às solicitações dos clientes e se comunicando, entre si e com os clientes, através de mensagens. Para viabilizá-lo, uma ou mais entidades, que não os próprios servidores, devem implementar os mecanismos básicos de criação de processos, sincronização e comunicação.

A solução mais comumente adotada é a de reunir esses mecanismos em um núcleo.

O núcleo é carregado como primeiro passo da inicialização do sistema operacional, tem segmentos de dados e código protegidos e executa em modo supervisor de processador. Ele é invocado através de interrupções de *software* sempre que algum de seus serviços é solicitado ou quando da ocorrência de alguma interrupção de *hardware*. No modelo cliente-servidor, os controladores de dispositivo podem ser implementados como servidores, sendo nesse caso necessário converter as interrupções de *hardware* relativas aos dispositivos em mensagens para os respectivos servidores.

No que diz respeito a manutenção do estado do sistema, o núcleo mantém somente informações sobre os recursos controlados por ele, tais como segmentos de memória associados aos processos, contexto dos processos, etc. Informações sobre os recursos gerenciados pelos servidores são mantidas pelos próprios servidores. Dessa forma, o estado do sistema fica distribuído pelos vários servidores do sistema, e não centralizado no núcleo.

Resumidamente, pode-se definir três classes de serviços que são de responsabilidade do núcleo do sistema:

- Criação de processos;
- Sincronização entre processos;
- Comunicação entre processos.

2.1.1 Processos

O conceito tradicional de processo [SIL 94] não se aplica muito bem ao modelo cliente-servidor, onde é comum um servidor criar um processo filho

apenas para executar um serviço e, em seguida, destruí-lo. A criação de processos nos sistemas convencionais é muito cara, sendo necessário aperfeiçoar esse conceito.

Os núcleos de sistemas operacionais cliente-servidor, geralmente, implementam o conceito de processo através da combinação de tarefas (*tasks*) e fluxos de execução (*threads*) [TAN 92].

O processo de compilação de um programa produz um arquivo executável que contém código e dados. Para que se possa executá-lo, tanto seu código como seus dados devem ser carregados para segmentos específicos de memória. Esses segmentos, mais algumas informações de controle, formam uma tarefa, que pode então ser entendida como um programa carregado na memória principal do computador. Recursos como arquivos e semáforos são alocados às tarefas, logo, seus descritores devem manter informações sobre esses recursos.

Para se executar um programa não basta carregá-lo na memória: é necessário que se defina seu contexto de execução e que se faça com que o processador comece a executar suas instruções. Em outras palavras, é necessário que se criem fluxos de execução para a tarefa. O contexto de execução de um fluxo inclui seu estado, o estado do processador, informações de controle e uma pilha de execução.

O núcleo deve prover serviços tanto para criação e destruição de tarefas, tendo como base arquivos binários contendo código e dados, como para a criação e destruição de fluxos de execução sobre elas.

Uma tarefa com um único fluxo de execução (figura 2.2 (a)) é equivalente ao conceito tradicional de processo. Uma tarefa com n fluxos de execução (figura 2.2 (b)) é algo mais simples e barato do que n processos, pois quando se cria um novo fluxo de execução para uma tarefa não é necessária a carga

de novos segmentos de memória com código e dados. O custo de criação de um fluxo de execução se reduz à criação de uma estrutura descritora e à alocação de uma nova pilha.

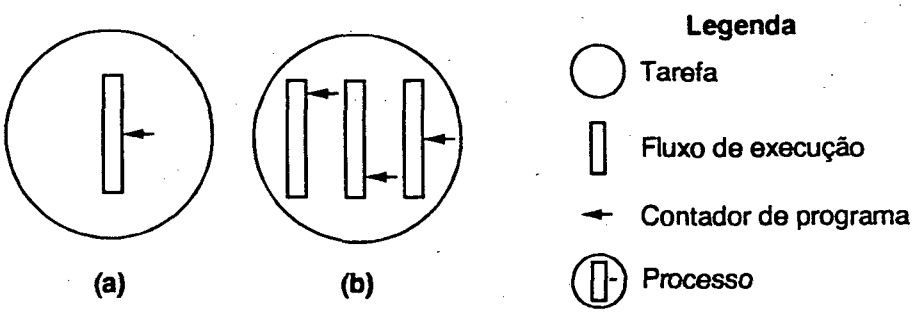

Figura 2.2: Tarefas e fluxos de execução

A construção de um servidor como uma tarefa com múltiplos fluxos de execução pode aumentar bastante o desempenho do sistema, porque, enquanto um fluxo estiver bloqueado esperando por um evento, outro poderá estar executando. Além disso, pode-se dispensar um mecanismo formal de comunicação entre os fluxos de um mesmo servidor, uma vez que eles compartilham o mesmo segmento de dados.

O conceito de múltiplos fluxos de execução de tarefas abordado nessa seção é derivado do conceito de processo leve (*lightweight process*) descrito em [TAN 92]. Modelos similares têm sido usados em sistemas operacionais como o AMOEBA [MUL 90] e o MACH [ACC 86] e parecem se consolidar a medida que os processadores adquirem mais facilidades para escalonar rapidamente os processos.

2.1.2 Caixas postais

A comunicação entre processos talvez seja o ponto mais crítico para o desempenho de um sistema baseado no modelo cliente-servidor, uma vez que todos os serviços são requisitados e respondidos através de mensagens.

As caixas postais (figura 2.3) constituem um mecanismo de comunicação apropriado para suportar o modelo cliente-servidor. Com esse mecanismo, as mensagens são endereçadas a caixas postais e não a processos. Para receber mensagens, um servidor solicita ao núcleo que crie uma caixa postal com um certo endereço. As mensagens que chegam referindo este endereço são colocadas na caixa postal e lá permanecem até que o servidor solicite ao núcleo que retire uma mensagem. Clientes e servidores podem, por convenção, definir um domínio de caixas postais, onde cada uma delas está previamente associada a um serviço.

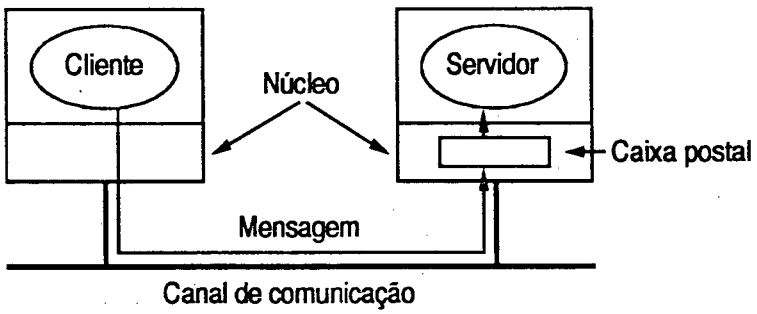

Figura 2.3: Comunicação com caixa postal

Para implementar as caixas postais, o núcleo mantém um conjunto de *buffers* para armazenar as mensagens e fornece, pelo menos, duas primitivas: "send" e "receive". A primeira envia uma mensagem a uma caixa postal, liberando o processo assim que a mensagem for enviada. A segunda recebe uma mensagem da caixa postal, bloqueando o processo se ela estiver vazia. Caso o núcleo não mais disponha de *buffers* para armazenar uma nova mensagem, ela poderá ser descartada e o mecanismo de confirmação de mensagens deverá detectar a falha.

Uma concepção elaborada do mecanismo de caixas postais pode agregar ao sistema um série de propriedades valiosas à implementação de sistemas distribuídos, entre as quais as possibilidades de:

- Troca de mensagens de tamanho variável;

- Transparência de localidade dos servidores;
- Sincronização de clientes com servidores;
- Replicação de servidores;
- Migração de servidores;
- Envio de mensagens a partir do núcleo sem bloqueá-lo.

Detalhes de como essas propriedades podem ser obtidas serão apresentados juntamente com a descrição do PYXIS.

2.1.3 Semáforos

Em sistemas onde se permite concorrência, os cuidados com sincronização são fundamentais para evitar condições de corrida. Algum mecanismo deve ser suprido para que os processos possam sincronizar, por exemplo, a execução de seções críticas. Uma vez que cada fluxo de execução possui sua própria pilha de execução onde são criadas variáveis locais, uma seção crítica somente ocorre quando os fluxos manipulam dados globais da tarefa. Nesse caso, um mecanismo simples e eficiente para a sincronização de fluxos de execução é o semáforo, uma vez que ele pode garantir as três condições para a execução de seções críticas [SIL 94]: exclusividade de execução, progresso de execução e espera limitada para execução. A idéia é associar um semáforo a cada estrutura de dados global que possa ser envolvida em uma condição de corrida. Assim, quando um fluxo desejar alterar uma estrutura global, ele antes chama o núcleo para validar o acesso através do semáforo associado.

2.2 Chamada de Procedimento Remoto

Embora o modelo cliente-servidor provenha uma maneira conveniente para a estruturação de sistemas operacionais distribuídos, a utilização das primitivas "send" e "receive" não é natural aos programadores. Para tornar o mecanismo de troca de mensagens transparente aos usuários, Birrel e Nelson [BIR 84] introduziram o conceito de chamada de procedimento remoto (*Remote Procedure Call [RPC]*) [SUN 88].

Através de *RPC*s, um processo executando em um nodo pode chamar procedimentos de processos executando em outros nodos. Quando um processo executa uma *RPC* ele passa parâmetros como em uma chamada de procedimento convencional. Os parâmetros são então encapsulados em uma mensagem e enviados ao processo remoto, que chama o procedimento local passando os parâmetros retirados da mensagem. Os parâmetros retornados pelo procedimento percorrem o caminho inverso. O processo executor da *RPC* fica bloqueado até que uma resposta chegue ou que o tempo máximo estipulado para execução da chamada expire, fazendo com que um código de erro seja retornado.

O esquema de uma *RPC* está representado na figura 2.4. A fim de tornar o mecanismo descrito transparente, as bibliotecas do sistema contêm *stubs* que manipulam as mensagens relativas às *RPC*s. Considere o exemplo de uma função de abertura de arquivo ("open"). Quando um usuário escreve um programa incluindo uma chamada à função "open", o compilador procura a função nas bibliotecas do sistema e inclui o respectivo *stub* no programa executável. Quando o processo chama a função "open", o fluxo é desviado para o *stub* associado, que gera a mensagem necessária à execução da *RPC* e fica bloqueado aguardando a resposta do servidor.

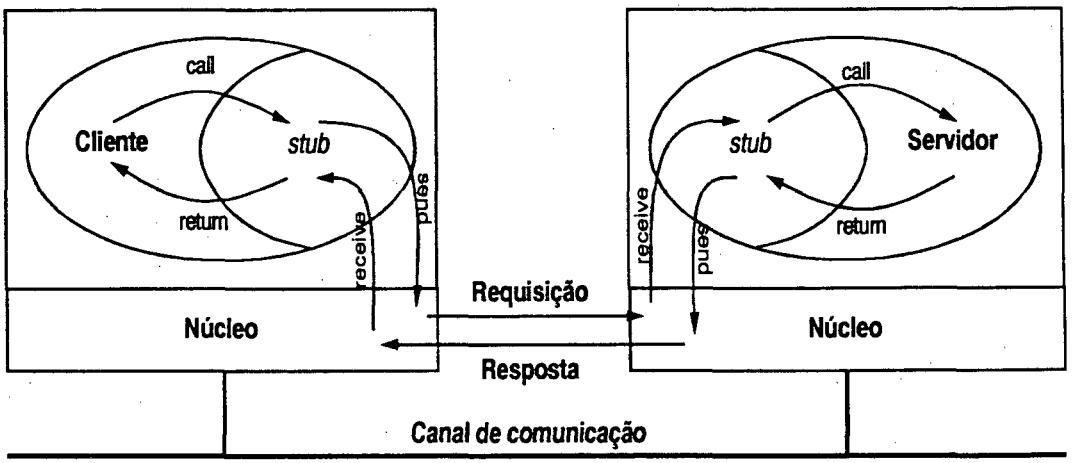

Figura 2.4: Chamada de procedimento remoto [*RPC*]

3 SISTEMA DE ARQUIVOS

As unidades processadoras hoje em uso manipulam dados exclusivamente em memória principal, contudo, essa memória geralmente é volátil e em quantidade insuficiente para atender a todos os usuários. Para superar esta limitação, a maioria dos sistemas computacionais faz uso de sistemas de arquivos capazes de armazenar grandes quantidades de informações de forma permanente.

O sistema de arquivos é uma das partes mais complexas de um sistema operacional, envolvendo uma série de conceitos e algoritmos, vários dos quais serão descritos a seguir.

3.1 Discos

Discos são os dispositivos de memória secundária que normalmente servem de suporte para a implementação de sistemas de arquivos. As tecnologias atuais de construção de discos divergem em muitos aspectos, mas, duas características são comuns e definem o termo disco: acesso a dados em blocos e acesso direto a blocos.

3.1.1 Blocos lógicos de dados

O bloco lógico de dados, ou simplesmente bloco, é a unidade utilizada pelo sistema de arquivos para acessar, de maneira homogênea, dados armazenados em disco. Uma vez que os blocos físicos (setores) variam de dispositivo para dispositivo, o bloco lógico de dados é um conceito fundamental para manter o sistema de arquivos independente de dispositivo.

A adoção de um tamanho fixo de bloco pelo sistema de arquivos implica em que os servidores de dispositivo façam o mapeamento de blocos em setores e vice-versa. Dessa maneira, o servidor de um dispositivo com setor menor do que um bloco terá que acessar mais do que um setor de disco para formar um bloco, enquanto um servidor de dispositivo com setor maior do que um bloco acessará um setor e selecionará a parte do mesmo correspondente ao bloco. Outro fator relevante para a escolha do tamanho do bloco é a granularidade do disco. Um disco muito grande com blocos pequenos pode ser de difícil gestão; um disco muito pequeno com blocos grandes pode apresentar uma fragmentação interna dispendiosa.

A definição do tamanho dos endereços dos blocos depende basicamente da quantidade de blocos que se pretende endereçar por disco. A realidade de hoje envolve discos com tamanhos que variam de poucos Mbytes até Tbytes, logo, deve-se escolher um tamanho de endereço que permita exprimir tais valores. Com blocos de 1 kbyte e 16 bits para representar os endereços de blocos, o tamanho máximo de um disco ficaria restrito a 64 Mbytes, o que é muito pouco. Com endereços de 24 bits, poder-se-ía endereçar 16 Gbytes, um valor que em pouco tempo poderá ser restritivo. Endereços de 32 bits possibilitam o endereçamento de 4 Tbytes e parecem se adequar, tanto ao tamanho dos discos que serão usados nesta década, quanto à arquitetura dos computadores que, provavelmente, executarão sistemas de arquivos distribuídos.

3.1.2 Volumes

Um volume de disco, ou disco lógico, é um conjunto contíguo de setores de um disco (partição), ao qual se atribui um nome e que é reconhecido pelo sistema como uma entidade independente. Somente os servidores de dispositivos conhecem a diferença entre um disco físico e um volume; o resto do sistema só reconhece volumes.

O conceito de volume traz, pelo menos, duas vantagens ao sistema:

1. Tolerância a falhas: caso algum dano ocorra a uma trilha de um disco físico, apenas um volume é comprometido enquanto o restante do disco continua operacional.

2. Facilidade de gestão: o particionamento de discos físicos grandes em volumes menores torna a gestão de espaço livre mais simples e eficiente.

A principal desvantagem do particionamento de um disco é a limitação imposta ao tamanho máximo de um arquivo, que fica restrito ao tamanho do volume. Todavia, essa desvantagem pode ser facilmente superada, uma vez que o volume é uma entidade lógica que pode ser expandida até o tamanho físico do disco.

O mapeamento de volume em disco físico também é atribuição dos servidores de dispositivos. Para tal, é mantida, em um setor especial do disco, uma tabela que descreve todas as suas partições. Essa tabela é lida e interpretada pelo servidor do dispositivo quando da inicialização do dispositivo, após o que, cada uma das suas partições passa a ser vista como um disco autônomo.

Daqui para frente, o termo disco será usado para designar qualquer dispositivo de memória secundária acessado a bloco, independentemente deste ser realmente um disco físico, um volume de disco ou outro tipo de dispositivo.

Como se pode ver na figura 3.1, um disco apresenta três componentes:

Figura 3.1: Organização de um disco

- Bloco descritor do disco: está presente em todos os discos do sistema, sempre localizado no endereço zero. Ele é responsável pela manutenção de uma série de informações sobre o disco, como o seu tamanho, o número de blocos livres, o tamanho da lista de descritores de arquivos, o número de descritores de arquivos disponíveis, datas de criação e alteração do disco, entre outras. O bloco descritor de disco contém ainda um campo booleano que é marcado como verdadeiro quando da ativação do disco, e só é marcado falso na sua desativação. Assim, se houver falta de luz ou violação de integridade do disco, na sua próxima ativação o referido campo será verdadeiro, ocasionando uma seqüência de recuperação do disco.

- Lista de descritores de arquivos: estrutura com uma entrada para cada arquivo armazenado no disco. Como cada arquivo está associado a um descritor, a definição do tamanho da lista de descritores de arquivos impõe um limite ao número de arquivos que se pode ter no disco. Esse limite apenas pode ser ultrapassado com a reformatação do disco e a definição de uma lista de descritores maior.

- Área de usuário: o conjunto de blocos disponíveis ou utilizados pelos usuários.

3.1.3 Gestão de blocos livres

O controle da ocupação dos blocos dos vários discos do sistema é uma tarefa de extrema importância. Uma falha no algoritmo que executa essa tarefa poderia, por exemplo, gerar uma situação onde arquivos diferentes compartilhassem um mesmo bloco.

A literatura apresenta basicamente duas técnicas para a gestão de blocos livres de disco [SIL 94]:

- Mapa de bits: Consiste, basicamente, em manter-se um vetor de bits onde cada bit representa o estado de um bloco de disco (livre ou ocupado). As principais vantagens dessa técnica são a facilidade de implementação e uma forte tendência a gerar arquivos com grandes faixas de blocos contíguos, o que otimiza muito o acesso seqüencial. Suas desvantagens advêm basicamente da dificuldade de gestão de grandes mapas de bits, uma vez que eles, em geral, não poderão ser mantidos em memória principal, além de consumir uma área fixa de disco que nunca será liberada para os usuários. Essa técnica é utilizada por sistemas operacionais como o do MACHINTOSH [APP 87] e o MINIX [TAN 87b].

- Lista encadeada: Consiste em manter-se uma lista encadeada com todos os blocos livres do disco. Para alocar um bloco, simplesmente retira-se o primeiro da lista e, quando um bloco é liberado, reinsere-se o mesmo na lista. Essa técnica é muito eficiente no que diz respeito a utilização de memória, pois, conforme a ocupação do disco cresce, o tamanho da lista de blocos livres diminui, até a sua extinção quando o disco estiver completamente alocado. É também bastante eficiente para operações corriqueiras de alocação e liberação de blocos individuais, mas pode gerar arquivos completamente dispersos pelo disco. Essa técnica é adotada pela maioria dos sistemas operacionais do tipo UNIX.

3.2 Arquivos

A manipulação de dados em disco envolve um conjunto de atividades difíceis de serem executadas pelos usuários, como o cálculo da localização do setor que contém os dados, o controle da alocação dos blocos do disco, a sincronização de acessos concorrentes, entre outras. A fim de tornar essas atividades transparentes, a funcionalidade do sistema de arquivos é passada aos usuários através do conceito de arquivo. Arquivo é uma abstração de dados armazenados em disco, pela qual os dados são vistos como uma seqüência linear de registros [1]. Para acessar um dado, o usuário identifica o arquivo e especifica a ordem do registro dentro dele.

Uma tentativa de uniformização, introduzida pelo UNIX, define uma interface homogênea para o acesso a dados contidos em arquivos, bem como, para o acesso aos dispositivos periféricos, diferenciando apenas dispositivos acessados a bloco dos acessados a caractere. Essa interface é hoje adotada por vários outros sistemas operacionais, tendo se tornado natural à maioria dos usuários.

3.2.1 Descritores de arquivos

Os arquivos são descritos, perante o sistema, por uma estrutura chamada descritor de arquivo [2]. Os descritores armazenam várias informações sobre os arquivos, como por exemplo, seu tipo, seu tamanho, datas de criação, alteração e acesso, identificação do dono, um contador de *links* e um índice dos blocos de dados.

[1] No UNIX, os arquivos são seqüências lineares de bytes.
[2] No UNIX, os descritores de arquivos são chamados "i-nodes" [BAC 87].

Quando um arquivo é criado, um descritor lhe é associado. Caso não existam descritores livres no disco, a solicitação de criação de arquivo é negada. Os arquivos são identificados, dentro de um disco, pelos números de seus descritores. A identificação de um arquivo perante todo o domínio envolve, além do número do descritor, a identificação do nodo e a identificação do disco.

3.2.2 Gestão de blocos alocados a arquivos

Existe um conjunto de características que devem ser consideradas para que o sistema de arquivos seja considerado de uso genérico, a saber:

- Eficiência na manipulação de arquivos pequenos, pois estatísticas mostram que o tamanho médio dos arquivos em um sistema UNIX é 1 Kbyte e que 99 % dos arquivos são menores que 64 Kbytes [MUL 84a];
- Possibilidade de criação, sob algum custo, de arquivos grandes;
- Possibilidade de acesso seqüencial a arquivos;
- Possibilidade de acesso direto a arquivos;
- Possibilidade de expansão de arquivos já criados.

A técnica que faz uso de índices de blocos [SIL 94] satisfaz todos esses requisitos. Ela consiste em manter-se, dentro do descritor de arquivo, uma tabela com os endereços dos primeiros blocos do arquivo, garantindo acesso direto eficiente a esses blocos. Caso esses endereços não sejam suficientes para descrever todos os blocos do arquivo, endereços indiretos, que referem blocos com endereços, são utilizados. A figura 3.2 ilustra essa técnica.

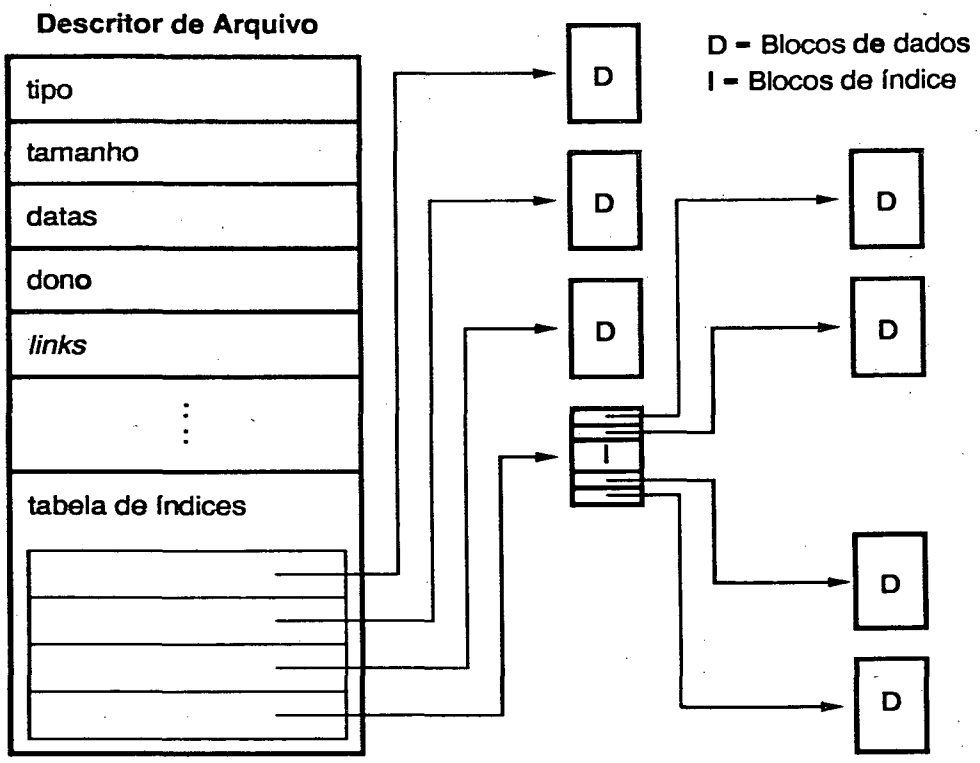

Figura 3.2: Localização dos blocos de um arquivo

3.2.3 Operações sobre arquivos

Diferentes sistemas de arquivos implementam diferentes operações para permitir que os usuários armazenem e recuperem dados em arquivos. As operações mais comumente suportadas são:

- Criação ("create"): Cria um arquivo sem dados. O propósito desta operação é anunciar o arquivo e definir alguns de seus atributos.
- Remoção ("remove"): Essa operação libera os recursos alocados a um arquivo que não mais será necessário.
- Abertura ("open"): A fim de acessar os dados contidos em um arquivo, um processo deve antes abri-lo. Na operação de abertura, em geral, o descritor do arquivo é trazido para tabelas internas do sistema, em memória principal, permitindo acesso rápido ao

mesmo. Além disso, a operação de abertura executa o procedimento de validação de acesso ao arquivo, impedindo que usuários não autorizados o acessem.

- Fechamento ("close"): A operação de fechamento é utilizada para indicar que o processo não mais acessará os dados do arquivo. Em alguns sistemas essa operação garante a atualização dos dados do arquivo em disco.

- Leitura ("read"): Lê dados de um arquivo. Normalmente, a leitura é feita a partir da posição corrente, sendo necessário ao processo especificar apenas a quantia de bytes a serem lidos e um ponteiro para a área de memória que os conterá.

- Escrita ("write"): Análoga à leitura, salvo que, quando a posição corrente for o fim do arquivo, ele será expandido ("append").

- Posicionamento ("seek"): Especifica a próxima posição corrente do ponteiro de arquivo, definindo de onde os dados serão lidos ou para onde serão escritos. Essa operação dá suporte a acesso direto ao arquivo.

- Leitura de atributos ("stat"): Essa operação possibilita que um processo conheça os atributos dos arquivos que vai manipular.

- Escrita de atributos ("chmod"): Alguns dos atributos de um arquivo podem ser alterados, como por exemplo, as permissões de acesso. A alteração desses atributos é feita por uma ou mais operações de escrita de atributos.

4 SISTEMA DE DIRETÓRIOS

As estruturas internas utilizadas pelo sistema para identificar os arquivos não são adequadas aos usuários, que preferem se referir a eles através de nomes. O relacionamento entre os nomes dados pelos usuários e os identificadores atribuídos pelo sistema é feito por diretórios similares aos catálogos telefônicos, que relacionam nomes a números de telefones.

4.1 Diretórios

Um diretório nada mais é do que uma tabela com uma linha para cada entidade e tantas colunas quantos atributos se deseja representar. O objetivo básico do sistema de diretórios é relacionar nomes de arquivos a seus identificadores internos, mas, nada impede que ele seja utilizado para relacionar nomes de outras entidades a atributos quaisquer. A tabela 4.1 apresenta um exemplo de diretório.

Nome	Tipo	Identificador
dissertação	texto	(disco 2, arquivo 23)
U2-One	audio	(disco 1, arquivo 10)
Schiffer	imagem	(disco 1, arquivo 15)

Tabela 4.1: Exemplo de diretório

4.1.1 Árvore de diretório

As tabelas utilizadas pelo sistema de diretórios para associar entidades a atributos são, freqüentemente, implementadas como arquivos ordinários do sistema. Sendo um arquivo, um diretório pode ser referido dentro de ou-

tro, de forma a definir estruturas hierárquicas. A estrutura mais comumente utilizada para organizar os diretórios é a árvore. Toda árvore de diretórios apresenta um nodo especial, denominado raiz ("/"), a partir do qual são encadeados sub-diretórios e arquivos. A figura 4.1 mostra um exemplo de árvore de diretórios.

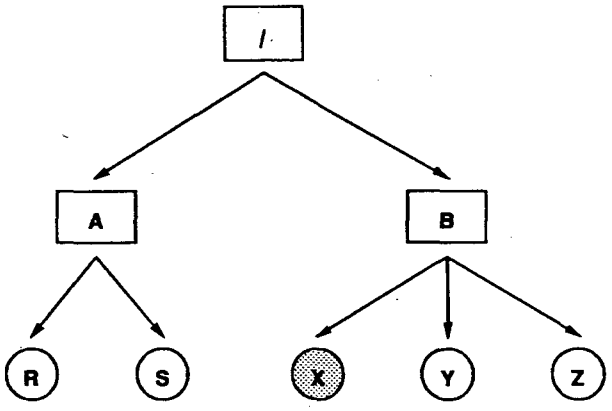

Figura 4.1: Exemplo de árvore de diretórios

Cada diretório possui duas entradas especiais: "." e "..". A entrada "." se refere ao próprio diretório e a ".." se refere ao diretório pai. Com essas duas entradas é possível se localizar um arquivo na árvore através da seguinte regra: um nome que começa com "/" é um nome absoluto, isto é, relativo à raiz da árvore; qualquer outro nome é relativo ao diretório corrente. Por exemplo, considere a árvore da figura 4.1 e suponha que o diretório corrente seja "A". Para se referir ao arquivo "X", pode-se especificar o nome "/B/X" ou "../B/X".

4.1.2 Operações sobre diretórios

As operações mais freqüentemente suportadas para permitir a manutenção dos diretórios são as seguintes:

- Criação ("mkdir"): Cria um diretório sem nenhuma entrada, a exceção de entradas para referir o próprio diretório (".") e o diretório pai (".."). Um diretório assim é dito vazio.

- Remoção ("rmdir"): Remove um diretório vazio.
- Inserção de item ("link"): Insere um item com nome e atributos em um diretório.
- Remoção de item ("unlink"): Remove um item, referido pelo nome, de um diretório.

Parte II

SISTEMAS DE ARQUIVOS DISTRIBUÍDOS

"...

There's so many different worlds
So many different suns
And we have just one world
But we live in different ones

..."

(Mark Knopfler)

5 NFS

O NFS (NETWORK FILE SYSTEM) [SUN 89] foi apresentado em 1985 pela SUN MICROSYSTEMS e, desde então, vem sendo largamente utilizado, tanto no mundo acadêmico como fora dele, a ponto de ser hoje considerado um padrão *de facto*.

O sucesso do NFS se deve a algumas características, definidas pela SUN como básicas para o projeto, entre as quais a portabilidade e a heterogeneidade. Para garantir tais características, junto com a primeira versão, a SUN tornou público o protocolo NFS. Esse protocolo define uma interface de *RPC*s que permite a um servidor exportar arquivos locais para acesso remoto. O protocolo nada diz sobre a implementação do servidor ou dos clientes, permitindo que vários fabricantes desenvolvessem suas próprias implementações de NFS, cada qual com características próprias e mesmo assim compatíveis. Hoje, pode-se encontrar implementações de NFS para quase todo o mundo UNIX, bem como para outros sistemas, entre os quais o MS-DOS.

Outro importante fator considerado na definição do NFS é a tolerância as falhas da rede. Com o objetivo de simplificar a recuperação dos servidores após uma falha, o NFS foi desenhado desconsiderando-se o estado dos clientes. Assim, cada *RPC* realizada por um cliente contém todas as informações necessárias a execução do serviço.

Devido ao grande número de implementações de NFS com características distintas, esse texto se restringirá a implementação da SUN, simplesmente por ser a mais documentada e referida pela literatura especializada.

5.1 A Implementação da Sun Microsystems

A implementação do NFS pela SUN tem como ponto de partida um mecanismo de indireção no núcleo do sistema operacional que permite que chamadas de sistema relativas ao sistema de arquivos sejam interceptadas e desviadas para servidores de arquivos locais ou remotos. Esse mecanismo faz uso de uma estrutura de dados chamada "v-node" [SUN 90], uma extensão do "i-node" tradicional do UNIX, que contém, entre outras, informações sobre a localização do arquivo.

O NFS trata todos os nodos como pares, sem distinção entre clientes e servidores. Um nodo pode ser um servidor, exportando alguns de seus arquivos, mas pode, ao mesmo tempo, ser um cliente, acessando arquivos remotos.

Os clientes NFS são geralmente configurados de forma a ter uma árvore de diretórios com raiz privada, isto é, cada nodo possui sua própria árvore de diretórios. Através de uma extensão do mecanismo de "mount" do UNIX [LEF 89], os clientes podem anexar sub-árvores remotas, exportadas por servidores NFS, à sua árvore privada. Desta forma, um arquivo compartilhado pode ser visto com nomes distintos ao longo do domínio. O NFS não provê mecanismos para garantir a homogeneidade do espaço de nomes de arquivos. Entretanto, um grupo de usuários pode, por convenção, obter um espaço de nomes homogêneo.

O NFS não implementa um mecanismo dinâmico de amarração de arquivos a nodos, pois não possibilita migrações ou replicações de arquivos. Cada cliente possui uma tabela de amarração relacionando sub-árvores a nodos, atualizada quando da anexação da sub-árvore. A adição de novos servidores ou o movimento de arquivos através dos servidores torna a tabela de amarração obsoleta, exigindo intervenção humana.

Enquanto o NFS estava sendo desenvolvido, várias empresas já haviam percebido um grande mercado para estações de trabalho sem disco, entre elas a própria SUN. Essas estações tinham um custo bastante reduzido, porém, estavam restritas ao baixo desempenho dos acessos remotos. Várias características do NFS foram definidas em função do desempenho de máquinas sem disco, como o mecanismo de *cache*, o tamanho dos blocos transferidos através da rede e a leitura antecipada de blocos de arquivos.

Os clientes NFS fazem *cache* de blocos de arquivos e diretórios remotos e de traduções de nomes de arquivos em "v-nodes" na memória principal. Discos locais, se presentes, não são utilizados para *cache*. Quando um cliente faz *cache* de um bloco de um arquivo remoto, ele armazena também uma informação temporal representando a última alteração do arquivo no servidor. Para validar um bloco, o cliente compara a informação temporal da *cache* com a do servidor. Se a informação do servidor for mais recente, o cliente invalida todos os blocos do arquivo na *cache* e os reobtém por demanda. Essa validação é realizada sempre que um arquivo é aberto e quando um novo bloco é solicitado ao servidor. Após a validação, os blocos do arquivo são considerados válidos por um período finito de tempo, especificado pelo cliente quando da anexação da sub-árvore correspondente. A primeira referência após esse período força uma revalidação dos blocos da *cache* relativos ao arquivo.

Quando um bloco da *cache* é modificado, ele é escalonado para atualização no servidor. A atualização é feita por um mecanismo assíncrono do núcleo do sistema operacional, ocorrendo após um tempo não definido. O núcleo, entretanto, garante que todos os blocos modificados serão atualizados antes do fim da operação de fechamento de arquivo. O NFS faz *cache* de diretórios de forma similar a dos arquivos, entretanto, as modificações em um diretório são atualizadas diretamente no servidor. Quando um arquivo é aberto, a informação temporal do diretório pai também é validada.

Objetivando melhorar o desempenho, o NFS transfere dados através da rede em grandes blocos, tipicamente 8 Kbytes. Além disso, a leitura antecipada de blocos é realizada para otimizar os acessos seqüenciais, sendo que arquivos correspondentes ao código executável de programas menores que um certo limite são transferidos de uma só vez.

A especificação original do NFS não suportava qualquer tipo de replicação de arquivos, contudo, versões mais recentes incluem um certo nível de replicação através de um mecanismo chamado "automounter" [CAL 89]. Esse mecanismo permite que uma sub-árvore remota seja especificada usando-se um conjunto de servidores ao invés de um único servidor. A primeira vez que uma sub-árvore é referida, uma requisição é enviada a cada um dos servidores do conjunto. O primeiro a responder à requisição é escolhido como servidor até a desanexação da sub-árvore. A propagação das modificações de um arquivo para as várias réplicas é desconsiderada pelo NFS e deve ser realizada manualmente. Além disso, a impossibilidade, temporária ou permanente, de comunicação com o servidor não ocasiona o acesso a um segundo servidor.

O controle de acesso aos arquivos remotos não recebe tratamento especial por parte do NFS. Cada *RPC* executada por um cliente carrega consigo a identificação do usuário que a gerou. O servidor assume, temporariamente, a identidade do usuário em questão e a validação de acesso ocorre como se o usuário estivesse executando uma sessão diretamente no nodo servidor. O controle de acesso é provido pelo sistema de arquivos original, sobre o qual se executa o NFS. No caso do UNIX, o esquema de bits de permissão [BAC 87] para o dono, grupo do dono e outros é utilizado para validar acesso a arquivos individuais.

As primeiras implementações de NFS pressupunham uma rede de computadores confiável, pois a identificação de um usuário era feita por um cliente e aceita pelos demais servidores sem maiores verificações. O nível de segurança de um ambiente executando NFS era, então, equivalente ao do

nodo menos seguro no ambiente. Para diminuir a vulnerabilidade, as *RPC*s executadas em nome do super-usuário "root" são tratadas pelos servidores como se tivessem sido executadas por um usuário inexistente "nobody". Desta forma, o super-usuário recebe o mais baixo nível de acesso aos arquivos remotos. Essa decisão diminuiu a vulnerabilidade dos ambientes que executam NFS em detrimento da facilidade de administração dos mesmos, pois, agora, o super-usuário precisa executar sessões remotas para manipular arquivos remotos.

Versões mais recentes de NFS podem ser configuradas para prover um nível mais alto de segurança através de um mecanismo de autenticação de clientes e servidores baseado em chave pública [NEC 90]. Mas, como os dados transferidos pela rede em um ambiente NFS não são criptografados, a vulnerabilidade à exposição ou à alteração de informações ainda está presente.

Maiores informações sobre implementações específicas de NFS podem ser obtidas com os fabricantes. A implementação da SUN está descrita em [SAN 85] e em [SUN 90]. O protocolo NFS está descrito em [SUN 89].

6 DOMAIN

O sistema DOMAIN começou a ser desenvolvido pela APOLLO COMPUTERS no início da década de 80. O projeto teve como meta a definição de um ambiente distribuído de estações de trabalho conectadas em rede local para serem utilizadas por um grupo de colaboradores engajados em projetos afins.

O DOMAIN foi projetado sobre uma plataforma proprietária da APOLLO, envolvendo redes com topologia de anel e taxa de transmissão de 12 Mbps. A oportunidade de influenciar o desenho do *hardware*, sem dúvida contribuiu para o bom desempenho do sistema, contudo, a escolha de uma plataforma proprietária fez com que o DOMAIN ficasse restrito a instalações APOLLO e, conseqüentemente, não atingisse o mesmo sucesso do NFS.

Em instalações que executam DOMAIN, é comum ver-se nodos dedicados a fornecer recursos (servidores) e nodos que apenas usam esse recursos (clientes). Entretanto, isso é apenas uma convenção: o DOMAIN trata todos os nodos como pares, sem distinção entre clientes e servidores.

O DOMAIN provê suporte à distribuição de arquivos através do OSS (OBJECT STORAGE SYSTEM), sobre o qual é implementado um sistema de arquivos com interface compatível com UNIX. O OSS suporta um esquema de tipagem de arquivos que permite que um usuário defina novos tipos de arquivos, bem como seus métodos de acesso. Quando um arquivo é aberto, o sistema verifica o seu tipo e, caso necessário, carrega dinamicamente o código correspondente a implementação dos métodos daquele tipo, expandindo o subsistema de entrada e saída.

Outras características importantes do DOMAIN incluem a transparência de localidade de arquivos, um mecanismo para garantir a integridade dos dados de arquivos compartilhados, um esquema de nomenclatura uni-

forme, bom desempenho e facilidades administrativas. Essas características serão detalhadas a seguir.

Os objetos de um ambiente DOMAIN estão associados a identificadores únicos que envolvem, entre outras informações, o identificador do nodo onde o objeto foi criado (identificador interno da APOLLO) e a data de criação do objeto. O identificador do nodo é utilizado apenas para garantir a unicidade do identificador de objeto, não contrariando a meta de transparência de localidade.

O DOMAIN não permite que objetos sejam replicados, e assim, um dado objeto, em um dado instante, está localizado em um único nodo. Entretanto, ele permite que um objeto migre de um nodo para outro, por exemplo, porque as unidades de armazenamento do nodo onde um objeto se encontra estão cheias e existem outros nodos com capacidade para armazenar o objeto.

A transparência de localidade é alcançada pelo DOMAIN através de um servidor heurístico, que avalia um conjunto de fatores para fornecer sugestões sobre a localização de um dado objeto. Entre os fatores considerados por este servidor estão o local onde o objeto foi criado e a localização do diretório pai do objeto. A fim de localizar um objeto, o OSS solicita uma sugestão ao servidor heurístico e então solicita ao OSS remoto (os servidores básicos que formam o OSS devem executar em todos os nodos) que confirme a presença do objeto. Caso a confirmação não ocorra, o OSS solicita uma nova sugestão e o procedimento se repete. Migrações aleatórias de objetos tornam esse esquema de localização ineficiente, uma vez que as sugestões fornecidas pelo servidor heurístico perdem significado e a probabilidade de acerto pode ser tão baixa quanto $\frac{1}{n}$, onde n é o número de nodos no domínio.

O DOMAIN utiliza um servidor de nomes para mapear nomes de objetos em identificadores. Esse servidor provê um espaço de nomes hierárquico, com estilo UNIX, para todos os objetos do domínio, ou seja, todos os nodos do

domínio enxergam uma mesma árvore de diretórios. A árvore de diretórios é implementada como uma base de dados que pode ser replicada em vários OSS ao longo do domínio.

O DOMAIN, através do OSS, faz *cache* de páginas de dados de objetos locais e remotos. As páginas modificadas são periodicamente atualizadas no nodo que contém o objeto. A consistência das páginas da *cache* é verificada da seguinte forma: cada objeto tem associada uma informação temporal relativa a sua última atualização; essa informação é armazenada junto com cada página do objeto na *cache*; para validar um acesso, o OSS compara a informação temporal da *cache* com a do objeto; caso o objeto tenha sido atualizado mais recentemente que as páginas da *cache*, todas as páginas relativas ao objeto são descartadas e reobtidas por demanda.

A manutenção da *cache* no DOMAIN está integrada ao mecanismo de controle de concorrência. Cada nodo executa um servidor de *locks* que sincroniza o acesso aos objetos nele contidos. Esse servidor fornece dois tipos de *locks*: o primeiro permite múltiplas operações de leitura provenientes de nodos distintos ou uma única operação de escrita; o segundo permite múltiplas operações de leitura e escrita em um único nodo. Desta forma, quando um processo obtém um *lock* de escrita, os processos leitores, ou não existem, ou estão localizados no mesmo nodo que o processo escritor. Isso simplifica a gestão da *cache*, pois as alterações das páginas dos objetos não precisam ser dinamicamente propagadas pelo domínio, mas, inviabiliza a implementação de aplicações distribuídas que envolvam múltiplos processos leitores e escritores.

A obtenção e liberação de *locks* é feita pelas aplicações. Quando uma aplicação obtém um *lock* de um objeto, as páginas da *cache* relativas ao objeto são validadas. Sempre que um *lock* de escrita é liberado, todas as páginas modificadas são atualizadas no objeto. O DOMAIN não provê qualquer mecanismo para tratar *deadlocks* ou má utilização de recursos, passando às aplicações essa responsabilidade.

A segurança de um ambiente DOMAIN está baseada na integridade física da rede e na confiança mútua entre os nodos participantes. Nenhum mecanismo de autenticação é implementado.

O controle de acesso ao sistema é feito de forma similar ao UNIX, através de uma base de dados com senhas criptografadas que mapeia nomes de usuários em identificadores. A diferença está na maneira como o DOMAIN define os identificadores de usuário, que incluem informações sobre os projetos com os quais o usuário está envolvido, sua instituição de origem (laboratório, por exemplo) e o nodo ao qual ele está conectado.

O acesso aos objetos é validado através de listas de acesso, associadas a cada objeto, que mapeiam identificadores de usuários em permissões. Parte dos identificadores de usuários podem assumir o valor "todos", por exemplo, todos os usuários envolvidos com o projeto "X" têm permissão para alterar o objeto.

A maneira como o DOMAIN define os identificadores de usuário permite que a administração do sistema seja distribuída, pois possibilita que múltiplos identificadores estejam associados a permissões irrestritas de acesso aos objetos de um projeto, ou de uma instituição. Isso simplifica a administração de grandes domínios, como a sede da APOLLO, que em 1989 contava com 3500 nodos [SAT 90].

Informações complementares sobre o sistema DOMAIN podem ser obtidas em [LEV 87] e [SAT 90].

7 ANDREW

O sistema ANDREW começou a ser desenvolvido na CARNEGIE MELLON UNIVERSITY em 1983. Trata-se de um ambiente distribuído de estações de trabalho, onde cada estação executa 4.3 BSD [LEF 89]. O projeto envolveu todo o campus da universidade, com aproximadamente 5000 nodos, fazendo com que o fator de escala assumisse papel fundamental no desenho do ANDREW.

Através de um conjunto confiável de nodos servidores dedicados, o ANDREW implementa um sistema de arquivos distribuído por todo o domínio. Esse sistema de arquivos é apresentado aos nodos com um espaço homogêneo de nomes, independente de localidade e com hierarquia similar a do UNIX.

O espaço de nomes de arquivos é dividido em duas partes: local e compartilhado. O espaço local é particular a cada nodo e contém apenas arquivos necessários à inicialização do nodo. O espaço compartilhado de nomes é independente de localidade e comum a todos os nodos. Os arquivos de usuário ficam no espaço compartilhado, possibilitando que os usuários se desloquem livremente pelo domínio.

O espaço compartilhado de nomes de arquivos é particionado em sub-árvores, sendo que cada uma delas está inteiramente contida em um único servidor. A localização dos arquivos é definida a partir de uma base de dados, replicada em todos os servidores, que mapeia sub-árvores em servidores.

O ANDREW permite que uma sub-árvore seja movida de um servidor para outro. Para evitar que inconsistências temporárias na base de dados de localização de sub-árvores gerem problemas ao sistema, sempre que uma sub-árvore é movida, informações sobre o seu paradeiro são deixadas no nodo onde ela se encontrava originalmente.

O ANDREW faz *cache* dos arquivos do espaço compartilhado de nomes nos discos locais das estações de trabalho, assim, todos os nodos participantes devem possuir discos e executar um gerente de *cache*. Quando um arquivo é aberto, o gerente de *cache* verifica a presença de uma cópia do arquivo no disco local. Caso tal cópia exista, a operação de abertura é tratada como uma abertura de arquivo local. Caso contrário, uma cópia atualizada do arquivo é obtida por demanda junto ao servidor correspondente. As operações de leitura e escrita em arquivos abertos são sempre locais e não envolvem tráfego na rede. Se a cópia local de um arquivo é modificada, ela é transferida de volta ao servidor quando o arquivo é fechado.

Quando um cliente copia um arquivo remoto para o disco local, o servidor que forneceu a cópia registra o arquivo e o cliente em suas tabelas. Caso o arquivo seja atualizado no servidor, todos os clientes com cópias daquele arquivo são notificados. Entretanto, a cópia local só será atualizada quando uma nova operação de abertura for solicitada, não afetando os processos que, eventualmente, tenham aberto o arquivo antes da atualização.

Apesar de exigir que uma sub-árvore esteja inteiramente contida num servidor, o ANDREW permite que sub-árvores sejam replicadas, apenas para leitura, em múltiplos servidores. Somente a sub-árvore original pode ser atualizada e a propagação das atualizações para as réplicas é feita por um procedimento administrativo não automático.

Os mecanismos operacionais do ANDREW estão baseados numa estrutura chamada volume. Um volume é um conjunto de arquivos fisicamente contidos em um mesmo servidor e que formam uma sub-árvore do espaço compartilhado de nomes, ou seja, o volume é a visão física da sub-árvore. O deslocamento de um volume de um servidor para outro é possível mesmo com o volume ativo. Réplicas, apenas para leitura, de um volume podem ser obtidas através de uma operação de clonagem. Outros procedimentos administrati-

vos, como a definição de quotas de disco e geração de *backups*, também são executados sobre volumes.

A segurança de um ambiente ANDREW está baseada na integridade de um pequeno número de nodos servidores. Esses servidores são fisicamente seguros, estão acessíveis apenas para um pequeno número de administradores confiáveis e executam apenas *software* confiável de sistema. Esses servidores não confiam na rede e nem nos demais nodos.

Para prover acesso seguro aos servidores, o ANDREW utiliza mecanismos de autenticação e transmissão segura baseada em criptografia. O ANDREW adota o sistema de autenticação KERBEROS, definido originalmente pelo MASSACHUSETTS INSTITUT OF TECHNOLOGY para o projeto ATHENA [STE 88]. Quando um usuário abre uma sessão, a sua senha é utilizada para estabelecer um canal seguro de comunicação com o servidor KERBEROS. Através desse canal, o usuário obtém um par de chaves de autenticação que será usado, no futuro, para estabelecer conexões seguras de *RPC* entre clientes e servidores. O servidor de autenticação pode ser replicado na forma de um servidor mestre e vários servidores escravos, sendo que todos devem executar em nodos fisicamente seguros.

O controle de acesso aos arquivos é feito através de uma extensão do esquema de bits de proteção do UNIX, que, para o ANDREW, serve apenas para dizer o que pode ser feito com o arquivo (leitura, escrita e execução). A identificação de quem pode executar essas operações sobre o arquivo, diferentemente do UNIX, é definida por uma lista de acesso associada a cada diretório. As listas de acesso envolvem usuários e grupos de usuários, sendo que os grupos definidos pelo ANDREW podem ser combinados para formar novos grupos.

Informações complementares sobre o sistema ANDREW podem ser obtidas em [MOR 86], [SAT 90] e [TAN 92].

8 AMOEBA

O sistema AMOEBA se originou na VRIJE UNIVERSITEIT de Amsterdam em 1981 como um projeto de pesquisa em computação distribuída. Diferentemente dos sistemas operacionais distribuídos convencionais, o AMOEBA não foi implementado como extensão de um sistema operacional tradicional. Trata-se, portanto, de um sistema completamente novo e sem quaisquer restrições de compatibilidade. O sistema de arquivos do AMOEBA também apresenta um conjunto de características peculiares, muitas das quais derivadas diretamente de mecanismos do núcleo do sistema operacional. Assim sendo, para que se possa descrever o sistema de arquivos, deve-se antes descrever algumas características do núcleo.

8.1 Núcleo

O AMOEBA é um sistema operacional baseado em objetos. Um objeto, para o AMOEBA, é uma estrutura de dados encapsulada, sobre a qual usuários autorizados podem executar funções bem definidas, independentemente da localização de usuários e objetos. Objetos são entidades passivas que não contém processos ou métodos. Ao invés disso, cada objeto está associado a um servidor que o gerencia.

Para executar uma operação sobre um objeto, remoto ou local, processos clientes utilizam um mecanismo de *RPC* implementado no núcleo do sistema operacional. O mecanismo de *RPC* do AMOEBA é síncrono, isto é, após iniciar uma *RPC*, o fluxo cliente fica, obrigatóriamente, bloqueado até receber a resposta do servidor.

8.1.1 Identificação e proteção de objetos

Os objetos de um ambiente AMOEBA são identificados e protegidos por capacidades (*capabilities*) [MUL 90]. Como se pode ver na figura 8.1, uma capacidade possui quatro campos:

- Número da caixa postal do servidor: identifica um canal de comunicação entre os clientes e o servidor que implementa os métodos do objeto. O número da caixa postal não inclui informações sobre a localização do servidor, que é determinada pelo núcleo através de um mecanismo que será descrito adiante.
- Número do objeto: é utilizado pelo servidor para identificar o objeto. No caso do servidor de arquivos, o número do objeto pode ser comparado ao número do "i-node" de um sistema UNIX.
- Permissões: identifica as operações que o detentor da capacidade pode executar sobre o objeto.
- Verificador: é utilizado pelo servidor para validar a capacidade.

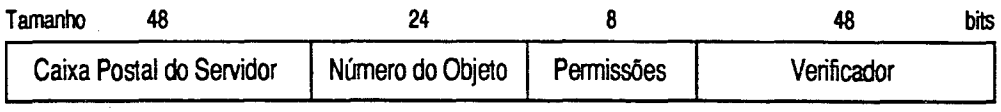

Figura 8.1: Uma capacidade no AMOEBA

Quando um cliente solicita a criação de um objeto, o sistema associa ao mesmo uma capacidade e a retorna ao cliente. Essa capacidade deve ser apresentada ao servidor sempre que o cliente desejar executar alguma operação sobre o objeto.

Para garantir a autenticidade de uma capacidade, o servidor, quando da criação da mesma, escolhe aleatoriamente um número (verificador) e o inclui, tanto na capacidade, como em suas tabelas internas. Todos os

bits de permissão de uma capacidade nova têm valor verdadeiro, o que identifica a capacidade do dono do objeto. Quando a capacidade é retornada ao servidor juntamente com uma solicitação de serviço, o verificador é conferido.

Para criar uma capacidade restrita, um cliente (dono de um objeto) envia uma mensagem ao servidor contendo a capacidade original e os novos bits de permissão. O servidor faz um ou-exclusivo (XOR) do verificador da capacidade original com os novos bits de permissão, e então passa o resultado por uma função sem inversa. A nova capacidade é retornada ao cliente, que pode repassá-la a quem lhe convier. Quando um cliente envia uma capacidade restrita ao servidor, o servidor a identifica pelos bits de permissão, busca em suas tabelas o verificador original, faz um XOR com os bits de proteção e passa o resultado pela função sem inversa. Se o resultado conferir com o campo verificador, a capacidade apresentada é considerada válida.

O algoritmo descrito é probabilístico, pois é possível que um usuário escolha, aleatoriamente, um valor válido para o verificador. Entretanto, a probabilidade de acerto é de 1 em 2^{48}. Outra restrição do algoritmo é que, como as capacidades são manipuladas diretamente pelos usuários, é possível que todas as capacidades que identificam um certo objeto sejam destruídas, tornando o objeto inacessível, apesar de reter recursos. Para reaver tais recursos, o AMOEBA, periodicamente, pesquisa todos os objetos para identificar e remover os que são inacessíveis.

8.1.2 Localização de objetos

A localização de um objeto em um ambiente AMOEBA está baseada na restrição de que, se um nodo contém um objeto, então ele também contém o servidor que implementa os métodos de manipulação daquele objeto. Desta forma, basta que se determine a localização dos servidores para que se tenha

a localização dos objetos. Isso justifica a presença do número da caixa postal do servidor nas capacidades dos objetos.

A determinação da localização de uma caixa postal é feita com o auxílio de uma *cache*, mantida pelo núcleo do AMOEBA no nível de implementação de *RPC*s, que associa caixas postais a processos. Quando um processo cria uma caixa postal, o núcleo registra o número (endereço) da caixa postal na *cache*. Quando um cliente executa uma *RPC* para uma caixa postal, o núcleo consulta a *cache* para determinar a identificação do processo destinatário. Caso a caixa postal não esteja registrada na *cache*, o núcleo envia uma mensagem de difusão requisitando informações sobre ela. Se mais do que um processo estiver atendendo em uma mesma caixa postal, todos eles serão incluídos na *cache* e um será escolhido para receber a *RPC*.

A *cache* descrita acima relaciona caixas postais a processos. Para determinar a localização dos processos, o AMOEBA faz uso de um protocolo chamado FLIP (FAST LOCAL INTERNET PROTOCOL) [TAN 92]. Cada processo criado no AMOEBA é associado a um número chamado endereço FLIP. Esse número acompanha o processo desde a sua criação até a sua destruição, independentemente das possíveis migrações que o processo venha a sofrer durante sua existência. O núcleo do sistema mantém uma segunda *cache* no nível de implementação do FLIP que associa processos a nodos. A atualização desta *cache*, como na *cache* do nível de *RPC*, é feita por difusão. A versão 5.0 AMOEBA não permite migrações de processo, mas quando esse mecanismo for habilitado em versões futuras, ele deverá incluir cuidados com a integridade da *cache*.

A principal vantagem em ter-se esse mecanismo de indireção {caixa postal – > FLIP – > nodo} ao invés de um mecanismo direto com {caixa postal – > nodo} é possibilitar a migração transparente de processos. Entretanto, como o AMOEBA pressupõe que os objetos estejam localizados junto com os processos servidores que implementam seus métodos, a migração dos servidores implica na migração de objetos. Além disso, a técnica de localização de

objetos adotada pelo AMOEBA obriga que os métodos de uma certa classe de objetos sejam todos implementados por um mesmo servidor. Por exemplo, se um objeto possui os métodos "lê", "escreve" e "compacta", como a capacidade do objeto identifica uma única caixa postal, mesmo que existam vários processos atendendo na caixa postal, todos devem implementar as três operações. O AMOEBA não permite que um servidor implemente os métodos "lê" e "escreve" e outro servidor implemente o método "compacta".

8.2 Sistema de Arquivos

O AMOEBA considera arquivos como objetos, portanto, todos os mecanismos descritos anteriormente se aplicam a arquivos. O sistema de arquivos é implementado, na versão 5.0, por três servidores: de diretórios, de arquivos e de replicação, que serão descritos a seguir.

8.2.1 Servidor de diretórios

A principal função do servidor de diretórios é mapear nomes de objetos em capacidades. Cada diretório é organizado como uma tabela, onde cada linha descreve um objeto. Os diretórios do AMOEBA também são objetos, e, como tal, estão associados a capacidades. Isso possibilita que diretórios sejam organizados hierarquicamente, formando árvores ou grafos. O servidor de diretórios provê métodos para criação e remoção de diretórios, inserção e deleção de linhas em diretórios já existentes e pesquisa de nomes em diretórios.

O AMOEBA, através do servidor de diretórios, permite que objetos replicados sejam acessados de forma eficiente. Cada linha de um diretório pode conter várias capacidades, em geral, referindo réplicas do objeto gerenciadas por servidores distintos. Quando um cliente solicita ao servidor de diretórios

que traduza um nome de objeto, o servidor retorna o conjunto de capacidades que está associado àquele objeto. O cliente pode, então, escolher uma das capacidades, e, se o servidor associado não estiver ativo, tentar outra.

Outra facilidade fornecida pelo servidor de diretórios é a possibilidade de definir-se diferentes domínios de proteção. Por exemplo, para simular o esquema de proteção do UNIX, um diretório poderia conter uma coluna para o dono, uma para o grupo do dono e outra para os demais usuários. Uma capacidade para um diretório é então uma capacidade para uma coluna de um diretório.

A organização típica de um diretório no AMOEBA envolve uma árvore com raiz privada para cada usuário e com algumas sub-árvores compartilhadas com os demais usuários. Assim sendo, cada usuário enxerga um espaço de nomes próprio. Por convenção, todas as árvores incluem o diretório "public", que é o começo do espaço compartilhado de objetos. Esse diretório contém, entre outras, capacidades para vários recursos do sistema.

Por ser um componente crítico do AMOEBA, o servidor de diretórios foi implementado de forma a ser tolerante a falhas. A estrutura básica do servidor de diretórios é um vetor de pares de capacidades que identificam objetos idênticos armazenados em servidores de arquivos distintos. Como o servidor de arquivos do AMOEBA não permite que objetos sejam modificados, esse vetor é fisicamente armazenado em uma partição de disco própria do servidor de diretórios, fora do controle do servidor de arquivos. Desta forma, o vetor de capacidades de diretórios pode ser atualizado sempre que um diretório for modificado. Esse vetor, além de referir duas cópias dos diretórios, se encontra, ele próprio, replicado em dois servidores de diretórios.

Quando um diretório é criado, o número de objeto de sua capacidade representa um índice para o vetor de capacidades de diretórios. Quando um diretório é modificado, o vetor de capacidades é atualizado e um novo arquivo

imutável é criado para ele. A segunda cópia do diretório é escalonada para geração quando a carga do servidor for baixa. Após a confirmação da criação das duas cópias do diretório e da atualização dos vetores nos dois servidores de diretórios, os arquivos originais relativos ao diretório são removidos. Esse esquema garante um alto grau de tolerância a falhas, mas, apresenta um custo relativamente alto, tanto em termos de ocupação de disco quanto em tempo de execução.

8.2.2 Servidor Bullet

BULLET é o nome do servidor de arquivos do AMOEBA. Esse nome lhe foi atribuído por tratar-se de um servidor muito rápido, com desempenho aproximadamente três vezes superior ao do NFS da SUN [RES]. Para alcançar esse desempenho, o BULLET apresenta uma organização bastante diferente dos servidores de arquivos convencionais. Em especial, os arquivos mantidos pelo BULLET são imutáveis; uma vez criados, eles não podem ser alterados.

Uma vez que os arquivos não podem ser modificados, o tamanho de um arquivo é sempre conhecido no momento de sua criação. Essa propriedade permite que os arquivos sejam armazenados contiguamente em disco, que se faça *cache* de arquivos inteiros em memória principal e que arquivos sejam transferidos em uma única *RPC*. A combinação dessas simplificações aliada a máquinas com grande capacidade de armazenamento (tanto memória principal como secundária) é responsável pelo alto desempenho do BULLET.

Como os arquivos do BULLET são imutáveis, para criar um arquivo, um cliente deve primeiro criar o arquivo inteiro em sua própria memória e, então, transmiti-lo em uma única *RPC* ao servidor, que o armazena e retorna uma capacidade para acessos futuros. Para modificar esse arquivo, o cliente envia a capacidade ao servidor, solicitando que este lhe envie o arquivo na

integra. O cliente, então, o modifica e o reenvia ao servidor. Essa operação cria um novo arquivo, com uma nova capacidade, portanto, após o recebimento da qual, o cliente deve solicitar ao servidor que remova o arquivo original. Esse modelo pode ser alterado para permitir que clientes com pouca memória acessem o servidor de arquivos, possibilitando que arquivos sejam criados e lidos em partes.

Cada disco sob o controle do BULLET possui uma tabela de arquivos com uma entrada para cada arquivo. Essa tabela é carregada em memória quando da ativação do disco e lá permanece até a sua desativação. A tabela de arquivos é similar a tabela de "i-nodes" do UNIX, entretanto, como o BULLET faz uso de alocação contígua, a descrição dos blocos formadores de um arquivo é feita apenas com um ponteiro e um contador de blocos.

A cópia em memória da tabela de arquivos é também utilizada para controle da *cache*. Uma vez que o BULLET faz *cache* de arquivos inteiros em memória, cada entrada da tabela de arquivos possui um ponteiro para a cópia do arquivo na *cache*, se existente. Como os arquivos são imutáveis, mecanismos de sincronização não são necessários. Essa técnica permite que os arquivos sejam acessados com rapidez, mas, gera fragmentação externa, tanto no disco como na memória. Para reduzir os efeitos da fragmentação, o BULLET implementa um mecanismo de compactação.

Para tratar o problema dos arquivos inacessíveis (por terem tido suas capacidades destruídas), o BULLET utiliza uma técnica de envelhecimento. Associado a cada arquivo existe um contador inicializado com um valor máximo. Periodicamente, o servidor executa a operação de envelhecimento, que decrementa esse contador de uma unidade. Qualquer arquivo cujo contador chegue a zero é removido. Para evitar que arquivos em uso sejam removidos, uma outra operação, que retorna o contador ao valor inicial, é executada periodicamente para todos os arquivos listados em algum diretório.

8.2.3 Servidor de replicação

Os objetos gerenciados pelo servidor de diretórios podem ser automaticamente replicados através do servidor de replicação. Quando um objeto é criado, inicialmente apenas uma cópia é feita. O servidor de replicação pode, então, ser chamado para gerar réplicas do objeto.

O servidor de replicação é mantido em execução durante todo o tempo, verificando, periodicamente, partes do sistema de diretórios. Sempre que uma entrada de diretório suposta a ter n capacidades, é encontrada com menos capacidades, o servidor de replicação contacta os servidores envolvidos para providenciar a geração das réplicas.

O servidor de replicação é quem executa o mecanismo de envelhecimento e de compactação utilizados pelo BULLET e outros servidores. Periodicamente, ele envia mensagens aos servidores para decrementar o contador de envelhecimento. Quando um objeto é eliminado, ele executa o procedimento de compactação.

9 COMPARAÇÃO ENTRE OS SISTEMAS APRESENTADOS

Nos capítulos anteriores foram apresentados quatro sistemas de arquivos distribuídos: NFS, DOMAIN, ANDREW e AMOEBA. A escolha desses sistemas levou em consideração, basicamente, dois aspectos: a disponibilidade bibliográfica e a presença de idéias essencialmente diferentes para a solução dos problemas comuns a sistemas de arquivos distribuídos, tornando a análise o mais abrangente possível. A exclusão do MACH, entretanto, merece considerações especiais.

O sistema MACH teve sua origem na CARNEGIE MELLON UNIVERSITY no início da década de 80. As diretrizes do projeto original [ACC 86] envolviam, entre outras coisas, compatibilidade binária com o sistema 4.3 BSD de Berkeley e um sistema de arquivos que suportava acesso remoto transparente a arquivos. Esse sistema de arquivos apresentava características interessantes, como a decisão de não fazer *cache* de dados remotos e a utilização de *links* remotos [1].

Na versão 3.0, o MACH sofreu uma revisão dos objetivos do projeto que alterou radicalmente o sistema. O núcleo foi separado e isolado do restante do sistema e a compatibilidade binária com o 4.3 BSD passou a ser suportada por um servidor específico. Essa nova versão do MACH está descrita em [TAN 92] e [SIL 94], sendo que nenhuma das referências faz menção à presença de qualquer sistema de arquivos distribuído. Desta forma, apesar de ser um sistema distribuído de inegável importância, a descrição do MACH em pouco contribuiria para uma discussão sobre sistemas de arquivos.

[1] O mecanismo de *link* remoto é adotado pelo PYXIS e será descrito no capítulo 15.

A análise dos sistemas de arquivos distribuídos apresentados neste texto permite que se identifique os temas de maior relevância ao assunto. Esses temas são sumarizados a seguir.

9.1 Espaço de Nomes de Arquivos

Uma implementação de sistema de arquivos distribuído pode, no que tange a definição do espaço de nomes de arquivos, adotar uma das seguintes alternativas:

- Espaço de nomes independente para cada nodo do domínio, possibilitando que um mesmo arquivo seja visto com nomes diferentes ao longo do domínio. O NFS e o AMOEBA adotam essa técnica.
- Espaço de nomes compartilhado por todos os nodos do domínio. Existe uma única maneira de se referir a um arquivo em todo o domínio. O DOMAIN define o espaço de nomes de arquivos desta maneira.
- Espaço de nomes misto, ou seja, uma parte privada a cada nodo e outra compartilhada com os demais nodos. É assim que o ANDREW define o espaço de nomes de arquivos.

A uniformidade de nomes ao longo do domínio simplifica o uso do sistema, pois, independentemente do nodo de trabalho, os usuários sempre designam os arquivos pelos mesmos nomes. Contudo, caso o domínio agregue um número grande de usuários com interesses distintos, a imposição de uniformidade dos nomes de arquivos pode se tornar um problema. Uma solução comum é adotar espaços de nomes privados a cada nodo e, caso seja conveniente, uniformizá-los por convenção entre os usuários.

9.2 Localização de Arquivos

A transparência de localidade dos arquivos para os usuários é um dos requisitos para que um sistema de arquivos seja considerado distribuído. Sob responsabilidade do sistema, a tarefa de localização de arquivos pode ser feita por um dos mecanismos abaixo:

- A localização do arquivo está embutida em seu identificador, de forma que a tradução do nome do arquivo em identificador já revela sua localização. Essa técnica é adotada pelo NFS e pelo ANDREW.
- A localização do arquivo é definida através de mensagens difundidas pelo domínio indagando sua localização. Essa é a estratégia utilizada pelo AMOEBA
- Servidores especiais localizam os arquivos a partir da análise de informações levantadas durante a utilização do sistema. O DOMAIN implementa um servidor heurístico para localizar os arquivos.

A decisão de embutir a localização de um arquivo em seu identificador dispensa mecanismos específicos para localização de arquivos, entretanto, confere um caráter estático ao sistema. A implantação de mecanismos específicos para a localização de arquivos permite, entre outras coisas, que um arquivo migre de um nodo para outro e seus usuários continuem acessando o arquivo automaticamente.

9.3 *Cache* de Dados Remotos

Um sistema de arquivos distribuído pode decidir fazer *cache* de dados remotos ou não, dependendo, basicamente, da velocidade do canal de

comunicação que interliga os vários nodos participantes. O gerenciamento de uma *cache* com dados remotos pode não se justificar em redes de alta velocidade. Caso se decida fazer *cache* de dados remotos, pode-se considerar as seguintes alternativas:

- Fazer *cache* apenas de dados imutáveis, isto é, de arquivos com autorização apenas para leitura, de forma que os dados da *cache* não precisem ser validados. Esse é o caso do AMOEBA.
- Fazer *cache* de dados de arquivos com múltiplos leitores e apenas um escritor, atualizando os dados nas *caches* dos leitores a cada modificação na *cache* do escritor. É assim que o DOMAIN implementa seu mecanismo de *cache*.
- Fazer *cache* de dados independentemente do número de leitores e escritores e passar para os usuários a responsabilidade de sincronizar os processo que compartilham arquivos. Esse mecanismo é utilizado pelo NFS.

No ANDREW cada nodo cliente obtém uma cópia dos arquivos remotos que deseja manipular. Esse arquivo só é atualizado no servidor quando é fechado. Dessa forma, a sincronização de acessos concorrentes a arquivos é tratada localmente. Além disso, o ANDREW é o único dos sistemas estudados que faz *cache* de dados remotos em disco, os demais utilizam exclusivamente memória principal.

9.4 Replicação de Arquivos

A grande motivação para se replicar um arquivo é aumentar sua disponibilidade, o que pode melhorar tanto o desempenho, como o grau de tolerância a falhas do sistema. A replicação automática de arquivos, entretanto,

implica em mecanismos complexos que, entre outras coisas, devem gerenciar a propagação das alterações realizadas nas várias réplicas.

Dos sistemas descritos, apenas o ANDREW e o AMOEBA implementam mecanismos que suportam replicação de arquivos, sendo que ambos os sistemas restringem o mecanismo a réplicas imutáveis. O ANDREW permite que os arquivos originais sejam modificados, porém, a propagação das alterações para as réplicas não é automática. O AMOEBA faz réplicas de arquivos de forma automática, contudo, todos os arquivos são imutáveis.

9.5 Segurança de Arquivos

A segurança dos arquivos de um sistema envolve, basicamente, três mecanismos:

- Controle de acesso aos arquivos, para garantir que somente usuários autorizados os acessem.
- Autenticação de clientes e servidores, de forma a garantir que suas identidades não sejam fraudadas.
- Criptografia dos dados, a fim de impedir que dados sigilosos trafegando em redes, cuja segurança física não se pode garantir, sejam expostos.

Todos os sistemas descritos implementam mecanismos de controle de acesso derivados do esquema de bits de proteção do UNIX. O DOMAIN e o AMOEBA não implementam qualquer outro mecanismo de segurança, confiando na integridade física da rede. O NFS implementa um mecanismo que permite garantir a autenticidade da identidade de usuários, clientes e servidores. O ANDREW adota o sistema de segurança KERBEROS, que implementa

mecanismos de autenticação e criptografia para garantir a transmissão segura dos dados.

Parte III

IMPLEMENTAÇÃO DO PYXIS

"Nunca ande pelo caminho traçado,
pois ele conduz somente até onde os outros já foram."

(Alexander Graham Bell)

10 OBJETIVOS DO PROJETO

Não é necessário muito esforço para se justificar um projeto de sistema de arquivos distribuído, uma vez que acessos remotos a arquivos fazem parte do cotidiano de uma parcela considerável da comunidade de usuários de informática. Contudo, esses acessos raramente são transparentes aos usuários, que se vêem envolvidos em atividades cuja natureza foge de sua compreensão.

O PYXIS é uma proposta de sistema de arquivos distribuído que visa prover métodos para que os usuários possam realizar acessos remotos a arquivos de forma transparente, eficiente e segura.

A fase de definição das metas do projeto, devido à complexidade inerente de um sistema de arquivos distribuído, se estendeu por longo tempo. Algumas das características do PYXIS foram definidas a priori, outras apenas durante o desenvolvimento da primeira versão. Mas houve uma linha de pensamento que esteve presente durante todas as etapas: tratar cada problema na sua essência, buscando as soluções mais simples e eficientes possíveis.

Resumidamente, o projeto tem como objetivo a definição de um sistema de arquivos com as seguintes características:

- Compatibilidade com POSIX.1: O PYXIS não é um projeto isolado, devendo ser englobado por um projeto de âmbito maior, além de ser usado como material de apoio ao ensino de sistemas distribuídos. Para facilitar a utilização do sistema por parte dos usuários e para suportar a execução de *software* aplicativo disponível comercialmente, o PYXIS inclui bibliotecas compatíveis com UNIX, mais especificamente, com a norma 1003.1 (POSIX.1) do INSTITUTE OF ELETRICAL AND ELECTRONICS ENGINEER [IEE 88].

- Portabilidade: A fim de permitir o porte para outras plataformas, o sistema foi escrito na linguagem C++ e traduzido para C ANSI.

- Heterogeneidade de equipamentos: O PYXIS é independente de qualquer aspecto de *hardware*, o que possibilita sua execução em ambientes heterogêneos. Toda a funcionalidade necessária à execução do sistema é fornecida por um núcleo que gerencia os processos e implementa um mecanismo de comunicação entre eles.

- Execução em redes de computadores: A INTERNET, através do protocolo TCP/IP, foi escolhida para caracterizar a implementação do PYXIS por ser a rede mais abrangente e mais estável dos dias atuais. Assim, os aspectos do sistema relativos a nomenclatura de nodos, identificação de nodos, roteamento de mensagens e outros, são diretamente herdados da INTERNET [POS 87]. Isso possibilita que o PYXIS tenha seu escopo de utilização equiparado ao da própria rede INTERNET.

- Paralelismo: Um aspecto muito explorado na definição do PYXIS foi o aproveitamento do paralelismo natural de um sistema de arquivos, tanto no sentido de paralelisar a execução de um serviço como no sentido de executar múltiplos serviços concomitantemente. O primeiro esforço para a paralelisação do PYXIS foi definir a sua implementação não como um único servidor, mas como uma coleção de servidores. O segundo foi definir que cada servidor fosse implementado como uma *task* com múltiplos *threads* de execução.

- Distribuição dos servidores: Nos primórdios do desenvolvimento do projeto não se tinha uma plataforma definida, as possibilidades envolviam plataformas tão diferentes como uma rede local de PCs e um multicomputador em desenvolvimento no CPGCC/UFSC [COR 93]. Isso influenciou o projeto no sentido de que não se tinha ao certo o nível de distribuição do sistema, que poderia variar de

uma rede de computadores, até o caso do multicomputador, onde cada servidor poderia executar em um processador próprio. O sistema foi, então, desenhado para ser flexível no que diz respeito a distribuição de seus servidores, de tal forma que uma alteração no nível de distribuição envolvesse apenas uma reconfiguração do sistema, dispensando recompilações e religações.

- Distribuição da árvore de diretórios: Uma vez que se possibilitou a distribuição dos servidores do sistema de arquivo, decidiu-se também permitir a distribuição da árvore de diretórios, de forma que um cliente pudesse acessar um arquivo remoto anexado à árvore de diretórios local de forma transparente.

- Idempotência: Uma questão muito relevante para sistemas distribuídos sobre redes de computadores é a tolerância às falhas da rede, assunto que não está no escopo deste projeto. Entretanto, para dar alguma robustez ao sistema, decidiu-se que o sistema de arquivos seria idempotente. Desta forma, os servidores não guardam informações sobre o estado dos clientes e cada mensagem de solicitação de serviço carrega todas as informações relativas ao serviço, o que, além de facilitar a definição de um mecanismo que permita a replicação de servidores, simplifica o processo de reativação dos servidores após uma falha.

- Replicação de servidores: Como o PYXIS é idempotente e foi desenhado para ser executado, também, em multicomputadores, decidiu-se incluir mecanismos para possibilitar a replicação de processos servidores. Dessa maneira, um certo nodo pode ter n cópias de um dado servidor executando ao mesmo tempo, elevando assim a taxa de execução de serviços.

- Autenticação de mensagens: Um sistema distribuído é muito mais vulnerável do que um centralizado, principalmente se fizer uso de re-

des abertas como a INTERNET. O PYXIS implementa um mecanismo que possibilita garantir a autenticidade das mensagens trocadas entre clientes e servidores.

11 MODELO ARQUITETURAL

O PYXIS está baseado no modelo cliente-servidor, portanto, o sistema de arquivos é implementado por servidores que são acessados pelos clientes através de *RPC*s. Um ambiente próprio à execução do PYXIS pressupõe a existência de um núcleo que suporte o modelo, bem como de servidores específicos para cada tipo de dispositivo periférico presente. Nem o núcleo, nem os servidores de dispositivo fazem parte do escopo desse projeto, contudo, foram desenvolvidos ou simulados para permitir a operacionalização do sistema de arquivos.

Na maioria dos sistemas operacionais, o sistema de arquivos é implementado por um único servidor. No caso do PYXIS, levou-se em conta a complexidade de um sistema de arquivos e, com vistas a aproveitar o seu paralelismo implícito, decidiu-se subdividi-lo em três servidores:

- Um servidor encarregado de traduzir nomes de arquivos em identificadores internos, implementando as estruturas de diretórios.
- Um servidor responsável pela gerência de memória secundária, implementando o conceito de arquivo e estendendo o mesmo aos dispositivos de entrada e saída.
- Um servidor para armazenar, temporariamente, dados de memória secundária, visando otimizar o acesso a eles.

Uma visão parcial de um sistema computacional executando PYXIS pode ser vista na figura 11.1.

A distribuição das tarefas do sistema de arquivos em três servidores pode, por si só, trazer grande ganho de desempenho ao sistema. Se, por exemplo, o PYXIS estiver executando em um multicomputador, pode-se ter o caso

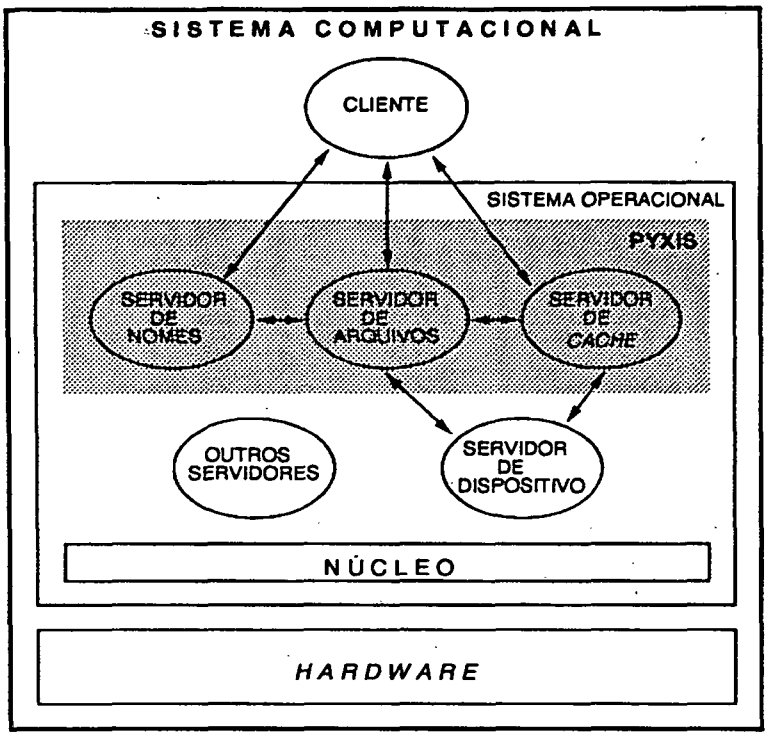

Figura 11.1: O PYXIS em um sistema computacional

onde cada um dos seus servidores executa paralelamente em um processador, elevando, indubitavelmente, a taxa de execução de serviços do sistema de arquivos. Mesmo quando o PYXIS estiver executando concorrentemente em um único processador, existem vantagens que compensam a perda de desempenho causada pela troca de mensagens, como por exemplo a facilidade de substituir-se ou acrescentar-se servidores com o sistema em execução, sem necessidade de religações de código.

Pelo que foi descrito até o momento, não se pode fazer considerações sobre a execução paralela dos serviços do PYXIS, uma vez que a maior parte deles é de natureza seqüencial. Considere o exemplo do serviço utilizado para se obter um descritor de arquivo: primeiramente o processo de usuário (cliente) envia uma mensagem para o servidor de nomes solicitando a tradução do nome do arquivo para um identificador interno. Para efetuar a tradução, o servidor de nomes solicita a leitura de diretórios, que são implementados como arquivos, ao servidor de arquivos. Isso termina bloqueando o servidor, não

pela maneira como ocorre a comunicação, que é assíncrona, mas pela natureza seqüencial do serviço solicitado. O servidor de nomes não pode responder à solicitação antes de ter recebido a resposta do servidor de arquivos.

Para permitir que os servidores atendam a mais do que uma solicitação de serviço ao mesmo tempo, decidiu-se implementar cada um deles como uma tarefa com múltiplos fluxos de execução. Assim, sempre que um serviço puder bloquear um servidor por tempo considerável, ele será executado por um novo fluxo. Mais especificamente, um novo fluxo é gerado sempre que um serviço envolver alguma operação de entrada e saída. Desta forma, cada um dos servidores componentes do sistema de arquivos constitui uma tarefa, com um fluxo básico que faz a inicialização das estruturas de dados e fica em um laço, recebendo mensagens de solicitação de serviços, chamando as respectivas funções (que podem ou não gerar um novo fluxo) e, dependendo da situação, enviando mensagens de resposta (em geral, a resposta de um serviço é enviada pelo fluxo que o executa). Um exemplo simplificado da codificação de um servidor pode ser visto a seguir:

```
main ()
{
  Mail_Box_Id m_box_id;
  Message     message;
 /* laco principal do servidor */
 do
 {
    /* recebe solicitacao de servico */
    receive(&m_box_id, &message, sizeof(Message));
    /* seleciona o servico a ser executado */
    switch (message.service)
    {
      case SERVICE_1: do_service_1(m_box_id, message);
                      break;
      case SERVICE_2: do_service_2(m_box_id, message);
                      break;
      default:        send(m_box_id, <SERVICO INVALIDO>, ...);
    }
  } while TRUE;
}

/* rotina que executa o servico 1 */
do_service_1(Mail_Box_Id m_box_id, Message message)
{
  /* faz o que deve ser feito */

  /* envia resposta do servico */
  send(m_box_id, <RESPOSTA DO SERVICO>, ...);
}

/* rotina que executa o servico 2 */
do_service_2(Mail_Box_Id m_box_id, Message message)
{
  int pid;
  /* cria um novo fluxo para executar o servico */
  if(pid = create_thread()) < 0)
  {
    send(m_box_id, <NAO PODE CRIAR O FLUXO>, ...);
    return;
  }
  /* fluxo pai retorna ao laco principal */
  if (pid > 0)
    return;

  /* fluxo filho faz o que deve ser feito */

  /* envia resposta do servico */
  send(m_box_id, <RESPOSTA DO SERVICO>, ...);
  /* fluxo filho termina */
  exit;
}
```

Figura 11.2: Estrutura simplificada de um servidor

12 IDENTIFICAÇÃO DAS ENTIDADES DO SISTEMA

Um sistema de arquivos envolve uma série de entidades que devem ser identificadas, tanto perante o conjunto de usuários quanto perante o sistema de arquivos, como por exemplo, os próprios usuários, os nodos formadores do domínio, os arquivos, os servidores (através de suas caixas postais), entre outras. Para os usuários, tais entidades estão associadas a nomes, porém, o sistema de arquivos faz uso de estruturas internas especiais. A seguir pode-se ver como essas entidades estão definidas no PYXIS.

12.1 Identificação Perante os Usuários

Todas as entidades do sistema de arquivos são referidas pelos usuários através de nomes alfanuméricos quaisquer, contudo, os nomes de nodos e de arquivos seguem convenções específicas.

12.1.1 Nomes de nodos

Um dos objetivos definidos para o projeto é a adequação à execução sobre a INTERNET, portanto, os nomes de nodos de um domínio PYXIS seguem a convenção estabelecida pelo NETWORK INFORMATION CENTER da INTERNET, cujo diagrama de sintaxe é apresentado na figura 12.1.

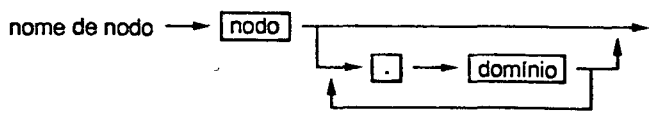

Figura 12.1: Diagrama de sintaxe de um nome de nodo

Se o último elemento de um nome de nodo for um dos sub-domínios oficiais da INTERNET [1], então o nome é absoluto e designa um nodo perante todo o mundo; caso contrário, ele é local ao domínio. Informações complementares sobre a convenção de nomenclatura de nodos na INTERNET podem ser obtidas em [POS 82], [MOC 87a] e [MOC 87b].

12.1.2 Nomes de arquivos

A figura 12.2 mostra, através de um diagrama de sintaxe, a definição de um nome de arquivo.

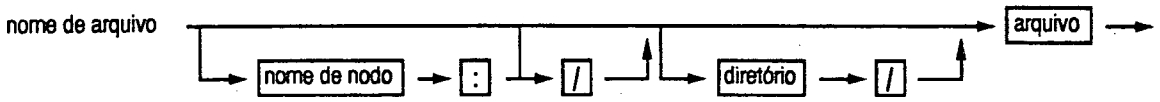

Figura 12.2: Diagrama de sintaxe de um nome de arquivo

Se um nome de arquivo começar com um nome de nodo, então o arquivo é remoto, salvo quando especificado o nodo local. Se começar com "/", ele é relativo à raiz da árvore de diretórios local. Caso contrário ele é relativo ao diretório de trabalho corrente.

12.2 Identificação Perante o Sistema de Arquivos

A fim de identificar as entidades de forma eficiente, o sistema de arquivos faz uso de estruturas especiais, das quais, as mais importantes são descritas a seguir.

[1] Os sub-domínios oficiais da INTERNET são "edu", "net", "mil", "org", "com", "gov" e as siglas oficiais dos países participantes ("br", "de", "jp", etc) [POS 82].

12.2.1 Identificadores de usuários

Usuários são os agentes que manipulam os arquivos do sistema, logo, é importante que o sistema os identifique, possibilitando assim a validação de acessos. Um usuário qualquer é identificado perante o sistema por uma estrutura de 64 bits (figura 12.3) com três campos:

- Identificador de nodo: 32 bits equivalentes ao número IP (INTERNET PROTOCOL) da INTERNET [POS 81a], que identificam um nodo perante o mundo;
- Identificador de grupo: 16 bits que identificam um grupo de usuários em um nodo;
- Identificador local de usuário: 16 bits que identificam um usuário dentro de um determinado grupo.

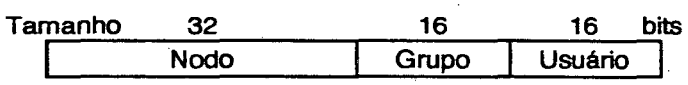

Figura 12.3: Identificador de usuário

12.2.2 Identificadores de caixas postais

As caixas postais utilizadas para a comunicação entre processos são identificadas globalmente por uma estrutura de 48 bits (figura 12.4) formada por dois campos:

- Identificador de nodo: 32 bits equivalente ao número IP da INTERNET;
- Identificador local de caixa postal: 16 bits que identificam uma caixa postal em um nodo.

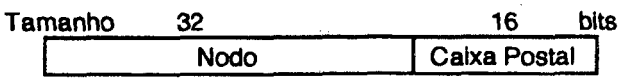

Figura 12.4: Identificador de caixa postal

12.2.3 Identificadores de arquivos

Os identificadores de arquivos adotados pelo PYXIS são estruturas de 96 bits (figura 12.5) formadas por três campos descritos como segue:

- Identificador de nodo: 32 bits equivalente ao número IP da INTERNET;
- Identificador de dispositivo: 32 bits que identificam um dispositivo em um nodo (os 16 bits de mais alta ordem identificam a classe do dispositivo e os 16 restantes a unidade);
- Identificador de arquivo: 32 bits que identificam o descritor de um arquivo em um dispositivo.

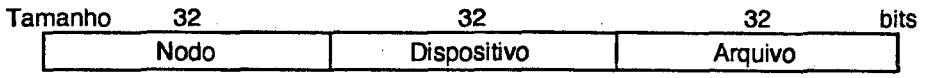

Figura 12.5: Identificador de arquivo

13 COMUNICAÇÃO E LOCALIZAÇÃO DOS SERVIDORES

A comunicação entre clientes e servidores, bem como entre os próprios servidores está baseada no conceito de caixa postal. Quando um processo deseja se comunicar com outros, ele cria uma ou mais caixas postais que serão usadas para receber e enviar mensagens. Desta forma, a entidade identificada na comunicação é a caixa postal e não o processo que a criou.

O uso de caixas postais para a comunicação entre processos permite que se defina um espaço "bem conhecido" de caixas postais, de forma similar aos *Well Known Services [WKS]* da INTERNET [POS 87]. Esse espaço engloba todos os serviços básicos do sistema, de tal forma que, quando um cliente deseje um serviço, ele conheça, a priori, a caixa postal correspondente. Da maneira como os *WKS* são implementados na INTERNET, se um dado serviço está disponível em um nodo, então o respectivo servidor também está disponível naquele nodo. Como essa é uma restrição que contraria as metas definidas para o projeto, por não permitir que os servidores do PYXIS executem em nodos distintos, decidiu-se expandir este mecanismo para que uma caixa postal local possa estar associada a um servidor remoto.

Nessa primeira versão do PYXIS, o sistema de comunicação inclui uma tabela de localização com uma entrada para cada "serviço bem conhecido". Cada entrada da tabela identifica até três servidores aptos a executar os serviços associados à caixa postal. Quando um cliente solicita um serviço, ele o faz enviando uma mensagem a uma caixa postal. Para proceder a entrega da mensagem, o sistema de comunicação consulta a tabela de localização em busca de um par (nodo, caixa postal) que identifica globalmente um servidor. Caso as três referências da tabela sejam nulas, uma mensagem indicando a indisponibilidade do serviço é retornada ao cliente. Caso alguma das entra-

das seja válida, o sistema de comunicação procede ao envio da mensagem à primeira caixa postal. Cada servidor tem associado um tempo máximo de execução do serviço, assim, o sistema de comunicação pode, decorrido o limite de tempo sem que se tenha recebido uma resposta do servidor, proceder ao envio da mensagem ao próximo servidor da lista. Um exemplo de tabela de localização de servidores pode ser visto na tabela 13.1.

SERVIÇO	SERVIDOR 0			SERVIDOR 1			SERVIDOR 2		
CP	Nodo	CP	Exp	Nodo	CP	Exp	Nodo	CP	Exp
0	local	0	5	local	4	5	-	-	-
1	local	1	5	vega	1	50	juno	8	100
2	vega	2	80	-	-	-	-	-	-
:	:	:	:	:	:	:	:	:	:
n	local	n	5	vega	n	50	baco	n	60

CP = Caixa postal Exp = Tempo de expiração

Tabela 13.1: Exemplo de tabela de localização de servidores

O mecanismo de localização de servidores descrito permite a replicação de servidores idempotentes, bem como a definição de serviços equivalentes, pois uma caixa postal pode ser redirecionada para outra no mesmo nodo. A tabela de localização de servidores pode ser alterada por meio de chamadas ao núcleo do sistema operacional, permitindo que, futuramente, um servidor externo faça reconfigurações dinâmicas na tabela, otimizando o tempo de resposta dos serviços. Na tabela 13.1, o serviço 0 é equivalente ao serviço 4; o serviço 1 está, também, disponível nos nodos "vega" e "juno"; o serviço 2 está disponível apenas remotamente no nodo "vega".

A tabela de localização de servidores permite que um cliente localize um servidor de forma transparente, porém o próprio sistema de arquivos está dividido em mais de um servidor, sendo necessário definir-se um mecanismo para que os servidores se localizem entre si. Como o PYXIS não permite a replicação transparente de arquivos, definiu-se que sempre que um nodo contiver um arquivo, ele conterá também a coleção de servidores necessários

para acessá-lo. Desta forma, pode-se tirar do próprio identificador do arquivo a informação sobre a localização dos servidores, ou seja, assim que o servidor de nomes obtém o identificador de um arquivo, as mensagens relativas a ele passam a ser enviadas diretamente ao nodo que o contém.

Um caso especial ocorre quando cada um dos servidores do PYXIS executa em um processador particular de um multicomputador. Neste caso, o multicomputador é visto como um único nodo, sendo que o espaço de "serviços bem conhecidos" é distribuído pelos processadores. O núcleo do sistema operacional ou o controlador de comunicações é responsável pela associação de caixas postais a processadores.

Para manter a interface do sistema de comunicações homogênea, todo o endereçamento de mensagens compreende o par (nodo, caixa postal), sendo que quando o campo "nodo" for nulo, o sistema tentará defini-lo através de uma pesquisa na tabela de localização de servidores.

14 DISTRIBUIÇÃO DOS SERVIDORES

O PYXIS foi desenhado de forma a possibilitar vários esquemas de distribuição, sem que se precise recompilar ou religar o sistema. Todas as alterações necessárias para reconfigurar o sistema se concentram no subsistema de comunicações e não se propagam pelo resto do sistema. Existem três configurações de especial importância que serão discutidas a seguir: computadores isolados, multicomputadores e computadores conectados em rede.

14.1 Computadores Isolados

Embora seja um sistema de arquivos distribuído, nada impede que o PYXIS execute em um único computador isolado. Neste caso, o sistema não apresenta qualquer nível de distribuição, pois todos os arquivos e servidores acessíveis são locais. Esta configuração pode ser vista na figura 14.1.

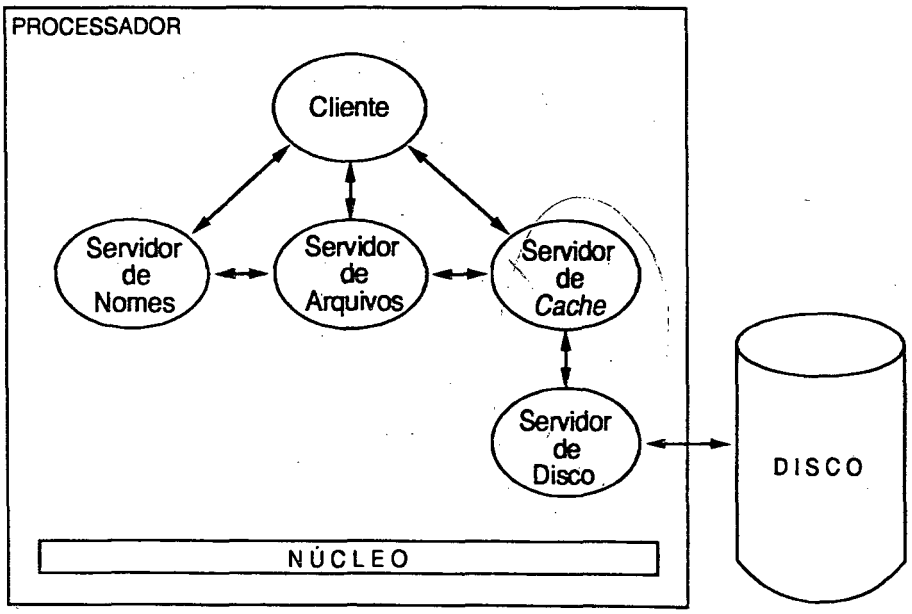

Figura 14.1: O PYXIS em um computador isolado

14.2 Multicomputadores

Para a análise da distribuição do PYXIS em um multicomputador, considerar-se-á que cada servidor executará em um processador distinto, o que pode ser visto na figura 14.2. Um multicomputador é visto como um único nodo, sendo que seus processadores compartilham um espaço comum de caixas postais. Desta forma, qualquer solicitação de serviço será local, porém, envolvendo a troca de mensagens entre os processadores. É do sistema de comunicações do multicomputador a responsabilidade de associar uma determinada caixa postal a um processador.

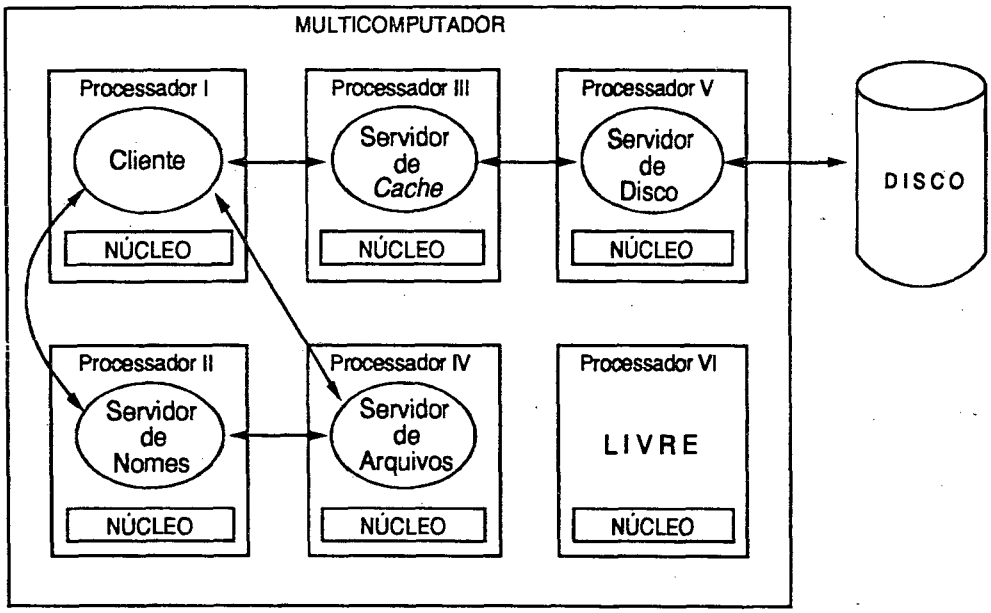

Figura 14.2: O PYXIS em um multicomputador

Reconfigurações dinâmicas do multicomputador, como migrações de processos, podem ser feitas, no que diz respeito ao PYXIS, simplesmente redefinindo-se as tabelas de localização de servidores em cada um dos processadores ou no processador de controle, dependendo da arquitetura da máquina.

14.2.1 Replicação de servidores

Um aspecto interessante da execução do PYXIS em um multicomputador é a possibilidade de replicação dinâmica de seus servidores. O sistema de comunicação do multicomputador poderia ser programado para disparar, automaticamente, réplicas de servidores sobrecarregados e eliminá-las quando ociosas por um determinado tempo.

Considere o seguinte exemplo: o PYXIS é inicializado como mostrado na figura 14.2. Em um dado instante, a fila de mensagens da caixa postal do servidor de arquivos apresenta uma grande quantidade de mensagens. O sistema de comunicação pode, desde que existam processadores livres, executar um procedimento de replicação do servidor de arquivos em um novo processador. A fila de mensagens pode agora ser dividida entre os dois servidores e, caso as solicitações de serviço refiram, eqüitativamente, dois dispositivos controlados por servidores distintos, ter-se-á um ganho de desempenho teórico de 100%.

O mecanismo de replicação descrito pode ser estendido a qualquer servidor idempotente e pode ser executado repetidas vezes, de acordo com a carga do sistema.

14.3 Redes de Computadores

Uma configuração comum para o sistema é distribui-lo através de nodos conectados por uma rede de computadores. Para tal, basta definir-se o estado inicial das tabelas de localização de servidores de acordo com a distribuição desejada. Duas configurações são de especial interesse: nodos sem disco, que acessam exclusivamente arquivos remotos; nodos com disco que compartilham arquivos com outros nodos.

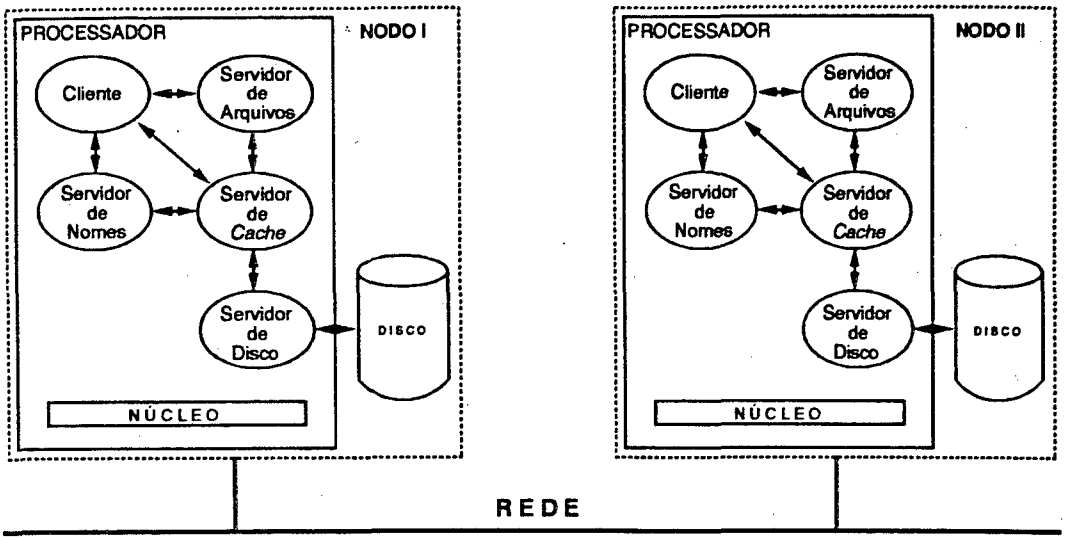

Figura 14.3: O PYXIS em uma rede de computadores

A figura 14.3 mostra o PYXIS configurado de forma a permitir o compartilhamento de arquivos entre dois nodos de uma rede. Nesta primeira versão do PYXIS, um serviço sobre um arquivo remoto é encaminhado ao nodo que contém o arquivo tão logo se conheça a sua localização. Por exemplo, um cliente no nodo I solicita o serviço "open" para um arquivo localizado no nodo II. A solicitação é enviada ao servidor de nomes do nodo I, que retornará o identificador do arquivo do nodo II. Se o cliente solicitar agora o serviço "read" fornecendo o identificador previamente obtido, a mensagem será enviada diretamente para o servidor de arquivos do nodo II. Isso acontece devido ao mecanismo de localização de servidores descrito anteriormente, que extrai do próprio identificador do arquivo a sua localização.

Uma segunda técnica para manter o serviço no nodo local até o nível físico poderia ser implementada dando-se preferência aos servidores locais. Desta forma, a cadeia de solicitações de serviço gerada por um cliente seria local até o nível dos servidores de dispositivo, implicando na buferização local de dados remotos e, conseqüentemente, aumentando o desempenho do sistema. Essa técnica não foi adotada pelo PYXIS por dois motivos: a complexidade do sistema aumentaria bastante, pois seria necessário a definição de um

esquema para garantir a propagação de um bloco de dados alterado em um servidor de *cache* para os outros servidores de *cache* que mantivessem uma cópia do mesmo bloco; e o crescente aumento da velocidade de transmissão de dados em redes locais atenua e talvez até compense a perda de desempenho.

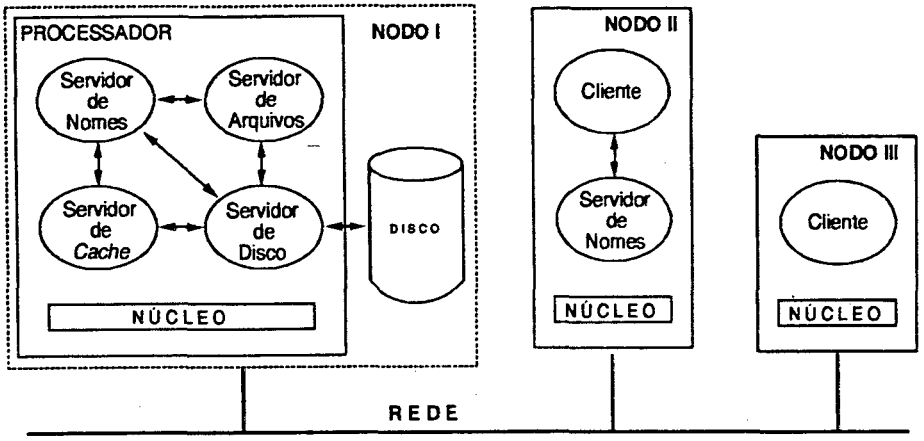

Figura 14.4: O PYXIS em uma rede de computadores com nodos sem disco

A figura 14.4 mostra o PYXIS executando uma rede com três nodos, dois dos quais sem disco. A execução do servidor de nomes em um nodo sem disco é opcional, caso decida-se executá-lo, deve-se fornecer um identificador de arquivo remoto para ser o diretório raiz da árvore de diretórios. Esse identificador deve ser obtido a priori, pois o servidor ainda não está executando, sendo impossível a tradução de nomes de arquivos. A vantagem de se executar o servidor de nomes em um nodo sem disco é a manutenção local de algumas estruturas, reduzindo o tráfego de mensagens entre os nodos. De qualquer maneira, toda a manipulação de dados de usuário será solicitada ao nodo que contém o disco. Outra opção é não se ter qualquer um dos servidores do PYXIS executando localmente, bastando definir-se uma tabela de localização de servidores adequada. O nodo III da figura 14.4 exemplifica esta situação.

Executar os servidores de arquivos, de *cache* ou de dispositivos em um nodo sem disco é absolutamente sem propósito. Nenhum identificador de arquivo fará referência a um nodo sem disco, logo, os servidores nunca receberão solicitações de serviço.

15 DISTRIBUIÇÃO DA ÁRVORE DE DIRETÓRIOS

Os clientes do PYXIS podem solicitar serviços relativos a arquivos remotos fornecendo um nome apropriado, entretanto, a existência de arquivos remotos freqüentemente acessados, torna esse método inconveniente. A alternativa de replicação de arquivos pode trazer grandes dificuldades para que se possa garantir a integridade dos arquivos que vierem a ser alterados em algum dos nodos envolvidos. A alternativa da definição de um espaço de nomes de arquivos único para todo o domínio, de tal forma que todos os arquivos sejam compartilhados por todos os usuários, pode se tornar inviável quando o domínio agregar um grande número de usuários com interesses distintos. Isso justifica o esforço para se definir um mecanismo de compartilhamento de arquivos entre nodos.

A fim de permitir o acesso transparente a arquivos remotos, o PYXIS implementa um servidor de nomes capaz de mapear nomes de arquivos em identificadores internos, independente de eles serem locais ou remotos. Para tal, o servidor de nomes mantém diretórios com entradas similares a da figura 15.1. Uma vez que os identificadores de arquivos adotados pelo PYXIS incluem a identificação do nodo e do disco que os contêm, um mesmo diretório pode apresentar arquivos de discos ou nodos diferentes. Esse mecanismo será designado pelo termo *link* remoto; um exemplo pode ser visto na figura 15.2.

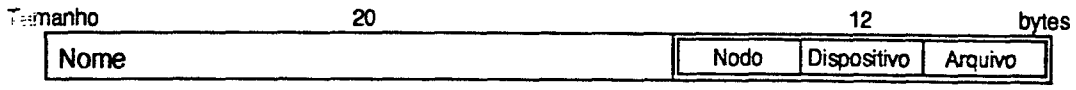

Figura 15.1: Entrada de diretório

A inicialização do servidor de nomes pressupõe o fornecimento do identificador do arquivo correspondente à raiz da árvore de diretórios. Caso se tenha apenas um servidor de nomes no domínio, ou então, todos os servidores

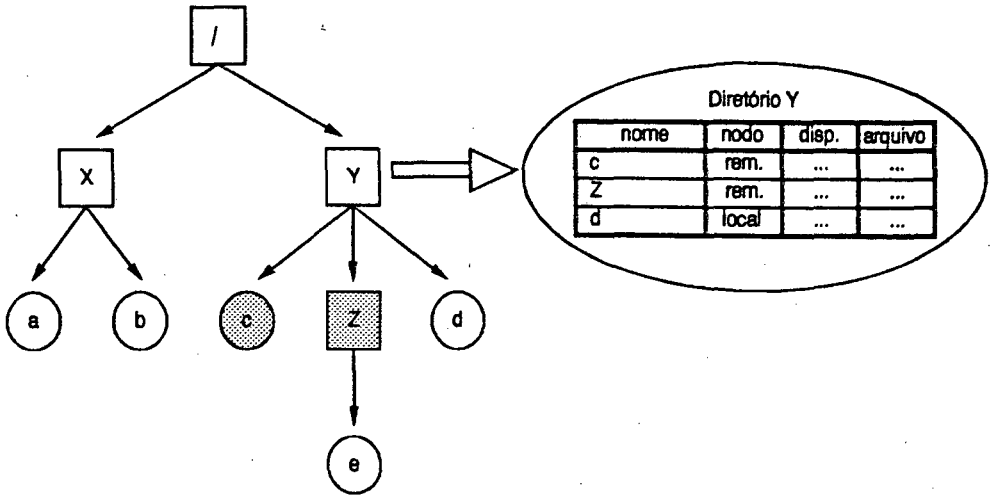

Figura 15.2: Exemplo de árvore de diretórios com *links* remotos

sejam inicializados com a mesma raiz, ter-se-á um espaço de nomes único. Se cada servidor de nomes for inicializado com uma raiz própria, poder-se-á ter espaços de nomes distintos no domínio. Nada impede que árvores de diretórios independentes tenham sub-árvores em comum. Assim sendo, cabe aos usuários decidir se desejam ter um espaço de nomes único, parcialmente compartilhado ou independente em cada servidor.

A utilização de *links* remotos pode degenerar a árvore de diretórios em um grafo cíclico, o que por sua vez, pode ocasionar laços infinitos em aplicações que realizam pesquisa na árvore, como "find" e "tar". Todavia, a maneira como os diretórios são implementados no PYXIS permite às aplicações diferenciar arquivos locais dos remotos, tornando possível a prevenção de laços infinitos e, por conseguinte, minimizando os efeitos negativos da ocorrência de ciclos no grafo de diretórios.

16 AUTENTICAÇÃO DE MENSAGENS

Um problema grave no que diz respeito à segurança de sistemas distribuídos é a autenticação das mensagens de solicitação de serviço. No caso do PYXIS, os servidores não guardam informações sobre o estado dos clientes, agravando bastante esse problema. Todas as mensagens enviadas aos servidores do sistema de arquivos contêm o identificador do arquivo a ser operado. Sem um mecanismo seguro, os servidores não poderiam dizer se o identificador de arquivo foi obtido do servidor de nomes com o respectivo serviço ou se ele foi arbitrado por um cliente descuidado ou mal intencionado.

Para autenticar uma mensagem, o PYXIS faz uso de uma chave gerada pelo servidor de nomes em resposta à solicitação de um serviço especial. Por ser idempotente, o PYXIS não implementa um serviço "open" convencional, resumindo esse serviço à obtenção do identificador de arquivo e da respectiva chave. Para solicitar qualquer outro serviço, o cliente deve apresentar o identificador do arquivo juntamente com a chave, que será usada para validar a solicitação. A chave é gerada como função do próprio identificador do arquivo, da data de criação do arquivo e do identificador do usuário.

A inclusão da data de criação do arquivo na composição da chave de autenticação se deu pelo fato dos servidores do sistema de arquivos não manterem uma tabela de arquivos abertos, mas sim uma *cache* com os descritores dos arquivos mais recentemente acessados. Assim, um cliente pode, por exemplo, obter o identificador de um arquivo e a respectiva chave e ficar algum tempo sem acessar o arquivo, de forma que seu descritor seja removido da *cache*. Um outro cliente pode agora solicitar a remoção deste arquivo. É teoricamente possível que um outro arquivo venha a ser criado e receba o mesmo identificador. O primeiro cliente que obteve o identificador do arquivo passará

a acessar, indevidamente, um novo arquivo. Incluindo a data da criação do arquivo na função geradora da chave elimina-se esse problema.

O identificador do usuário foi envolvido na geração da chave de autenticação para evitar que um usuário abra um arquivo, obtenha o seu identificador e a chave, e transfira autorizações de acesso a um outro usuário originalmente não autorizado. Considere um exemplo onde as autorizações de um arquivo permitam sua leitura por qualquer usuário do grupo do dono do arquivo, mas não pelos demais usuários. Um processo de um usuário do grupo do dono (cliente) poderia abrir o arquivo e mandar uma mensagem contendo o identificador e a chave a um processo de um usuário de fora do grupo do dono do arquivo, ludibriando o esquema de controle de acesso aos arquivos.

17 CONCLUSÕES

Este texto apresentou o PYXIS, um sistema de arquivos distribuído, portável, com alto grau de paralelismo e desenhado de forma a permitir sua execução em ambientes com diferentes níveis de distribuição.

A grande maioria dos sistemas de arquivos distribuídos disponíveis, tanto a nível comercial quanto acadêmico, foram implementados a partir de sistemas centralizados, freqüentemente derivados do UNIX. O PYXIS foi projetado sem qualquer compromisso de compatibilidade com outros sistemas; foram considerados inúmeros algoritmos tradicionais e propostos outros tantos, sempre objetivando um sistema de arquivos distribuído. A compatibilidade com POSIX.1 é provida por uma biblioteca de *stubs* independente dos servidores.

Visando o aproveitamento do paralelismo interno presente em um sistema de arquivos, o PYXIS foi subdividido em três servidores autônomos: servidor de nomes, servidor de arquivos e servidor de *cache*, cada um dos quais implementado por uma tarefa com múltiplos fluxos de execução. O uso de múltiplos fluxos de execução é comumente empregado no desenho de servidores, porém a subdivisão do sistema de arquivos em servidores independentes está presente em um número restrito de sistemas.

A forma pela qual o PYXIS alcança a transparência de localidade de servidores e arquivos é resultado de uma combinação de idéias simples, mas que se mostraram de grande eficiência. A definição de um espaço de caixas postais conhecido por todos os clientes para os serviços básicos e a extração da localização dos servidores secundários dos identificadores de arquivos resolvem por completo a questão da localização dos servidores. A extensão do mecanismo de *link* possibilita que arquivos remotos sejam acessados transpa-

rentemente, além de dispensar a ativação explícita ("mount") das sub-árvores remotas.

A característica de idempotência apresentada pelos servidores do PYXIS, além de simplificar o restabelecimento da comunicação entre clientes e servidores após uma falha, permitiu que se definisse um mecanismo automático de replicação dos servidores, aproveitando o paralelismo provido por arquiteturas multiprocessadoras. Entre os sistemas de arquivos distribuídos estudados, o PYXIS é o único que implementa servidores idempotentes e que aproveita o paralelismo externo apresentado pelo *hardware*.

A proposta original do PYXIS, descrita em [FRO 93], supunha a existência de um núcleo de sistema operacional que suportaria a execução de seus servidores. Esse núcleo se encontra em desenvolvimento no CPGCC/UFSC e só deverá estar operacional no fim deste ano. Decidiu-se então simular esse núcleo sobre um outro sistema operacional, o SUNOS 4.1.3 da SUN MICROSYSTEMS, o que impossibilitou a avaliação do desempenho do sistema, que ficou condicionado ao do sistema hospedeiro.

Os resultados até agora obtidos são muito satisfatórios. Existe uma versão completamente operacional do PYXIS e a simplicidade e clareza de sua implementação permitirão que muito em breve o sistema seja transferido para uma plataforma própria, possivelmente integrando um projeto cooperativo, envolvendo UFSC, UFRGS e UFSM, para o desenvolvimento de um multicomputador e de um ambiente de programação paralela associado ele.

BIBLIOGRAFIA

[ACC 86] ACCETTA, M. et alli, *Mach: a New Kernel Foundation for UNIX Development*, Pittsburgh: Proceedings of the Summer 1986 USENIX Conference, p. 93-112, julho de 1986.

[APP 87] APPLE Computer Inc., *Apple Techinical Introduction to the Macintosh Family*, Reading: Addison-Wesley, 1987.

[BAC 87] BACH, M., *The Design of the UNIX Operating System*, Englewood Cliffs: Prentice-Hall, 1987.

[BIR 84] BIRREL, A. & NELSON, B., *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, v. 2, fevereiro 1984.

[BOO 91] BOOCH, G., *Object Oriented Design with Applications*, Redwood City: Benjamin Cummigs, 1991.

[BRO 85] BROWN, M. & KOLLING, K. & TAFT, E., *The Alpine File System*, ACM Transactions on Computer Systems, v. 3, n. 4, novembro de 1985.

[CAL 89] CALLAGHAN, B. & LYON, T., *The Automounter*, San Diego: Proceedings of the Winter 1989 USENIX Conference, 1989.

[CAS 90] CASTRO, A., *Um Servidor de Arquivos Paralelo para o MINIX*, Porto Alegre: UFRGS, 1990 (Trabalho de Diplomação).

[COM 84] COMER, D., *Operating System Design: The XINU Approach*, Englewood Cliffs: Prentice-Hall, 1984.

[COR 93] CORSO, T., *Ambiente para Programação Paralela em Multicomputador*, Florianópolis: UFSC/CTC/INE, 1993 (Relatório Técnico).

[DIJ 75] DIJKSTRA, E., *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, Communications of the ACM, v. 18, n. 8, agosto de 1975.

[FRO 92] FROHLICH, A., *Um Sistema de Arquivos para o S.O.S.*, Porto Alegre: UFRGS, 1992 (Trabalho de Diplomação).

[FRO 93] FROHLICH, A., *Um Sistema de Arquivos Experimental.*, Florianópolis: UFSC, 1993 (Trabalho Individual).

[GAR 93] GARFINKEL, S. & SPAFFORD, G., *Pratical Unix Security*, Sebastopol: O'Reilly & Associates, 1993.

[HOA 85] HOARE, C., *Communicating Sequential Processes*, Englewood Cliffs: Prentice-Hall, 1985.

[HOW 88] HOWARD, J. et alli, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, v. 6, n. 1, fevereiro de 1988.

[IEE 88] INSTITUTE OF ELETRICAL AND ELECTRONICS ENGINEERING, *POSIX - Portable Operating System Interface for Computing Environments*, IEEE Computing Society, 1988.

[ISO 81] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, *ISO Open Systems Interconnection - Basic Reference Model*, ISO/TC 97/SC 16 N 719, agosto de 1981.

[KER 84] KERNIGHAN, B. & RITCHIE, D., *The C Programming Language*, Englewood Cliffs: Prentice-Hall, 1984.

[KLE 85] KLEINROCK, L., *Distributed Systems*, Communications of the ACM, v. 28, n. 11, novembro de 1985.

[KOR 90] KORN, D. & KRELL, E., *A New Dimension for the UNIX File System*, Software - Practice and Experience, v. 20, n. 1, 1990.

[LEF 89] LEFFLER, S. at alli, *The Desing and Implementation of The 4.3 BSD Unix Operating System*, Reading: Addison-Wesley, 1989.

[LEV 87] LEVINE, P., *The Apollo DOMAIN Distributed File System*, Theory and Practice of Distributed Operating Systems, Springer-Verlag, 1987.

[MOC 87a] MOCKAPETRIS, P., *Domain Names - Concept and Facilities*, Menlo Park: Network Information Center, 1987 (RFC 1034).

[MOC 87b] MOCKAPETRIS, P., *Domain Names - Implementation and Specification*, Menlo Park: Network Information Center, 1987 (RFC 1035).

[MOR 86] MORRIS, J. et alli, *Andrew: A Distributed Personal Computing Environment*, Communications of the ACM, v. 25, n. 24, abril de 1986.

[MUL 84a] MULLENDER, S. & TANEMBAUM, A., *Immediate Files*, Software - Practice and Experience, v. 14, n. 4, p. 368-368, abril de 1984.

[MUL 84b] MULLENDER, S. & TANENBAUM, A., *Protection and Resource Control in Distributed Operating Systems*, Computer Networks, v. 8, p. 412-432, novembro de 1984.

[MUL 90] MULLENDER, S. et alli, *Amoeba: a Distributed System for the 1990s*, IEEE Computer, p. 44-53, maio de 1990.

[NEC 90] NECHVATAL, J., *Public Key Cryptography*, National Institute of Standards and Technology, 1990.

[OUS 88] OUSTERHOUT, J. et alli, *The Sprite Network Operating System*, IEEE Computer, v. 21, fevereiro de 1988.

[POS 81a] POSTEL, J., *Internet Protocol*, Marina del Rey: USC, 1981 (RFC 791).

[POS 81b] POSTEL, J., *Transmission Control Protocol*, Marina del Rey: USC, 1981 (RFC 793).

[POS 82] POSTEL, J. & SU, Z., *The Domain Naming Convention for Internet User Applications*, Menlo Park: Network Information Center, 1982 (RFC 819).

[POS 87] POSTEL, J. & REYNOLDS, J., *Official Internet Protocols*, Marina del Rey: USC, 1987 (RFC1011).

[RES] RESESSE, R. & TANENBAUM, A. & WILSCHUT, A., *The Design of a High-Performance File Server*, p. 72-82.

[RES 85] RESESSE, R. & TANENBAUM, A., *Distributed Operating Systems*, Computing Surveys, v. 17, n. 4, dezembro de 1985.

[SAN 85] SANDBERG, R. et alli, *Design and Implementation of the Sun Network File System*, Portland: Proceedings of the Summer 1985 USENIX Conference, 1985.

[SAT 90] SATYANARAYANAN, M., *A Survey of Distributed File Systems*, Annual Review of Computer Science, v. 4, p. 73-104, 1989.

[SIL 94] SILBERSCHATZ, A. & GALVIN, P., *Operating Systems Concepts*, Reading: Addison-Wesley, 1994.

[STE 88] STEINER, J. & NEUMAN, C. & SCHILLER, J., *Kerberos: An Authentication Service for Open Network Systems*, Dallas: Proceedings of the Winter 1988 USENIX Conference, 1988.

[STE 92] STEIN, B., *Projeto do Núcleo de um Sistema Operacional Distribuído*, Porto Alegre: UFRGS, 1992 (Dissertação de Mestrado).

[SUN 88] SUN MICROSYSTEMS, *RPC: Remote Procedure Call Protocol Specification (version 2)*, Menlo Park: Network Information Center, 1988, (RFC 1057).

[SUN 89] SUN MICROSYSTEMS, *NFS: Network File System Protocol Specification*, Menlo Park: Network Information Center, 1989, (RFC 1094).

[SUN 90] SUN MICROSYSTEMS, *Network Programming Guide*, Mountain View: Sun Microsystems, 1990.

[TAN 87a] TANENBAUM, A., *A UNIX Clone with Source Code for Operating Systems Courses*, Operating Systems Review, v. 21, n. 1, p. 20-29, janeiro de 1987.

[TAN 87b] TANENBAUM, A., *Operating Systems: Design and Implementation*, Englewood Cliffs: Prentice-Hall, 1987.

[TAN 89] TANENBAUM, A., *Computer Networks*, Englewood Cliffs: Prentice-Hall, 1989.

[TAN 92] TANENBAUM, A., *Modern Operating Systems*, Englewood Cliffs: Prentice-Hall, 1992.